# Capítulo 3

# Web das Coisas: Conectando Dispositivos Físicos ao Mundo Digital

Tiago C. de França, Paulo F. Pires, Luci Pirmez, Flávia C. Delicato, Claudio Farias

***Abstract***

*In the near future the number of physical devices (also called smart objects) connected to the internet will be massive. Those devices can be wireless sensors, mobile phones or any other electronic device of people's daily lives. The Web of Things (WoT) proposal is to integrate smart objects into the Web so that user will be able to access those physical objects via URLs, browse them, and reuse the resources provided by them in the building of composite Web applications, called mashups. This short course introduces the concepts and technologies involved in the WoT. Following, we present the underlying software architecture currently employed in WoT projects. Finally, we present practical examples of application development using the Sun SPOT platform.*

***Resumo***

*Brevemente será imenso o número de dispositivos físicos (chamados objetos inteligentes) conectados a internet. Esses dispositivos podem ser sensores sem fio, celulares ou qualquer outro aparelho eletrônico do cotidiano das pessoas. A Web das Coisas (WoT, do inglês, Web of Things) propõe que os objetos inteligentes sejam integrados a Web, permitindo, desta forma, que os usuários possam acessar tais objetos através de URLs, realizando pesquisas e reutilizando os recursos dos mesmos em aplicações Web chamadas mashups. Este minicurso introduz os conceitos e tecnologias da WoT e apresenta a arquitetura de software subjacente atualmente empregada. Em seguida, são apresentados exemplos práticos de construção de aplicações baseadas no paradigma da WoT utilizando a plataforma Sun SPOT.*

## 3.1. Introdução

Graças ao impressionante progresso no campo de dispositivos embarcados, objetos físicos tais como eletrodomésticos, máquinas industriais, sensores sem fio e atuadores podem atualmente se conectar a internet. Exemplos desses pequenos e versáteis dispositivos são: Chumby [Chumby 2011], Gumstix [Gumstix 2011], Sun SPOTs [Sun 2011], Ploggs [Ploggs 2011], Nabaztag [Nabaztag 2011], Pokens [Pokens 2011], etc.

Segundo a Aliança IPSO (do inglês, IP *for Smart Objects*), em um futuro próximo um grande número de dispositivos embarcados irá suportar o protocolo IP [Hui 2008]. Assim, muitos objetos do dia a dia (como geladeiras, equipamentos de ar-condicionado, dentre outros) brevemente estarão conectados diretamente a internet. A conexão desses objetos do dia a dia com a internet é denominada Internet das Coisas (do inglês, *Internet of Things* - IoT) [Duquennoy 2009]. A IoT oferece novas oportunidades de projetos para aplicações interativas as quais, além de conter documentos estáticos, conterão informação em tempo real referentes a lugares e objetos do mundo físico.

Recentemente surgiu um novo paradigma de desenvolvimento de aplicações inspirado na idéia da IoT, conhecido como Web das Coisas (do inglês *Web of Things* - WoT) [Duquennoy 2009], [Guinard 2010]. Esse novo conceito se baseia no uso de protocolos e padrões amplamente aceitos e já em uso na Web tradicional, tais como HTTP (*Hypertext Transfer Protocol*) e URIs (do inglês, *Uniform Resource Identifier*). O objetivo da WoT é alavancar a visão de conectividade entre o mundo físico e o mundo digital, fazendo com que a Web atual passe a englobar também objetos do mundo físico (chamados objetos inteligentes, do inglês "*smart things*") os quais passarão a ser tratados da mesma forma que qualquer outro recurso Web.

Na WoT, o protocolo HTTP não é utilizado apenas como protocolo de comunicação para transportar dados formatados em conformidade com alguma especificação (como no caso das tecnologias de serviços Web). Em vez disso, o protocolo HTTP é utilizado como mecanismo de suporte padrão a toda interação com os objetos inteligentes. Essa interação ocorre por meio dos principais métodos (GET, POST, PUT e DELETE) desse protocolo, os quais permitem que as funcionalidades dos objetos sejam expostas em interfaces Web bem definidas. Tais interfaces são construídas de acordo com os princípios REST (do inglês, *Representational State Transfer*) [Fielding 2000], [Guinard e Trifa 2009] os quais permitem que os serviços dos objetos inteligentes sejam expostos como recurso em uma abordagem ROA (do inglês, *Resource-Oriented Architecture*) [Guo 2010], [Mayer 2010]. Além da padronização e simplificação no processo de desenvolvimento, a utilização do protocolo HTTP também elimina problemas de compatibilidade entre diferentes fabricantes, protocolos e formatos específicos [Duquennoy 2009].

A realização da visão da WoT requer, portanto, que a *World Wide Web* atual seja estendida de modo que objetos do mundo real e dispositivos embarcados possam ser incorporados a ela de forma transparente. Essa extensão é obtida através da utilização do protocolo HTTP e dos princípios REST na criação de APIs (do inglês, *Application Programming Interface*) RESTful que façam com os objetos inteligentes se tornem recursos Web. A forma como tais objetos inteligentes são representados e expostos como recursos na Web tem diferentes granularidades, podendo um recurso ser definido como sendo um objeto ou dispositivo sensor individual, como uma rede de

sensores sem fio (RSSF), ou até mesmo como dados agregados oriundos de diferentes RSSFs. Além disso, através do uso de plataformas de *middleware*, por exemplo, podem ser providos serviços no topo dos recursos conectados a Web, de modo a facilitar a rápida combinação de múltiplos recursos para criar aplicações de valor agregado denominadas *mashups* físicos [Delicato et al. 2010], [Guinard et al. 2009]. Esses *mashups* são aplicações *ad-hoc* da Web 2.0 que permitem colaboração e compartilhamento de informações através da composição de recursos disponíveis na Web [Bezerra et al. 2009].

O objetivo geral deste minicurso é apresentar o estado da arte no desenvolvimento de aplicações para a "Web das Coisas". Seus objetivos específicos são: (i) fornecer uma visão geral do conceito de "Internet das Coisas" e sua evolução para a "Web das Coisas"; (ii) apresentar a arquitetura de software atualmente empregada nos projetos de "Web das Coisas"; (iii) apresentar soluções atuais da WoT; e, por fim, (iv) apresentar como aplicações baseadas no conceito da Web das Coisas podem ser desenvolvidas. A plataforma alvo abordada no curso para os dispositivos embarcados será a Sun SPOT [Sun SPOT 2011].

## 3.2. Da Internet das Coisas a Web das Coisas

Espera-se que em um futuro próximo tanto computadores como objetos físicos estejam conectados a internet [Atzori et al. 2010]. Essa interconexão de dispositivos na internet, chamada Internet das Coisas (IoT, do inglês *Internet of Things*), possibilitará que tais dispositivos sejam utilizados remotamente por humanos ou até mesmo por outros dispositivos [Tan e Wang 2010]. Dentre as propostas existentes na IoT, está a Web das Coisas, a qual propõe a adoção dos padrões Web a fim de oferecer uma base comum para que diferentes tipos de dispositivos possam ser beneficiados pelas tecnologias existentes na Web, além de facilitar o desenvolvimento de aplicativos para tais dispositivos. Nesta Seção são descritas as principais características da Internet das Coisas, seguida da apresentação da Web das Coisas.

### 3.2.1. Internet das Coisas

Atualmente, a Internet das Coisas (IoT) vem ganhando grande destaque no cenário das telecomunicações e está sendo considerada a revolução tecnológica que representa o futuro da computação e comunicação [Tan e Wang 2010], [Atzori et al. 2010]. Devido a importância da IoT, o Conselho Nacional de Inteligência dos EUA (NIC) a considera como uma das seis tecnologias civis mais promissoras e que mais impactarão a nação no futuro próximo. O NIC prevê que em 2025 todos os objetos do cotidiano (por exemplo, embalagens de alimento, documentos e móveis) poderão estar conectados a internet [NIC 2008].

Graças ao paradigma IoT, estima-se que uma grande quantidade de objetos estarão conectados à internet, se tornando os maiores emissores e receptores de tráfego da rede. Esses objetos podem ser quaisquer dispositivos, tais como eletrodomésticos, pneus, sensores, atuadores, telefones celulares, entre outros, que possam ser identificados e interligados a internet para trocar informações e tomar decisões para atingir objetivos comuns [Atzori et al. 2010]. A ITU (*International Telecommunication Union*) *Internet Reports* (2005) apontou que na Internet das Coisas qualquer objeto capaz de ser conectado em rede poderá se comunicar a qualquer tempo e em qualquer lugar. A Figura 3.1 mostra as novas dimensões do mundo das tecnologias da

comunicação e informação da internet no futuro. Nessa figura é possível observar “o que” pode ser conectado a internet, “quando” pode ser conectado e “onde” pode se conectar.

Obviamente, a ampla difusão do paradigma IoT acarretará um forte impacto na vida cotidiana dos usuários. Isso ocorrerá porque diversas aplicações estarão a disposição desses usuários, entre elas: aplicações de controle de ambiente; aplicações de assistência a vida em ambientes de saúde; aplicações de automação e produção industrial, logística, segurança, entre outras [Atzori et al. 2010], [Yun e Yuxin, 2010]. As mudanças proporcionadas pela IoT também trarão novas oportunidades de negócio que, impulsionadas pelas demandas da população, contribuirão de maneira inestimável para a economia [ITU Internet Reports 2005], [Tan e Wang 2010], [Yongjia 2010].

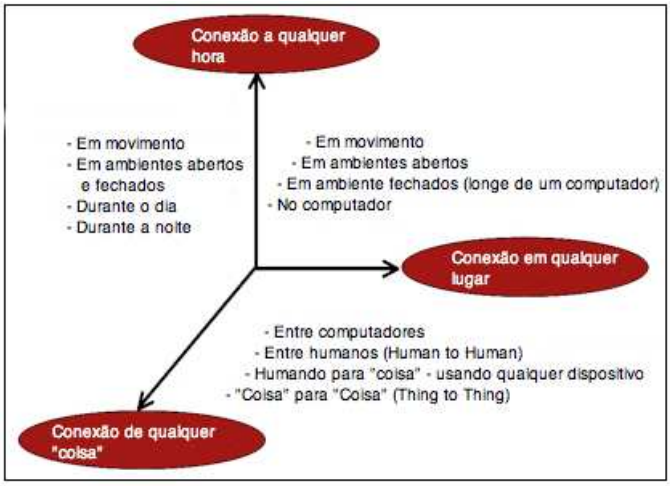


**Figura 3.1 - Uma Nova Dimensão, adaptado de [ITU Internet Reports 2005]**

O termo IoT recebeu diferentes definições na literatura. Algumas definições e pesquisas focaram no termo “internet” [Atzori et al. 2010] e observaram a IoT do ponto de vista de redes [Atzori et al. 2010] . Outras definições focaram no termo genérico “coisas” e as pesquisas que focam nesse termo buscam a integração de objetos em um arcabouço comum. Também surgiram definições que focaram em questões semânticas, observando a IoT do ponto de vista da comunicação entre dispositivos distintos. De fato, para a TIC (Tecnologia da Informação e Comunicação), a expressão composta “Internet das Coisas” representa uma rede mundial de objetos heterogêneos e endereçáveis, interligados e se comunicando através de protocolos de comunicação padronizados. A Figura 3.2 ilustra o fato exposto acima, de que o paradigma da IoT pode ser visto de acordo com  três visões principais: uma focada nas coisas, outra focada na semântica e ainda outra cujo foco é a internet [Atzori et al. 2010].

Os trabalhos focados nas “coisas” buscam apresentar propostas que garantam o melhor aproveitamento dos recursos dos dispositivos e sua comunicação [Atzori et. al 2010]. Por outro lado, os trabalhos que apresentam propostas focadas na semântica dos objetos da IoT são importantes devido a grande quantidade de itens que estarão conectados a internet em um futuro próximo. Tais trabalhos apresentam propostas que estão focadas na representação, armazenamento, interconexão, pesquisa e organização da informação gerada na IoT, buscando soluções para a modelagem das descrições que permitam um tratamento adequado para os dados gerados pelos objetos [Atzori et al. 2010].

Os trabalhos que focam na visão orientada a internet procuram criar modelos e técnicas para a interoperabilidade dos dispositivos em rede. Um exemplo é o padrão IPSO (IP for *Smart Objects*), o qual apresenta a proposta 6LowPAN, na qual o protocolo IP é adaptado para ser utilizado em dispositivos que possuem recursos de hardware reduzidos [RFC4944 2007], [Hui e Culler 2008].

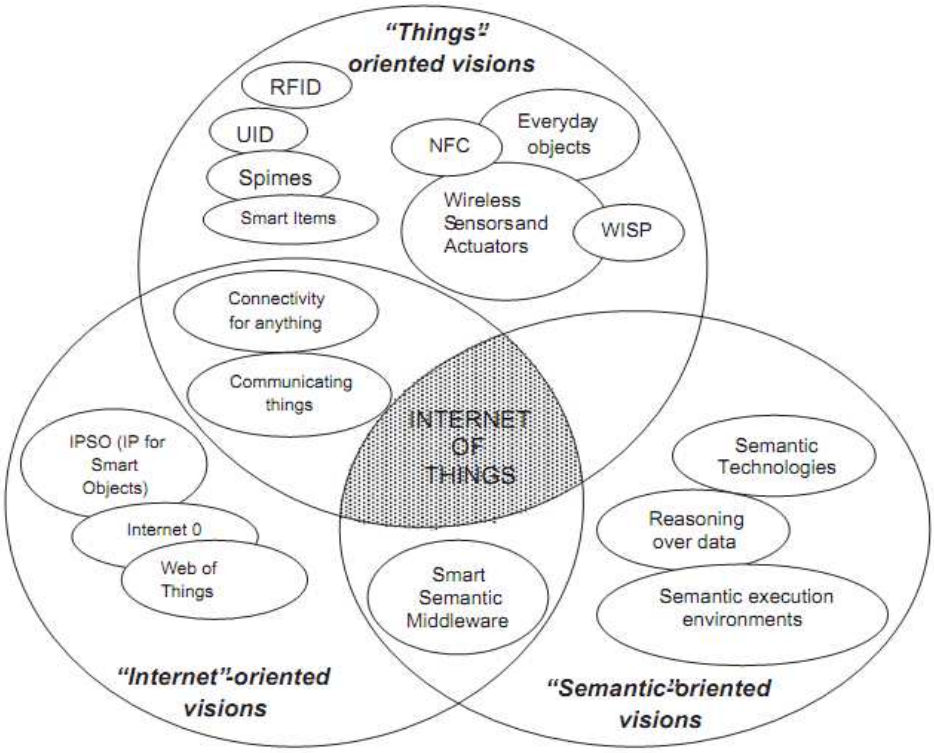


**Figura 3.2 - "*Internet of Things*" como resultado de diferentes visões [Atzori et al. 2010]**

Apesar de existirem diversos trabalhos na literatura que tratam temas relacionados à IoT, ainda é necessário superar uma série de desafios tecnológicos e sociais para que tal paradigma seja amplamente utilizado e difundido. Um tipo de desafio tecnológico está relacionado com os baixos recursos computacionais e de energia dos dispositivos da IoT. Portanto, os trabalhos nessa área, além de apresentarem propostas que sejam escaláveis, dado que será potencialmente enorme o número de dispositivos que farão parte da IoT, precisam também apresentar soluções que utilizem os recursos dos dispositivos de forma eficiente. Um outro tipo de desafio consiste na definição de modelos de desenvolvimento de aplicações que tratem questões tais como a padronização do acesso aos serviços e informações oferecidos pelos dispositivos, segurança e privacidade, modelo de programação, etc. O paradigma Web das Coisas, descrito na seção a seguir, se preocupa em apresentar soluções para esse tipo de desafio.

### 3.2.2. Web das Coisas

A inclusão de dispositivos físicos e aparelhos eletrônicos (redes de sensores sem fio, telefones celulares, etc.) na internet traz consigo inúmeras possibilidades de novas aplicações, as quais podem utilizar as informações e serviços desses dispositivos com diferentes propósitos. Entretanto, a maioria dos objetos são atualmente conectados a internet (e algumas vezes a Web) utilizando softwares e interfaces proprietárias, o que

torna onerosa a criação de aplicações que integram dados e serviços providos por diferentes dispositivos [Guinard 2010]. Além disso, o uso das linguagens, protocolos e interfaces específicas de cada tipo de dispositivo também faz com que o desenvolvimento de aplicativos para os mesmos seja uma tarefa complexa, pois é necessário que o desenvolvedor possua conhecimento especializado para cada dispositivo utilizado no projeto. Dessa forma, para facilitar o desenvolvimento de serviços no topo desses dispositivos, permitindo também que os serviços e os dados dos mesmos sejam compostos em diferentes aplicações, é necessário utilizar uma linguagem comum a diferentes dispositivos [Guinard e Trifa 2009], [Guinard 2010].

A WoT propõe que os protocolos Web sejam utilizados como linguagem comum para integração de dispositivos físicos no meio digital. A inclusão dos dispositivos físicos na Web permite que os seus dados e serviços sejam reutilizados em diferentes aplicações [Duquennoy et al. 2009]. A integração dos dispositivos a Web ocorre no nível de aplicação, isto é, acima da conectividade de rede [Guinard et al. 2010]. Tal integração permite que ferramentas e técnicas da Web (por exemplo, navegadores, ferramentas de busca e sistemas de *cache*), linguagens da Web (por exemplo, HTML e *JavaScript*) e técnicas de interação com o usuário (por exemplo, navegação, vinculação e *bookmarking*) possam ser aplicadas para objetos do mundo real [Guinard e Trifa 2009], [Guinard et al. 2010]. Desta forma, a WoT possibilita a agregação de valor às informações providas pelos objetos físicos através da utilização de todos os recursos disponíveis na Web (por exemplo, *cache*, balanceamento de carga, indexação e pesquisa), o que por sua vez alavanca a concretização da visão da IoT [Guinard e Trifa 2009].

Atualmente é possível encontrar trabalhos que apresentam propostas de sistemas que integram objetos com a internet. Um exemplo são os projetos que promovem a integração de redes de sensores com a internet, tais como o SenseWeb [SenseWeb 2011] e o Pachube [Pachube 2011]. Ambos oferecem uma plataforma para que as pessoas compartilhem os dados coletados por sensores através de serviços Web. Porém, essas abordagens não são tão abrangentes quanto à proposta da Web das Coisas, pois elas utilizam servidores que recebem e armazenam dados de sensores de forma centralizada. A WoT por outro lado, preconiza que qualquer objeto físico pode enviar seus dados para pontos descentralizados e esses dados podem ser utilizado e reutilizados em diferentes aplicações [Guinard 2010].

Uma possível abordagem para implementar o conceito de WoT são os padrões WS-* (como o SOAP). Os WS-* geralmente utilizam o protocolo HTTP para realizar tarefas de comunicação na pilha de protocolos utilizada pelos dispositivos. Nesse caso, o HTTP é utilizado para transportar a mensagem SOAP (a qual é codificada em XML), evitando assim possíveis problemas com *firewalls*, já que geralmente a porta 80 (usada pelo HTTP) é liberada nos *firewalls* [Guinard e Trifa 2009], [Shelby 2010]. Contudo, a interoperabilidade obtida através do emprego dos padrões WS-* é alcançada através da adição de uma camada de software nas aplicações. Tal adição implica uma maior complexidade de software, não sendo, desta forma, a solução mais adequada para ser aplicada em dispositivos com recursos limitados [Guinard e Trifa 2009].

