UFABC - Universidade Federal do ABC

Gustavo Arthur da Costa

Implementação de uma rede de sensores com a utilização de protocolos padronizados para Internet das Coisas.

Santo André

2015

Implementação de uma rede de sensores com a utilização de protocolos padronizados para Internet das Coisas.

Relatório Final do Projeto do Trabalho de Graduação do curso de Engenharia de Informação da Universidade Federal do ABC.

Orientador: Prof. Dr. João Henrique Kleinschmidt

Gustavo Arthur da Costa

_____

João Henrique Kleinschmidt

_____

Santo André

2015

Resumo

O projeto abordado neste relatório foi desenvolvido como Trabalho de Graduação do curso de Engenharia de Informação na Universidade Federal do ABC e tem como objetivo principal a implementação de uma rede de sensores com a utilização de protocolos já estabelecidos no âmbito da internet das coisas.

Internet das coisas ou IoT, do inglês *Internet of Things,* é um tema que está em ampla discussão e a ideia básica é que objetos do dia a dia possuam uma conexão com a internet. A adição de tecnologias de comunicação a esses objetos não só proporciona a eles a capacidade de se comunicar, como também abre possibilidades para novas aplicações que venham a utilizar os dados obtidos para proporcionar novos serviços.

Esse projeto aborda a descrição e implementação de protocolos que vêm sendo utilizados como tentativa de padronização de uma pilha de protocolos para IoT como 6LoWPAN, RPL e CoAP e tecnologias específicas para o uso em dispositivos de IoT como o sistema operacional Contiki e a especificação IEEE 802.15.4.

A implementação dos protocolos em hardware e o desenvolvimento de alguns exemplos de aplicações para a rede de sensores proposta fazem parte do escopo do projeto como uma forma de exemplificar as possibilidades geradas por esses novos protocolos e tecnologias.

Abstract

The Project covered by this report was developed for the Graduation Project for the course of Information Engineering of Federal University of ABC and has as its main objective the implementation of a sensor network with the usage of well-established Internet of things protocols.

Internet of things or IoT is a topic that is being widely discussed nowadays and the basic idea is that common objects will start having an Internet connection. The usage of communication technologies on this objects will not only provide them the ability to communicate but also will open new possibilities for new applications that can take advantage of the data collected by them to create new services.

This project will cover the choice of protocols that are being used as an attempt to standardize a protocol stack for IoT, like 6LoWPAN and CoAP and technologies designed to be used with IoT devices, such as the operational system Contiki and the IEEE 802.15.4 specification.

The hardware implementation of this protocols and the development of some example applications for the sensor's network are also on the scope of this project as a way to demonstrate possibilities created by this protocols and technologies.

Sumário

1. Introdução

O termo *Internet of Things* (IoT) ou Internet das Coisas, vem se tornando cada vez mais conhecido e é um tema de pesquisa e nicho de mercado com crescente importância atualmente [1]. O conceito de IoT está relacionado com a conexão com a internet de objetos do dia a dia, partindo de etiquetas RFID (*Radio-Frequency Identification),* smartphones, computadores, que já estão atualmente conectados para objetos "comuns" como roupas, aparelhos domésticos, móveis e carros. A utilização de sensores é um ponto chave de aplicações IoT por ser necessário coletar dados do ambiente para uma tomada de decisão baseada nos mesmos.

Redes de sensores sem fio (RSSF) são geralmente classificadas como um tipo de rede ad-hoc e são um tópico que vem se desenvolvendo muito nos últimos anos. Com a notabilidade que o termo IoT ganhou e com as características compartilhadas entre esses dois temas, as RSSF começaram a ser classificadas como parte de IoT.

A adição de tecnologias diversas de comunicação a esses objetos como, por exemplo, Bluetooth, ZigBee e WI-FI não só proporciona aos mesmos capacidade de se comunicar, como também abre possibilidades de agregar um certo nível de inteligência a esses dispositivos, o que abre um grande leque de novos serviços que podem surgir a partir dessa conectividade [1].

IoT traz várias características novas por conectar objetos diversos à internet. Essas características podem trazer oportunidades de inovação mas também trazem novos problemas a serem enfrentados. Abaixo serão descritas algumas dessas características:

- **Facilidade de Crescimento (*Scalability*)** – Com a adição de diversos novos objetos à internet, a comunicação entre os mesmos precisa considerar o um número de dispositivos extremamente maior do que o encontrado atualmente, com isso problemas de endereçamento e roteamento podem surgir. Por

outro lado a com a quantidade de dados que poderão ser coletados por esses novos dispositivos, aplicações de Big Data podem surgir para tratar esses dados e gerar novas aplicações. Por exemplo, em um cenário em que todos os carros de uma cidade possuem diversos sensores conectados com dados sobre localização, velocidade, quantidade de combustível, entre outros, é possível desenvolver sistemas de melhor controle de fluxo de veículo. O próprio veículo ao receber informações da cidade, pode tomar melhores decisões de rota ou informações sobre postos de combustível mais próximos.

- **Gerenciamento de Energia –** A utilização de dispositivos diversos com rádios para comunicação e poder de processamento é afetada diretamente pela capacidade de sua bateria. Com um número muito alto de dispositivos, o processo de carga ou substituição de baterias se torna extremamente complicado, portanto a duração da bateria desses dispositivos é parte vital da sua utilização. Com isso surgem diversos desafios de gerenciamento de energia nesses dispositivos que devem ser considerados tanto na aplicação quanto nos protocolos utilizados por eles.

- **Segurança –** A ideia de existir conexão em dispositivos do dia a dia pode gerar oportunidades com relação a segurança, como em sistemas de vigilância que podem coletar dados de diversas fontes, mantendo um controle maior sobre o ambiente o qual se deseja monitorar. Por outro lado, novos desafios surgem, estratégias bem estabelecidas de criptografia, troca de chaves e autenticação de mensagens enfrentam complicações principalmente pelas limitações de processamento e bateria dos dispositivos utilizados em aplicações IoT.
- **Padronização –** Como a IoT está se baseando em novos dispositivos que estão sendo desenvolvidos em um ritmo acelerado, a padronização dos mesmos é algo complicado por

existirem diversos fabricantes de dispositivos, principalmente relacionado a protocolos de comunicação. [2]

Aplicações de IoT estão presentes em diversas áreas de atuação como em transporte e logística, área da saúde e ambientes inteligentes. Em transporte e logística existem aplicações diversas de rastreamento sendo realizadas por pequenos sensores e etiquetas RFID para identificação. Aplicações na área da saúde estão sendo desenvolvidas principalmente na área de sensoriamento em tempo real de pacientes. Diversos dispositivos e aplicações voltadas para o desenvolvimento de ambientes inteligentes foram implementadas recentemente. O conceito de casa inteligente com interruptores, lâmpadas e termostatos conectados já está mais difundido em mercados como dos Estados Unidos e Europa. [1]

Recentemente várias empresas de tecnologia têm entrado no mercado de IoT. O Google anunciou seu projeto Brillo como uma plataforma para IoT que tem base no Android. O Projeto vai consistir em um sistema operacional para dispositivos IoT e uma linguagem de programação chamada Weave para comunicação entre os dispositivos IoT e dispositivos Android.

As versões *Developer Preview* do sistema operacional e da linguagem Weave estão disponível via cadastro no site do projeto.[8]

A Microsoft anunciou que juntamente com o lançamento do Windows 10 estaria lançando o Windows 10 IoT Core que tem no início um foco maior em plataformas como Arduino e Raspberry Pi. Ele permite desenvolvimento em C# utilizando a IDE Visual Studio e embora ainda não tenha suporte para a grande maioria dos dispositivos é uma iniciativa importante para desenvolvimento de sistemas IoT. [7]

Um exemplo de dispositivo para IoT foi lançado recentemente pela empresa Amazon, que apresentou um dispositivo chamado *Amazon Dash Button*. Ele é um dispositivo conectado com a conta do cliente na Amazon que está relacionado a um produto específico e ao ser pressionado ele realiza um pedido automaticamente daquele produto no site Amazon.com. No exemplo na Figura 1 abaixo o *Amazon Dash Button* está relacionado a lâminas de barbear da Gillete, e pode ser colado em uma superfície de fácil acesso, então quando o cliente perceber que suas lâminas estão acabando,

ele só precisa pressionar o botão e novas laminas serão enviadas a sua residência.

Figure 1 - Amazon Dash Button da Gillete

Esse exemplo de aplicação é simples porém demonstra as possibilidades desse novo panorama de aplicações: dispositivos cotidianos podem ser conectados à internet e criar novas possibilidades de interação.

Com as possíveis novas oportunidades de desenvolvimento vêm também novos desafios. Os protocolos padrão de comunicação para redes comuns não são compatíveis com as características das redes de sensores sem fio ou das redes IoT. Assim é necessária a utilização de protocolos desenvolvidos levando em consideração as características dessas redes [3].

Muitos dos protocolos propostos para atender essas características são proprietários, para aplicações especificas ou para hardware específicos, o que torna o desenvolvimento de aplicações IoT mais complicado por uma falta de padronização. Entretanto algumas organizações propuseram protocolos e especificações padrão para redes IoT.

O objetivo deste projeto é implementar uma rede de sensores sem fio utilizando os protocolos 6LoWPAN (*IPv6 over Low Power Wireless Personal Area Network*), RPL (*Routing Protocol for Low-power and Lossy Networks*) e CoAP (*Constrained Application Protocol*) e a especificação IEEE 812.15.4. Esse objetivo foi definido pois os protocolos escolhidos fazem parte de estratégias de padronização de entidades como o IEEE e IETF para melhorar a interoperabilidade entre dispositivos de diferentes fabricantes [3].

Todos os protocolos utilizados levam em conta os desafios apresentados por aplicações IoT e seus dispositivos e a escolha dos mesmos para a aplicação a ser desenvolvida fez parte do escopo do projeto.[2]

Os objetivos específicos do projeto são :

- Definição dos protocolos e tecnologias.
- Estudo prático e teórico no sistema operacional Contiki.
- Estudo dos protocolos utilizados na proposta e implementação dos mesmos.
- Definição de uma arquitetura de rede.
- Implementação de aplicações exemplo utilizando a rede proposta.

Na seção 2 deste relatório é apresentada uma fundamentação teórica com as principais tecnologias utilizadas. A seção 3 apresenta a implementação do projeto com a descrição do hardware e software e da arquitetura de rede utilizada. A seção 4 possui aplicações exemplos para a implementação realizada e por fim na seção 5 são apresentadas as conclusões.

2. Fundamentação Teórica

Com o desenvolvimento de IoT, diversas tecnologias e protocolos surgiram com o intuito de melhorar as aplicações e a utilização dos dispositivos. Com isso algumas organizações como a IEEE e IETF propuseram alguns frameworks para padronização dos protocolos de comunicação para esses novos sistemas. O padrão IEEE 802.15.4 definiu uma camada física/enlace de baixa potência (*Low-power Physical Layer*) [3], e com base nesse padrão foram desenvolvidas várias tecnologias para IoT como o 6LoWPAN, RPL, CoAP e inclusive a especificação ZigBee.

Em [3] é descrita uma pilha de protocolos para IoT que utiliza especificações como o IEEE 802.15.4 para endereçamento MAC, IETF ROLL RPL para roteamento, IETF 6LoWPAN para endereçamento e CoAP como protocolo na camada de aplicação. RPL, 6LoWPAN e CoAP formam o principal conjunto de protocolos para IoT, por isso a padronização dessa pilha em [3] é de extrema relevância para o desenvolvimento de aplicações IoT.

Com o objetivo desse projeto de realizar uma implementação de protocolos padrão da internet das coisas, essa pilha de protocolos foi adotada e implementada.

Em uma comparação com o modelo OSI (*Open System Interconnect*) e o modelo de camadas Wi-Fi, é possível observar a introdução de uma camada de adaptação entre a camada de enlace (*Data Link*) e a camada de rede (*Network*). Assim o 6LoWPAN atua tratando os frames IPv6 vindo da camada de rede e os tornando compatível com as camadas MAC e física do IEEE 802.15.4. A comparação dessa estrutura de camadas pode ser vista na Figura 2 abaixo:

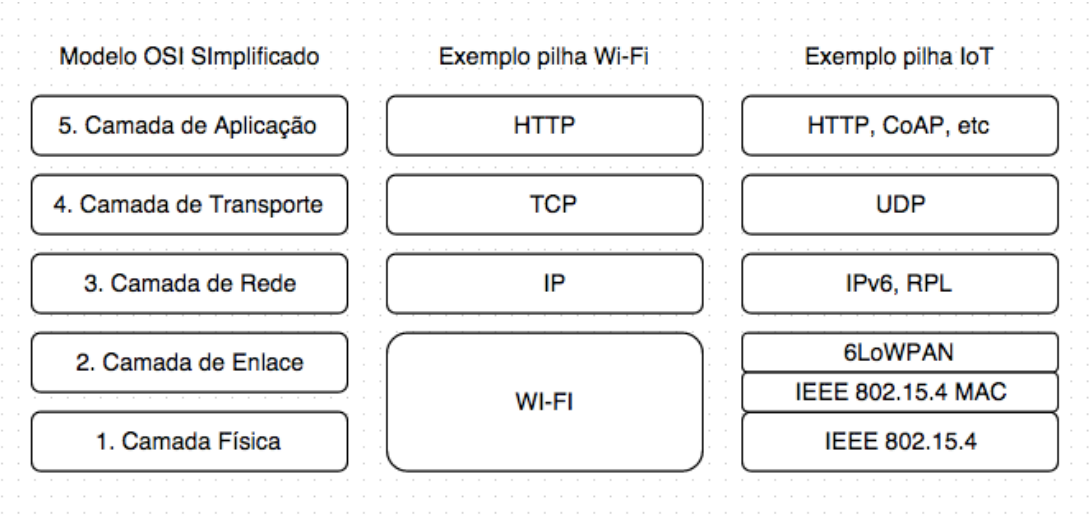

Figure 2 - Modelo Simplificado OSI, camadas Wi-Fi e camadas 6LoWPAN - adaptado de [14]

Os protocolos e tecnologias abordadas no projeto estão detalhados a seguir, iniciando pela especificação IEEE 802.15.4, seguido dos protocolos 6LoWPAN, RPL e CoAP e por fim do sistema operacional Contiki.

2.1 IEEE 802.15.4

A especificação IEEE 802.15.4 estabelece uma camada física e uma camada de acesso ao meio (MAC – *Medium Access Control*) para dispositivos de baixa capacidade de processamento e foi a fundação para a maioria das tecnologias IoT. O ponto de partida para a criação da especificação é a existência de dispositivos de rádio de baixo alcance, em que os dispositivos IoT geralmente estão classificados. As restrições de energia e processamento desses dispositivos requer uma camada física que leve em consideração suas limitações.

A camada física da especificação IEEE 802.15.4 proporciona uma boa relação entre alcance, eficiência energética e taxa de transferência, com foco na criação de redes de grande tamanho.

A faixa de frequência de 2.4 a 2.5 GHz é a mais comumente utilizada em todo o mundo por ser uma banda que não necessita de licença. A modulação *Offset-Quadrature Phase-Shift Keying* (O-QPSK) é utilizada com uma taxa de transferência de 2 Mbps. São definidos 16 canais de frequência de 2MHz de largura localizados a cada 5MHz na faixa de frequência.

O protocolo MAC definido pela especificação é a camada interagindo diretamente com o rádio dos dispositivos e define um cabeçalho para a comunicação e como os dispositivos podem comunicar-se entre si. Esse protocolo é voltado a redes do tipo estrela, onde todos os dispositivos se comunicam com um mote especial de coordenação. A última versão do protocolo MAC na especificação IEEE 802.15.4 é o *Time Synchronized Channel Hopping* (TSCH) que é considerado um protocolo altamente confiável e para dispositivos de baixa capacidade. Sua estrutura se baseia em sincronização entre motes para eficiência energética e *Channel Hoping* para maior confiabilidade na transmissão.

Em TSCH a sincronização de motes é feita com base em *slotframes*, que são grupos de slots que se repetem com o tempo. Cada mote segue uma programação que diz o que ele pode fazer em cada slot, em cada slot ele pode transmitir ou receber ou dormir. Em um slot que ele está dormindo o rádio não é ligado, em um slot ativo ele encontra na programação para qual vizinho ele vai transmitir ou receber e qual canal utilizar.

Como pode ser visto na Figura 3 abaixo, um único slot possui uma limitação de tamanho, onde é possível enviar o tamanho máximo de pacote mais o espaço para receber a mensagem de *Acknologement* (ACK) do receptor, para indicar que a mensagem foi recebida com sucesso. O tamanho do slot pode ser configurado dependendo da aplicação, porém um valor possível sugerido é 10 ms. [3]

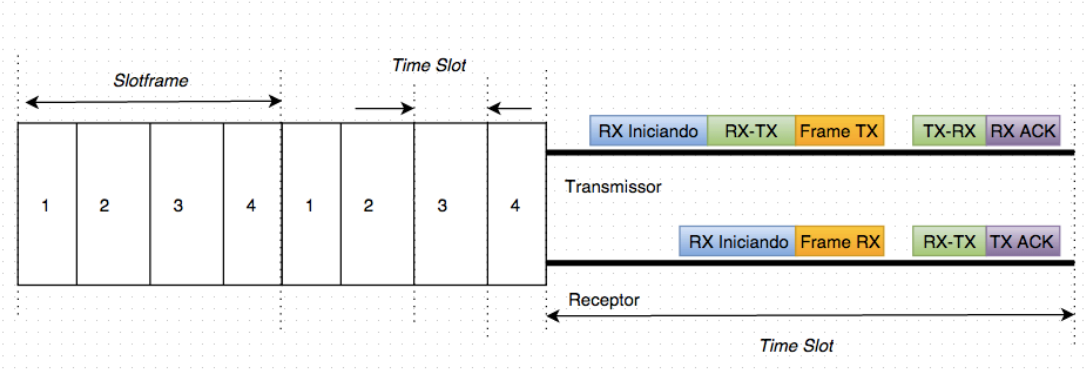

Figure 3 - Estrutura do slot frame e sequências de ações em um time slot - adaptado de [3]

Quando uma camada superior gera um pacote e envia para a camada MAC, esse pacote é armazenado em uma fila. A cada slot de transmissão a camada MAC checa se possui um pacote na fila para o destino do slot. Em caso negativo o protocolo volta para o modo dormindo sem ativar o rádio. No caso positivo a camada envia o pacote e aguarda pela mensagem de ACK. Quando essa mensagem chega o pacote é removido da fila; caso a mensagem não chegue, o pacote volta para a fila e o número de tentativas de transmissão do pacote é acrescido. Este número tem um limite de retransmissões para evitar que o pacote fique sempre sendo reenviado.

A cada slot de recepção o mote liga o seu rádio, caso ele receba um pacote em que ele é o destinatário, ele envia o ACK, move o pacote para a camada superior e desliga o rádio. Caso não receba nada dentro de um período de tempo ele volta o rádio para a função de dormir.

Sincronização entre os dispositivos é necessária para manter a conexão entre nós vizinhos em uma rede baseada em *slotframes*. Duas técnicas são utilizadas para sincronização da rede, Sincronização baseada em ACK e baseada em frame.

A sincronização baseada em ACK utiliza a diferença de tempo esperado para o recebimento de um frame e o recebimento real do mesmo e enviando essa informação para o mote que enviou o pacote para que ele sincronize seu clock com o do receptor. Já na sincronização baseada em frames o receptor realiza o mesmo cálculo de diferença entre tempo estimado e tempo real e utiliza essa informação pra sincronizar seu clock com o do mote que transmitiu a informação.

O TSCH do IEEE 802.15.4 adiciona *channel hopping* para o acesso baseado em time slots. Este *channel hopping* implica o uso de diferentes frequências para mitigar os efeitos de interferência co-canal e desvanecimento. O uso de frequências diferentes aumenta também a capacidade de transmissão da rede, pois vários motes podem transmitir ao mesmo tempo utilizando canais diferentes.

ZigBee é uma especificação de rede sem fio projetada para o uso em dispositivos com baixa capacidade tanto de processamento quanto de comunicação. É utilizada a especificação IEEE 802.15.4 como camada de enlace/física e as camadas superiores são especificadas pelo consórcio

ZigBee. Pode ser comparada as tecnologias Wi-Fi e Bluetooth porém com um foco maior em menor consumo de energia e tendo assim geralmente um menor alcance. Essa tecnologia vem sofrendo forte pressão das últimas versões do *Bluetooth Low Energy* que vem sendo adotado por diversos fabricantes.

ZigBee é um padrão para arquiteturas de malha de baixo custo e baixa potência, suportando nativamente as topologias em estrela e árvore e possui três tipos de dispositivos: o ZigBee coordenador, que é mais completo e tem objetivo de controlar a rede, o ZigBee roteador, que pode funcionar como receptor e também como roteador para transmitir a informação entre dois nós e o ZigBee dispositivo final, que só tem a função de se comunicar com o nó desejado, não podendo transmitir informação de outros nós.

2.2 6LoWPAN

O protocolo *IETF 6LoWPAN* ou *IETF IPv6 over Low Power WPAN [15]* é baseado no conceito de *Wireless Personal Area Network* (WPAN) ou Rede sem fio de área pessoal. WPAN é uma rede de pequeno alcance que tem como características um alto número de dispositivos, pacotes de dados de tamanho pequeno, pequena largura de banda, posição dos nós indefinida e longos períodos de inatividade entre os processos de comunicação. Uma rede de sensores ou uma rede de dispositivos IoT pode ser considerada uma WPAN.

O 6LoWPAN é uma camada de adaptação destinada a transportar pacotes IPv6 dentro de redes com frames de tamanho reduzido na camada de comunicação, como na especificação IEEE 802.15.4. Assim é necessário realizar um processamento dos frames IPv6 usando de compressão e supressão de dados redundantes dos pacotes IPv6. A compressão do cabeçalho IP é baseada na premissa que vários pacotes IP vão possuir os mesmos cabeçalhos dentro de uma WPAN, o que permite uma menor repetição no envio de informação. Algumas características do IPv6 ainda não são suportadas pelo 6LoWPAN como o suporte para mecanismos de segurança e integridade.[3]

O protocolo é utilizado na conexão de uma rede IoT com uma rede convencional na figura de um roteador de borda 6LoWPAN. Esse roteador trata os pacotes chegando e saindo da rede IoT e não possui informação relacionada a aplicação da rede, o que diminui a necessidade de poder de processamento e consequentemente requer um menor consumo de energia.

Considerando que a camada MAC do IEE 802.15.4 pode limitar o frame para 81 bytes, um pacote IPv6 comum com 40 bytes de cabeçalho IPv6, 20 bytes de TCP e 8 bytes de UDP se torna incompatível. Com isso a compressão do cabeçalho IPv6 no 6LoWPAN utiliza os campos em comum dos cabeçalhos IPv6 e UDP e também os campos que podem ser extraídos das camadas inferiores [14].

Redes 6LoWPAN utilizam um processo de auto organização sem estado, onde os dispositivos dentro da rede geram seus próprios endereços IPv6. Esse processo também é utilizado na compressão do cabeçalho IPv6, o que faz com que os protocolos de roteamento utilizados não influenciem na compressão do cabeçalho.

Na Figura 4 abaixo, três exemplos de comunicação são apresentados com suas diferenças de compressão.

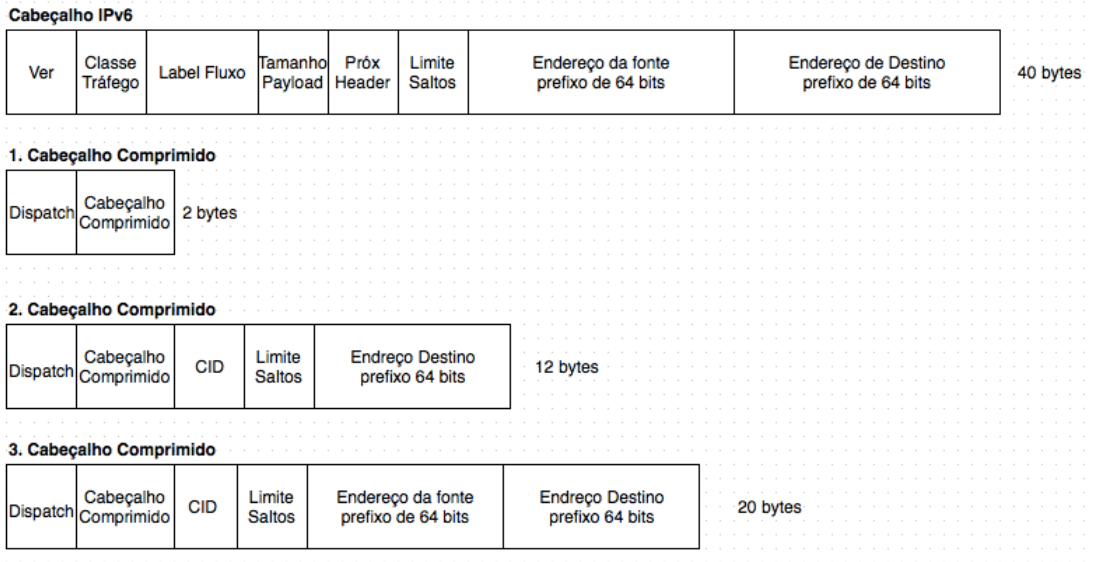

Figure 4 - Três exemplos de comunicação e cabeçalho IPv6 - adaptado de [14]

Primeiramente é possível ver o cabeçalho IPv6 completo com seus 40 bytes. No exemplos número 1, o cabeçalho de somente 2 bytes é possível

por se tratar de dois dispositivos dentro da mesma rede 6LoWPAN, utilizando o endereço local da camada de enlace.

No exemplo 2 um cabeçalho de 12 bytes é utilizado pois a comunicação é destinada a um dispositivo fora da rede 6LoWPAN e o prefixo da rede externa já é conhecido. Por fim no exemplo 3 o cabeçalho comprimido possui 20 bytes e também é para comunicação com um dispositivo em uma rede externa à 6LoWPAN, porém o prefixo da rede externa não é conhecido.

No pior caso de compressão uma redução de 50% é atingida e no melhor chega a 95%. No primeiro caso a comunicação só é possível entre dois nós vizinhos da rede, o que pode impossibilitar o envio de dados de aplicação em um cenário com múltiplos saltos, porém é extremamente importante para o protocolo de roteamento [14].

Para a transmissão de frames IPv6 sobre IEEE 802.15.4 é necessário fragmentar os frames recebidos em vários segmentos menores. Para isso é necessário adicionar dados extras nos cabeçalhos para realizar a remontagem dos frames. Na remontagem esses dados adicionais são removidos e os pacotes são retornados ao formato inicial.

2.3 RPL

RPL [16. é um protocolo de roteamento IPv6 para redes consideradas de baixa potência e com muitas perdas ou *Low-Power and Lossy Networks* (LLNs). Esse protocolo suporta uma variedade de camadas de rede, incluindo as que possuem recursos limitados e com grandes perdas atreladas a dispositivos também limitados, o que é comum em aplicações IoT.