Diferentemente da abordagem WS-*, a Web das Coisas propõe que a Web atual seja estendida de modo a incorporar objetos e dispositivos embarcados do mundo real como qualquer outro serviço Web. A extensão da Web proposta pela WoT é realizada

através da adoção do protocolo HTTP como protocolo de aplicação. Isso significa que esse protocolo deve ser utilizado como interface base para realizar toda a interação com os recursos disponíveis [Guinard e Trifa 2009], [Shelby 2010], e não apenas para transportar passivamente as mensagens trocadas.

Uma abordagem que está sendo bastante utilizada juntamente com protocolo HTTP na criação de sistemas distribuídos na Web é o estilo arquitetural REST [Mayer 2010] (do inglês, *Representation State Transfer*). Esse estilo arquitetural pode ser empregado para desenvolver sistemas que seguem uma arquitetura orientada a recursos (ROA, do inglês *Resource Oriented Architecture*) [Mayer 2010]. O REST define um conjunto de princípios que, ao serem adotados, dão origem a sistemas RESTful. Os sistemas RESTful são menos acoplados, mais leves, eficientes e flexíveis do que os sistemas Web baseados em WS-* e podem ser facilmente reutilizados [Guinard e Trifa 2009], [Sandoval 2009]. Além disso, os princípios REST podem ser mapeados nos métodos básicos do protocolo HTTP (GET, POST, UPDATE e DELETE) para criar sistemas CRUD (*Create, Read, Update, Delete*) de uma aplicação RESTful. Os recursos dos sistemas RESTful são identificados e encapsulados por um URI. A utilização do protocolo HTTP como protocolo de aplicação admite que os recursos possuam várias representações e permite que os clientes selecionem, dentre as representações disponíveis, aquela que melhor se adéqüe as necessidades da aplicação [Sandoval 2009]. Essas características fizeram do REST a opção mais adequada para construção de APIs Web para acesso a objetos do mundo real [Guinard e Trifa 2009], [Ostermaier et al. 2010].

A Web das Coisas emprega os princípios REST para disponibilizar as funcionalidades dos dispositivos inteligentes na Web utilizando duas abordagens. Na primeira abordagem, são implantados servidores Web embarcados em dispositivos inteligentes e as funcionalidades desses dispositivos são disponibilizadas na forma de recursos RESTful. Na segunda abordagem, quando um objeto inteligente não possui recursos de hardware suficientes para executar um servidor embarcado, é possível utilizar outro dispositivo como ponte para disponibilizar as funcionalidades do dispositivo inteligente na Web através de uma interface RESTful.

Embora a utilização de uma arquitetura RESTful permita que objetos físicos se tornem parte da Web, a WoT também propõe a utilização de mecanismos que foquem no desenvolvimento e prototipagem de aplicativos interativos que façam com que os recursos dos objetos físicos sejam utilizados em diferentes aplicações [Guinard e Trifa 2009], [Guinard 2010]. Nesse sentido, os trabalhos da WoT propõem que sejam utilizadas aplicações da Web 2.0 chamadas *mashups*. Os *mashups* são aplicativos criados a partir da composição de recursos Web. Como qualquer aplicação da Web 2.0, os *mashups* são construídos com base em um conjunto de tecnologias (por exemplo, Atom [Atom 2011]) que dão suporte ao desenvolvimento de interfaces altamente interativas e simples ao usuário, semelhante ao que acontece com aplicativos *desktop* [Bezerra et al. 2009]. Os *mashups* criados a partir da composição de dados e serviços de dispositivos físicos com outros recursos Web são chamados *mashups* físicos. Esse tipo de *mashup* foca no reuso e prototipagem de objetos físicos do mundo real em diferentes aplicações. [Duquennoy et al. 2009], [Guinard 2010].

Logo, é possível resumir que na WoT o protocolo HTTP é adotado como linguagem comum entre diferentes dispositivos e o seu emprego em conformidade com

o princípio arquitetural REST permite que as funcionalidades desses dispositivos sejam expostas como recursos Web que possuem interfaces bem definidas. Isso irá permitir que os dados e recursos dos dispositivos sejam reutilizados em diversas aplicações. A função dos *mashups* físicos é permitir que os usuários construam aplicações formadas a partir da composição dos recursos e dados desses dispositivos os quais se tornam passíveis de serem combinados e recombinados em tempo de execução para resolverem requisitos de mais alto nível.

## 3.3. Concretizando a Web das Coisas: REST & ROA

Para uma melhor compreensão da Web das Coisas, esta Seção apresenta os conceitos envolvidos com o desenvolvimento de aplicações Web RESTful. Além dos conceitos, também é apresentada uma abordagem prática da aplicação dos mesmos no contexto de aplicações RESTful.

### 3.3.1. REST

O termo REST, descrito por Roy Fielding (2000) em sua tese de doutorado, define um conjunto de princípios que podem ser aplicados na construção de sistemas com uma arquitetura orientada a recursos (ROA). O REST é um estilo de arquitetura de software que pode ser aplicado no desenvolvimento de sistemas denominados sistemas RESTful [Sandoval 2009]. ROA e REST são utilizados na concepção da implementação de sistemas focados em recursos [Mayer 2010]. Um recurso pode ser qualquer componente de uma aplicação que seja importante o suficiente para ser endereçado na Web através de pelo menos uma URI. Ou seja, um recurso é tudo aquilo que deve ser acessado pelo cliente e transferido entre o mesmo e um servidor [Mayer 2010]. Por exemplo, um recurso pode ser uma lista de cursos de uma instituição, ou mesmo cada curso dessa instituição, os quais possuem disciplinas que por sua vez são subrecursos do curso e assim por diante. Esta Subseção aborda os princípios REST, o modelo de arquitetura orientada a recursos e a construção de sistemas RESTful.

#### 3.3.1.1. Princípios REST

Os princípios REST podem ser facilmente empregados (e explicados) com o protocolo HTTP [Pautasso 2009]. Por esse motivo, esse protocolo tem sido amplamente utilizado no desenvolvimento de sistemas RESTful [Lucchi et al. 2008].

O HTTP é um protocolo da camada de aplicação para sistemas de hipermídia, colaborativos e distribuídos, baseado no modelo de comunicação requisição/resposta que pode ser utilizado para realizar diferentes tarefas. O HTTP é um protocolo sem manutenção de estado (*stateless*) e define como é feita a troca de mensagens entre o cliente e o servidor; ou seja, esse protocolo define como um cliente requisita recursos em um servidor e como este responde a tais requisições. O HTTP define um conjunto de métodos que são utilizados nas requisições, dentre os quais se destacam os métodos GET, POST, PUT e DELETE [Lucchi et al. 2008]. Outra característica importante desse protocolo é a negociação da representação de dados, a qual pode ser feita através da utilização dos cabeçalhos (por exemplo, *Accept* ou *Accept-Language*) da requisição.

A Figura 3.3 mostra o formato de uma requisição HTTP. Nessa figura é possível observar os campos do cabeçalho de uma requisição HTTP contendo o método GET, o caminho (*path*) do recurso solicitado e a versão do protocolo HTTP. Uma requisição

contendo o método GET pode possuir dados incluídos no *path* da requisição. Esses dados são separados do *path* do recurso pelo caractere de interrogação "?" e diferentes dados são separados pelo caractere "&" (cada dado possui uma variável e um valor os quais são separados pelo sinal de igualdade). O campo *Host* especifica o endereço do servidor na internet e o número da porta do recurso solicitado. O campo *User-Agent* contém informações sobre o agente de usuário (nessa figura, o agente é o mozilla/5.0) que originou a requisição. O campo *Accept* indica quais representações o cliente espera receber (nesse caso as representações podem ser HTML, XML OU JSON [JSON 1999]). O campo *Connection* permite que o emissor da requisição especifique algumas informações concernentes a uma requisição em particular (o valor *keep-alive* que aparece na requisição apresentada na figura é utilizado para indicar que uma conexão do protocolo de transporte deve permanecer aberta para ser reusada no envio e recepção de múltiplas requisições e respostas). Para maiores detalhes sobre o protocolo HTTP ver [RFC2068 1997] e [RFC2616 1999].

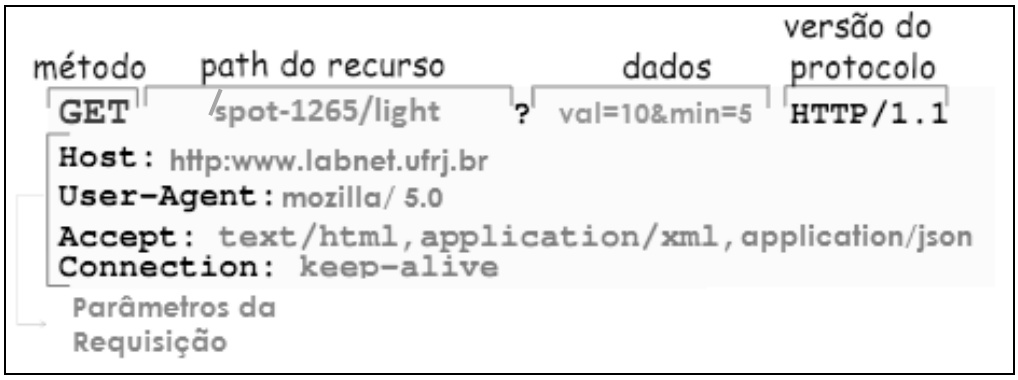


**Figura 3.3 - Requisição HTTP contendo o método GET**

Como o HTTP possibilita a utilização de diferentes representações, as aplicações podem ser construídas independentemente da forma como os dados serão transferidos [RFC2616 1999]. As características do HTTP fornecem o suporte necessário para a realização dos princípios REST. Esses princípios são: identificação única e global para os recursos, uma interface uniforme de acesso aos recursos, endereçabilidade dos recursos (permite que os mesmos sejam vinculados), suporte a múltiplas e independentes representações para o recurso e interação sem manutenção de estado [Sandoval 2009].

O princípio da **identificação dos recursos** está relacionado ao uso de URIs que fornecem endereços únicos e globais para identificação de um recurso [Pautasso et al. 2008]. URIs também são utilizadas para indicar o escopo da informação provendo meios que permitem a navegação entre recursos que interagem entre si. Isso significa que um identificador pode indicar os subrecursos relacionados com um recurso em um dado momento. Por exemplo, o *path* do recurso "/spot-1265/light" presente na requisição da Figura 3.3 indica que *light* é um subrecurso de spot-1265. Uma URI também é utilizada para endereçar a interação entre recursos sem a necessidade do uso do corpo da resposta. No contexto da Web, o uso de URIs possibilita a utilização de *links* para recursos os quais podem ser estabelecidos utilizando esquemas bem conhecidos [Mayer 2010].

O princípio da **interface uniforme** define que o servidor deve ser capaz de determinar o que deve ser feito ao receber uma requisição HTTP em uma URI apenas

observando do método presente nessa requisição [Pautasso et al. 2008], [Mayer 2010]. Os métodos das requisições HTTP (GET, POST, PUT ou DELETE) são utilizados para indicar ao provedor do recurso a ação que deve ser realizada. Assim, cada método representa uma ação específica sobre o recurso, podendo ser executadas quatro ações (operações): obter a representação de um recurso (GET), criar um novo recurso (POST), alterar um recurso (PUT) e remover um recurso (DELETE). Essas operações geralmente são descritas como sendo o CRUD (*Create, Retrieve, Update e Delete*) da aplicação [Sandoval 2009]. Por exemplo, o método GET da requisição exemplificada na Figura 3.3 indica ao servidor (localizado em *www.labnet.ufrj.br*) que o cliente espera receber uma representação do recurso "/light" o qual é subrecurso de "/spot-1265". Essa figura é um exemplo de uma requisição que está em conformidade com os princípios de endereçabilidade e interface uniforme.

O princípio da **vinculação de recursos** está relacionado com o uso da abordagem HATEOAS (do inglês, *Hypermedia as Engine Of Application State*). Nessa abordagem, a vinculação dos recursos é realizada através de *links* que são criados a partir das URIs dos recursos. Os *links* podem apontar para qualquer recurso na Web, até mesmo recursos externos ao provedor que está sendo acessado em um dado momento. Isso é possível graças a utilização de um esquema de nomes global que possibilita que qualquer recurso Web seja ligado a outro. Tal característica permite que o servidor ofereça *links* para que o cliente mude de um estado da aplicação para outro, tornando a aplicação dinâmica. Ou seja, as aplicações são consideradas como uma máquina de estado onde cada página representa um estado e os *links* representam todas as possíveis transições de estado a partir do estado corrente. Sempre que possível, os *links* de navegação entre recursos relacionados devem ser disponibilizados na representação do primeiro recurso acessado. Por exemplo, considere um sistema de uma instituição de ensino que possui um recurso chamado "lista_cursos", o qual lista todos os cursos dessa instituição quando tal recurso é acessado. O provedor do recurso deve oferecer *links* criados a partir da URI de identificação de cada recurso que representa um curso da instituição. Esses *links* são oferecidos com o objetivo de permitir que cada curso possa ser acessado a partir da representação obtida do recurso "lista_cursos".

O princípio da **representação de um recurso** define como serão os formatos das mensagens trocadas entre cliente e servidor. A representação de um recurso representa o valor do dado no momento em que foi recebida a requisição [Sandoval 2009]. Um recurso pode possuir várias representações. As opções de representações de recurso permitem que o consumidor do recurso escolha, entre as representações disponíveis, aquela que ele deseja receber. Exemplos de representações de recurso são o XML, JSON, HTML, entre outros. A especificação das representações que o cliente deseja receber podem ser passadas no campo *Accept* do cabeçalho HTTP. Por exemplo, o campo *Accept* da requisição apresentada na Figura 3.3 indica três opções de representações que o cliente espera receber, as quais são: text/html (indicando formato HTML), application/xml (formato XML) e application/json (indicando formato JSON).

O último princípio REST define que um servidor deve ser *stateless*, ou seja, **sem estado**. Esse princípio define que não pode haver manutenção de informações sobre o usuário em sessões no lado do servidor. Assim, cada requisição enviada ao provedor do recurso deve ter todas as informações necessárias para a sua compreensão. Se alguma informação deve ser mantida sobre um recurso, esta deve ser mantida no lado do cliente. Essa abordagem está relacionada principalmente a escalabilidade do sistema,

pois se um servidor mantivesse as sessões com informações para cada usuário ele poderia ter seu desempenho afetado quando estivesse tratando de muitos acessos concorrentes [Pautasso et al. 2008]. Além disso, esse princípio diminui a dependência do cliente com relação ao provedor do recurso, pois as requisições submetidas são independentes de um servidor específico. Um exemplo dessa situação ocorre quando um cliente que recebeu um *link* ao acessar o recurso em um servidor, o qual ficou inativo logo após responder ao cliente (por problemas de hardware, por exemplo) e foi automaticamente substituído por outro servidor, pode utilizar esse *link*, pois ele irá funcionar da mesma forma que antes e a troca do servidor será transparente para o usuário [Webber et al. 2010].

### 3.3.2. ROA

Apesar dos princípios REST terem sido apresentados neste minicurso juntamente com o uso das estruturas de URI e dos mecanismos do HTTP, como também fazem muitos outros autores ([Sandoval 2009], [Webber et al. 2010], [Mayer 2010]), o REST é independe de tecnologia e seus princípios não estão obrigatoriamente interligados a Web. O REST também não é um si uma arquitetura, mas possui termos genéricos que podem ser utilizados para definir uma arquitetura. A falta de uma arquitetura que possa ser empregada no desenvolvimento de aplicações que seguem os princípios REST pode levar o desenvolvedor ou arquiteto de software a empregar de forma equivocada esses princípios [Pautasso et al. 2008]. Por exemplo, as aplicações que empregam os princípios REST podem se tornar um híbrido de REST com RPC (*Remote Procedure Call*) [Richardson e Ruby 2007], [Pautasso et al. 2008], o que não é desejável.

Em 2007, Richardson e Ruby apresentaram uma arquitetura orientada a recursos (do inglês, *Resource-Oriented Architecture* – ROA) especificando em detalhes como empregar os princípios REST juntamente com as tecnologias da Web (HTTP e URI) em um modelo de boas práticas que podem ser aplicadas no desenvolvimento de serviços RESTful. Esta Subseção apresenta as principais características dos conceitos e propriedades de ROA conforme propostos por [Richardson e Ruby 2007].

#### 3.3.2.1. URIs

Em uma abordagem ROA, as URIs dos recursos devem possuir uma correspondência intuitiva com o recurso que elas identificam e a sua estrutura deve variar de forma previsível. Por exemplo, as URIs http://www.labnet.nce.ufrj.br/rssf/temperatura, http://www.labnet.nce.ufrj.br/alunos/john ou http://www.labnet.nce.ufrj.br/redes/redesemfio/zigbee permitem que o cliente infira quais são os subrecursos disponíveis em um diretório da URI. Ou seja, ao observarem essas URIs, os clientes esperarão que um dado de temperatura possa ser acessado a partir do diretório “rssf/temperatura” e que um aluno (por exemplo, John) possa ser acessado a partir do diretório “alunos”.

Outra característica relacionada ao uso de URIs abordada em ROA diz respeito ao relacionamento entre URIs e recursos. Um recurso deve possuir no mínimo uma URI, mas nada impede que ele possua várias. Porém, sempre que possível, um recurso deve possuir apenas uma URI. Isso porque, utilizar várias URIs para identificar o mesmo recurso diminui a importância de uma URI, pois dificulta o relacionamento da URI com o recurso que ela identifica [Richardson e Ruby 2007]. Por exemplo, se um recurso possui várias URIs será mais difícil para um *proxy* HTTP, por onde passam as requisições de uma rede, identificar se uma resposta contendo uma representação de um

recurso já está armazenada em seu *cache*. Do ponto de vista de um usuário, é mais difícil associar a URI com um recurso quando este possui vários URIs.

#### 3.3.2.2. Endereçabilidade

A propriedade ROA que define que um recurso deve ser endereçável pode ser facilmente observada na Web. Essa propriedade está relacionada com o uso de uma URI para identificar e localizar um recurso de um sistema RESTful. A endereçabilidade também é comum para as aplicações REST-RPC (aplicações que são um híbrido entre REST e RPC), pois toda aplicação na Web precisa ser endereçável para poder ser acessada. Richardson e Ruby (2007) apontaram que, para o usuário final, a endereçabilidade é o aspecto mais importante das páginas ou serviços Web. Um recurso endereçável pode ser acessado sempre que sua URI for utilizada na Web. Por exemplo, esses recursos podem ser acessados através de *links* de hipertexto, ou mesmo quando a URI é enviada de um usuário para outro através do e-mail. Ser endereçável também permite que um documento enviado como resposta a uma requisição possa ser mantido em *cache* nos *proxies* HTTP. Por exemplo, a página endereçada pela URI http://www.labnet.nce.ufrj.br/projetos/webofthings pode ser mantida em *cache* para ser devolvida como reposta a uma segunda requisição HTTP que passe por esse *proxy* e que seja enviada para a mesma URI.

#### 3.3.2.3. Sem Estado

Enquanto o REST define que uma aplicação RESTful deve ser sem estado, ROA define como construir uma aplicação sem estado. Ser sem estado significa que uma requisição HTTP deve ser independente de outras requisições anteriores. Para que isso seja possível, as requisições HTTP devem ser auto-contidas, ou seja, uma requisição HTTP deve ter em si toda informação necessária para que possa ser processada. Um servidor que não mantém estado deve ser capaz de processar uma requisição sem precisar utilizar informações de requisições anteriores. Se uma informação é suficientemente importante para ser mantida como uma sessão no servidor, então esta informação deve ser transformada em um recurso.