RPL pode rotear diferentes tipos de tráfego e a sinalização utilizada entre os nós da rede depende do fluxo de dados da aplicação em que ele é utilizado. Para cada nó da rede o protocolo gera um *Destination Oriented Directed Acyclic Graph* (DODAG) ou grafo direcionado acíclico orientado a destino que é calculado com base no custo do link com relação a distância entre os nós, atributos do nó como capacidade restante da bateria e uma função objetivo, que mapeia os requerimentos para otimização da potência utilizada [3].

Em uma configuração padrão do RPL os nós são conectados por meio de múltiplos saltos a alguns nós raízes da rede. Esses nós são responsáveis por coleta de dados e coordenação da rede.

DODAG *Information Option* (DIO), ou mensagens DIO são trocadas periodicamente entre todos os membros da rede e contém informação como o Rank atual, criado com base nos custos de link, informações da função objetivo entre outras. Essas mensagens são utilizadas para a formação da topologia da rede onde um nó recebendo essa informação se junta a um DODAG.

A formação da topologia é baseada em um ranking de métricas e tem o valor decrescendo do DODAG em direção ao destino final. A Figura 5 abaixo representa três DODAGs distintos. É importante notar que quando um nó deseja enviar informação para um nó que não é o nó raiz do DODAG, esse pacote vai percorrer o caminho subindo até o DODAG e depois descer até o seu nó de destino.

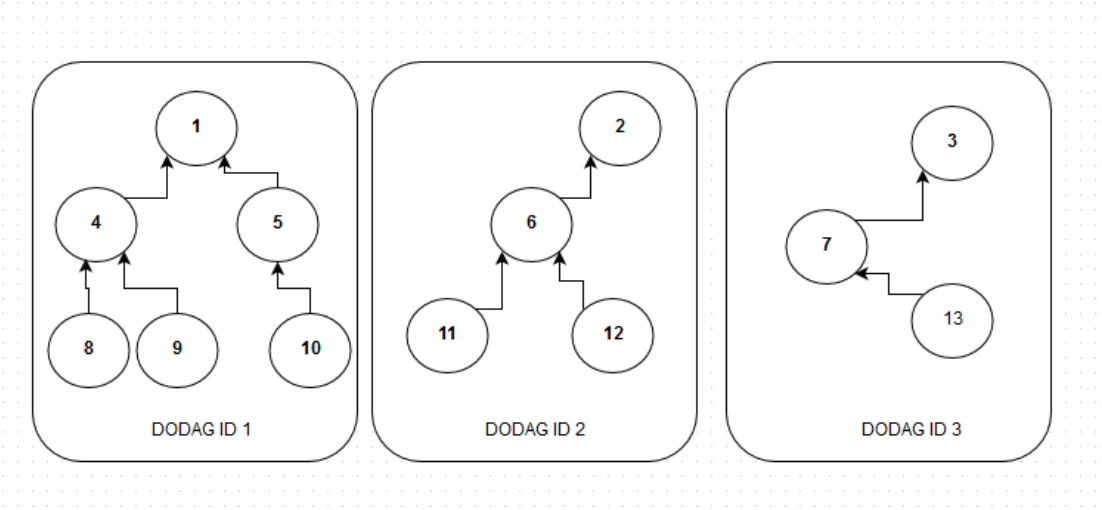

Figure 5 - Particionamento de Topologia RPL adaptado de [19]

O RPL utiliza o status dos nós da rede para cálculo de métricas para um enlace. O status pode conter dados como utilização de CPU, memória disponível e bateria, isso faz com que o RPL considere características importantes para aplicações IoT para determinação das rotas. Duas métricas importantes utilizadas por ele são o consumo de energia em um nó e o número esperado de transmissões.

A informação de consumo de energia em um nó é utilizada pelo RPL quando qualquer nó deseja enviar uma mensagem. O transmissor considera o nível de energia nos seus nós vizinhos para escolher o melhor caminho para enviar a mensagem. Caso o nível de energia de um nó esteja muito baixo, ele pode ser preterido mesmo que seja o nó que permita o melhor caminho. O número esperado de transmissões é o número de transmissões necessário para que o pacote chegue ao destino final. Em redes de sensores o rádio consome muita energia de um nó. Então esse parâmetro tem um peso considerável na decisão do caminho a ser tomado, por exemplo, um caminho com o nível de sinal maior pode ser preterido caso tenha várias transmissões até o pacote atingir o seu destino.

Na Figura 6 abaixo se pode observar um caso em que o nó 2 deve ser acionado constantemente por ser o nó mais próximo do nó raiz do DODAG (Nó 1). Assim caso o nível de energia do nó 2 esteja ficando baixo, o RPL deve começar a sugerir o nó 7 como possível caminho alternativo. É necessário um período de transição para que o RPL sugira a rota através do nó 7, assim caso o nó 2 esteja indisponível, alguns pacotes podem ser perdidos.

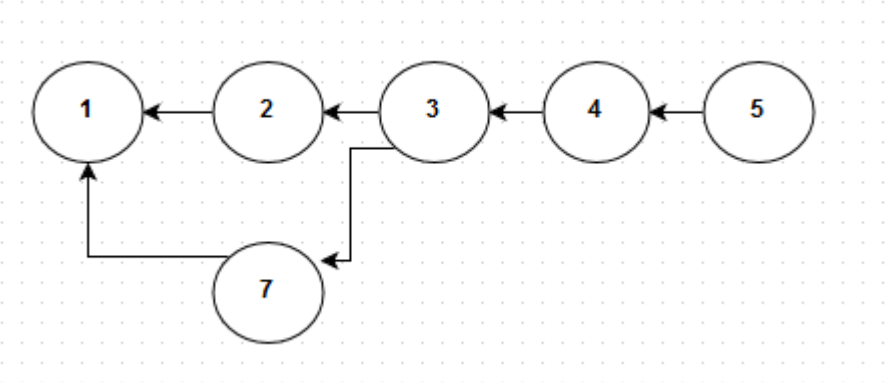

Figure 6 - Topologia em Cadeia – adaptado de [19]

A função objetivo no RPL utiliza as métricas calculadas pelos nós para gerar um ranking que modela a distância do nó destino até o nó raiz mais próximo. Assim ela permite que um nó faça a escolha do DODAG que vai fornecer um melhor caminho para troca de mensagens.

2.4 CoAP

Constrained Application Protocol (CoAP) [17] é um protocolo na camada de aplicação que tem como objetivo principal prover interoperabilidade entre as aplicações IoT e o protocolo HTTP. CoAP é um protocolo assíncrono de requisições e respostas com uma arquitetura cliente-servidor um pouco diferenciada, pois os nós não tem os papeis fixos, assim os dois pontos na comunicação atuam como clientes e servidores.

A arquitetura CoAP representada na Figura 7 mostra como o protocolo é divido em duas camadas, as camadas de Requisições e Repostas e a camada de mensagens. Novamente em uma analogia com o modelo OSI o protocolo CoAP estaria situado entre a camada de transporte e aplicação. Em redes IoT geralmente é utilizado o protocolo *User Datagram Protocol* (UDP) para a camada de transporte. UDP é mais simples e utiliza mensagens com tamanho menor que o TCP, o que o faz com que ele seja mais indicado para aplicações IoT.

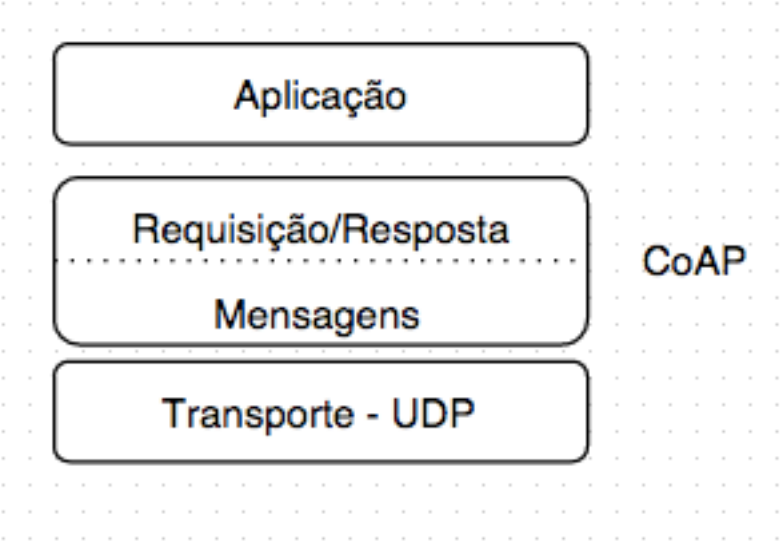

Figure 7 - Arquitetura CoAP - adaptado de [3]

A camada de mensagens é responsável pela garantia de entrega de pacotes (transferência confiável), pelos números de sequência dos pacotes e pela troca de mensagens pelo protocolo da camada de transporte UDP. O formato das mensagens de requisições e de respostas é o mesmo e a

camada de mensagens controla as mensagens com um ID único de cada mensagem que previne mensagens duplicadas e garante que uma resposta seja enviada para a requisição correta.

A camada de requisição e repostas é responsável por controlar a semântica das mensagens enviadas e provê suporte a multicast. Esta camada controla também o processo de retransmissão de mensagens para garantia de confiabilidade e o processo de detecção de duplicidade, para evitar que mensagens duplicadas sejam respondidas. [3]

O frame de uma mensagem CoAP tem um cabeçalho de tamanho fixo, como pode ser visto na Figura 8 abaixo.

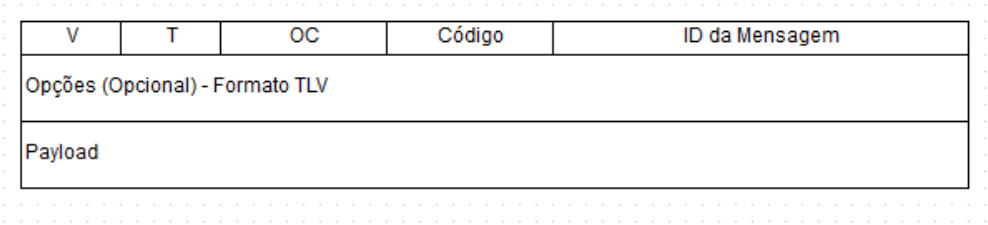

Figure 8 - Formato de frame CoAP - adaptado de [3]

Os campos presentes no frame CoAP são:

- Ver(Versão) – Com 2 bits de comprimento esse campo indica a versão do CoAP implementada.
- T (Tipo) – 2 bits que indicam o tipo de mensagem sendo enviada.
- OC (Countador Opcional) – 4 bits que indicam a quantidade de opções que estão sendo utilizadas na mensagem.
- Código – 8 bits que indicam se a mensagem é uma requisição ou uma resposta.
- ID da Mensagem – 16 bits que formam um ID único de uma mensagem para ser confirmado com as mensagens recebidas e evitar duplicidade de mensagens.
- *Payload* - O conteúdo real da mensagem enviada.

Os métodos básicos da utilização do protocolo são:

- GET – Método utilizado para requisitar e receber informação de um serviço disponível no servidor CoAP.
- POST – Requisita um processamento no servidor CoAP, geralmente resulta na atualização de um recurso.
- PUT – Requisita a criação ou atualização de um recurso do servidor CoAP
- DELETE – Esse método requisita que um recurso seja deletado do servidor CoAP.

Como no HTTP, o protocolo CoAP possuiu códigos de identificação de repostas. Códigos 2. XX são códigos que indicam sucesso na transação. Códigos 4. XX são retornados quando o cliente retornou algum erro na requisição. Códigos 5. XX são retornados quando o servidor CoAP não conseguem tratar a informação e exibe um erro interno.

2.1 Contiki OS

Contiki OS [6] é um sistema operacional Open Source desenvolvido especialmente para dispositivos utilizados em aplicações IoT, assim ele tem suporte para os padrões atuais da internet como IPv4 e IPv6 juntamente com os novos padrões para IoT. Ele possui suporte para diversas plataformas de hardware das empresas mais conhecidas pela fabricação de dispositivos IoT como a Texas Instruments, ST Microeletronics, Zolertia, entre outras.

O desenvolvimento em Contiki é facilitado por existir um ambiente de desenvolvimento totalmente preparado com uma máquina virtual Ubuntu, o que faz com que nenhuma configuração de ambiente seja necessária, e um simulador de redes chamado Cooja, que possibilita testes antes de utilizar o código em hardware. Por fim todo o desenvolvimento é realizado em C.

Contiki OS possui algumas características que facilitam o desenvolvimento de aplicações IoT, sempre levando em consideração os desafios enfrentados por dispositivos e aplicações IoT. Algumas dessas características são:

- Alocação de Memória – O Contiki possui ferramentas para alocação de memória de maneira otimizada como

memoryblockallocation, *managedmemoryallocator* e também possui o alocador de memória padrão do C. [6].

- Suporte a Redes IP – Contiki suporta todos os protocolos utilizados por redes IP como UDP, TCP e HTTP e também os novos protocolos como 6LowPan, RPL e CoAP. Toda a pilha de protocolos IP implementados no SO foram desenvolvidos pela Cisco. [6]
- Gerenciamento de Energia – O SO foi desenvolvido para ser operado com uso mínimo de energia e possui mecanismos para acompanhamento do gasto de energia das aplicações e estimativa de duração de bateria. [6]
- Download de Código Dinâmico – Contiki possui a capacidade que os programas a serem executados nele sejam carregados enquanto o mote já está em execução, não sendo necessário fazer o download do SO completo a cada nova alteração de código. Essa característica é importante para aplicações IoT onde o software de vários sensores precisa ser atualizado ao mesmo tempo e é necessário transmitir esse software via rede.
- Sistema Híbrido – O SO possui um Kernel direcionado a eventos pois assim não é necessário a alocação de uma pilha de memória para cada thread. Porém algumas aplicações como algoritmos criptográficos podem levar um tempo elevado para ser processadas nesse tipo de SO, então o Contiki também implementa um multi-thread preemptivo como uma biblioteca de aplicação que pode ser utilizado caso uma aplicação necessite.

Contiki é composto pelo Kernel, bibliotecas, carregador de programas e seus processos. Processos podem ser tanto uma aplicação carregada ou um serviço do Kernel. Um serviço é a implementação de uma funcionalidade do SO que pode ser utilizada por mais de uma aplicação. Tanto processos como serviços podem ser alterados em tempo de execução. A comunicação entre os processos é sempre realizada através do Kernel.

Uma camada de abstração de hardware não está implementada, porém é possível a comunicação entre uma aplicação e os drivers de um dispositivo diretamente.

O sistema operacional é divido em duas partes, o Core e os programas carregados, conforme a Figura 9 abaixo. [12]

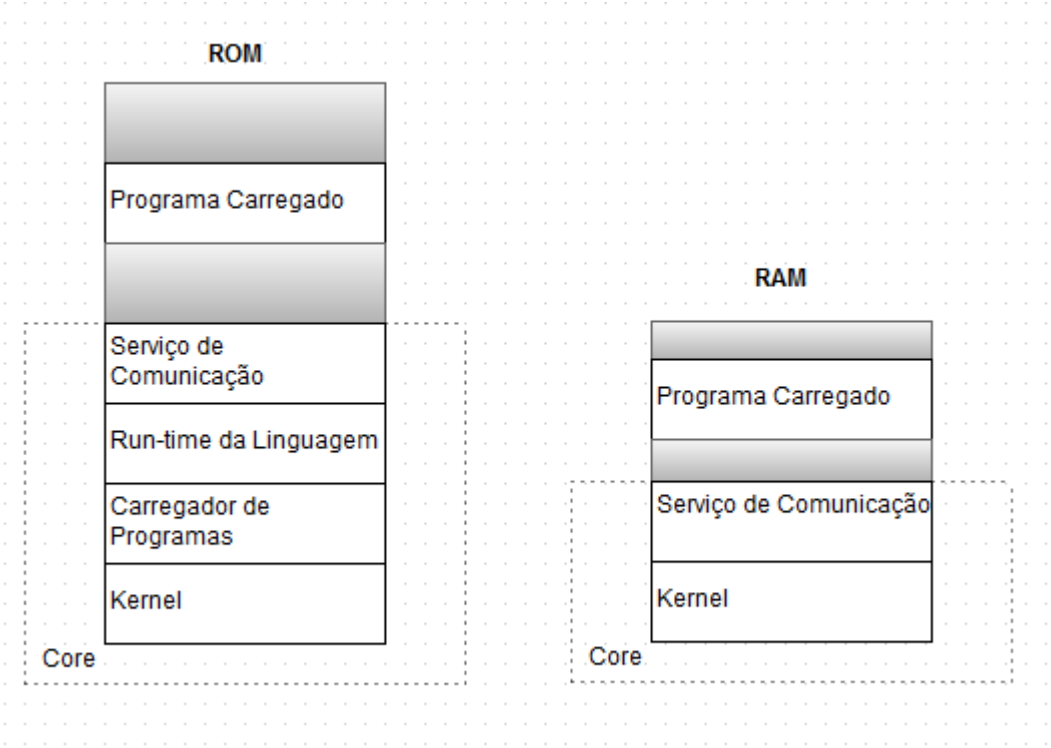

Figure 9 - Estrutura do sistema operacional Contiki - adaptado de [12]

Essa divisão é realizada durante a compilação do sistema operacional e tipicamente o Core possui o Kernel, o carregador de programas, as partes mais comuns da linguagem de tempo de execução, as bibliotecas de suporte e o serviço de comunicação. O Core é armazenado como uma imagem binária no dispositivo e geralmente não é modificado depois que o SO é carregado no mote. O Carregador de programas é responsável por armazenar as aplicações no sistema, e pode utilizar a camada de comunicação ou armazenamento externo do mote.

O Kernel do Contiki envia eventos para os processos e periodicamente chama alguns processos. Não possui mecanismos de economia de energia já implementados, porém permite que as aplicações implementem tais mecanismos. Ele disponibiliza a fila de eventos e o tamanho de cada evento

para que, caso necessário, uma aplicação possa decidir desligar o sistema ou processador.

Serviços no Contiki são os processos que podem ser usados por várias aplicações e geralmente são os protocolos de comunicação, drivers de dispositivos e algoritmos de tratamento de dados de sensores. A camada de serviços é quem faz o gerenciamento dos mesmos e mantém um controle de todos os serviços disponíveis.

3. Implementação

O desenvolvimento do projeto foi baseado na implementação da rede de sensores com os devidos protocolos e tecnologias descritas no Capítulo 2, a definição da arquitetura da rede e desenvolvimento de algumas aplicações práticas para exemplificar a usabilidade dessa rede. Foram utilizados três motes rodando o sistema operacional Contiki, dois deles atuam como sensores (servidores CoAP) e um atua como roteador de borda da rede. Por fim um computador foi utilizado como cliente da rede (Cliente CoAP). O hardware e software utilizados para o desenvolvimento, a arquitetura da rede e as aplicações propostas estão descritos abaixo.

3.1 Hardware e Software

O Hardware utilizado no projeto são três motes do modelo MTM-CM500-MSP que são fabricados pela empresa Advanticsys que foram desenvolvidos utilizando a plataforma *open-source* TelosB/Tmote Sky. Essa plataforma estabelece um padrão de criação de mote e características necessárias para um dispositivo de comunicação sem fio de baixa potência.

O Mote é compatível com sistemas operacionais como o TinyOS e Contiki e possui um chip RF da Texas Instruments, atuando na faixa de 2,4 a 2,485GHz com uma sensitividade de recepção de -95dBm com um alcance de aproximadamente 120m em lugares abertos e de 20 a 30m em lugares fechados. O mote é compatível com o padrão IEEE 802.15.4, utilizando o padrão ZigBee. O mote possui dois sensores de luminosidade e um sensor de temperatura e umidade e trabalha com 3V de alimentação (Duas baterias AA). A imagem do mote pode ser vista na Figura 10 e a Tabela 1 possui todas as especificações técnicas do mote. [5]

Figure 10 - Mote MTM-CM5000- MSP

Table 1 - Especificações Técnicas do Mote

Item	Especificação	Descrição
Processador		
Modelo	Texas Instruments® MSP430F1611	Texas Instruments® MSP430 Family
Memória	48KB	Flash de Programa
	10KB	RAM
	1MB	Flash externa (ST® M25P80)
ADC	12bit resolution	8 canais
Interfaces	UART, SPI, I2C	Interfaces Serial
	USB	Interfaces Serial Externas (FTI® FT232BM)
Rádio		
Chip RF	Texas Instruments® CC2420	Módulo Wireless IEEE 802.15.4 2.4GHz
Banda de Frequência	2.4GHz ~ 2.485GHz	Compatível IEEE 802.15.4
Sensibilidade	-95dBm	Sensibilidade de Recepção
Taxa de Transferência	250Kbps	Compatível com IEEE 802.15.4
Potência RF	-25dBm ~ 0dBm	Configurável via Software
Alcance	~120m(outdoor), 20~30m(indoor)	Alcance pode ser aumentado com antena SMA opcional
Corrente	RX: 18.8mA TX: 17.4mA Sleep mode: 1uA	Modos RF de baixa potência reduzem o consumo
Energia RF	2.1V ~ 3.6V	Potência de entrada CC2420
	Antena Dipolo / Antena PCB	Conector SMA adicional disponível pra antena extra
Sensores		
Luminosidade 1	Hamamatsu® S1087 Series	Faixa Visível (Comprimento de onda de pico de sensibilidade - 560 nm)
Luminosidade 2	Hamamatsu® S1087 Series	Faixa visível & Faixa infravermelho (Comprimento de onda de pico de sensibilidade - 960 nm)
Temperatura e	Sensirion® SHT11	Faixa de Temperatura : -40 ~ 123.8 °C

Umidade		Resolução da Temperatura: : ± 0.01(típico)
		Precisão da Temperatura ± 0.4 °C (típico)
		Faixa de Umidade: 0 ~ 100% RH
		Resolução de Umidade 0.05 (típico)
		Precisão de Umidade: ± 3 %RH (típico)
Características Físicas		
Dimensões	81.90mm x 32.50mm x 6.55mm	Incluindo conector USB
Peso	17.7g	Sem Baterias
Energia	3V (2xAA Baterias)	Regulador de Potência MICREL® MIC520

Como mencionado anteriormente Contiki OS é o sistema operacional utilizado no projeto. A implementação do código fonte no mote pode ser algo complexo, porém existe um ambiente de desenvolvimento Contiki já pré configurado chamado Instant Contiki.

Instant Contiki é um ambiente de desenvolvimento Contiki completo preparado em uma imagem de máquina virtual de Ubuntu com toda a configuração necessária já realizada. Possui uma ferramenta chamada Cooja que permite a simulação de redes de sensores em que é possível configurar motes de diversas plataformas e executar códigos fontes de aplicações desejadas [4]. Após os testes das aplicações no Cooja é possível então realizar o upload dos códigos fontes em hardware para então implementar as aplicações práticas.

Uma das dificuldades encontradas no projeto foi relacionado ao ambiente Instant Contiki, alguns bugs já conhecidos e não muito bem documentados e alguns comportamentos não esperados atrasaram o desenvolvimento.

A aplicação cliente utilizada nos exemplos de implementação de redes é um computador rodando a extensão Copper do Firefox. Copper implementa um cliente para servidores CoAP, sendo possível acessar os serviços desses servidores através do navegador. Esse cliente acessa os serviços oferecidos pelos sensores que são considerados servidores nessa estrutura. Uma representação da interface do Copper pode ser visto na Figura 11 [10].

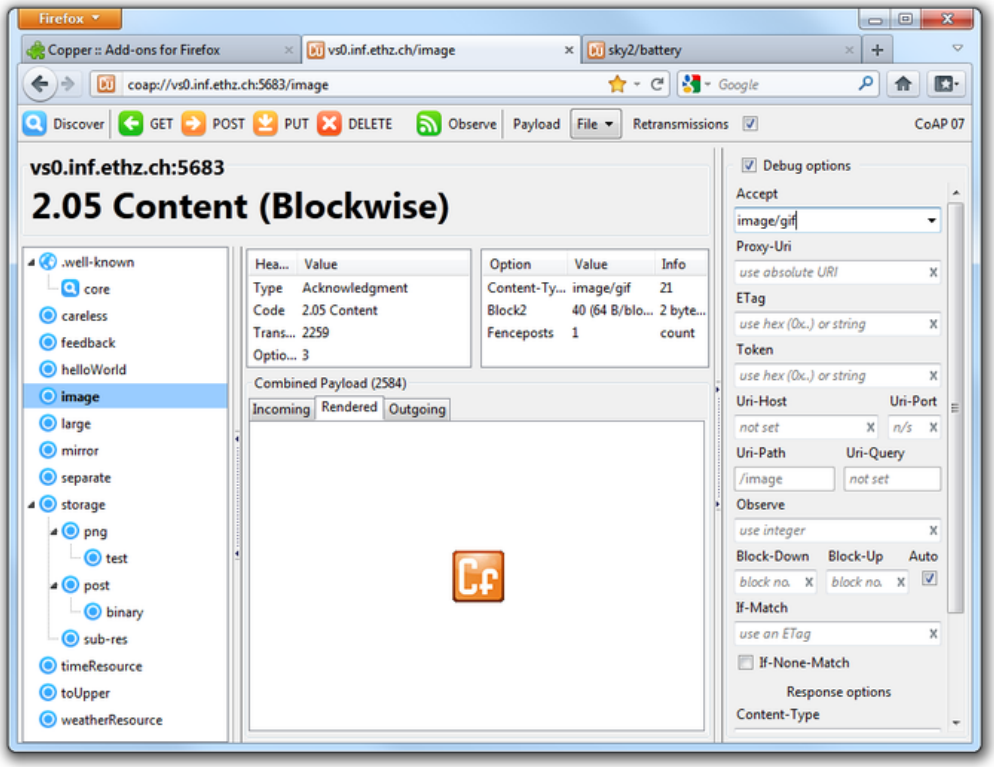

Figure 11 - Interface da extensão Copper no Firefox

3.2 Arquitetura da Rede

A arquitetura da rede foi definida levando em consideração o hardware disponível para implementação e as aplicações práticas.

Assim a estrutura da arquitetura da rede utilizada pode ser observada na Figura 12 com o sentido de comunicação e todos os integrantes da rede. Podemos ver na arquitetura os dois sensores na parte de baixo, se comunicando diretamente com o roteador de borda e o roteador de borda se comunicando com o cliente da rede.