Apesar da expressão “sem estado”, ROA define dois tipos de estado: o estado da aplicação, o qual deve ser mantido no cliente; e o estado do recurso, que deve ser mantido no servidor. Um exemplo de estado da aplicação pode ser observado nos sistemas de busca da Web (por exemplo, o sistema de busca do Google). Nesses sistemas, o cliente faz uma busca enviando os parâmetros da consulta (geralmente esses parâmetros são passados na URI) para o provedor do recurso. Ao receber o resultado da consulta, o cliente poderá navegar entre páginas que mostram esses resultados. A navegação ocorre quando o cliente passa de uma página contendo um subconjunto do resultado da busca para outra página contendo outro subconjunto do resultado dessa busca. Para tanto, o cliente mantém os parâmetros da consulta (as palavras chave utilizadas) e a identificação da página atual na qual está localizado (página um, dois, três, etc.). Ao navegar entre as páginas que contêm o resultado da busca, o cliente deve incluir nas requisições os parâmetros da consulta e a página que está querendo acessar. Dessa forma, o servidor só precisará tratar o estado da aplicação ao receber uma requisição.

O estado do recurso deve ser mantido no servidor e deve ser igual para todos os clientes. Por exemplo, quando um serviço de fotos da Web (como o Flickr) salva uma

nova foto, essa deve ser considerada um novo recurso. Essa figura ganha sua própria URI e por isso poderá ser acessada nas próximas requisições. Ao ser transformada em um recurso, a figura poderá ser pesquisada, acessada, alterada ou apagada pelos clientes. A figura é parte do estado do recurso e permanece no servidor até que o cliente que possui as permissões necessárias a apague.

#### 3.3.2.4. Representação

As representações de um recurso podem ser enviadas tanto do cliente para o servidor quanto do servidor para o cliente. As representações são enviadas pelo cliente quando este deseja criar ou atualizar um recurso no servidor (é isso que acontece quando um cliente envia uma foto para ser salva no Flickr). O sentido contrário do envio de uma representação ocorre quando um cliente envia uma requisição para o servidor interessado em obter uma representação de um recurso.

Quando o cliente deseja obter uma representação de um recurso do servidor, esse cliente deve ser o mais explícito possível sobre qual representação deseja receber, já que um recurso pode possuir mais de uma representação. ROA define duas formas de especificar uma representação. A primeira forma é utilizar a URI para indicar qual é a representação que o cliente deseja receber. Por exemplo, na requisição http://twitter.com/statuses/public_timeline.{xml, json, rss} a URI é utilizada para especificar o tipo de representação (XML, JSON ou RSS). Segundo Richardson e Ruby (2007) utilizar URI é a forma mais explícita com que o cliente anuncia o tipo de representação que ele espera receber. A segunda forma de especificar a representação é utilizar os cabeçalhos *Accept* ou *Accept-Language* do protocolo HTTP para indicar ao servidor a representação (XML ou JSON, por exemplo) ou o idioma (inglês, português, etc.) do recurso que o cliente deseja receber.

#### 3.3.2.5. *Links* e Conectividade

As representações dos recursos podem ser utilizadas para fornecer *links* para outros recursos relacionados com o primeiro recurso acessado. Por exemplo, quando uma busca é feita no serviço de busca do Google, a representação enviada para o cliente contém *links* para cada resultado dessa consulta. O termo conectividade definido pelo ROA é um sinônimo de HATEOAS, visto que o conceito é o mesmo: os recursos podem conter *links* para outros recursos em suas representações.

#### 3.3.2.6. Interface Uniforme

Em uma abordagem ROA as operações realizadas sobre os recursos são mapeadas nos quatro métodos básicos do protocolo HTTP. O método GET é utilizado para recuperar a representação de um recurso. A resposta a uma requisição GET inclui em seu corpo uma representação do recurso. O método DELETE é utilizado para apagar um recurso já existente. A resposta para uma requisição DELETE pode conter uma mensagem indicando o status da operação solicitada ou apenas um código HTTP.

Em ROA os métodos utilizados para criar recursos são o PUT e o POST. O PUT também é utilizado para atualizar um recurso já existente. Apesar do ROA especificar as situações quando utilizar o método POST ou PUT para criar um novo recurso, muitos sistemas RESTful utilizam apenas o POST com essa finalidade, visto que o método PUT também é utilizado para atualizar um recurso [Sandoval 2009], [Webber et al.

2010]. As requisições PUT e POST podem conter em seu corpo uma representação do recurso que será criado ou atualizado.

Além desses métodos, ROA define como utilizar os métodos HEAD e OPTIONS do HTTP para obter informações sobre um recurso. O HEAD é utilizado para obter metadados sobre um recurso sem que a resposta para uma requisição possua a representação completa desse recurso. O OPTIONS é utilizado para verificar qual método HTTP um recurso suporta. A resposta a uma requisição que contém o método OPTIONS fornece o subconjunto da interface uniforme (HTTP) que este recurso suporta através do cabeçalho *Allow* (por exemplo, *Allow*: GET, HEAD indicam que um recurso suporta esses dois métodos).

Outra definição apresentada em ROA diz respeito aos efeitos que uma requisição causa sobre um recurso. As requisições podem ser seguras (*Safe*) ou idempotente. As requisições seguras são aquelas que são utilizadas para obter uma representação de um recurso ou alguma outra informação sobre esse recurso sem causar alteração no estado do servidor. Por exemplo, uma requisição GET pode ser enviada centenas de vezes para obter uma representação de um recurso sem que nenhuma dessas requisições cause alterações no recurso solicitado. As requisições que alteram o estado do recurso são consideradas idempotente quando outras requisições iguais a primeira não são capazes de causar uma alteração diferente a causada pela primeira requisição. O conceito de idempotente é análogo ao da matemática, que utiliza esse termo para indicar, por exemplo, que na multiplicação o zero é idempotente, pois qualquer número multiplicado por zero sempre vai ser igual zero. Analogamente, uma requisição é idempotente se a alteração ocasionada por essa requisição for mantida mesmo que outras requisições semelhantes sejam enviadas em seguida. Por exemplo, ao apagar um recurso ele continuará sem existir ainda que outras requisições DELETE sejam enviadas para esse recurso.

### 3.3.3. Desenvolvimento de Serviços RESTful

Desenvolver um serviço Web RESTful não é diferente de desenvolver uma aplicação Web tradicional. É necessário se preocupar com os requisitos do negócio, satisfazer as necessidades dos usuários os quais manipularão os dados e lidar com limitações de hardware e arquiteturas de software. A principal diferença, contudo, é que o foco reside na identificação dos recursos e na abstração sobre as ações específicas a serem tomadas por esses recursos.

É possível comparar o desenvolvimento de serviços Web RESTful com o desenvolvimento orientado a objetos, pois existem algumas similaridades. No desenvolvimento orientado a objetos é realizada a identificação dos dados que se deseja representar juntamente com as ações que esse objeto pode realizar. Porém, as similaridades acabam na definição da estrutura do dado, porque com os serviços Web RESTful existem chamadas específicas que fazem parte do próprio protocolo de troca de mensagens.

Os princípios do desenvolvimento de um serviço web RESTful podem ser resumidos em quatro passos:

1. **Levantamento de requisitos** – Esse passo é similar às tradicionais práticas de coleta de requisitos do desenvolvimento de software.

2. **Identificação de recursos** – Esse passo é similar ao desenvolvimento orientado a objetos onde é realizada a identificação dos objetos, mas sem se preocupar com a troca de mensagens entre objetos.
3. **Definição de representação de recursos** – Para viabilizar a troca de mensagens entre clientes e servidores, é necessário definir o tipo de representação que será usada. Geralmente o XML é utilizado, porém atualmente o formato JSON está sendo cada vez mais popular [Sandoval 2009]. Porém, qualquer forma de representação de recursos pode ser utilizada. Por exemplo, é possível utilizar o XHTML ou qualquer outra forma binária de representação, embora seja necessário deixar os requisitos serem os guias das escolhas sobre as representações.
4. **Definição de URI** – O último passo é a definição do ponto de acesso ao recurso, o qual consiste em especificar as URIs para que os clientes possam endereçar os servidores a fim de trocar as representações de recursos.

Esse processo de desenvolvimento não é feito de forma estática, pelo contrário, ele deve ser realizado através de passos iterativos os quais giram em torno dos recursos [Sandoval 2009]. Por exemplo, pode acontecer que, durante a etapa de definição das URIs, descobre-se que uma das repostas da URI não é coberta em um recurso identificado. Nesse caso, deve-se voltar para definir um recurso adequado. Na prática, o que acontece é que na maioria dos casos os recursos já definidos cobrem os requisitos da aplicação, então basta combinar os recursos dentro de meta-recursos para tratar os novos requisitos [Sandoval 2009].

#### 3.3.3.1. Requisitos do Serviço Web de Exemplo

Para explicar melhor como funciona o desenvolvimento de serviços RESTful, é descrita nesta Subseção a modelagem de um desses serviços. O serviço Web RESTful modelado é uma aplicação Web de um laboratório de faculdade formado por um grupo de pessoas (usuários) onde cada pessoa é responsável por um ou vários trabalhos. Esse simples exemplo permite que seja apresentado o emprego dos princípios REST na prática, sem preocupações com regras complexas de lógica do negócio.

A modelagem da aplicação é feita seguindo um processo orientado a objetos [Sandoval 2009] e apenas o necessário para o entendimento do desenvolvimento de sistemas RESTful é apresentado. Desta forma, várias questões existentes na modelagem de software e assuntos afins foram omitidas.

Considere-se que, após a etapa inicial de levantamento de requisitos, foram especificados os seguintes casos de uso da aplicação (essas são as principais funcionalidades da aplicação): (i) um usuário pode criar uma conta com um nome de usuário e uma senha; (ii) um usuário pode publicar os trabalhos sob sua responsabilidade (uma publicação nesse caso consiste de uma página Web contendo uma descrição do trabalho); (iii) qualquer pessoa, registrada ou não, pode ver os trabalhos publicados na página do laboratório; (iv) qualquer pessoa, registrada ou não, pode ver o perfil dos usuários; (v) os usuários cadastrados podem atualizar seus dados (por exemplo, alterando sua senha); e (vi) qualquer pessoa pode pesquisar por termos para encontrar publicações cadastradas.

Nas próximas Subseções são endereçados os passos seguintes do desenvolvimento de nosso sistema Web de exemplo.

#### 3.3.3.2. Identificação de Recursos

A especificação dos recursos dos serviços é uma etapa posterior a listagem dos casos de uso. Com base nos requisitos, é possível perceber que serão necessários usuários e trabalhos. O recurso *usuários* retorna um usuário ou uma lista de usuários. Além disso, cada usuário pode publicar seus trabalhos. Assim, também serão necessários recursos para um trabalho e para uma lista de trabalhos. Com base nessa observação os seguintes recursos foram identificados: usuário, lista de usuário, trabalho e lista de trabalhos.

#### 3.3.3.3. Representação de Recursos

Nesta etapa são definidas as representações dos recursos da aplicação. Sabe-se que o protocolo HTTP permite a definição de qualquer tipo de representação (inclusive formatos proprietários de dados). Contudo, é recomendado que sejam utilizadas estruturas padrões, entre as quais estão o XML e o JSON. Logicamente, essa escolha deve ser feita com base nos requisitos da aplicação, os quais irão ditar que tipo de representações devem ser providas [Sandoval 2009]. Sempre que possível, é desejável que sejam fornecidas várias representações para um mesmo recurso. Assim, os consumidores do recurso poderão escolher dentre as opções disponíveis, aquela que é mais adequada para ele.

O formato ideal de uma representação é uma questão de concepção. É necessário considerar quais ações serão realizadas pelo servidor e a finalidade com a qual os clientes utilizarão os recursos. Como regra geral, XML deve sempre ser considerada como potencial representação, porque muitas linguagens oferecem bibliotecas para processar *streams* XML [Sandoval 2009] (isso facilita o processamento das mensagens e favorece a interoperabilidade entre as aplicações).

Após a definição do formato, é necessário definir o encadeamento (*linkability*) das representações. O encadeamento das representações define o um tipo de mecanismo de descoberta de recursos, o qual permite que os recursos possam ser ligados a outros recursos. Por exemplo, a lista de usuarios retornada pelo provedor dos recursos pode conter URIs (disponibilizada por meio de *links*) para cada elemento usuário da lista.

O servidor do exemplo desta Subseção irá utilizar XML e JSON para representar os recursos. A representação XML será utilizada pelo servidor para enviar uma representação do estado de um recurso. O cliente utiliza o XML para criar e atualizar os recursos no lado do servidor. O JSON será utilizado apenas no envio de representações de recurso do servidor para o cliente.

A Figura 3.4 ilustra uma representação do recurso *usuario* no formato XML. Apenas o conteúdo dos elementos *nome*, *nome_usuario* e *senha* são armazenados no servidor, pois eles são parte de um recurso do usuário. O *nome* indica apenas o nome do usuário. O *nome_usuario* deve ser único em todo sistema, pois ele será utilizado para identificar o usuário. O elemento *link* é utilizado para apontar algum recurso no serviço Web e esse *link* é criado pelo servidor com base em *nome_usuario*.Um *link* para um recurso pode ser associado a esse recurso assim que o mesmo é criado ou quando uma representação do recurso é solicitada. Por exemplo, o *link* para o usuário identificado pelo *nome_usuario* “johnSmith” pode ser criado assim que o provedor do recurso

recebe uma requisição para criar esse novo *usuario* ou quando o provedor recebe uma requisição para um recurso que lista todos os usuários do sistema.

```
<usuario>
        <nome> john </nome>
        <nome_usuario> johnSmith </nome_usuario>
        <senha> abc123 </senha>
        <link> /johnSmith </link>
</usuario>
```

**Figura 3.4 - Representação XML do recurso *usuario***

A segunda estrutura criada define uma lista de usuários (Figura 3.5). O documento XML da Figura 3.5 declara uma lista de usuários dentro do elemento *usuarios*. Nessa estrutura é possível observar o conceito de encadeamento na prática: com a lista de usuários é possível buscar por um usuário específico usando o valor do elemento *link*. Por exemplo, o primeiro *usuario* da lista apresentada na Figura 3.5 (identificado por johnSmith) pode ser acessado no *path* /johnSmith. Dessa forma, quando o cliente obtiver uma representação contendo os usuários do sistema ele poderá acessar cada *usuario* da lista através do seu respectivo *link*.

```
<usuarios>
        <contagem> 50 </contagem>
        <link></link>
                <usuario>
                        <nome> john </nome>
                        <nome_usuario> johnSmith </nome_usuario>
                        <senha> abc123 </senha>
                        <link> /johnSmith </link>
                </usuario>
                ...
                <usuario>
                        <nome> henrique </nome>
                        <nome_usuario> hribeiro </nome_usuario>
                        <senha> ribeir0123 </senha>
                        <link> /hribeiro </link>
                </usuario>
</usuários >
```

**Figura 3.5 - Representação XML do recurso lista de usuários**

A estrutura da representação do recurso trabalho é utilizada na inclusão de um novo trabalho. Essa estrutura é apresentada naFigura 3.6. Um trabalho possui o identificador (*id_trabalho*), o corpo da mensagem (definido pelo elemento *conteudo*), e o usuário que postou a mensagem. Dependendo do que se pretende fazer com essa representação, não será necessário passar todas as informações desse recurso para o servidor. Por exemplo, quando um cliente cria um novo recurso trabalho, ele não sabe qual é o valor do identificador (*id_trabalho*), pois na abordagem apresentada aqui, tal valor será criado no lado do servidor. Dessa forma, a estrutura com representação do trabalho será passada para o servidor, o qual irá definir o valor de *id_trabalho*.

A Figura 3.7 apresenta a estrutura XML de representação da lista de trabalhos. Essa estrutura contém uma coleção de trabalhos, e cada trabalho contém o usuário que o postou.

```
<trabalho>
        <id_trabalho> WebOfThings </id_trabalho>
        <conteudo> A Web englobando o mundo físico... </conteudo>
        <link> /webofthings </link>
        <usuario>
                <nome> tiago </nome>
                <nome_usuario> tcruzfranca </nome_usuario>
                <senha> abc123 </senha>
                <link> /tcruzfranca </link>
        </usuario>
</trabalho>
```

**Figura 3.6 - Representação XML do recurso *trabalho***

```
<trabalhos>
        <contador> 50 </contador>
        <link> /trabalhos </link>
        <trabalho>
                <id_trabalho> smarbuild </id_trabalho>
                <conteudo> edifícios inteligentes... </conteudo>
                <link> /smartbuild </link>
                <usuario>
                        <nome> claudio </nome>
                        <nome_usuario> cmiceli </nome_usuario>
                        <senha> m1c3l1 </senha>
                        <link> /cmiceli </link>
                </usuario>
        </trabalho>
        ...
        <trabalho>
                <id_trabalho> sutil </id_trabalho>
                <conteudo> ... </conteudo>
                <link> /sutil </link>
                <usuario>
                        <nome> jaime </nome>
                        <nome_usuario> jaime </nome_usuario>
                        <senha> 224466 </senha>
                        <link> /jaime </link>
                </usuario>
        </trabalho>
</trabalhos>
```

**Figura 3.7 - Representação XML do recurso *trabalhos***

As representações JSON possuem os mesmos elementos chaves para os mesmos recursos. Uma definição da representação JSON do recurso *usuario* pode ser vista na Figura 3.8, enquanto a representação JSON do recurso *lista_usuarios* pode ser vista na Figura 3.9. A Figura 3.10 apresenta a estrutura JSON de representação de todos os usuários.

```
{"usuario":{"nome_usuario":"juan","senha":"123456", "link":"/usuario/juan"}}
```

**Figura 3.8 - Representação JSON do recurso *usuário***

{"lista_usuarios":{"contador":"6","usuarios":[{"nome_usuario":"igor","senha":"zw
u987","link":"/usuario/igor"},...,{"nome_usuario":"jane","senha":"abc000","link":"/
usuario/paula"}]}}

**Figura 3.9 - Representação JSON do recurso lista_usuarios**

{"usuarios":[{"nome_usuario":"hsalmon","senha":"123456","link":"/usuario/
hsalmon"},...,{"nome_usuario":"erico","senha":"987456","link":"/usuario/
erico"}]}

**Figura 3.10 - Representação JSON de todos os recursos *usuario***

A representação JSON do *trabalho* é apresentada na Figura 3.11 e a Figura 3.12 apresenta a estrutura JSON de *lista_trabalhos*.

{"trabalho":{"id_trabalho":"id_1","conteudo":"algumConteudo","link":"/trabalhos/i
d_1","usuario":{"usuario":{"nome_usuario":"renato","senha":"rttr0ll","link":"/usuar
io/renato"}}}

**Figura 3.11 - Representação JSON do recurso *trabalho***

{"lista_trabalhos":{"contador":"6","link":"/trabalhos","trabalhos":[{"id_trabalho":"id
_1","conteudo":"algum-conteudo","link":"/trabalho/id_1",
"usuario":{"nome_usuario":"smelo","senha":"t3nbal","link":"/usuario/smelo"}},
...,{"id_trabalho":"id_2", "conteudo":"algum_conteudo", "link":"/trabalho/ id_2",
"usuario":{" nome_usuario ":"mmelo", "senha":"4pi4rab", "link":"/usuarios/
mmelo"}}]}}

**Figura 3.12 - Representação JSON de *lista_trabalhos***

### 3.3.3.4. Definição de URI

A definição das URIs é uma etapa crucial, pois elas definirão a API do sistema. A API definida deve ser lógica, hierárquica, e o mais estável possível. Uma boa API é aquela que é utilizada facilmente e não muda com freqüência. Além disso, a idéia de uma API RESTful é manter uma URI única e confiável para cada recurso.