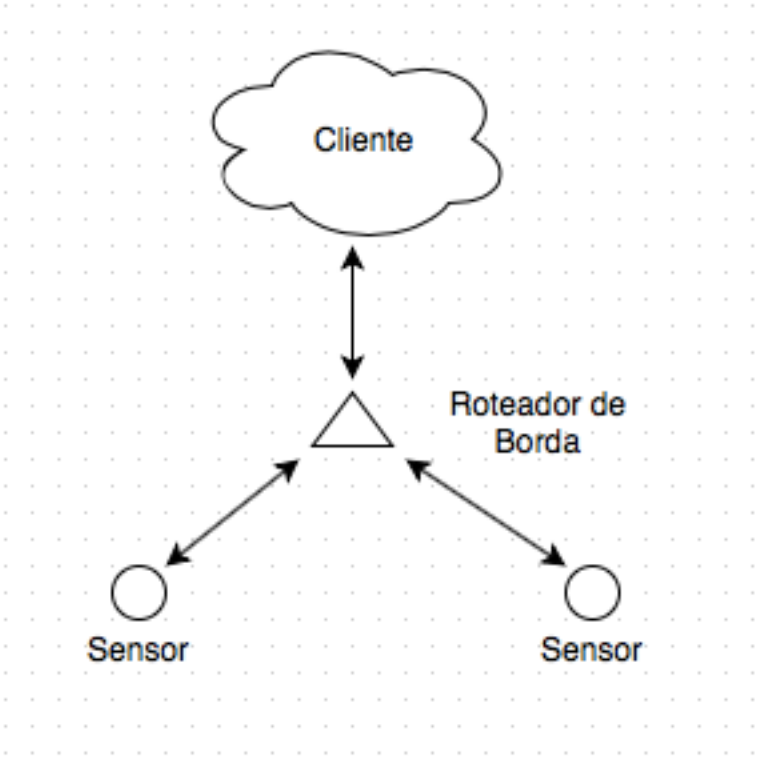

Figure 12 - Arquitetura da Rede

Com isso foram definidos os três componentes principais da rede:

Sensores – Os sensores da redes foram implementados como servidores CoAP. Esses servidores possuem serviços que podem ser acessados pelos clientes. Os serviços no projeto variam de acordo com a aplicação prática implementada, mas são em sua maioria dados coletados pelos sensores dos motes como luminosidade, temperatura e umidade.

O desenvolvimento dos servidores CoAP se baseou no modelo já implementado como exemplo no Contiki na pasta contiki/examples/er-rest-server. Assim os sensores disponibilizam seus serviços e ficam aguardando por requisições de aplicações clientes. Quando uma requisição chega ao sensor, ele processa essa requisição e devolve com a resposta equivalente e volta para o modo de repouso, onde aguarda uma próxima requisição.

Esse código foi utilizado em todas as aplicações práticas, pois foi desenvolvido para que os serviços oferecidos fossem configuráveis via código, sendo necessário somente ativar ou desativar os serviços.

Os serviços disponíveis nessa implementação são:

- Hello World – Serviço simples de leitura de informação em forma de texto armazenado internamente no mote.
- Bateria – Serviço que disponibiliza a carga atual da bateria do sensor.
- Temperatura – Serviço que fornece a medição de temperatura fornecida pelo sensor SHT11; o dado retornado pelo sensor é tratado pelos cálculos disponíveis nos anexos.
- Umidade - Serviço que fornece a medição de umidade fornecida pelo sensor SHT11; o dado retornado pelo sensor é tratado pelos cálculos disponíveis nos anexos.
- Luminosidade - Serviço que fornece a medição da luminosidade em Lux fornecida pelo sensor de luminosidade; o dado retornado pelo sensor é tratado pelos cálculos disponíveis em [13].
- Acender LED – Serviço simples que envia informação para que o sensor acenda o seu LED vermelho, sendo útil para teste de comunicação com um sensor específico em uma rede com múltiplos sensores.

Uma dificuldade encontrada no desenvolvimento da aplicação do servidor CoAP foi a limitação de memória do mote. Devido a limitação de memória não foi possível carregar o código com muitos serviços no mote. Essa limitação era observada no momento de upload do código do programa no hardware, onde o código ocupava mais que o espaço disponível.

Outra situação em que a limitação de memória impactou o desenvolvimento foi no momento do cálculo da conversão dos dados de temperatura e umidade. S eria necessário utilizar variáveis do tipo FLOAT para se obter uma precisão maior, entretanto ao utilizar essas variáveis para cálculo o programa também estourava os endereços de memória disponíveis. Uma solução paliativa foi utilizar os valores sem casas decimais no código, o que fez com que a precisão fosse menor.

O código fonte do servidor CoAP encontra-se na seção de anexos desse projeto.

Cliente CoAP – O Cliente CoAP no projeto é um computador rodando o navegador Firefox e a extensão Copper. A extensão permite a troca de informações entre um cliente e servidor CoAP e assim o cliente consegue acessar todos os serviços que o servidor disponibiliza.

Na Figura 13 é possível ver a interface do Copper acessando os serviços de um sensor da rede, com os seus respectivos serviços disponíveis. É possível observar que no topo da interface os botões de *Discover*, *Ping*, *Get*, *Post*, *Put* e *Delete* estão disponíveis para interação com os serviços do sensor e na esquerda uma lista dos serviços disponíveis está apresentada. Para acesso a interface do Copper é necessário entrar uma URL no navegador com o endereço IP externo do sensor e a porta de acesso ao protocolo.

No exemplo abaixo essa URL é *coap://[aaaa::212:7400:16bf:83a4]:5683*, onde *aaaa::212:7400:16bf:83a4* é o endereço IP externo do sensor e 5683 é a porta utilizada para acesso ao protocolo.

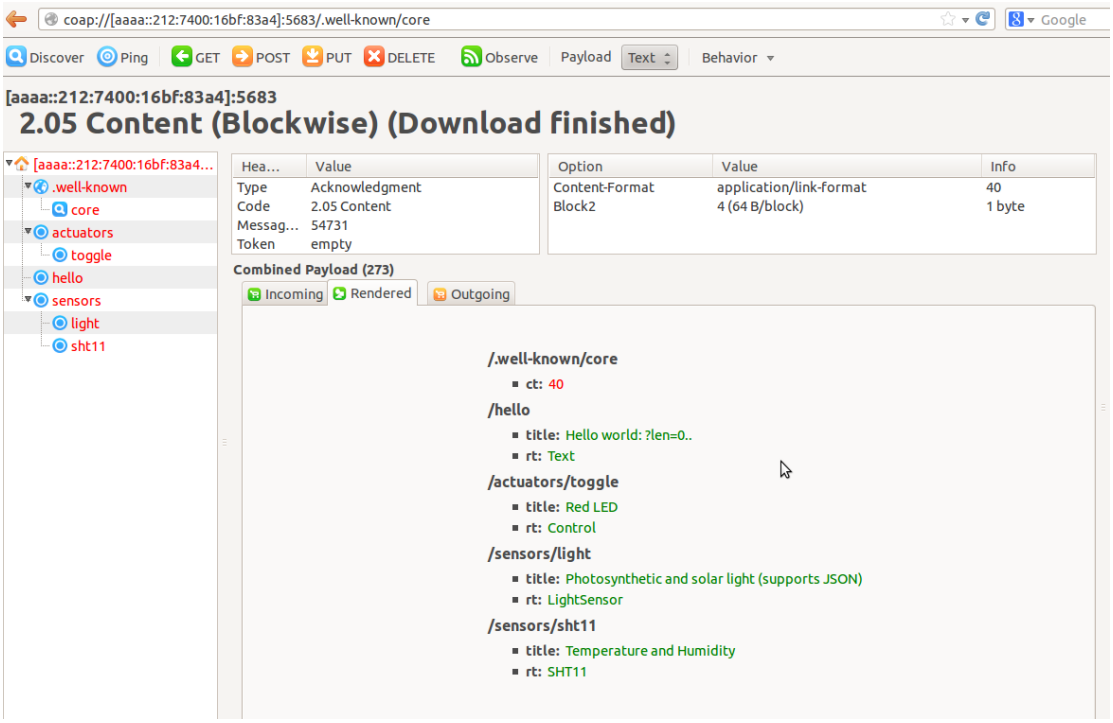

Figure 13 - Interface do Copper acessando os serviços de um sensor

Roteador de Borda – O roteador de borda RPL implementado define inicialmente o prefixo IPv6 da rede de sensores e inicia a criação da árvore de roteamento RPL da rede. Quando o roteador encontra um novo nó na rede ele atribui um IP interno para esse nó. Depois realiza o cálculo da melhor rota para esse nó até o roteador de borda.

O código fonte utilizado para o roteador encontram-se no diretório contiki/examples/ipv6/rpl-border-router na documentação do Contiki OS. O Roteador fica na “ponta“ da rede, fazendo a ligação entre a rede de sensores e a rede externa, no caso da aplicação cliente (Copper). [11]

O algoritmo do protocolo RPL vai gerar o grafo da rede automaticamente e montar os caminhos com os hops necessários. Nos testes realizados o protocolo não calculou caminhos com múltiplos saltos por questão de proximidade física dos sensores com o roteador de borda.

Como o roteador de borda nas aplicações práticas implementadas estava sempre conectado a porta do USB do computador, é possível observar a saída do algoritmo RPL ao acessar o endereço IPV6 do roteador no browser.

Na Figura 14 é possível observar a saída o roteador de borda com os endereços internos e externos dos sensores presentes na rede.

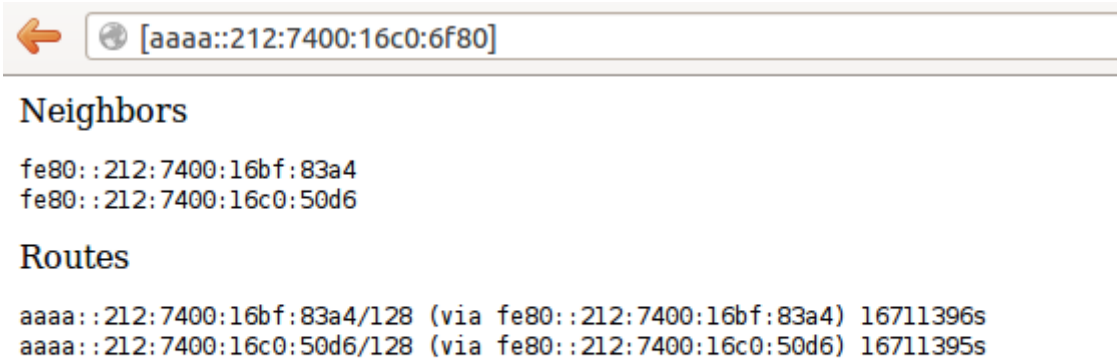

Figure 14 - Saída do Roteador de Borda com os sensores da rede

Na Figura 14 é possível observar que o endereço IP do roteador de borda é o *aaaa::212:7400:16c0:6f80* e que ele possuí dois vizinhos, com os endereços fe80::212:7400:16bf:83a4 e fe80::212:7400:16c0:50d6 . Assim ele calcula as duas rotas de acesso para esses vizinhos e fornece dois endereços IP externos para eles, *aaaa::212:7400:16bf:83a4* e *aaaa:212:7400:16c0:50d6.* Como mencionado acima pela proximidade dos sensores as duas rotas não possuem nenhum salto entre o roteador e os sensores.

4. Aplicações Exemplo

Para a demonstração das funcionalidades da implementação da rede de sensores com os componentes e arquitetura já descritos, algumas aplicações práticas simples foram realizadas. Os cenários estão descritos abaixo:

Aplicação Agrícola – Estufa/ambiente controlado

O cenário consiste em um sistema de monitoramento ambiental onde os sensores estariam distribuídos no ambiente e pela aplicação cliente seria possível monitorar as informações de temperatura e umidade.

Cada sensor da rede é um servidor que provê seus serviços (dados coletados pelos sensores) para o cliente, assim os sensores estariam distribuídos no ambiente de modo a coletar esses dados em diferentes pontos. Para demonstração cada sensor foi posicionado em uma posição diferente em um ambiente de teste, este ambiente era composto de uma sala de 3 metros de largura, 4 metros de comprimento e 2,4 metros de altura, os sensores estavam localizados a 2 metros de distância do roteador de borda. Os dados de temperatura e umidade foram coletados durante 4 horas de 30 em 30 minutos. Esses dados para os dois sensores estão presentes na Tabela 2 e 3 abaixo.

Table 2 - Dados de temperatura e umidade coletados no sensor 1.

Hora	Temperatura (C)	Umidade (%)
14:00	28	59
14:30	28	59
15:00	28	59
15:30	28	59
16:00	28	60
16:30	28	60
17:00	29	60
17:30	29	60
18:00	29	60

Table 3 - Dados de temperatura e umidade coletados no sensor 2

Hora	Temperatura (C)	Umidade (%)
14:00	27	63
14:30	27	63
15:00	28	63
15:30	28	63
16:00	28	64
16:30	28	64
17:00	29	64
17:30	29	64
18:00	29	64

Como o teste foi realizado em ambiente fechado as variações de temperatura e umidade não foram significativas, porém é possível acompanhar seu comportamento com as horas.

Um segundo teste foi realizado onde seis medições foram realizadas em cada sensor, sendo uma no inicio e outra no fim do dia durante três dias. Os dados obtidos podem ser vistos nas Tabelas 4 e 5.

Table 4 - Dados de temperatura e umidade coletados no intervalo de três dias no sensor 1.

Medidas	Temperatura (C)	Umidade (%)
1	21	72
2	26	70
3	21	68
4	27	68
5	25	79
6	26	79

Table 5 - Dados de temperatura e umidade coletados no intervalo de três dias no sensor 2

Medidas	Temperatura (C)	Umidade (%)
1	23	72
2	24	70
3	21	62
4	30	62
5	25	79
6	25	79

Com o segundo teste é possível observar uma variação maior de temperatura entre os sensores com o passar dos dias. Nos gráficos abaixo é possível observar o comportamento da temperatura e umidade nos três dias para os sensores.

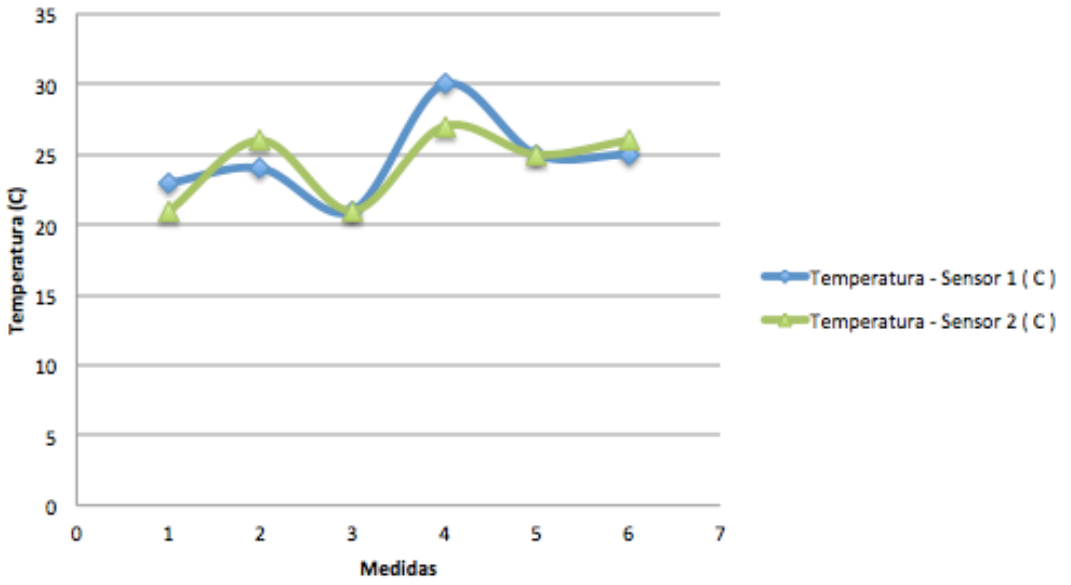

Figure 15 - Variação da temperatura e umidade no período de três dias.

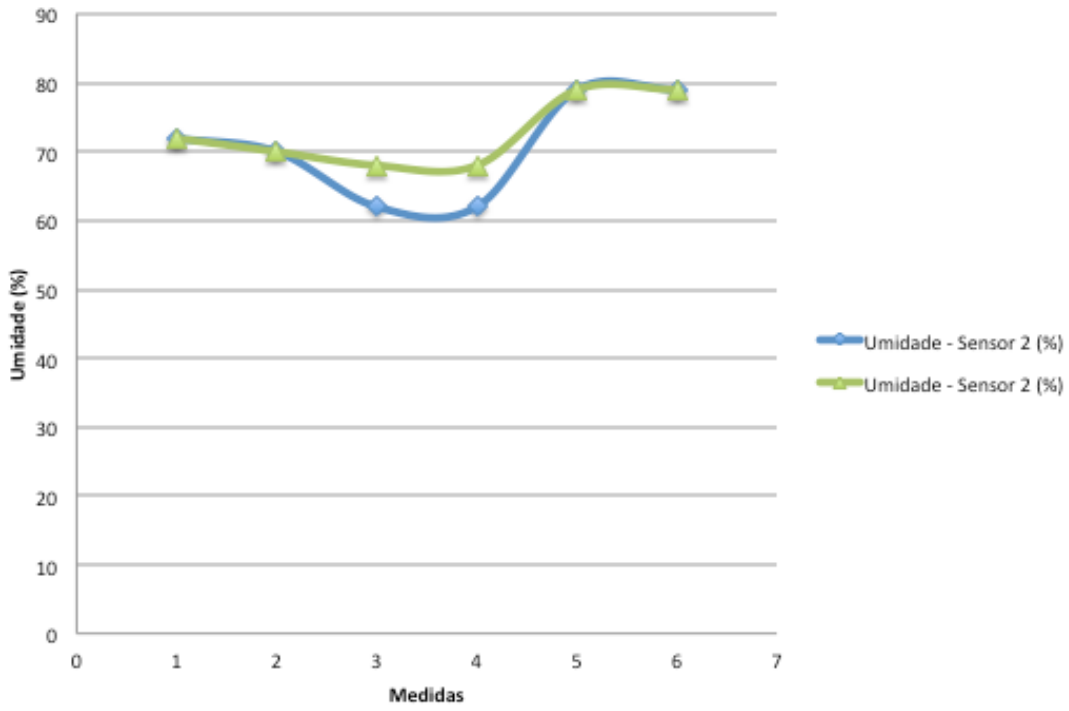

Figure 16 - Variação de temperatura e umidade no período de três dias.

A aplicação cliente consegue então acompanhar em tempo real a situação do ambiente monitorado. Com os dados obtidos pelos sensores seria possível a tomada de decisão de refrigerar ou acionar a irrigação do ambiente, sendo possível também agregar os dados dentro de um período de tempo, como temperatura ou umidade média nas últimas 24 horas ou ainda dados como relação entre temperatura e umidade para determinadas horas do dia.

Controle de Luminosidade e Climatização

Um sistema simples de controle de luminosidade e climatização de um quarto foi simulado utilizando novamente a mesma arquitetura de rede. Os dois sensores agora fornecem dados de luminosidade e temperatura de um quarto de 2 metros de largura por 2,5 metros de comprimento e 2,5 metros de altura. Durante o intervalo de 4 horas, medidos de 30 em 30 minutos. Os dados obtidos para os dois sensores estão nas Tabelas 6 e 7 abaixo:

Table 6 - Temperatura e luminosidade coletadas no intervalo de 4 horas para o sensor 1.

Hora	Temperatura (C)	Luminosidade (lux)
14:00	25	16
14:30	25	16
15:00	25	16
15:30	25	16
16:00	25	38
16:30	25	38
17:00	25	38
17:30	25	38
18:00	25	38

Table 7 - Temperatura e luminosidade coletadas no intervalo de 4 horas para o sensor 2.

Hora	Temperatura (C)	Luminosidade (lux)
14:00	26	18
14:30	26	18
15:00	26	18
15:30	26	18
16:00	26	37
16:30	25	37
17:00	25	37
17:30	25	37
18:00	25	37

É possível notar a grande diferença no valor da luminosidade as 16:00. Isso se deve ao fato que para objetivo de simulação os dois sensores terem sido colocados inicialmente em um quarto totalmente escuro e as 16:00 as luzes terem sido acessas.

Com as informações obtidas é possível então novamente tomar decisões para acionar um equipamento de ar condicionado do quarto, e podendo ainda obter a informação que a luz está acessa ou apagada no quarto em questão. Com o adendo de um atuador na rede funcionando como interruptor de luz, seria possível acionar um serviço para ligar as luzes do quarto.

Exemplo de Ativação LED

O exemplo mais simples de utilização de um serviço dos sensores é o de ativar um LED no mote através da aplicação cliente. Essa ativação pode ser útil em uma aplicação com vários sensores em que seja necessário fazer a manutenção de um sensor específico e assim ao acionar o LED no mesmo, seria possível identificá-lo.

A Figura 17 mostra o sensor real com o LED Vermelho acionado após o envio do comando pela aplicação cliente.

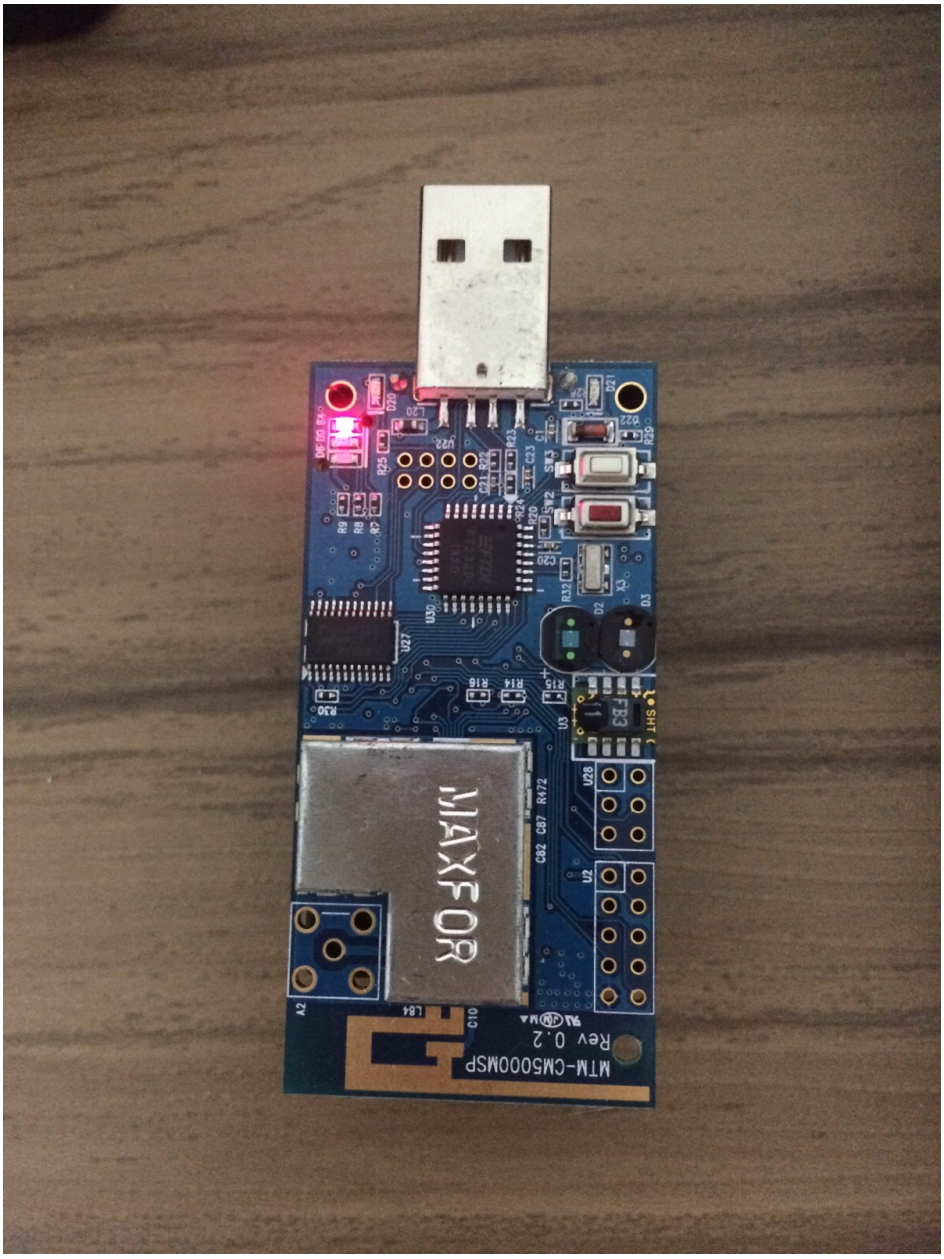

Figure 17 - Imagem do mote com LED vermelho acionado

Coleta de Informação da Bateria

Uma variável extremamente importante em redes de sensores sem fio é a duração da bateria de um sensor. Como as aplicações IoT devem funcionar com o mínimo de intervenção humana, é indispensável que a bateria dos sensores dure o máximo possível.

O sistema operacional Contiki possibilita a implementação de mecanismos para um baixo consumo de energia, porém é importante para algumas aplicações a consulta da capacidade de bateria dos sensores. Assim um serviço foi implementado de forma que com aplicação cliente seja possível obter a informação da carga da bateria de cada sensor da rede.

Para definição da carga máxima de bateria, foram adicionadas pilhas novas ao mote e anotado o valor obtido na leitura da carga da bateria, esse valor foi 2600 mV. Assim todas as medições de baterias foram feitas como uma porcentagem desse valor máximo.

A Figura 18 mostra a aplicação cliente recebendo a informação da carga da bateria de um sensor específico. No caso abaixo a carga da bateria do sensor era de 99 %, a precisão do valor da carga não possui casas decimais devido ao fato da utilização de variáveis float consumirem um espaço maior na memória.

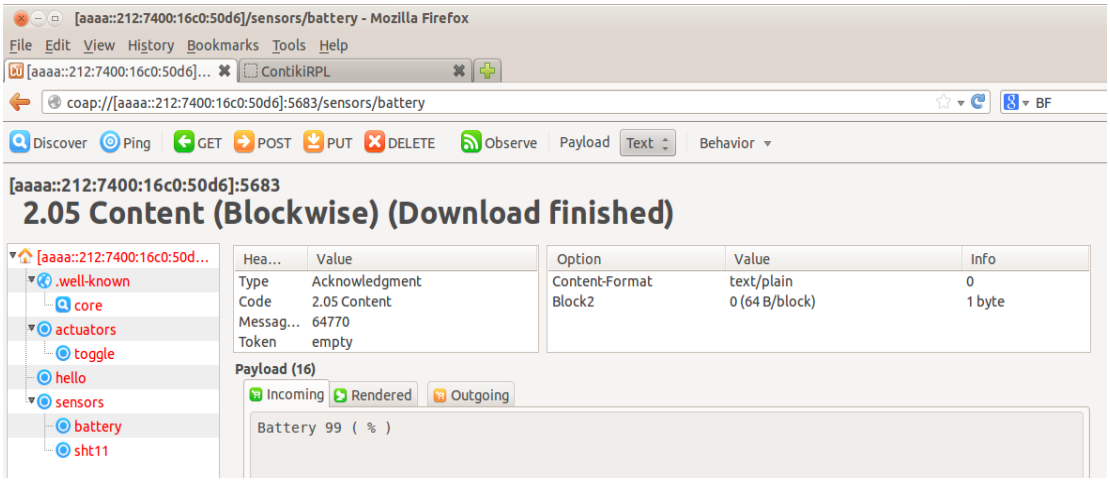

Figure 18 - Aplicação Cliente lendo a carga atual da bateria em um sensor.