Na definição da URI, a primeira coisa necessária é um endereço Web, no nosso exemplo utilizaremos o endereço *http://www.dcc.ufrj.br/*. Ainda, serão adotadas duas convenções na definição das URIs RESTful. Primeiro, os itens e identificadores que não mudam serão nomeados utilizando palavras-chave como parte da URI (por exemplo, a palavra “usuários” foi utilizada para definir a URI que aponta para o recurso lista de usuários). A segunda convenção é a utilização de palavras-chaves entre chaves “{}”. Ao aplicar essa convenção para definir a lista de URIs para os recursos, as URIs obtidas são:

- *http:// www.dcc.ufrj.br /usuarios* - uma requisição com o método GET enviada para essa URI irá retornar uma lista de usuários. Se o método POST for utilizado, será criado um novo *usuario*. Nesse caso, o corpo da mensagem conterá uma representação XML do usuário que deve ser criado. Os outros métodos (PUT e DELETE) não serão suportados, pois as listas de usuários não podem ser alteradas ou apagadas;

- *http:// www.dcc.ufrj.br /usuarios/{nome_usuario}* - uma requisição contendo o método GET enviada para essa URI irá retornar uma representação de um usuário contendo um identificador *nome_usuario*. Se o método PUT for utilizado, o recurso acessado (*usuario*) será atualizado. Já o método DELETE é utilizado para excluir um usuário;
- *http://www.dcc.ufrj.br/trabalhos* - uma requisição com o método GET enviada para essa URI retornará uma lista com todos os trabalhos. Se o método POST for utilizado, um novo trabalho será criado;
- *http://www.dcc.ufrj.br/trabalhos/{id_trabalho}* – se o método GET for utilizado, o acesso a essa URI retornará uma representação para um trabalho associado ao *id_trabalho* enviado. O método DELETE irá indicar ao servidor que o cliente espera que o trabalho seja apagado. Os métodos POST e PUT não serão utilizados para esse recurso; e
- *http://www.dcc.ufrj.br/trabalhos/usuarios/{nome_usuario}* – se o método GET for utilizado em uma requisição submetida para essa URI, uma lista de todos os trabalhos do usuário com o *nome_usuario* será retornada (nenhum outro método é suportado).

O uso adequado da URI (conforme os princípios REST e ROA) fornece a informação semântica necessária para interação com os recursos. O uso inadequado da URI pode causar interpretações confusas. Por exemplo, quando uma URI é utilizada para indicar o tipo de representação do recurso, como faz o *twitter*, por exemplo, dúvidas podem ser geradas no consumidor do recurso. O *twitter* oferece diferentes representações através da URI http://twitter.com/statuses/public_timeline.{xml, json, rss}. Essa abordagem poderá gerar dúvidas no cliente que consome o recurso, pois segundo os princípios REST o protocolo de comunicação deveria ser utilizado para indicar o tipo de representação desejado. Por exemplo, o que aconteceria se uma requisição HTTP indicando que o cliente espera receber uma representação XML através do campo *Accept* do HTTP fosse submetida para a URI http://twitter.com/statuses/public_timeline.json? Não seria possível especificar se a representação obtida poderia ser um XML ou um JSON. A pesar disso, ROA incentiva o uso de URI para indicar o tipo de representação de um recurso que o servidor deve retornar, porque essa seria a forma mais simples e explícita do cliente indicar o tipo de representação que espera receber.

A definição sobre qual abordagem usar para indicar a representação do recurso é uma decisão do desenvolvedor ou arquiteto de software. Essa decisão deve ser tomada com base na observação de qual será a melhor abordagem para os usuários do sistema. Por exemplo, uma abordagem semelhante a do *twitter* pode facilitar o desenvolvimento de clientes (aplicações que vão utilizar esse recurso), pois, para recursos que são apenas para leitura, é possível utilizar somente uma requisição HTTP (com o método GET) contendo o tipo de representação embutida na URI. Enviar esse tipo de requisição é mais fácil do que instanciar uma nova requisição HTTP e modificar o cabeçalho *Accept* a cada nova requisição.

### 3.3.4. Descritor WADL

A linguagem WADL (*Web Application Description Language*) é uma especificação formal baseada em XML utilizada para descrever aplicações Web baseadas no protocolo HTTP [Hadley 2006]. Quando os provedores de aplicações Web RESTful desejavam fornecer descrições sobre os recursos, eles utilizavam descrições informais e personalizadas ou ofereciam bibliotecas para permitir que os clientes consumissem seus recursos [Hadley 2009], [Ferreira Filho 2010]. A WADL foi proposta para ser um padrão de descrição de aplicações Web RESTful, cujo propósito é semelhante ao do descritor WSDL utilizado para descrever serviços SOAP [WSDL 2011]. Porém, a WADL foi criada especialmente para descrever interfaces RESTful [Ferreira Filho 2010]. Os principais benefícios na utilização de descritores como o WADL é a possibilidade de automatizar a criação de código através de um formato padronizado e portável que independe de linguagem ou aplicação específica [Webber et al. 2010].

A Tabela 3.1 apresenta os principais elementos de um documento WADL juntamente com uma descrição de cada um deles [Hadley 2009], [Ferreira Filho 2010].

**Tabela 3.1 – Principais elementos de um descritor WADL**

| Elemento | Descrição |
|---|---|
| application | É o elemento raiz e contém os demais elementos do documento WADL |
| resources | Esse elemento atua como um contêiner para cada recurso fornecido pela aplicação, a URI do provedor dos recursos é indicado neste elemento através do atributo base |
| resource | Cada recurso é descrito por este elemento que inclui o atributo *path* que identifica o recurso naquele provedor de recursos. |
| method | Esse elemento descreve a entrada e a saída de um método do protocolo HTTP que deve ser aplicado a um recurso. |
| response | Esse elemento descreve a saída resultante da realização de um método do HTTP no recurso. |
| representation | Esse elemento descreve uma representação do estado de um recurso. |

A Figura 3.13 é um exemplo de documento WADL gerado pela ferramenta Jersey [Jersey 2011] que é uma ferramenta para implementação de serviços RESTful utilizando a linguagem Java. No exemplo é apresentada a descrição do recurso identificado pelo *path* "/spot-1265/light" e que pode retornar dois tipos de representação: XML e JSON.

Apesar das vantagens do uso de um descritor, os defensores dos princípios REST vão de encontro à idéia de utilizar documentos formais que estabeleçam contratos entre os clientes e os provedores de recursos. Segundo eles, a idéia de utilizar descritores vem de uma mentalidade herdada dos serviços Web baseados em SOAP cuja filosofia é contrária ao REST. Segundo Webber et al. (2010), o emprego adequado de hipermídia como mecanismo de manutenção do estado da aplicação fornece toda semântica necessária para que o consumidor realize toda manipulação que deseja sobre os recursos. Desta forma, não seria necessária a utilização de um contrato formal (como a WADL) o que resulta em um menor acoplamento entre cliente e servidor. Resumidamente, as principais desvantagens da utilização de WADL são [Webber et al. 2010]:

- Existem poucas ferramentas para manipulação dessa linguagem;

- As aplicações Web passam a ser vistas como aplicações estáticas devido ao uso de uma linguagem de descrição de interfaces;
- Há um aumento no acoplamento entre o cliente e o servidor, que acaba fazendo com que alterações no lado do servidor gerem conseqüências no lado do cliente como acontece com os serviços Web que utilizam WSDL; e
- A linguagem de descrição não fornece informações suficientes para direcionar a interação com os recursos, isto é, o consumidor do documento WADL não sabe qual interação que o servidor espera que ocorra sobre um recurso.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <application xmlns="http://research.sun.com/wadl/2006/10">
    <doc xmlns:jersey="http://jersey.dev.java.net/" jersey:generatedBy="Jersey: 1.0.3.1 08/14/2009 04:21 PM" />
  - <resources base="http://www.ufrj.labnet/sensor/resources/">
    - <resource path="WebOfthings">
      - <resource path="/spot-1265/light">
        - <method name="GET" id="getLight">
          - <response>
              <representation mediaType="application/json" />
              <representation mediaType="text/xml" />
            </response>
          </method>
        </resource>
      </resource>
    </resources>
  </application>
```

**Figura 3.13 - Exemplo de documento WADL**

### 3.3.5. REST versus WS-* SOAP

O REST e os Serviços Web WS-* (SOAP, WSDL, etc.) são técnicas de integração de aplicações distribuídas que visam manter o baixo acoplamento entre as partes envolvidas [Guinard e Trifa 2009]. Na Web das Coisas os princípios REST têm sido amplamente aplicados na integração dos dispositivos inteligentes a Web porque esses princípios parecem ser mais adequados para dispositivos com poucos recursos de hardware [Wilde 2007], [Guinard e Trifa 2009]. Esta Subseção apresenta algumas comparações entre REST e WS-* SOAP que podem facilitar a compreensão de tal escolha (maiores detalhes sobre essa comparação podem ser vistos em [Pautasso et al. 2008]).

A primeira diferença entre o REST e o WS-* SOAP está na forma como o protocolo HTTP é empregado. Com REST, o HTTP é utilizado para definir a interação entre o cliente da aplicação e o provedor do recurso. Nesse caso, toda a semântica presente nos quatro métodos (GET, POST, PUT, DELETE) do protocolo HTTP é utilizada na definição da interface do sistema [Pautasso et al. 2008]. Os WS-* SOAP, por outro lado, utilizam o protocolo HTTP para transportar as mensagens no formato SOAP com objetivo de integrar aplicações. Nessa abordagem, as mensagens SOAP são adicionadas ao corpo do HTTP para comunicação remota através de *firewalls* [Pautasso et al. 2008], [Guinard e Trifa 2009]. Nos WS-* SOAP o método POST do HTTP é utilizado na troca de mensagens entre clientes e servidores e a informação sobre qual funcionalidade deve ser executada está presente na mensagem SOAP e não na requisição HTTP [Pautasso et al. 2008]. É por esse motivo que se diz que para os

serviços Web SOAP o protocolo HTTP desempenha funcionalidade de transporte mesmo ele sendo um protocolo do nível de aplicação. Enquanto as aplicações WS-* SOAP utilizam a Web como meio de troca de mensagens, as aplicações Web RESTful são parte da Web a qual é vista como um terreno comum para as aplicações [Pautasso et al. 2008].

Além do HTTP, as mensagens SOAP também podem ser encapsuladas e transportadas dentro de outros protocolos (por exemplo, os protocolos TCP e SMTP podem ser utilizados para esse propósito) [Pautasso et al. 2008]. Isso é possível porque o SOAP possui um formato próprio de mensagem (baseado em XML), porém ele requer que outro protocolo seja utilizado para transferir essa mensagem. O formato da mensagem SOAP ocasiona um maior consumo de banda do que o ocasionado pelo protocolo HTTP, devido ao tamanho da mensagem SOAP [Tyagi 2006]. Esse é um dos motivos para o REST apresentar vantagens para ser utilizado em dispositivos com pouca capacidade de *hardware* ou restrição de banda de rede disponível [Tyagi 2006]. Além disso, um formato pré-definido de mensagem força o cliente a tratar aquele tipo de mensagem caso ele deseje utilizar um serviço. Com REST é possível oferecer diferentes tipos de representações. Assim, um cliente pode, por exemplo, escolher se para ele é mais adequado receber uma representação JSON ou um XML. Porém, fornecer vários formatos exige mais esforço no processo de desenvolvimento, pois diferentes tipos de mensagem precisam ser providos.

Do ponto de vista do acoplamento, tanto REST quanto o SOAP fomentam o desenvolvimento de sistemas distribuídos com acoplamento fraco entre as partes. Porém, avaliar qual das duas abordagens atinge esse objetivo conseguindo um menor acoplamento é uma tarefa subjetiva, pois para definir o acoplamento vários aspectos devem ser observados (como tempo/disponibilidade, clareza de localização, e evolução do serviço) [Pautasso et al. 2008]. Geralmente o termo “fraco acoplamento” é relacionado à capacidade de fazer modificações no provedor do serviço sem afetar o cliente. Nesse caso, os serviços Web RESTful são menos acoplados, pois as operações sobre os recursos não mudam, já que tais operações são baseadas nos métodos do HTTP, os quais não mudam mesmo quando ocorre alguma alteração no serviço. Contudo, quando acontecem mudanças nos parâmetros passados nas mensagens, ambos (SOAP e REST) compartilham o mesmo nível de acoplamento [Pautasso et al. 2008].

Outra diferença entre REST e SOAP está na forma como essas abordagens utilizam URIs. Com REST a URI não é utilizada apenas para identificar o recurso, mas também para encapsular toda informação necessária para identificar e localizar os recursos sem a necessidade de um registro centralizado [Pautasso et al. 2009]. Entre outros benefícios, empregar URIs dessa forma permite que os recursos sejam marcados e que *links* hipermídia sejam fornecidos para o cliente para que o mesmo interaja com os recursos do sistema. O WS-* também utiliza URI, mas não da mesma forma que em uma abordagem REST, o que acarreta na necessidade de utilização de outros mecanismos para agregar informação ao serviço [Pautasso et al. 2009].

Prosseguindo com as comparações entre os serviços Web SOAP e as aplicações RESTful, é possível observar diferenças entre a forma como os clientes consomem os serviços. Com SOAP, os clientes utilizam um documento de descrição formal dos serviços disponíveis. Esse documento é o WSDL (*Web Service Description Language*) [WSDL 2007]. O uso de descritores fornece aos clientes meios que permitem a geração

automática de código para consumir esses serviços. Porém, sua utilização pode ocasionar falhas no cliente caso ocorram modificações no servidor [Pautasso et al. 2008]. As aplicações RESTful não precisam utilizar um contrato formal (apesar disso ser possível, conforme apresentado na Subseção 3.3.4) entre o cliente e o servidor. Freqüentemente os recursos de uma aplicação RESTful são descritos de forma textual ou por meio da documentação da API da aplicação [Pautasso et al. 2008]. Além dessas abordagens, também surgiram alguns provedores de serviços RESTful que oferecem bibliotecas (em diferentes linguagens de programação) para serem usadas pelos clientes que desejam consumir os recursos que o servidor oferece [Ferreira Filho 2010].

Por ser mais simples, por tornar as aplicações criadas com base em seus princípios parte da Web permitindo que todos os recursos e disponíveis na Web possam ser utilizados para o mundo físico e por causa do menor tamanho das suas mensagens os princípios REST são considerados mais adequados para serem utilizados na integração de dispositivos com baixa capacidade de hardware na Web [Wilde 2007]. Porém, ainda existe uma série de desafios que precisam ser superados. Dentre eles destaca-se o modelo de *pull* da Web ocasionado pelo HTTP que não prevê um modelo de comunicação assíncrona onde o cliente é notificado da ocorrência de um evento.

## 3.4. Estendendo ROA para a Web das Coisas

Apesar do REST parecer adequado para dispositivos embarcados, estes nem sempre possuem um endereço IP (*Internet Protocol*) e não são, portanto, diretamente localizáveis/endereçáveis na internet. No entanto, é muito provável que mais e mais dispositivos do mundo real se tornem habilitados para o IP e incorporem servidores HTTP (em especial com 6LoWPAN), tornando-os capazes de compreender as linguagens e protocolos da Web [Mayer 2010]. Desta forma, tais dispositivos poderão ser diretamente integrados a Web e assim as suas funcionalidades serão acessadas através de interfaces RESTful.

Embora tais dispositivos habilitados sejam susceptíveis de serem amplamente distribuídos em um futuro próximo, a integração direta de dispositivos do mundo real com a Web ainda é uma tarefa bastante complexa. Principalmente nos casos em que os dispositivos não suportam IP e/ou HTTP, como ocorre normalmente no contexto de redes de sensores sem fio (RSSF), por exemplo. Quando o endereço IP não é suportado, é necessário utilizar um padrão de integração diferente. Nessas situações, nas quais os dispositivos não são capazes de se comunicar via IP, é possível utilizar um dispositivo intermediário chamado *Smart Gateway*. *Smart Gateways* possuem duas funções básicas: fornecer uma interface RESTful com URIs que identificam e fornecem acesso aos objetos físicos (dispositivos inteligentes) e seus subrecursos; e realizar a comunicação com os objetos físicos utilizando as APIs destes. Em outras palavras, o *Smart Gateway* atua como uma ponte entre a Web e os dispositivos inteligentes, ao fornecerem uma interface Web RESTful de acesso aos recursos e subrecursos dos dispositivos e ao se comunicarem com tais dispositivos através de suas APIs.

Cada *gateway* tem um endereço IP, executa um servidor HTTP e compreende os protocolos proprietários dos diferentes dispositivos conectados a ele através do uso de controladores (*drivers*) dedicados. Como exemplo, considere uma requisição para um nó sensor proveniente da Web através da API RESTful. O *gateway* mapeia essa requisição em uma solicitação da API proprietária do dispositivo e a transmite usando o

protocolo de comunicação que tal dispositivo compreende (por exemplo, usando o protocolo *Zigbee*). Um *Smart Gateway* pode suportar vários tipos de dispositivos através de uma arquitetura de controladores. A Figura 3.14, mostra um exemplo de *Smart Gateway* que suporta três tipos de dispositivos, comunicando-se com eles através de seus protocolos de comunicação correspondentes.

Os *Smart Gateways* também podem ser usados para orquestrar a composição de vários serviços de baixo nível em serviços Web de mais alto nível. Esses serviços Web de mais alto nível são *mashups* criados a partir da composição dos recursos dos dispositivos disponibilizados através da API RESTful oferecida pelo *gateway*. Por exemplo, se um dispositivo embarcado oferece monitoramento do consumo de energia dos aparelhos, o *Smart Gateway* poderia fornecer um serviço que retorna a soma de todos os consumos de energia monitorados por todos os dispositivos embarcados conectados ao *gateway*.

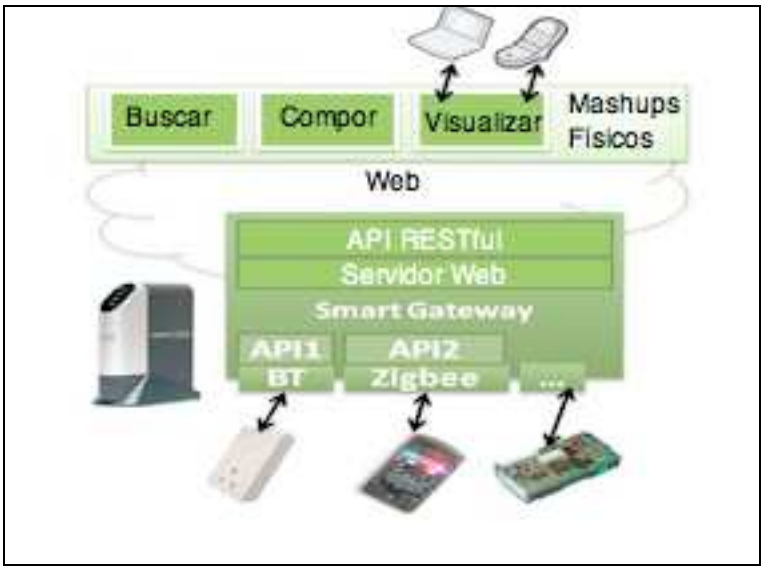


**Figura 3.14 - *Smart Gateway*, adaptado de [Guinard e Trifa 2009]**

Outro dispositivo da WoT muito similar aos *Smart Gateways* são os *Smart Hubs* [Mayer 2010]. Os *Smart Hubs* permitem que sejam criadas infraestruturas que interligam dispositivos inteligentes à Web a fim de prover serviços avançados no topo desses dispositivos. Desta forma, os *Smart Hubs* oferecem a possibilidade de ter dispositivos interagindo entre si para que sejam fornecidas funcionalidades que vão além do que cada dispositivo poderia prover individualmente. O *Smart Hub* utiliza serviços de descoberta de dispositivos inteligentes a fim de obter e armazenar informações sobre esses dispositivos. O processo de descoberta e armazenamento de informações sobre novos dispositivos realizado pelo *Hub* é conhecido como vinculação de recurso (*attaching resource*). Os *Smart Hubs* oferecem serviços de consulta e publicação de recursos dentro da infraestrutura da WoT e são implementados para estabelecer conexões entre si de forma que seja mantida uma infraestrutura em árvore. Essa infraestrutura permite que os recursos de diferentes dispositivos não sejam centralizados em um único *Hub*. Assim como os *Smart Gateway*, os *Smart Hubs* utilizam o conceito de controlador para se comunicarem com diferentes dispositivos utilizando as linguagens e protocolos destes [Trifa et al., 2009], [Mayer, 2010].

No contexto da WoT, um controlador é um software utilizado para fazer a ponte entre o mundo físico e a Web com o intuito de expor objetos inteligentes como serviços RESTful. Um controlador é tipicamente formado por: um módulo que controla o dispositivo permitindo que o mesmo seja acessado a partir de um computador; e por um componente que disponibiliza um subconjunto de funcionalidades do dispositivo na Web, estabelecendo a comunicação entre clientes HTTP e esse dispositivo. Porém, a implementação de um controlador WoT pode diferir na forma como o dispositivo é acessado pelo software. Um controlador pode atuar como um *proxy* reverso encaminhando requisições HTTP para um dispositivo que possui um servidor embarcado ou pode dissociar o processo de comunicação entre o cliente HTTP e o dispositivo (nesse caso, o objeto físico é acionado por meio de sua API para realizar a tarefa especificada na requisição HTTP).

### 3.4.1. Comunicação Orientada a Eventos

Os *Smart Hubs* e *Smart Gateway* fornecem serviços de comunicação orientada a eventos. Esse serviço é importante porque a comunicação com o protocolo HTTP é síncrona (o modelo de comunicação é requisição e resposta – *request/reply*), o que não permite que o cliente seja notificado da ocorrência de um evento. Um exemplo de evento pode ser observado quando sensores utilizados para monitorar a temperatura de uma área detectam que está ocorrendo um incêndio. Não é possível prever quando ocorrerá um incêndio, por isso é necessário prover meios que permitam que o cliente saiba da ocorrência dos eventos de interesse (no caso, a ocorrência de incêndio).

Para monitorar a ocorrência de um evento, os clientes podem consultar os dispositivos constantemente utilizando AJAX, por exemplo. Porém, essa abordagem não é adequada, pois poderia fazer com que a bateria dos dispositivos se esgotasse rapidamente. Para evitar tal situação, é possível utilizar o *Smart Gateway* e o *Smart Hub* para fornecer um modelo de comunicação assíncrona baseado no modelo *publish/subscribe*. Nesse modelo, o cliente publica um interesse para ser notificado com uma resposta assincronamente [Guinard et al. 2010].

Um modelo de comunicação assíncrona que pode ser utilizado é o Atom *Publishing Protocol* (AtomPub) [Atom 2004], [RFC4287 2005]. O Atom é um formato XML que encapsula dados para serem publicados na Web (esse formato de disponibilização de conteúdo é chamado *Syndication*). O AtomPub é o protocolo utilizado para publicar mensagens Atom. Nessa abordagem, o cliente deve assinar os *feeds* para monitorar as mensagens dos dispositivos. Assim, os clientes deixam de consultar os dispositivos e passam a consultar um servidor a parte. Os dispositivos, por sua vez, publicam os dados referentes a ocorrência de um evento nesse servidor.

Outros meios de comunicação assíncrona também podem ser utilizados, tais como: e-mail, *twitter* e URIs de serviços externos que devem ser acessados pelo *gateway* ou *hub* para publicação do evento. Por exemplo, se um usuário deseja que os dados do dispositivo sejam publicados em sua conta no *twitter*, ele deve fornecer informações sobre a sua identificação e as credenciais do *twitter*. Então, qualquer cliente que deseje obter as informações do dispositivo precisa apenas se associar a conta do mesmo no *twitter*.

Outra alternativa de comunicação assíncrona na Web é o modelo Comet, o qual pode ser utilizado para que o servidor envie dados para o cliente [Duquennoy et al.

2009]. Porém esse modelo consome mais recurso no lado do servidor, pois é necessário que o servidor armazene informações sobre cada cliente para poder anunciar os dados referentes a um evento [Duquennoy et al. 2009].

## 3.5. Web das Coisas: Desenvolvendo Aplicações na Plataforma Sun SPOT

Esta Seção descreve como dispositivos físicos e suas funcionalidades podem ser disponibilizados na Web conforme as soluções propostas na WoT. Para tal, são apresentados exemplos práticos utilizando os dispositivos embarcados Sun SPOT (*Sun Small Programmable Object Technology*), os quais são programados usando a linguagem Java [Sun SPOT 2011]. Nesta Seção também são apresentados dois *mashups* que compõe os recursos dos SPOTs com outros recursos da Web.

A Subseção 3.5.1 apresenta a plataforma Sun SPOT. A Subseção 3.5.2 apresenta como os objetos físicos podem ser integrados a Web, incluindo exemplos práticos de como disponibilizar as funcionalidades de dispositivos Sun SPOT na forma de recursos RESTful. A Subseção 3.5.3 apresenta a construção de *mashups* físicos que compõem os dados e funcionalidades disponibilizados pelos SPOTs.

### 3.5.1. Introdução a Plataforma Sun SPOT

O dispositivo Sun SPOT é uma plataforma de sensores desenvolvida pela Sun Microsystems/Oracle [Sun SPOT, 2011]. Esse dispositivo é acionado por código Java e pode ser utilizado em diversos projetos com os mais variados propósitos. Eles podem, por exemplo, ser utilizado em carros robóticos ou no monitoramento de fenômenos físicos devido aos dispositivos de sensoriamento acoplados a eles [Sun SPOT, 2011]. Existem dois tipos de dispositivos Sun SPOT: os *free-range* SPOTs e a estação base [Sun SPOT, 2011]. A Figura 3.15 ilustra esses dois tipos de dispositivos (uma estação base e dois *free-ranges* SPOTs).

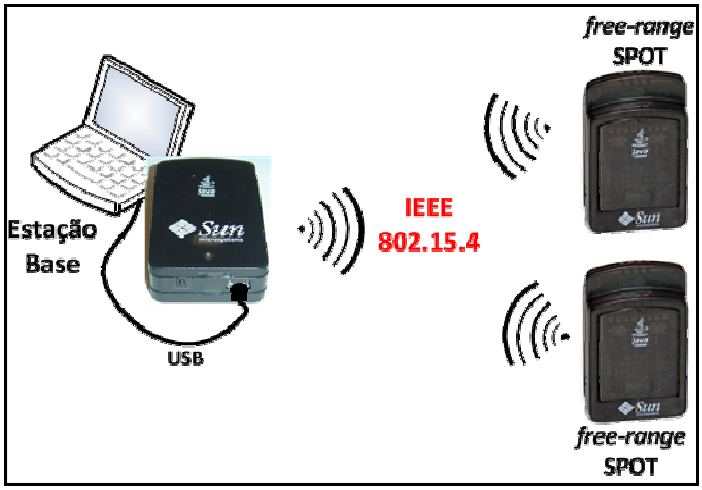


**Figura 3.15 - Utilizando uma EB e dois *free-range* SPOTs**

Os *free-range* SPOTs (ou simplesmente SPOTs) possuem uma bateria recarregável, dispositivos de sensoriamento, sendo um de temperatura, outro de luminosidade e um acelerômetro. Além disso, eles também possuem dois botões (os botões podem ser utilizados para acender leds do SPOT ou até mesmo para mudar algum parâmetro na aplicação), oito *leds* de três cores (vermelho, verde e azul), quatro pinos genéricos de entrada e saída (que podem ser utilizados para acoplar outros

dispositivos eletrônicos), quatro pinos de saída de alta tensão e uma interface USB [Sun SPOT, 2011]. Atualmente os SPOTs possuem as seguintes configurações de hardware: um processador ARM de 32 bits que opera a uma freqüência de 400 MHz; 1 MB de memória RAM; 4 ou 8 MB de memória *flash*; e um *Chipcon* 2420 para comunicação a rádio em redes aderentes ao padrão IEEE 802.15.4 dispondo de 11 canais de 2,4 GHz [Sun SPOT, 2011]. Os SPOTs podem ser identificados pelo seu endereço MAC IEEE de 64 bits. Além disso, esse dispositivo oferece um protocolo de roteamento o LQRP (do inglês *Link Quality Routing Protocol*), o qual pode ser utilizado ou estendido pelo usuário. O código Java da aplicação é executado em uma máquina virtual chamada *Squawk* VM (*Virtual Machine*), compatível com a plataforma Java ME (*Micro Edition*) na configuração CLDC1.1 (do inglês, *Connected Limited Device Configuration*), que permite que o código seja executado sem depender de um sistema operacional. Portanto, é a *Squawk* que realiza todas as funcionalidades necessárias para execução de uma aplicação, a qual deve possuir uma classe que estende a classe *MiDlet* [Sun SPOT, 2011].

A estação base (EB) possui apenas uma interface de comunicação sem fio baseada em rádio e uma interface USB. A EB comunica-se com os *free-range* SPOTs usando o radio e com um dispositivo de maior poder computacional (um computador, por exemplo) via interface USB.

Aplicações para a plataforma Sun SPOT podem ser desenvolvidas em um computador pessoal e posteriormente instaladas nos SPOTs. Para tal, é necessário instalar no computador o JDK (*Java Development Kit*), o SDK (*Sun Development Kit*) e o Ant [Ant 2010], [Oracle 2010], [Sun SPOT 2011]. O Ant é uma ferramenta que independe de sistema operacional e pode ser utilizada na automatização do processo de compilação, implantação, geração de documentos e execução de uma aplicação. Essa ferramenta é usada principalmente na construção de aplicações Java e pode ser utilizada em conjunto com IDEs (*Integrated Development Environment*) [Ant 2010]. Desse modo, o desenvolvedor pode utilizar IDEs bem conhecidas tais como NetBeans [NetBeans, 2011] e Eclipse [Eclipse, 2011] para desenvolver aplicações para Sun SPOT. A implantação do código de uma aplicação nos SPOTs pode ser feita transferindo o código através da interface USB ou utilizando uma abordagem OTA (“*over the air”*). A implantação do código através do cabo USB exige que o SPOT esteja conectado fisicamente ao computador que contém o código compilado. A implantação de código via OTA utiliza o enlace de rede sem fio IEEE 802.15.4 para enviar o código a um SPOT específico. Utilizando essa última abordagem, a estação base é utilizada para enviar esse código e o endereço MAC do SPOT de destino deve ser especificado.

Uma aplicação para Sun SPOT deve possuir uma classe que estende a classe *MiDlet*. A classe *MiDlet* possui três métodos abstratos que devem ser implementados pela classe que a estende: *startApp*(), *pauseApp*(), e *destroyApp*(*boolean unconditional*). O método que é executado inicialmente pela *Squawk* é o *startApp*, sendo executado assim que o SPOT for iniciado. Os outros dois métodos *pauseApp* e *destroyApp* podem conter uma implementação vazia. A

Figura 3.16 mostra um exemplo básico de uma classe que estende *MiDlet*. A *Squawk* executa uma aplicação por vez; isto significa que apenas uma classe estendendo MiDlet pode ser executada. Porém, a *Squawk* permite que uma aplicação execute várias

*threads* de forma paralela. Maiores detalhes sobre a plataforma Sun SPOT podem ser obtidas em [Sun SPOT, 2011], onde também é possível obter informações sobre um emulador que pode ser utilizado para testar a maioria dos projetos desenvolvidos para o Sun SPOT.

```
package br.ufrj.labnet;

import javax.microedition.midlet.*;
import java.io.IOException;
import com.sun.spot.io.j2me.udp.UDPConnection;
import com.sun.spot.io.j2me.udp.UDPDatagram;
import javax.microedition.io.Connector;
import javax.microedition.io.DatagramConnection;
import com.sun.spot.resources.transducers.ILightSensor;
import com.sun.spot.resources.Resources;

public class Exemplo extends MIDlet {

   public static DatagramConnection conn = null;
   public static int maxlen = 0;
   public static int port = 100;

   protected void startApp() throws MIDletStateChangeException {

          try {
              conn = (UDPConnection) Connector.open("udp://:" + port);
              int response = 0;
              ILightSensor lightSensor = (ILightSensor)
                           Resources.lookup(ILightSensor.class);
              UDPDatagram dg = (UDPDatagram)
                     conn.newDatagram(conn.getMaximumLength());
              response = lightSensor.getValue();
              dg.write(response);
              conn.send(dg);

          } catch (IOException ex) {
                       ex.printStackTrace();
          }