É importante notar que os sensores foram utilizados por um largo período para monitoramento constante, o que resultou na bateria sendo utilizada de forma relativamente rápida. Esse comportamento é insatisfatório para uma os objetivos da rede proposta e seria necessário implementar medidas mais eficazes de economia de energia.

Leitura de Dados estáticos nos sensores

Embora a memória interna dos sensores seja extremamente limitada, é possível armazenar informação em texto nos sensores que pode ser acessada. Um serviço chamado Hello World foi implementado no sensor para essa função. Com essa função foi possível acessar pela aplicação cliente a informação armazenada no sensor, como pode ser visto na Figura 19 abaixo:

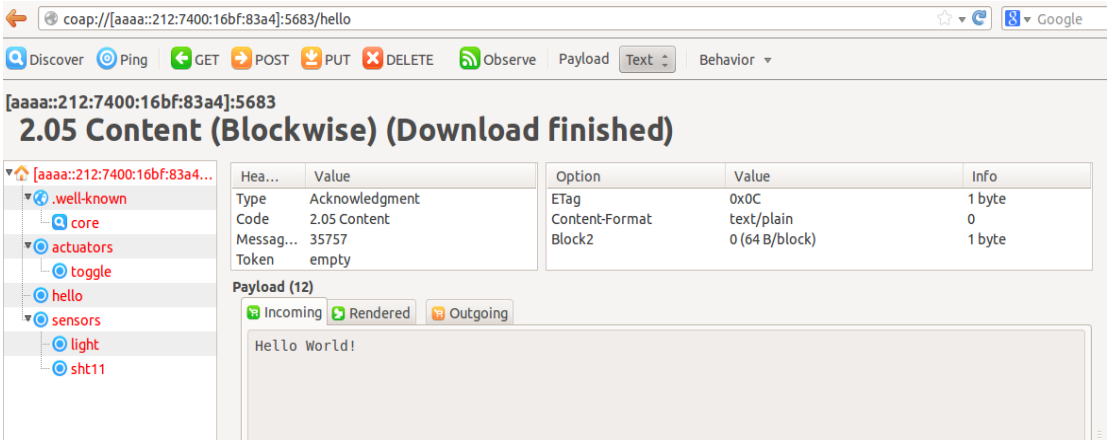

Figure 19 - Aplicação Cliente lendo dado em texto armazenado no sensor

5. Conclusões

Com o desenvolvimento do projeto foi possível realizar a implementação da rede de sensores utilizando o sistema operacional Contiki e os protocolos CoAP, 6LowPan e RPL de maneira satisfatória, sendo possível acessar por meio do cliente da rede os dados coletados pelos sensores e obtendo assim acesso aos serviços disponíveis pelas aplicações exemplo projetadas.

O sistema operacional Contiki possui várias características importantes para o desenvolvimento de aplicações IoT. Sua capacidade de carregar os programas em tempo de execução e o suporte para os diversos hardwares são grandes diferenciais.

Entretanto ele apresenta algumas dificuldades de implementação pois possui alguns bugs, embora alguns já sejam conhecidos, o fato de não possuir uma extensa documentação de fácil acesso torna o processo de desenvolvimento complexo. Isso foi uma das grandes dificuldades encontradas no desenvolvimento do projeto.

Os protocolos utilizados no projeto são totalmente compatíveis e recomendados para aplicações IoT. Sua implementação já é suportada por vários hardwares de mercado e a compatibilidade entre hardwares de diferentes plataformas e fabricantes será um ponto focal no desenvolvimento futuro de IoT.

As arquitetura e implementação propostas são compatíveis com a internet, o que não acontece com redes de sensores com protocolos não padronizados. Essa compatibilidade cria oportunidades de novas aplicações que não estão restritas ao interior da rede ou ao ambiente onde a rede está instalada.

As aplicações práticas propostas são simples porém podem resolver vários problemas atuais de maneira eficiente. A informação de temperatura e umidade em uma aplicação agropecuária para economia de água aborda um tema extremamente atual de crise hídrica.

Assim o desenvolvimento do projeto se mostrou complexo pelos desafios encontrados na implementação, entretanto conforme o desenvolvimento de aplicações IoT se intensifique ainda mais, o que o

cenário atual indica, mais documentação e informação será disponibilizada, o que deve facilitar o desenvolvimento.

Possíveis melhorias no projeto poderiam abordar aplicações práticas mais complexas e com mais detalhamento, para buscar utilizar toda a capacidade hardware e software presentes nas tecnologias e protocolos utilizados.

Hoje a aplicação cliente realiza um monitoramento passivo. Um monitoramento ativo onde a aplicação enviaria dados em certo intervalo de tempo e um tratamento desses dados seria realizado de forma automatizada pode ser implementado.

Por fim os protocolos mais utilizados em sistemas IoT são totalmente funcionais em hardwares vastamente disponíveis e de fácil acesso, podendo ser implementados com relativa facilidade e abrindo possibilidades para diversas aplicações no contexto de IoT.

Referências

[1] - L. Atzori et al., The Internet of Things: A survey, Comput. Netw. (2010).

[2] - D. Miorando et al., Internet of Things: Vision, applications and research challenges, Ad Hoc Networks 10 (2012)

[3] - M. R. Palattella et al., Standardized Protocol Stack for the Internet of (Important) Things, IEEE COMMUNICATIONS SURVEYS & TUTORIALS (2013)

[4] – Tutorial Inicial – Contiki OS – http://www.contiki-os.org/start.html - Visitado em 03/05/2015

[5] – Advanticsys – CM500 Especificações - http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html - Visitado em 03/05/2015

[6] – Contiki OS - http://www.contiki-os.org/ - Visitado em 06/05/2015

[7] – Windows 10 IoT Core - https://dev.windows.com/en-us/iot - Visitado em 05/08/2015

[8] - Projeto Brillo - Google - https://developers.google.com/brillo/ - Visitado em 05/12/2015

[9] – Cooper - https://addons.mozilla.org/en-US/firefox/addon/copper-270430/ - Visitado em 20/08/15

[10] – Cognizzi, P. I; Duquennoy, S., Hands on Contiki OS and Cooja Simulator, Internet of things and Smart Cities PH.D. School (2013)

[11] - RPL Border Router, http://anrg.usc.edu/contiki/index.php/RPL_Border_Router - Visitado em 18/07/2015

[12] - Tmote Sky datasheet, http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf

[13] – Transformation for sensor raw data to SI units, Tmote Sky, http://tinyos.stanford.edu/tinyos-wiki/index.php/Boomerang_ADC_Example

[14] – Olsson, J., 6LowPan Desmystified, Texas Instruments White Paper (2014).

[15] – Montenegro, et all., Transmission of IPv6 packets over IEEE 802.15.4 Networks, RFC 4944 (2007)

[16] – Winter, et all., RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, Internet Engineering Task Force, RFC 6550 (2012)

[17] – Shelby, et all., The Constrained Application Protocol (CoAP), Internet Engineering Task Force, RFC 7252 (2014)

[18] – IEEE Computer Society, IEEE 802.15.4 Standard for Local and Metropolitan area Networks, Part 15.4: Low-Rate Wireless Personal Area Networks (2011)

[19] - Tsvetkov, T., RPL: IPv6 Routing Protocol for Low Power and Lossy Networks, Seminar SN SS (2011)

Anexos

Tutoriais de Utilização

Execução do código fonte no Mote

1. Com o InstantContiki aberto é necessário abrir o terminal.
2. No terminal é necessário navegar até o diretório em que o código fonte que se deseja executar no mote está localizado, por exemplo:

   cdcontiki/examples/hello-world

3. Agora na pasta é necessário configurar o tipo de mote que o código será executado, como o mote utilizado no projeto é baseado na plataforma TelosB/Tmotesky é necessário configurar o target como sky, para isso deve-se digitar:

   make TARGET = sky savetarget
4. Agora é necessário compilar o código fonte com o mote como target, por exemplo como o programa do exemplo se chama hello-world deve-se digitar :

   Make hello-world.upload

5. Caso um erro ocorra na compilação do código no mote e a seguinte mensagem seja apresentada:

   make: msp430-gcc: Command not found

   É necessário executar o seguinte comando:

   #apt-file search msp430-gcc
   gcc-msp430: /usr/bin/msp430-gcc g
   cc-msp430: /usr/bin/msp430-gcc-4.6.3
   #apt-get install gcc-msp430

6. Com isso o código será compilado dentro do mote e é possível observar o processo pelo terminal, ao fim do mesmo para se obter o resultado do programa, é necessário ler a portal do serial que é por onde o mote se comunica com o ambiente do instantcontiki, para isso é necessário digitar:

   Make login

7. Na primeira execução é necessário resetar o mote para que nenhum lixo que esteja na porta serial do instantcontiki apareça, assim é necessário pressionar o botão 2 do mote, que está localizado próximo a porta usb e tem coloração vermelha.

   Após esses passos é possível observar o output do mote no terminal do instantcontiki.

Cálculos para conversão da leitura dos sensores

Para uma melhor compreensão dos dados lidos nos sensores do mote, é necessário relizar a conversão nos valores lidos. Os valores utilizados nos cálculos são baseados no datasheet dos sensores e podem também ser encontrados em [13].

As formulas utilizadas no projeto são:

- Temperatura
  - Temperatura (C) = ((0,01* (ValorSensor) – 39,60)
- Umidade
  - Umidade (%) = (((0.0405*ValorSensor) - 4) + ((-2.8 * 0.000001)*ValorSensor*ValorSensor))

- Luminosidade
    - Luminosidade (Lux) = ValorSensor * 0.4071)