   }

   protected void pauseApp() {}

   protected void destroyApp(boolean unconditional)
   throws MIDletStateChangeException {}
} //fim da classe Exemplo
```

**Figura 3.16 - Classe de exemplo de uma aplicação para a plataforma Sun SPOT**

### 3.5.2. Integração de Dispositivos na Web

Essa Subseção apresenta os procedimentos de como os dispositivos físicos podem ser disponibilizados na forma de serviços Web RESTful. A disponibilização dos dispositivos pode ocorrer de duas formas: através da inclusão de um servidor embarcado executado diretamente no dispositivo; ou por meio de *Smart Gateways*, que permitem que até mesmo os dispositivos que não possuem um servidor embarcado (devido a restrições de *hardware* ou por restrição do projeto) tenham suas funcionalidades disponibilizadas na Web [Trifa et al. 2009].

Os dispositivos que possuem um servidor embarcado podem ter seus recursos acessados por requisições HTTP. Além disso, o servidor também permite que esses

recursos sejam acessados por meio de uma interface RESTful [Guinard e Trifa 2009]. No caso dos dispositivos não suportarem a pilha de protocolo IP, é necessário fazer uso de um *proxy* para receber as requisições Web da rede TCP/IP e enviar tais requisições para o dispositivo por meio do enlace da rede utilizado pelo mesmo[Guinard e Trifa 2009]. Nesse *proxy* podem ser implementadas outras funcionalidades as quais agregam valor aos serviços do dispositivo. Por exemplo, o *proxy* pode implementar serviços de descoberta, que permitem que os dispositivos de um mesmo tipo sejam adicionados (quando estiverem na área de atuação do *proxy*) e removidos do *proxy* (se o dispositivo se tornar inacessível) [Guinard e Trifa 2009], [Mayer 2010].

A Subseção 3.5.2.1 apresenta um servidor embarcado para plataforma Sun SPOT que permite que as funcionalidades desse dispositivo sejam providas na forma de serviços RESTful. Nessa mesma Subseção é apresentado um *proxy* que permite que requisições Web sejam encaminhadas para os SPOTs. A Subseção 3.5.2.2 apresenta o protótipo de um *Smart Gateway* e demonstra a associação de dois SPOTs ao Gateway. Um SPOT possui um servidor embarcado enquanto o outro SPOT não. Dessa forma, é exemplificado um cenário básico da inclusão de dispositivos físicos na Web. A Subseção 3.5.2.3 apresenta como os SPOTs associados ao *Smart Gateway* são acessados na Web. A Subseção 3.5.2.4 mostra como obter representações dos estados dos recursos através de exemplos dessas representações.

#### 3.5.2.1. Técnica de Implementação de Servidores Web em Dispositivos Embarcados

A utilização de servidores embarcados em objetos físicos permite que as funcionalidades de tais objetos sejam disponibilizadas como recursos Web. Porém, as metodologias utilizadas na criação de serviços Web não foram projetadas para serem empregadas em dispositivos que possuem hardwares restritos e são alimentados por bateria (por exemplo, sensores sem fio) [Shelby, 2010]. Portanto, para que os servidores Web sejam utilizados em dispositivos embarcados, eles devem atender uma série de requisitos. Em [Shelby, 2010] foram apresentados os requisitos e padronizações para servidores embarcados. Um exemplo de requisito a ser atendido de forma padronizada é a compressão das mensagens do protocolo HTTP.

Esta Subseção apresenta as principais características de um servidor embarcado desenvolvido para a plataforma Sun SPOT disponibilizado entre os projetos de demonstração que acompanham a nova versão do SDK lançada em Outubro de 2010 [Gupta 2010], [Gupta e Simmons, 2010], [Sun SPOT, 2011]. Esse servidor faz parte do projeto *WebOfThings* e está disponível em [Sun SPOT, 2011].

O projeto *WebOfThings* da plataforma Sun SPOT segue os *drafts* Chopan (*Compressed HTTP Over PANs*, 2009) e Reverse HTTP (2009) e a RFC 5785 (2010) da IETF [Gupta 2010], [Gupta e Simmons, 2010]. O *draft* Chopan descreve como o protocolo HTTP pode ser compactado em mensagens binárias a fim de reduzir o tamanho das mensagens que utilizam esse protocolo. As mensagens HTTP compactadas são transmitidas sobre UDP em redes sem fio de baixa taxa de transmissão e pequena largura de banda como, por exemplo, as redes IEEE 802.15.4/*ZigBee*. O *draft* Reverse HTTP descreve um método que permite que as requisições HTTP sejam enviadas para dispositivos que não podem ser acessados diretamente (por exemplo, dispositivos que estejam atrás de um *firewall* ou de NAT - *network address translation*). A RFC 5785

descreve como serviços Web podem obter meta-informações sobre o servidor acessando o *path* "/.well-know/" que simboliza as localizações dos recursos.

O projeto *WebOfThings* provê uma solução para que as funcionalidades dos SPOTs sejam disponibilizadas sob a forma de recursos Web. Conforme apresentado na Subseção 3.5.1, os SPOTs se comunicam entre si e com a EB utilizando padrão IEEE 802.15.4 e esses dispositivos não suportam o protocolo IP. Por isso, no projeto *WebOfThings* a EB (conectada a um computador que é capaz de se comunicar utilizando o protocolo IP) é utilizada para receber as requisições Web retransmitindo-as para os SPOTs [Gupta 2010], [Gupta e Simmons, 2010]. Para isso, a EB deve manter uma referência para cada SPOT. Para tal, a EB utiliza como registro de identificação do SPOT parte do endereço físico (MAC) desse SPOT. O serviço de descoberta é o responsável por realizar o registro dos SPOTs na EB. Esse serviço envia mensagens em *broadcast* em uma porta pré-definida e aguarda mensagens de algum SPOT que deseja se registrar. A mensagem de registro de um SPOT deve conter sua identificação. Essa identificação é utilizada para disponibilizar o SPOT como um recurso Web e as funcionalidades desse SPOT são subrecursos do mesmo. Por exemplo, o dispositivo de sensoriamento de temperatura do spot-1265 (identificação de um SPOT) é um subrecurso desse SPOT. Após a etapa de registro, os SPOTs podem receber requisições Web.

As requisições advindas da Web são enviadas para a estação base, a qual faz uso da URI da requisição para identificar qual recurso está sendo solicitado. Um exemplo de URI é *http://localhost:8888/spot-1265/light*. O trecho "spot-1265" é utilizado para identificar o SPOT, enquanto o trecho "light" é utilizado para identificar qual o recurso desse SPOT que o cliente deseja acessar (nesse caso, o recurso é o dispositivo de sensoriamento de luminosidade desse SPOT). Após a EB identificar qual SPOT está sendo solicitado na requisição HTTP, ela comprime a requisição recebida e a retransmite através do enlace de rede sem fio IEEE 802.15.4 para o respectivo SPOT e fica aguardando uma resposta do dispositivo solicitado (como exemplificado na Figura 3.17).

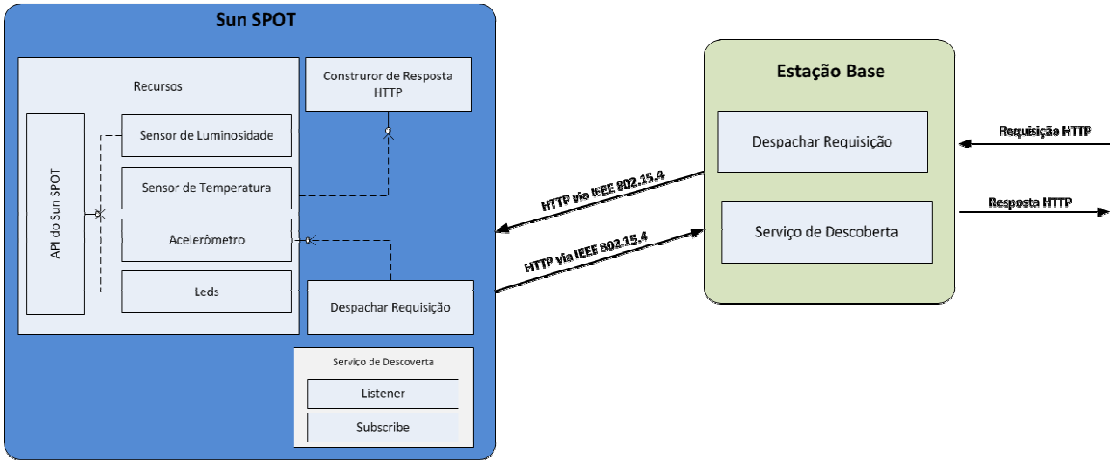


**Figura 3.17 - Tratamento de uma requisição HTTP para um SPOT que contém um servidor embarcado.**

Quando uma requisição HTTP é recebida pelo SPOT, o servidor HTTP é utilizado para descompactar essa requisição e criar um objeto contendo todas as informações incluídas nessa mensagem (por exemplo, o objeto criado, dentre outras

informações, pode ter o *path* de um recurso e o método presente na requisição). Esse objeto que representa a requisição HTTP é utilizado para identificar o recurso solicitado por meio do *path*, além de indicar qual operação deve ser realizada sobre esse recurso. As operações que podem ser executadas sobre um recurso são: criar, recuperar, atualizar e apagar. Essas operações são executadas com base no respectivo método da requisição HTTP (GET, POST, PUT ou DELETE) em conformidade com a propriedade ROA e o princípio REST de interface uniforme. É necessário ressaltar que os recursos não precisam tratar todos os métodos HTTP utilizados em serviços RESTful. Por exemplo, os dispositivos de sensoriamento apenas enviam o valor sensoriado; logo esses recursos são acessados por requisições que contém o método GET do HTTP e os demais métodos não são suportados, pois não possuem utilidade para esse recurso. Os *leds* por outro lado, podem retornar uma representação contendo as cores com que eles estão acesos (por exemplo, utilizando valores RGB – *red, green, blue*) quando recebem uma requisição com o método GET. Ou podem alterar sua cor com base em parâmetros de uma requisição que contém o método PUT.

Após o tratamento da requisição HTTP, o servidor do SPOT envia uma mensagem de resposta HTTP com um código de sucesso ou um código de erro (por exemplo, um 200 OK que indica que o conteúdo da requisição foi processado com sucesso ou um 404 que indica que o recurso não foi encontrado). As mensagens HTTP de sucesso podem conter uma representação do recurso solicitado (por exemplo, as requisições HTTP que contém o método GET). Nesse caso, o próprio recurso retorna uma representação (em HTML, JSON ou XML, por exemplo) que é incluída na mensagem de resposta HTTP. Em seguida, a mensagem HTTP de resposta deve ser compactada e enviada para a EB onde será descompactada e enviada ao cliente que fez a requisição. A Figura 3.18 e a Figura 3.19 apresentam as principais classes do projeto *WebOfThings* que fazem com que os SPOTs disponibilizem suas funcionalidades na Web na forma de serviços RESTful.

A Figura 3.18 apresenta as principais classes do projeto *WebOfThings* que são executadas na EB. Essas classes implementam as funcionalidades que permitem que os SPOTs registrados sejam acessados como qualquer serviço Web RESTful. Nessa classe é criada uma instância de *NanoAppServer*. Em *NanoAppServer* estão implementadas as funcionalidades que permitem que as requisições sejam encaminhadas para o SPOT adequado, como se esse dispositivo fosse um recurso Web. A classe *Main* também instancia *TCPHandle* passando a referência do objeto *NanoAppServer* criado anteriormente.

*TCPHandle* tem como objetivo receber as requisições HTTP advindas da Web em uma porta pré-definida. Para cada requisição advinda da Web, *TCPHandle* devolve uma resposta via Web ao emissor de tal requisição. Por exemplo, ao receber uma requisição HTTP advinda da Web, *TCPHandle* cria um objeto do tipo *HttpRequest* que contém todas as informações da requisição HTTP recebida da Web. Para tal, *TCPHandle* utiliza o método estático *parse* da classe *HttpRequest* passando como parâmetro a requisição HTTP recebida da Web. A instância de *HttpRequest* que equivale a requisição HTTP advinda da Web possui todos os campos da mensagem HTTP compactados (por exemplo, os campos do cabeçalho da requisição são representados por *bytes* específicos). Em seguida, *TCPHandle* acessa *NanoAppServer* para obter a resposta para essa requisição HTTP.

Em *NanoAppServer* é acessado o método *processRequest* da classe *MainApp* passando como parâmetro a instância de *HttpRequest* criada anteriormente. Em *MainApp* é utilizado o *path* do recurso que identifica o SPOT que deve ser acessado (por exemplo, "spot-1265"). Esse *path* é passado para a instância de *DeviceManager* que identifica se o SPOT solicitado está entre os SPOTs cadastrados na EB. Se um SPOT for identificado, a instância de *HttpRequest* é encaminhada para esse SPOT por meio do método *sendAndGetResponse* de *UDP6Forwarder*. O método *sendAndGetResponse* de *UDP6Forwarder* encaminha a requisição e aguarda a resposta do SPOT. Finalmente, a resposta enviada pelo SPOT é entregue por *TCPHandler* ao cliente que fez a requisição (qualquer erro em algumas das etapas anteriores fará com que uma resposta contendo um código de erro seja enviada para o cliente Web).

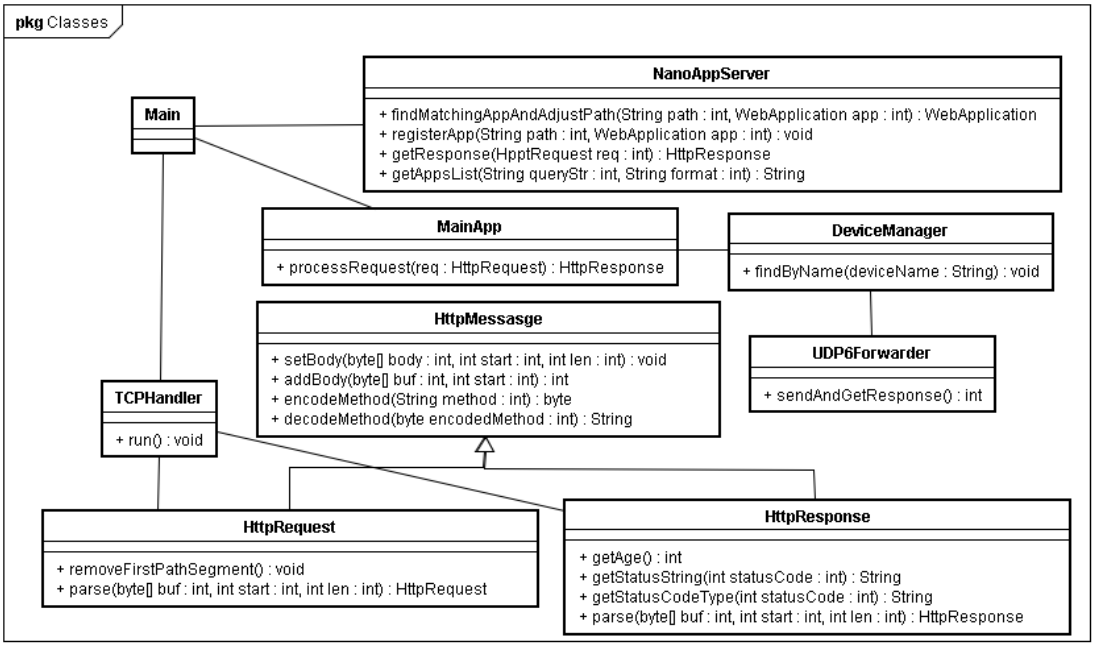


**Figura 3.18 - Principais classes do projeto *WebOfThings* executadas na Estação Base**

A Figura 3.19 apresenta as principais classes que são executadas no Sun SPOT para que as funcionalidades dessa plataforma sejam disponibilizadas como recursos RESTful. A classe *WoTServer* estende *MIDlet*, logo essa classe *MIDlet* é utilizada pela *Squawk* para iniciar a aplicação. Em *WoTServer* é instanciada a classe *NanoAppServer* que implementa as funcionalidades do servidor HTTP. Nos SPOTs, *NanoAppServer* armazena as instâncias das classes que implementam as funcionalidades dos recursos, bem como uma identificação para cada uma dessas instâncias. Essa identificação é utilizada para relacionar um recurso com o *path* de uma requisição. Por exemplo, uma instância da classe *LightSensor* que permite que o dispositivo de sensoriamento de luminosidade seja disponibilizado como recurso Web é identificada pela *String* "/light" e uma requisição para esse recurso deve conter o *path* "/light", pois é esse *path* que será utilizado para identificar o recurso.

Cada classe que implementa as funcionalidades de um recurso deve estender *WebApplication* (por exemplo, a classe *LightSensor* estende a classe *WebApplication*). A classe *WebApplication* é uma classe abstrata que define a interface necessária para as classes que implementam as funcionalidades de um recurso.

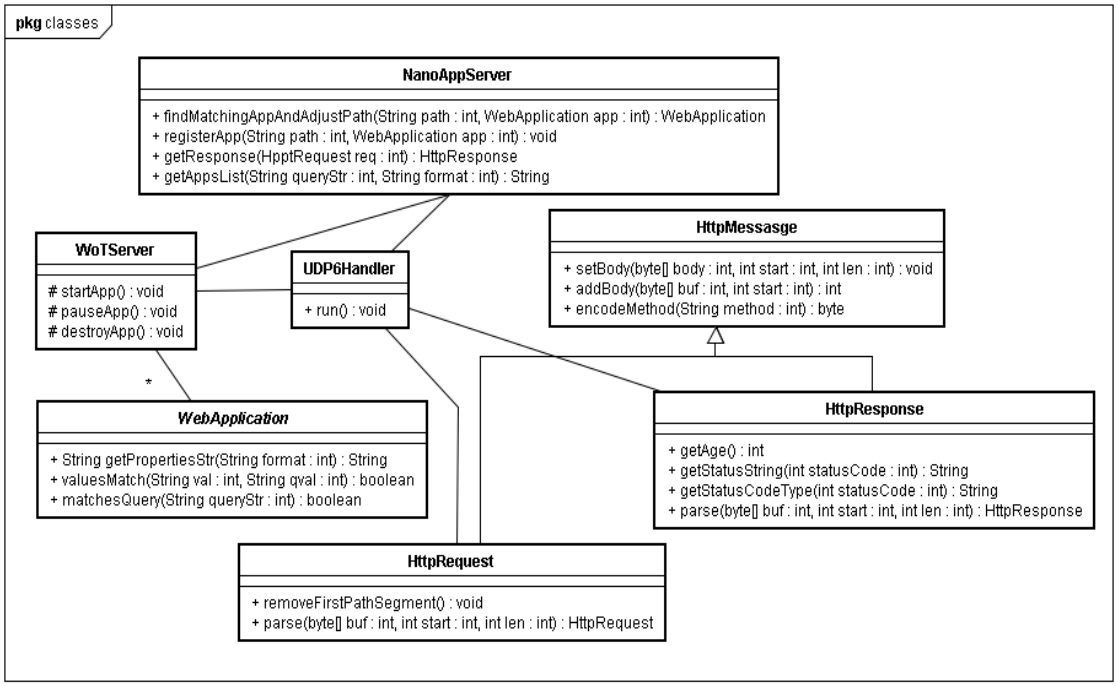


**Figura 3.19 - Principais Classes do projeto *WebOfThings* Executadas nos SPOTs**

Em *WoTServer* também é instanciada a classe *UDP6Handler* que realiza a tarefa de comunicação utilizando os níveis de rede da plataforma Sun SPOT. As mensagens no nível de rede são recebidas em *UDP6Handler* e o conteúdo de cada mensagem é um *array* de *bytes* que equivale a requisição HTTP compactada. Esse *array* de *bytes* é passado para o método estático *parse* da classe *HttpRequest* que retorna uma referência para um objeto do tipo *HttpRequest* que representa a requisição descompactada. *UDP6Handler* faz uso do objeto que equivale a requisição para obter um objeto *HttpResponse* que equivale a resposta adequada para a requisição recebida. Em seguida, *UDP6Handler* solicita que a resposta HTTP seja compactada para finalmente enviá-la à EB.

As classes *HttpRequest* e *HttpResponse* estendem a classe abstrata *HttpMessage* e contêm todas as funcionalidades necessárias para manipulação das requisições e respostas, respectivamente. A classe *HttpMessage* contém os métodos que realizam a compressão e descompressão das mensagens HTTP (cabeçalhos, tipos MIME, entre outros). A classe *HttpRequest* fornece todos os métodos para acesso aos cabeçalhos e parâmetros da requisição HTTP (por exemplo, obtenção do método da requisição HTTP). Os objetos do tipo *HttpResponse* são criados com base no recurso indicado no *path* da requisição (por exemplo, uma requisição utilizada para obter uma representação do valor da temperatura faz com que uma resposta seja gerada na instância da classe que implementa as funcionalidades desse recurso). Quando um recurso não existe ou o método indicado na requisição não é suportado pelo recurso, é criada uma instância de *HttpResponse* com um código de erro.

A Figura 3.20 apresenta o diagrama de seqüência das mensagens trocadas durante a execução do servidor embarcado nos SPOTs: da etapa de inicialização ao tratamento das requisições dos clientes que desejam obter uma representação de algum recurso disponibilizado. A classe *WoTServer* instancia a classe *NanoAppServer* e todas

as classes dos recursos. Em seguida, as instâncias das classes que implementam os recursos são passadas para *NanoAppServer* juntamente com o *path* que os identifica através do método *registerApp* dessa classe. A próxima classe a ser instanciada é *UDP6Handler* que recebe um parâmetro do tipo *int* e uma referência para a instância de *NanoAppServer*. O parâmetro *int* indica a porta onde deve ser aberta a conexão no nível de rede da plataforma Sun SPOT. Essa porta deve ser a mesma utilizada pela EB para encaminhar as requisições Web para o SPOT.

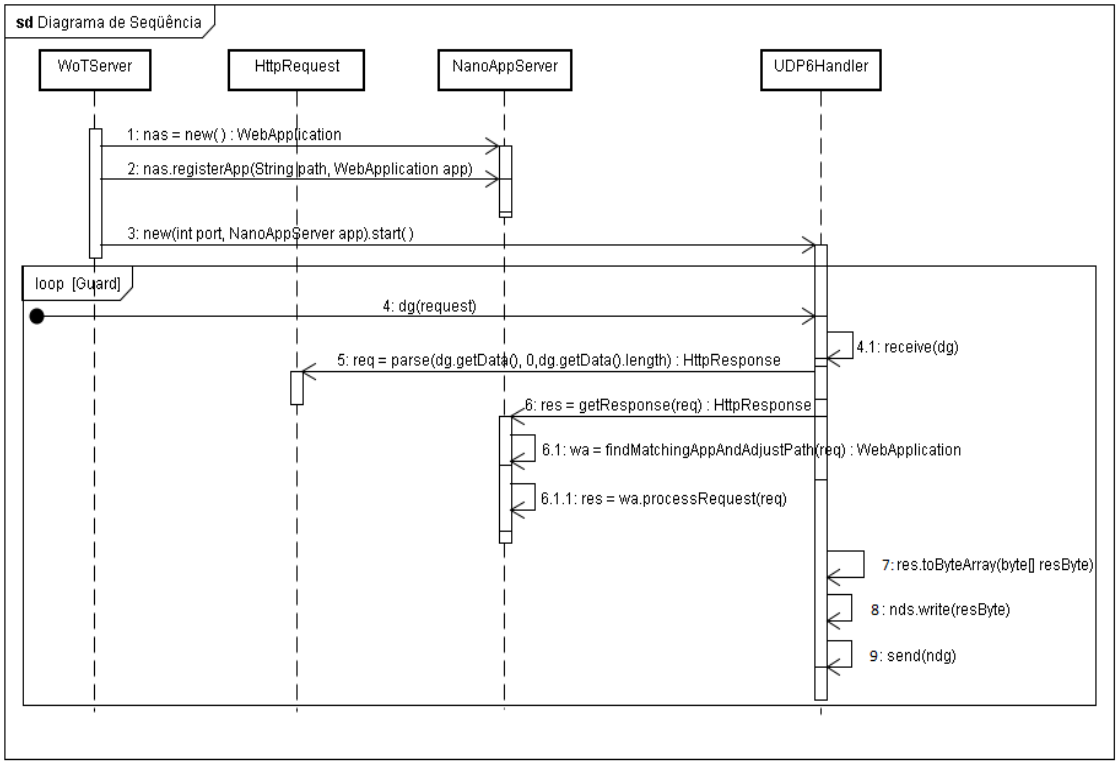


**Figura 3.20 - Diagrama de seqüência das mensagens trocadas durante a execução do servidor embarcado nos SPOTs**

O objeto do tipo *UDP6Handler* deve permanecer em um laço contínuo ouvindo mensagens na porta indicada. As requisições e respostas HTTP trafegam no enlace de rede sem fio dentro de datagramas. A comunicação utilizando datagramas é realizada através dos métodos da classe *UDPConnection* que estende a interface *DatagramConnection* disponível no SDK. Após receber o datagrama, *UDP6Handler* passa o conteúdo do datagrama como parâmetro do método estático *parse* da classe *HttpRequest* a fim de receber uma referência para um objeto do tipo *HttpRequest* que representa a requisição HTTP recebida (mensagem 5 do diagrama de seqüência).

A referência para o objeto que representa a requisição é utilizada por *UDP6Handler* para obter um objeto *HttpResponse* que equivale a resposta para essa requisição. Essa referência do objeto que representa a requisição é passada para *NanoAppServer* através do método *getResponse* (mensagem 6 do diagrama de seqüência). Então, *NanoAppServer* utiliza a instância do objeto que representa a requisição como parâmetro do método *findMatchingAppAndAdjustPath* para buscar entre a coleção dos recursos aquele que está sendo solicitado. Nesse momento, é extraído o *path* da requisição o qual identifica o recurso desejado. Quando o recurso solicitado é identificado, a instância da classe que implementa as funcionalidades do

recurso processa a requisição gerando uma resposta adequada (se ocorrer algum erro, uma resposta é criada contendo um código de erro). Finalmente, a resposta é compactada (mensagem 7 do diagrama de seqüência) através do método *toByteArray* da classe *HttpResponse* que repassa a tarefa de compressão para a classe *HttpMessage*. Um *array* de bytes é obtido desse processo o qual é inserido no datagrama, através do método *write*. Então o datagrama que contém a resposta HTTP é enviado através do método *send* em *UDP6Handler*.

### 3.5.2.2. Implementação de *Gateways*

Esta Subseção contém uma descrição do protótipo de um *Smart Gateway* desenvolvido a partir da extensão do *proxy* reverso apresentado na Subseção 3.5.2.1. A descrição desse protótipo tem como objetivo fornecer uma visão mais aprofundada sobre um *Smart Gateway*. No protótipo, dois tipos de dispositivos foram conectados ao *gateway* através do uso de controladores. Ambos os dispositivos são Sun SPOTs, porém um dispositivo possui um servidor embarcado e o outro não. No segundo caso, o *gateway* se comunica com o SPOT utilizando a API deste através de um controlador desenvolvido para tal finalidade. Desta forma, pretende-se exemplificar como um dispositivo que não possui servidor embarcado pode ter seus recursos disponibilizados na Web através de um *Smart Gateway*.

A Figura 3.21 mostra os principais componentes do *gateway*. Os controladores de cada dispositivo possuem um módulo de descoberta personalizado para o tipo de dispositivo que controla. Os controladores enviam mensagens em *broadcast* periodicamente em uma porta pré-definida. Os dispositivos que querem se associar ao *gateway* devem responder a essas mensagens enviando uma resposta para o controlador. Quando um novo dispositivo é adicionado ao *Smart Gateway*, o *path* de acesso a esse novo recurso é negociado entre um módulo do *gateway* que faz a gerência de *paths* e o controlador do dispositivo a fim de evitar *paths* repetidos. Normalmente o *path* segue a seguinte estrutura: */{tipoDispositivo}/dispositivo-id/recurso* (por exemplo, */spotapi/spot-0f40/temperature*).

Outro módulo apresentado na Figura 3.21 é o *DespacharRequisicao*. Esse módulo utiliza as URIs das requisições para obter a informação necessária para selecionar o controlador do tipo de recurso (dispositivo) que está sendo solicitado (o tipo de dispositivo pode ser um SPOT com servidor embarcado, por exemplo). Nessa etapa, uma referência de um objeto Java contendo as informações da requisição é passada para o controlador. O controlador, por sua vez, utiliza as informações da requisição para adquirir o recurso diretamente no dispositivo solicitado. O recurso obtido do dispositivo é devolvido ao componente *DespacharRequisicao*, o qual retorna uma resposta contendo o dado solicitado ao cliente que submeteu a requisição. Se algum erro ocorrer durante esse processo, uma resposta contendo um código de erro é devolvida ao cliente.

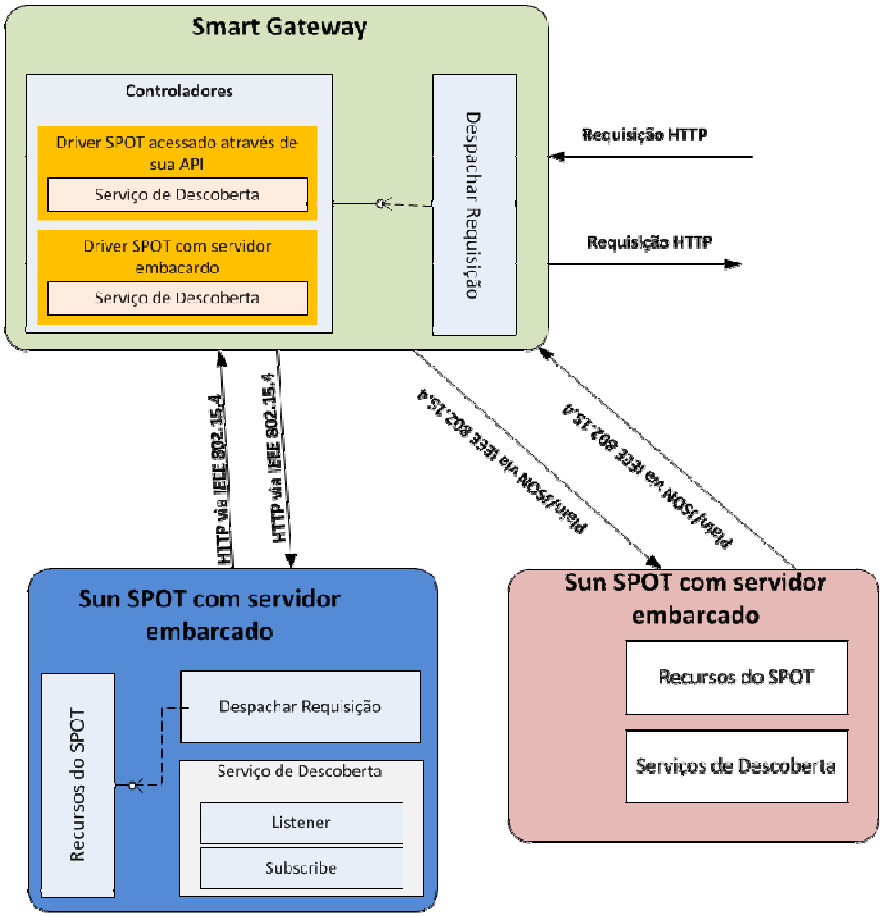


**Figura 3.21 - Componentes básicos do *Smart Gateway* e de dois tipos de dispositivo (um SPOT com servidor embarcado e o outro sem sendo a comunicação realizada através da API deste)**

As principais classes do *Smart Gateway* utilizadas para registrar e identificar controladores, além de encaminhar requisições da Web para esses controladores são apresentadas na Figura 3.22. Nessa figura, a classe *MainApp* é chamada a cada nova requisição. As requisições são encaminhadas para o controlador adequado com base no *path* dessas requisições.

A Figura 3.23 apresenta as principais classes de um controlador. Tais classes serão explicadas no contexto de um exemplo. O controlador utilizado para SPOTs que possuem um servidor embarcado foi desenvolvido de forma análoga, mas é apresentado na Subseção 3.5.2.1. Os controladores estendem a classe *WebApplication* e devem ser registrados na classe *MainApp* quando uma instância do controlador juntamente com o *path* que será utilizado para identificá-lo são passados para *NanoAppServer*. Os controladores são *Singletons*, isto é, só existe uma instância de cada controlador por *Smart Gateway*.

A classe *DeviceDiscovery* é implementada como uma *Thread* que executa em intervalos pré-definidos a tarefa de enviar anúncios sobre o *gateway* a fim de cadastrar novos dispositivos dinamicamente. Essa classe também é utilizada para controlar a saída de dispositivos que não possam mais ser alcançados (devido ao esgotamento da bateria do dispositivo, ou por qualquer outro motivo). Por exemplo, se um dispositivo conectado não responder a três anúncios consecutivos ele pode ser considerado

inalcançável, enquanto nos dois primeiros anúncios o controlador pode considerar que tal dispositivo está em modo *sleep* para economizar energia.

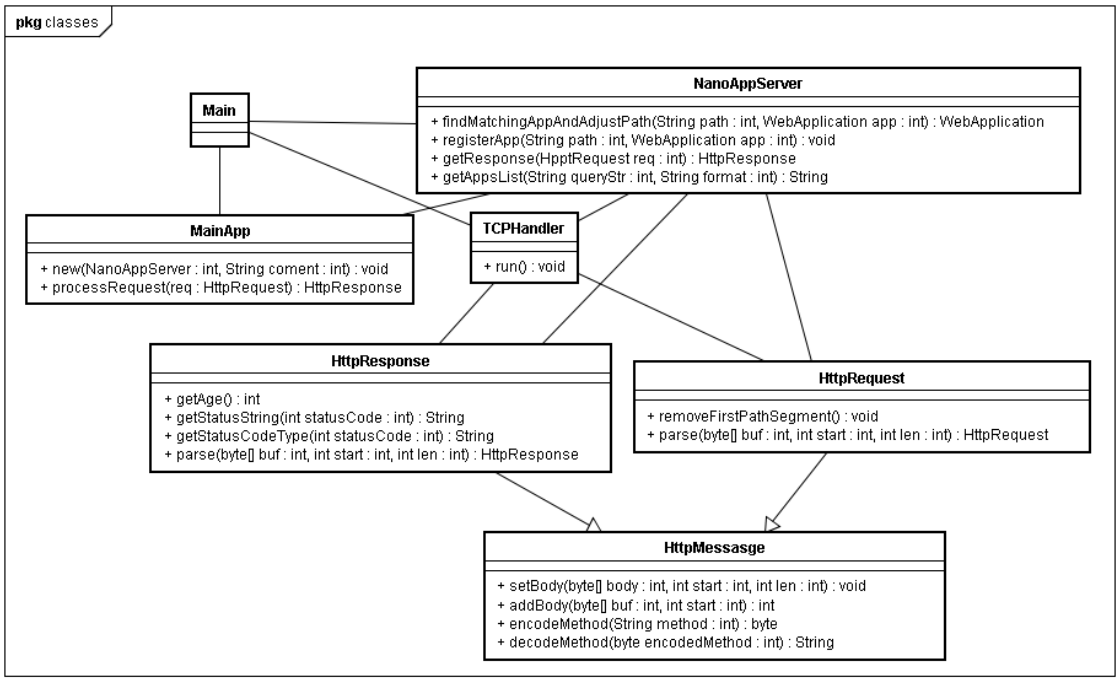


**Figura 3.22 - Principais classes do *Smart Gateway***

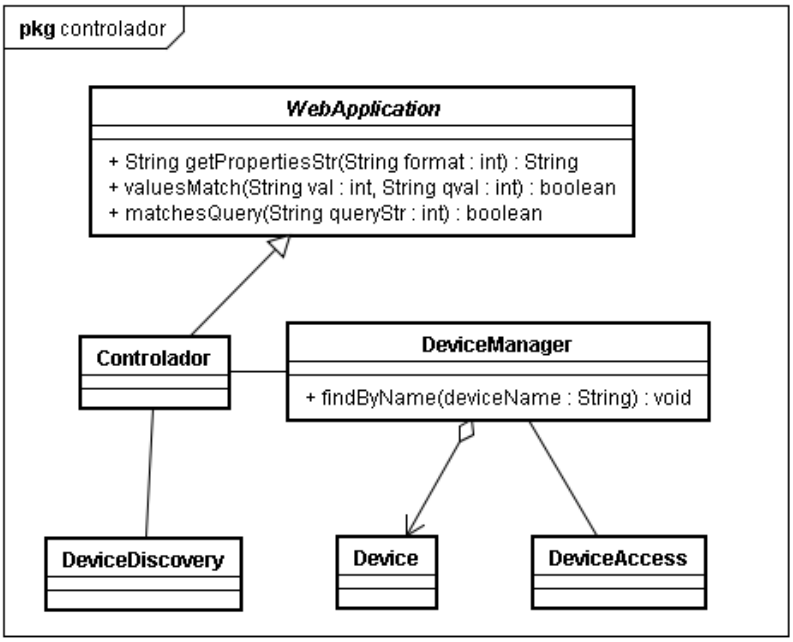


**Figura 3.23 - Principais classe de um controlador**

A classe *DeviceManager* mantém uma lista de todos os dispositivos e realiza o acesso aos recursos dos mesmos através da classe *DeviceAccess*. Uma requisição identifica através do *path* um recurso (representado pelo controlador), um dispositivo específico (considerado subrecurso do controlador) e alguma funcionalidade do dispositivo (considerada seu subrecurso). Por exemplo, o *path /spotApi/spot-0f40/temperatura*, a primeira parte do *path* "/spotApi" identifica o controlador desse

tipo de dispositivo. A segunda parte "/dispositivo-0f40" identifica o SPOT onde "0f40" são os quatro últimos dígitos do endereço MAC desse SPOT. Por último, o trecho "/temperatura" é utilizado para identificar o dispositivo de sensoriamento de temperatura desse SPOT.

A classe *Device* contém toda informação necessária para que seja realizado um acesso específico a um dispositivo. Uma nova instância dessa classe é criada e adicionada a lista de *DeviceManager* sempre que um novo dispositivo é encontrado. A classe *DeviceAccess* é utilizada para acessar os dispositivos por meio da API dos mesmos. Essa classe acessa o dispositivo utilizando os dados de uma instância de *Device*. Um controlador pode evitar o acesso contínuo aos tipos de dispositivo que representa utilizando *cache* dos dados com o objetivo de diminuir o consumo de energia desses dispositivos.

Um controlador utilizado para acessar SPOTs por meio da API destes pode enviar mensagens de anúncio do *gateway* na porta 201. Neste trabalho, os SPOTs que se comunicam com o *Smart Gateway* utilizando apenas a API enviam uma mensagem JSON contendo as seguintes informações: MAC, nome, *timestamp* e uma lista contendo as funcionalidades (recursos) dos SPOT. A Figura 3.24 ilustra a estrutura JSON utilizada para prover as informações necessárias para que os SPOTs sem servidor embarcado possam se cadastrar no *gateway*. A Figura 3.25 apresenta a estrutura da mensagem utilizada por esses SPOTs para se associarem a um *gateway*. A Figura 3.26 mostra as classes dos SPOTs que são acessadas por meio de sua API.

```
{"gateway_id":"MAC_gateway","timestamp":"timestamp"}
```

**Figura 3.24 - Estrutura JSON utilizada no serviço de descoberta do controlador que se comunicam com os SPOTs por meio da API destes**

```
{"spot-api":{"MAC":"0014.4F01.0000.0F20","nome":"spot-api","timestamp":"
1300223762980","recursos":{"temperatura":"/temperature","luminosidade":"l
ight"}}
```

**Figura 3.25 - Estrutura JSON de anúncio de capacidade enviada pelos SPOTs que não possuem servidor embarcado para o *gateway***

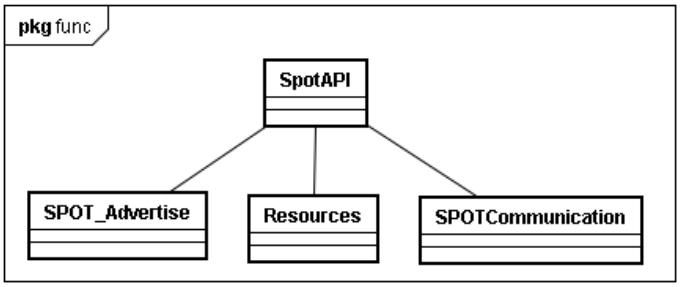


**Figura 3.26 - Diagrama de classes dos SPOTs que não possuem servidor embarcado**

A classe *SpotAPI* é a classe executada pela *Squawk*, pois estende *MiDlet*. A classe *SPOT_Advertise* é responsável pelas tarefas de associação a um *Smart Gateway*. A classe *Resource* é responsável por acessar o recurso adequado quando recebe uma mensagem do *gateway*. A classe *SPOTCommunication* é utilizada para ouvir e responder mensagens direcionadas aos recursos. Na implementação realizada neste protótipo o SPOT fica ouvindo na porta 200 mensagens cujo conteúdo é um documento

JSON. O JSON indica qual tipo de dado (temperatura ou de luminosidade) que o SPOT deve retornar.

#### 3.5.2.3. Exposição de Dispositivos como Recursos REST

Como visto extensivamente ao longo deste Capítulo, na Web das Coisas as funcionalidades dos dispositivos são acessadas por meio de URIs. Dessa forma, os recursos de um SPOT associado a um *gateway* devem ser acessados através de uma URI. Tal URI é utilizada para localizar o *gateway*, identificar um dispositivo e especificar um recurso do mesmo. A URI permite que os recursos sejam identificados de forma única na Web e fazem com que os SPOTs sejam vistos como recursos do *gateway* enquanto as suas funcionalidades são vistas como seus subrecursos [Mayer 2010].

A URI de acesso ao recurso de um SPOT tem o formato http://{endereço do gateway}/{identificação do controlador}/{identificação do SPOT}/{recurso do SPOT}. O endereço do *Smart Gateway* o identifica na Web; a identificação do controlador indica o tipo de dispositivo associado ao *gateway* que o cliente deseja acessar; e a identificação do SPOT indica qual dispositivo associado ao *gateway* que deve ser acessado (os SPOTs são identificados pelos quatro últimos dígitos do seu endereço MAC). O recurso especificado na URI indica qual funcionalidade do SPOT o cliente deseja acessar. Por exemplo, o acesso ao sensor de temperatura do SPOT cujos quatro últimos dígitos do MAC são "1265" pode ser feito através da URI *http://localhost:8888/spotserver/spot-1265/light*, onde "/light" identifica o subrecurso do SPOT. Os SPOTs que não possuem servidor embarcado também são identificados por uma URI com o formato semelhante. Por exemplo, de forma similar ao exemplo do SPOT que possui servidor embarcado, o SPOT sem servidor embarcado que possui os quatro últimos dígitos do MAC "0F20" pode ser acessado em *http://localhost:8888/spotapi/other-0f20/temperature*, onde "other-0f20" identifica um SPOT e "/temperatura" identifica a funcionalidade desse dispositivo que se deseja acessar.

Esse uso de URI também permite que o sistema forneça *links* para que os clientes naveguem entre os subrecursos de um recurso. Por exemplo, uma requisição enviada para *http://localhost:8888/spotserver/spot-1265* retorna uma listagem com *links* para todos os recursos disponíveis nesse SPOT. Esses *links* permitem que o cliente mude da representação do estado de um recurso para outras representações de diferentes recursos. Além disso, também é possível utilizar *links* que conduzem o cliente nas interações que podem ser realizadas com o recurso.

#### 3.5.2.4. Representação de Estado

Os dispositivos conectados a Web fornecem dois tipos de representação de recursos: HTML e JSON. A representação HTML foi adotada para simplificar a interação dos clientes (humanos) como os recursos disponibilizados, permitindo a navegação dentro da estrutura por meio de *links* para os recursos filhos (subrecursos). Esse tipo de representação HTML é retornado pelo *gateway* e pelo SPOT que possui um servidor embarcado. O *gateway* retorna um HTML contendo uma lista com os dispositivos conectados a ele separados por tipo. Cada dispositivo da lista possui um *link* associado a ele que permite que o usuário acesse tal dispositivo (a Figura 3.27 mostra um exemplo da página principal do *gateway*). Os SPOTs que possuem um

servidor embarcado enviam uma representação HTML contendo seus subrecursos com *links* para cada um deles (ver Figura 3.28).

As representações JSON são disponibilizadas quando um SPOT que não possui servidor embarcado é acessado ou quando uma requisição especificando que a representação deve ser JSON é submetida a um SPOT com servidor embarcado. No caso dos SPOTs sem servidor embarcado, a representação JSON é utilizada pelo SPOT para retornar seus dados ou a listagem dos seus recursos (a Figura 3.29 mostra um exemplo de dado de temperatura retornado por um desses SPOTs).


**Figura 3.27 - Representação HTML da página inicial do *Smart Gateway***

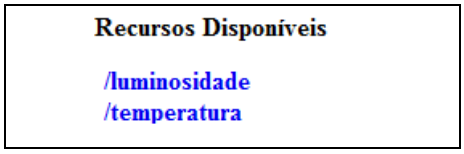


**Figura 3.28 - Listagem dos recursos disponíveis em um SPOT com servidor embarcado**

{"resource":{"typeOfResource":"temperature","scale":"Celsius","value","32","timestamp":"1300223762980","deviceId":"temperatura","link":"/temperatura"}}

**Figura 3.29 - Representação JSON do recurso temperatura**

### 3.5.3. Técnicas de implementação de Mashups Físicos

Essa Subseção apresenta dois exemplos de composição, em *mashups* físicos, dos recursos dos dois SPOTs (com e sem servidor embarcado) conectados ao *Smart Gateway*. Os *mashups* físicos têm como objetivo compor as informações e funcionalidades dos dispositivos inteligentes em serviços mais sofisticados agregando-lhes valor. Os *mashups* aqui apresentados foram desenvolvidos com a ferramenta Presto

[Presto 2010] a qual utiliza a EMML [EMML 2010] que é uma linguagem padrão de desenvolvimento de *mashups*.

O primeiro *mashup* apresenta os dados dos dois SPOTs em um gráfico (ver Figura 3.30). Esse gráfico apresenta a variação da temperatura das últimas doze amostras obtidas antes do momento da requisição. Para a construção do *mashups*, foi implementado um serviço Web que acessa os dispositivos de sensoriamento de temperatura dos SPOTs a cada hora e armazena a representação do recurso acessado em um repositório (para esses exemplos foi utilizado um banco de dados relacional) que atua como histórico de dados. É importante lembrar que as representações de recursos dos dois SPOTs foram obtidas como as de qualquer outro recurso na Web. O SPOT responsável pela geração dos dados representados pela linha laranja do gráfico foi colocado dentro de um ambiente fechado que possui ar condicionado. O SPOT responsável pela geração dos dados representados pela linha azul do gráfico foi colocado em uma ambiente aberto (e obviamente sem ar condicionado) para coletar a temperatura do ambiente.

A integração na Web permitiu que os dados dos SPOTs fossem utilizados em gráficos que podem ser visualizados na maioria dos navegadores Web. O *mashup* criado possibilita, por exemplo, que o responsável pela central do ar condicionado tenha uma visualização de alto nível sobre os dados de temperaturas obtidos pelos sensores, permitindo que esse usuário possa avaliar de forma simples e rápida se o sistema de refrigeração está mantendo a temperatura interna do laboratório independentemente da variação de temperatura do ambiente externo.

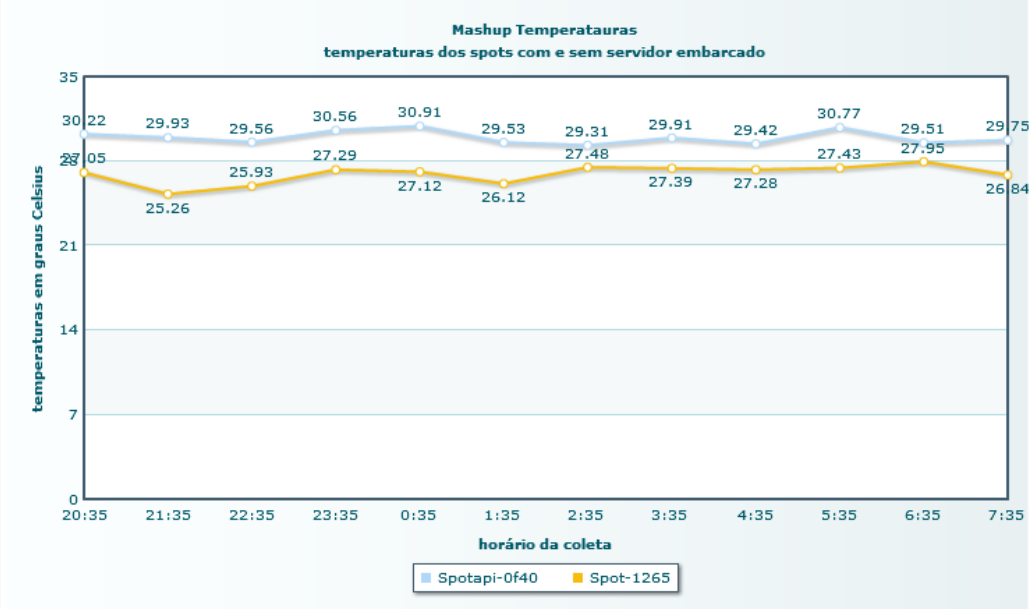


**Figura 3.30 - Gráfico de Temperaturas coletadas pelos SPOTs colocados em diferentes ambientes em um intervalo de 12 horas**

A Figura 3.31 apresenta a integração de um SPOT com um mapa Web [Google Maps Api 2010]. O mapa possui uma marcação com um ícone em uma localidade pré-definida. Quando esse ícone é acionado através de um clique do *mouse*, é aberta uma caixa com uma mensagem contendo a temperatura do local em que se encontra o SPOT no momento do clique. Quando o usuário clica no ícone apresentado no mapa, é obtida uma representação JSON de um dado coletado pelo dispositivo de sensoriamento do

SPOT. Essa representação contém o valor da temperatura monitorada pelo dispositivo de sensoriamento, o horário em que o dado foi coletado e uma descrição do local onde o sensor está localizado (nesse caso, dentro de um laboratório). A localização no mapa foi fornecida manualmente através da inserção das coordenadas geográficas do local. É possível utilizar algum mecanismo (um GPS, por exemplo) que forneça coordenadas geográficas de forma automatizada para o *mashup*. Porém, como o objetivo aqui é apenas ilustrar a integração de recursos da Web com dispositivos físicos, as coordenadas geográficas da localização do SPOT foram fornecidas manualmente.

O *mashup* apresentado na Figura 3.31 permite que o administrador de um sistema possa realizar alguma ação remotamente com base na informação oferecida pelo *mashup*. Por exemplo, o administrador de vários servidores espalhados por diversas instituições de ensino do país poderia desligar remotamente um servidor caso a temperatura de um local onde tal servidor se encontra estivesse acima do ideal para o bom funcionamento do mesmo. Essa ação emergencial seria tomada com base na informação do *mashup* que combina a localização (oferecida pelo mapa) com o dado de temperatura (fornecido pelos sensores).


**Figura 3.31 - *Mashup* físico que apresenta em uma mapa da Web a localização de um SPOT e o valor da temperatura local obtida através do acesso Web ao dispositivo de sensoriamento desse SPOT**

## 3.6. Conclusão

Neste trabalho foram descritos os princípios básicos da *Web* das Coisas. As tecnologias Web foram apresentadas como base viável para a integração de dispositivos inteligentes entre si e desses com a Web. Foi apresentada uma arquitetura para a Web das Coisas baseada nos conceitos de REST e na abordagem de arquitetura orientada a recursos (ROA), e seus componentes foram descritos. Também foram apresentados projetos de implementação de alguns desses componentes.

Graças ao baixo acoplamento, simplicidade e escalabilidade da arquitetura RESTful e a ampla disponibilidade de bibliotecas e clientes HTTP, tal arquitetura está

se tornando uma das mais onipresentes e leves dentre as arquiteturas de integração de dispositivos [Guinard e Trifa 2009]. Devido a isso, o uso de padrões Web para dar suporte a interação entre e com dispositivos inteligentes tem sido considerado uma promissora solução. Embora HTTP introduza uma sobrecarga de comunicação e aumente a latência média, os parâmetros de qualidade providos em geral são suficientes que tais atrasos não prejudiquem a comunicação entre dispositivos. Entretanto, outros trabalhos buscaram soluções de compressão desse protocolo a fim de reduzir o aumento da mensagem ocasionado pelo uso do HTTP, [Shelby, 2010], [Chopan,2009].

As vantagens oferecidas pela introdução de suporte para os padrões da Web diretamente no nível de dispositivo são benéficas para o desenvolvimento de uma nova geração de aplicações que conterão informações do mundo real além dos conteúdos estáticos como acontece atualmente. O emprego dos padrões Web também como base comum para diferentes dispositivos torna muito mais simples a programação dos mesmos. Aplicar os mesmos princípios de *design* que foram responsáveis para o sucesso da Web, em particular abertura e simplicidade, pode alavancar significativamente a ubiqüidade e versatilidade da Web como um terreno comum para dispositivos de rede e aplicações. Além disso, como a maioria das linguagens de programação suportam HTTP, percebe-se a enorme comunidade de desenvolvedores da Web como potenciais desenvolvedores de aplicações para a Web das Coisas.

## Agradecimentos

Este trabalho foi parcialmente suportado pelas agências de fomento CNPq, FINEP e FAPERJ, sob os processos de número 201090/2009-0, 306938/2008-1 e 477229/2009-3 para Flávia C. Delicato; 480359/2009-1 e 311515/2009-6 para Paulo F. Pires; e 309270/2009-0, 4781174/2010-1, 01.100064.00 1979/09 e 101.360/2010 para Luci Pirmez.

## Referências

Ant. Apache Ant, 2010. Disponível em <http://ant.apache.org/>. Acessado em 08 de março de 2011.

Atom. Atom Enable, 2004.Disponível em <http://www.atomenabled.org/>. Acessado em 19 de Fevereiro de 2011.

Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A survey, 2010. Computer Networks 54 (2010), p. 2787–2805.

Bezerra, R. S.; Santos, C. R. P.; Granville, L. Z.; Bertholdo L. M.; Tarouco, L. R. Um Sistema de Gerenciamento de Redes Baseado em Mashups, 2009. Disponível em: < http://www.sbc.org.br/bibliotecadigital/download.php?paper=1550>. Acessado em 14 de Novembro de 2010.

Chumby, 2011. Disponível em <http://www.chumby.com/>. Acessado em 5 de Janeiro de 2011.

Delicato, F. C.; Pires, P. F.; Pirmez, L.; Batista, T. Wireless Sensor Networks as a Service, 2010. IEEE Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops, p. 410 – 417.

Duquennoy, S.; Grimaud, G.; Vandewalle, J. The Web of Things: interconnecting devices with high usability and performance, 2009. In International Conferences on

Embedded Software and Systems. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5066664>. Acessado em 05 de Novembro de 2010

Eclipse, 2011. Disponível em <http://www.eclipse.org/>. Acessado em 08 de março de 2011.

EMML. Enterprise Mashup Markup Language, Open Mashup Alliance, 2010. Disponível em <http://www.openmashup.org/>. Acessado em 18 de Novembro de 2010.

Ferreira Filho, O. F. Serviços semânticos: uma abordagem RESTful. Dissertação de mestrado, USP, 2010. Disponível em <http://www.teses.usp.br/teses/disponiveis/3/3141/tde-11082010-120409/pt-br.php>. Acessado em 3 de Fevereiro de 2011.

Fielding, Roy Thomas (2000), Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, Irvine.

Google Maps API, 2010. Disponível em <http://code.google.com/intl/pt-BR/apis/maps/index.html>. Acessado em 3 de Março de 2011.

Guinard, D. e Trifa, V., "Towards the Web of Things: Web Mashups for Embedded Devices.", In Proceedings of Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web, International World Wide Web Conferences, Madrid, Spain, 2009.

Guinard, D. Towards Opportunistic Applications in a Web of Things. In IEEE International Conference on Pervasive Computing and Communications Workshops, 2010.

Guinard, D.; Trifa, V.; Pham, T.; Liechti, O. Towards Physical Mashups in the Web of Things. In Proceedings of IEEE Sixth International Conference on Networked Sensing Systems, Pittsburgh, USA, June 2009.

Guinard, D.; Trifa, V.; Wilde, E. Architecting a Mashable Open World Wide Web of Things. Em Technical Report, 663. Institute for Pervasive Computing, 2010. Disponível em <http://www.bibsonomy.org/bibtex/2dcc3fabe2de456144254afdcc8e06776/flint63>. Acessado em 28 de Janeiro de 2011.

Gumstix, 2010. Disponível em <http://www.gumstix.com/>. Acessado em 5 de Janeiro de 2011.

Guo, X.; Shen J.; Yin Z. (2010) On software development based on SOA and ROA. In Control and Decision Conference (CCDC), pages 1032 – 1035. Publishing Press.

Gupta, V. The Web of Things and Sun SPOTs, 2010. Disponível em <http://blogs.sun.com/vipul/entry/the_web_of_things_and>. Acessado em 20 de Fevereiro de 2011.

Gupta, V.; Simmons, D. G. Building the Web of Things with Sun SPOTs, 2010. Disponívelem<http://www.sunspotworld.com/S314730_Sun_SPOTs_Web_Of_Things/index.html>. Acessado em 20 de Fevereiro de 2011.

Hadley, J. H. WADL (Web Application Description Language). GlassFish, WADL, 2006. Disponível em <http://wadl.java.net/wadl20060802.pdf>. Acessado em 12 de Dezembro de 2010.

Hadley, J. H. WADL (Web Application Description Language). GlassFish, WADL, 2009. Disponível em <http://wadl.java.net/>. Acessado em 3 de Fevereiro de 2011.

Hui, J. W. e Culler D. E. "Extending IP to Low-Power, Wireless Personal Area Networks." Internet Computing, IEEE 12, no. 4 (2008), p. 37-45.

ITU Internet Reports. ITU Internet Reports 2005: The Internet of Things. Em International Telecomunications Union, 2005.

Jersey, 2011. Disponível em <http://jersey.java.net/>. Acessado em 16 de Março de 2011.

JSON. JavaScript Object Notation, 1999. Disponível em <□JavaScript Object Notation>. Acessado em 18 de Março de 2011.

Lucchi, R.; Millot, M.; Elfers, C. Resource Oriented Architecture and REST, Assessment of impact and advantages on INSPIRE, 2008.

Mayer, S. Deployment and Mashup Creation Support for Smart Things, Institute for Pervasive Computing Department of Computer Science ETH Zurich, 2010. Disponível em <http://www.vladtrifa.com/files/publications/Mayer10.pdf>. Acessado em 02 de Dezembro de 2010.

Nabaztag, 2009. Disponível em <http://www.nabaztag.com/brasil/index.html>. Acessado em 5 de Janeiro de 2011.

NetBeans, 2011. Disponível em <http://netbeans.org/>. Acessado em 08 de Março de 2011.

NIC (National Intelligence Council), Disruptive Civil Technologies – Six Technologies with Potential Impacts on US Interests Out to 2025 –Conference Report CR 2008-07, 2008. Disponível em <http://www.dni.gov/nic/NIC_home.html>. Acessado em 10 de Fevereiro de 2011.

Oracle, JDK (Java SE Development Kit), 2010. Disponível em <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Acessado em 08 de Março de 2011.

Ostermaier, B. Schlup, F. Romer, K. WebPlug: A framework for the Web of Things, em Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference, 2010, p. 690-695.

Pautasso, C.; Zimmermann, O.; Leymann, F. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. Em Proceeding of the 17th international conference on World Wide Web, 2008. Disponível em <http://portal.acm.org/citation.cfm?id=1367606>. Acessado em 02 de Dezembro de 2010.

Plogg. Wireless Energy Management, 2010. Disponível em <http://www.plogginternational.com/>. Acessado em 5 de Janeiro de 2011.

Poken, 2010. Disponível em <http://www.poken.com/>. Acessado em 5 de Janeiro de 2011.

Presto. Mashup Developer Community, 2010. Disponível em <http://www.jackbe.com/enterprise-mashup/>. Acessado em 20 de Janeiro de 2011.

Reverse HTTP. draft-lentczner-rhttp-00. IETF, 2009. Disponível em <http://tools.ietf.org/html/draft-lentczner-rhttp-00>. Acessado em 21 de Janeiro de 2011.

RFC 2068. Hypertext Transfer Protocol -- HTTP/1.1, 1997. Disponível em <http://tools.ietf.org/html/rfc2068>. Acessado em 20 de Fevereiro de 2011.

RFC 2616. Hypertext Transfer Protocol - HTTP/1.1, 1999. Disponível <http://tools.ietf.org/html/rfc2616>. Acessado em 20 de Fevereiro de 2011.

RFC 4287. The Atom Syndication Format, 2005. Disponível em <http://tools.ietf.org/html/rfc4287>. Acessado em 19 de Fevereiro de 2011.

RFC 4944. Transmission of IPv6 Packets over IEEE 802.15.4 Networks, IETF 2007. Disponível em <http://tools.ietf.org/html/rfc4944>. Acessado em 10 de Março de 2011.

RFC 5785. Defining Well-Known Uniform Resource Identifiers (URIs), 2010. Disponível em <http://tools.ietf.org/html/rfc5785>. Acessado em 20 de Fevereiro de 2011.

Richardson L.; Ruby, S. RESTful Web Services. O'Reilly Media, 2008, cap. 4, p. 79-105.

Sandoval, J. RESTful Java Web Services, Master core REST concepts and create RESTful web services in Java. Packt Publishing BIRMINGHAM – MUMBAI 2009, p. 20-81.

Shelby, Z. Embedded web services. Em Wireless Communications, IEEE , 2010. Volume 17, p. 52-57.

Sun SPOT World, Disponível em: <http://www.sunspotworld.com/>. Acessado em 06 de Dezembro de 2010.

Tan, L.; Wang, N. Future Internet: The Internet of Things, em Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference, vol. 5, p. 376-380.

Trifa, V.; Wiel S.; Guinard, D.; Bohnert, T. Design and implementation of a gateway for web-based interaction and management of embedded devices, 2009. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.155.4806>. Acessado em 15 de Fevereiro de 2011.

Tyagi, S. RESTful Web Services. Oracle, 2006. Disponível em <http://www.oracle.com/technetwork/articles/javase/index-137171.html>. Acessado em 3 de Março de 2011.

Webber, J.; Parastidis, S.; Robinson, I. REST in Practice, Hypermedia and Systems Architecture. O´Reilly, 2010, p. 86-95; 386-387.

Wilde, E. Putting Things to REST, 2007. Disponível em <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.7936>. Acessado em 15 de Fevereiro de 2011.

Wilde, E.; Guinard, D.; e Trifa, V. Architecting a Mashable Open World Wide Web of Things, Institute for Pervasive Computing, ETH Zürich, Zürich, Switzerland, No. 663, February 2010

WSDL. Web Service Description Language, W3C, 2007. Disponível em <http://www.w3.org/TR/wsdl20/>. Acessado em 15 de março de 2011.

Yongjia, H. The Plan of Elderly Smart Community: Based on the Concept of Internet of Things, 2010.Em Management and Service Science (MASS), 2010 International Conference, p. 1-4.

Yun, M.; Yuxin, B. Research on the architecture and key technology of Internet of Things (IoT) applied on smart grid, em Advances in Energy Engineering (ICAEE), 2010 International Conference, p. 69-72.