Códigos

Servidor CoAP

```
/*
 * Copyright (c) 2013, Matthias Kovatsch
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE

 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL

 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS

 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)

 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT

 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY

 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF

 * SUCH DAMAGE.

 *

 * This file is part of the Contiki operating system.

 */


/**

 * \file

 *      Erbium (Er) REST Engine example (with CoAP-specific code)

 * \author

 *      Matthias Kovatsch <kovatsch@inf.ethz.ch>

 *          Modifications By Gustavo Costa <gustavoarthurcosta@gmail.com>


 */


#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "contiki.h"
```

```
#include "contiki-net.h"


/* Define which resources to include to meet memory constraints. */
#define REST_RES_HELLO 0
#define REST_RES_CHUNKS 0
#define REST_RES_SEPARATE 0
#define REST_RES_PUSHING 0
#define REST_RES_EVENT 0
#define REST_RES_LEDS 0
#define REST_RES_TOGGLE 1
#define REST_RES_LIGHT 1
#define REST_RES_TEMP 1
#define REST_RES_BATTERY 1
#define REST_RES_RADIO 1
#define REST_RES_MIRROR 0 /* causes largest code size */


#include "erbium.h"


#if defined (PLATFORM_HAS_BUTTON)
#include "dev/button-sensor.h"
#endif
#if defined (PLATFORM_HAS_LEDS)
#include "dev/leds.h"
#endif
#if defined (PLATFORM_HAS_LIGHT)
#include "dev/light-sensor.h"
```

```
#endif

#if defined (PLATFORM_HAS_BATTERY)

#include "dev/battery-sensor.h"

#endif

#if defined (PLATFORM_HAS_SHT11)

#include "dev/sht11-sensor.h"

#endif

#if defined (PLATFORM_HAS_RADIO)

#include "dev/radio-sensor.h"

#endif



/* For CoAP-specific example: not required for normal RESTful Web service. */

#if WITH_COAP == 3

#include "er-coap-03.h"

#elif WITH_COAP == 7

#include "er-coap-07.h"

#elif WITH_COAP == 12

#include "er-coap-12.h"

#elif WITH_COAP == 13

#include "er-coap-13.h"

#else

#warning "Erbium example without CoAP-specifc functionality"

#endif /* CoAP-specific example */


#define DEBUG 0

#if DEBUG

#define PRINTF(...) printf(__VA_ARGS__)

#define                                                    PRINT6ADDR(addr)
PRINTF("[%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%0
2x:%02x%02x]", ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)[2], ((uint8_t
*)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5], ((uint8_t *)addr)[6], ((uint8_t
```

```
*)addr)[7], ((uint8_t *)addr)[8], ((uint8_t *)addr)[9], ((uint8_t *)addr)[10], ((uint8_t *)addr)[11], ((uint8_t *)addr)[12], ((uint8_t *)addr)[13], ((uint8_t *)addr)[14], ((uint8_t *)addr)[15])

#define PRINTLLADDR(lladdr) PRINTF("[%02x:%02x:%02x:%02x:%02x:%02x]",(lladdr)->addr[0], (lladdr)->addr[1], (lladdr)->addr[2], (lladdr)->addr[3],(lladdr)->addr[4], (lladdr)->addr[5])

#else

#define PRINTF(...)

#define PRINT6ADDR(addr)

#define PRINTLLADDR(addr)

#endif

/*---------------------------------------------------------------------------*/

#if REST_RES_HELLO

/*

 * Resources are defined by the RESOURCE macro.

 * Signature: resource name, the RESTful methods it handles, and its URI path (omitting the leading slash).

 */

RESOURCE(helloworld, METHOD_GET, "hello", "title=\"Hello world: ?len=0..\";rt=\"Text\"");

/*

 * A handler function named [resource name]_handler must be implemented for each RESOURCE.

 * A buffer for the response payload is provided through the buffer pointer. Simple resources can ignore

 * preferred_size and offset, but must respect the REST_MAX_CHUNK_SIZE limit for the buffer.

 * If a smaller block size is requested for CoAP, the REST framework automatically splits the data.

 */

void

helloworld_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{
```

```
  const char *len = NULL;

  /* Some data that has the length up to REST_MAX_CHUNK_SIZE. For more,
see the chunk resource. */

  char const * const message = "Hello World!";

  int length = 12; /*          |<-------->| */


  /* The query string can be retrieved by rest_get_query() or parsed for its key-
value pairs. */

  if (REST.get_query_variable(request, "len", &len)) {

    length = atoi(len);

    if (length<0) length = 0;

    if (length>REST_MAX_CHUNK_SIZE) length = REST_MAX_CHUNK_SIZE;

    memcpy(buffer, message, length);

  } else {

    memcpy(buffer, message, length);

  }


  REST.set_header_content_type(response,       REST.type.TEXT_PLAIN);       /*
text/plain is the default, hence this option could be omitted. */

  REST.set_header_etag(response, (uint8_t *) &length, 1);

  REST.set_response_payload(response, buffer, length);

 }

 #endif


 /*******************************************************************************/

 #if REST_RES_MIRROR

 /* This resource mirrors the incoming request. It shows how to access the
options and how to set them for the response. */

 RESOURCE(mirror, METHOD_GET | METHOD_POST | METHOD_PUT |
METHOD_DELETE,       "debug/mirror",       "title=\"Returns       your       decoded
message\";rt=\"Debug\"");


 void
```

```
mirror_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{

  /* The ETag and Token is copied to the header. */

  uint8_t opaque[] = {0x0A, 0xBC, 0xDE};


  /* Strings are not copied, so use static string buffers or strings in .text memory (char *str = "string in .text";). */

  static char location[] = {'/','f','/','a','?','k','&','e', 0};


  /* Getter for the header option Content-Type. If the option is not set, text/plain is returned by default. */

  unsigned int content_type = REST.get_header_content_type(request);


  /* The other getters copy the value (or string/array pointer) to the given pointers and return 1 for success or the length of strings/arrays. */

  uint32_t max_age_and_size = 0;

  const char *str = NULL;

  uint32_t observe = 0;

  const uint8_t *bytes = NULL;

  uint32_t block_num = 0;

  uint8_t block_more = 0;

  uint16_t block_size = 0;

  const char *query = "";

  int len = 0;


  /* Mirror the received header options in the response payload. Unsupported getters (e.g., rest_get_header_observe() with HTTP) will return 0. */


  int strpos = 0;

  /* snprintf() counts the terminating '\0' to the size parameter.

   * The additional byte is taken care of by allocating REST_MAX_CHUNK_SIZE+1 bytes in the REST framework.
```

```
         * Add +1 to fill the complete buffer, as the payload does not need a
terminating '\0'. */

         if (content_type!=-1)

         {

          strpos += snprintf((char *)buffer, REST_MAX_CHUNK_SIZE+1, "CT %u\n",
content_type);

         }



         /* Some getters such as for ETag or Location are omitted, as these options
should not appear in a request.

          * Max-Age might appear in HTTP requests or used for special purposes in
CoAP. */

         if (strpos<=REST_MAX_CHUNK_SIZE && REST.get_header_max_age(request,
&max_age_and_size))

         {

          strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"MA %lu\n", max_age_and_size);

         }

         /* For HTTP this is the Length option, for CoAP it is the Size option. */

         if (strpos<=REST_MAX_CHUNK_SIZE && REST.get_header_length(request,
&max_age_and_size))

         {

          strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"SZ %lu\n", max_age_and_size);

         }



         if (strpos<=REST_MAX_CHUNK_SIZE && (len =
REST.get_header_host(request, &str)))

         {

          strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"UH %.*s\n", len, str);

         }



        /* CoAP-specific example: actions not required for normal RESTful Web
service. */
```

```
#if WITH_COAP > 1

  if (strpos<=REST_MAX_CHUNK_SIZE && coap_get_header_observe(request,
&observe))

  {

    strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"Ob %lu\n", observe);

  }

  if (strpos<=REST_MAX_CHUNK_SIZE && (len =
coap_get_header_token(request, &bytes)))

  {

    strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"To 0x");

    int index = 0;

    for (index = 0; index<len; ++index) {

      strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-
strpos+1, "%02X", bytes[index]);

    }

    strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"\n");

  }

  if (strpos<=REST_MAX_CHUNK_SIZE && (len =
coap_get_header_etag(request, &bytes)))

  {

    strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"ET 0x");

    int index = 0;

    for (index = 0; index<len; ++index) {

      strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-
strpos+1, "%02X", bytes[index]);

    }

    strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"\n");

  }

  if (strpos<=REST_MAX_CHUNK_SIZE && (len =
coap_get_header_uri_path(request, &str)))

  {
```

```
            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"UP ");

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"%.*s\n", len, str);

          }

         #if WITH_COAP == 3

          if          (strpos<=REST_MAX_CHUNK_SIZE          &&          (len          =
coap_get_header_location(request, &str)))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"Lo %.*s\n", len, str);

          }

          if  (strpos<=REST_MAX_CHUNK_SIZE  &&  coap_get_header_block(request,
&block_num, &block_more, &block_size, NULL)) /* This getter allows NULL pointers to
get only a subset of the block parameters. */

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"Bl %lu%s (%u)\n", block_num, block_more ? "+" : "", block_size);

          }

         #else

          if          (strpos<=REST_MAX_CHUNK_SIZE          &&          (len          =
coap_get_header_location_path(request, &str)))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"LP %.*s\n", len, str);

          }

          if          (strpos<=REST_MAX_CHUNK_SIZE          &&          (len          =
coap_get_header_location_query(request, &str)))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"LQ %.*s\n", len, str);

          }

          if  (strpos<=REST_MAX_CHUNK_SIZE  &&  coap_get_header_block2(request,
&block_num, &block_more, &block_size, NULL)) /* This getter allows NULL pointers to
get only a subset of the block parameters. */

          {
```

```
            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"B2 %lu%s (%u)\n", block_num, block_more ? "+" : "", block_size);

          }

          /*

           * Critical Block1 option is currently rejected by engine.

           *

          if (strpos<=REST_MAX_CHUNK_SIZE && coap_get_header_block1(request,
&block_num, &block_more, &block_size, NULL))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"B1 %lu%s (%u)\n", block_num, block_more ? "+" : "", block_size);

          }

          */

        #endif /* CoAP > 03 */

        #endif /* CoAP-specific example */



          if (strpos<=REST_MAX_CHUNK_SIZE && (len = REST.get_query(request,
&query)))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"Qu %.*s\n", len, query);

          }

          if (strpos<=REST_MAX_CHUNK_SIZE && (len =
REST.get_request_payload(request, &bytes)))

          {

            strpos += snprintf((char *)buffer+strpos, REST_MAX_CHUNK_SIZE-strpos+1,
"%.*s", len, bytes);

          }



          if (strpos >= REST_MAX_CHUNK_SIZE)

          {

             buffer[REST_MAX_CHUNK_SIZE-1] = 0xBB; /* '»' to indicate truncation */

          }
```

```
  REST.set_response_payload(response, buffer, strpos);


  PRINTF("/mirror options received: %s\n", buffer);


  /* Set dummy header options for response. Like getters, some setters are not implemented for HTTP and have no effect. */

  REST.set_header_content_type(response, REST.type.TEXT_PLAIN);

  REST.set_header_max_age(response, 17); /* For HTTP, browsers will not re-request the page for 17 seconds. */

  REST.set_header_etag(response, opaque, 2);

  REST.set_header_location(response, location); /* Initial slash is omitted by framework */

  REST.set_header_length(response, strpos); /* For HTTP, browsers will not re-request the page for 10 seconds. CoAP action depends on the client. */


 /* CoAP-specific example: actions not required for normal RESTful Web service. */

 #if WITH_COAP > 1

  coap_set_header_uri_host(response, "tiki");

  coap_set_header_observe(response, 10);

 #if WITH_COAP == 3

  coap_set_header_block(response, 42, 0, 64); /* The block option might be overwritten by the framework when blockwise transfer is requested. */

 #else

  coap_set_header_proxy_uri(response, "ftp://x");

  coap_set_header_block2(response, 42, 0, 64); /* The block option might be overwritten by the framework when blockwise transfer is requested. */

  coap_set_header_block1(response, 23, 0, 16);

  coap_set_header_accept(response, TEXT_PLAIN);

  coap_set_header_if_none_match(response);

 #endif /* CoAP > 03 */

 #endif /* CoAP-specific example */

 }
```

```
#endif /* REST_RES_MIRROR */


/*-----------------------------------------------------------------------------*/
#if REST_RES_CHUNKS
/*
 * For data larger than REST_MAX_CHUNK_SIZE (e.g., stored in flash) resources must be aware of the buffer limitation
 * and split their responses by themselves. To transfer the complete resource through a TCP stream or CoAP's blockwise transfer,
 * the byte offset where to continue is provided to the handler as int32_t pointer.
 * These chunk-wise resources must set the offset value to its new position or -1 of the end is reached.
 * (The offset for CoAP's blockwise transfer can go up to 2'147'481'600 = ~2047 M for block size 2048 (reduced to 1024 in observe-03.)
 */
RESOURCE(chunks, METHOD_GET, "test/chunks", "title=\"Blockwise demo\";rt=\"Data\"");

#define CHUNKS_TOTAL    2050

void
chunks_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  int32_t strpos = 0;

  /* Check the offset for boundaries of the resource data. */
  if (*offset>=CHUNKS_TOTAL)
  {
    REST.set_response_status(response, REST.status.BAD_OPTION);
    /* A block error message should not exceed the minimum block size (16). */

    const char *error_msg = "BlockOutOfScope";
```

```
    REST.set_response_payload(response, error_msg, strlen(error_msg));
    return;
  }

  /* Generate data until reaching CHUNKS_TOTAL. */
  while (strpos<preferred_size)
  {
    strpos += snprintf((char *)buffer+strpos, preferred_size-strpos+1, "|%ld|",
*offset);
  }

  /* snprintf() does not adjust return value if truncated by size. */
  if (strpos > preferred_size)
  {
    strpos = preferred_size;
  }

  /* Truncate if above CHUNKS_TOTAL bytes. */
  if (*offset+(int32_t)strpos > CHUNKS_TOTAL)
  {
    strpos = CHUNKS_TOTAL - *offset;
  }

  REST.set_response_payload(response, buffer, strpos);

  /* IMPORTANT for chunk-wise resources: Signal chunk awareness to REST
engine. */
  *offset += strpos;

  /* Signal end of resource representation. */
  if (*offset>=CHUNKS_TOTAL)
```

```
  {
    *offset = -1;
  }
 }
#endif

/******************************************************************************/
#if REST_RES_SEPARATE && defined (PLATFORM_HAS_BUTTON) && WITH_COAP > 3
/* Required to manually (=not by the engine) handle the response transaction. */
#if WITH_COAP == 7
#include "er-coap-07-separate.h"
#include "er-coap-07-transactions.h"
#elif WITH_COAP == 12
#include "er-coap-12-separate.h"
#include "er-coap-12-transactions.h"
#elif WITH_COAP == 13
#include "er-coap-13-separate.h"
#include "er-coap-13-transactions.h"
#endif
/*
 * CoAP-specific example for separate responses.
 * Note the call "rest_set_pre_handler(&resource_separate, coap_separate_handler);" in the main process.
 * The pre-handler takes care of the empty ACK and updates the MID and message type for CON requests.
 * The resource handler must store all information that required to finalize the response later.
 */
RESOURCE(separate, METHOD_GET, "test/separate", "title=\"Separate demo\"");

/* A structure to store the required information */
```

```
typedef struct application_separate_store {

  /* Provided by Erbium to store generic request information such as remote address and token. */

  coap_separate_t request_metadata;

  /* Add fields for addition information to be stored for finalizing, e.g.: */

  char buffer[16];

} application_separate_store_t;


static uint8_t separate_active = 0;

static application_separate_store_t separate_store[1];


void

separate_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{

  /*

   * Example allows only one open separate response.

   * For multiple, the application must manage the list of stores.

   */

  if (separate_active)

  {

    coap_separate_reject();

  }

  else

  {

    separate_active = 1;


    /* Take over and skip response by engine. */

    coap_separate_accept(request, &separate_store->request_metadata);

    /* Be aware to respect the Block2 option, which is also stored in the coap_separate_t. */
```

```
            /*
             * At the moment, only the minimal information is stored in the store (client address, port, token, MID, type, and Block2).
             * Extend the store, if the application requires additional information from this handler.
             * buffer is an example field for custom information.
             */
            snprintf(separate_store->buffer, sizeof(separate_store->buffer), "StoredInfo");
          }
        }

        void
        separate_finalize_handler()
        {
          if (separate_active)
          {
            coap_transaction_t *transaction = NULL;
            if ( (transaction = coap_new_transaction(separate_store->request_metadata.mid, &separate_store->request_metadata.addr, separate_store->request_metadata.port)) )
            {
              coap_packet_t response[1]; /* This way the packet can be treated as pointer as usual. */

              /* Restore the request information for the response. */
              coap_separate_resume(response, &separate_store->request_metadata, REST.status.OK);

              coap_set_payload(response, separate_store->buffer, strlen(separate_store->buffer));

              /*
```

```
            * Be aware to respect the Block2 option, which is also stored in the
coap_separate_t.

            * As it is a critical option, this example resource pretends to handle it for
compliance.

            */

            coap_set_header_block2(response,                    separate_store-
>request_metadata.block2_num, 0, separate_store->request_metadata.block2_size);



            /* Warning: No check for serialization error. */

            transaction->packet_len = coap_serialize_message(response, transaction-
>packet);

            coap_send_transaction(transaction);

            /* The engine will clear the transaction (right after send for NON, after acked
for CON). */



            separate_active = 0;

          }

          else

          {

            /*

             * Set timer for retry, send error message, ...

             * The example simply waits for another button press.

             */

          }

        } /* if (separate_active) */

      }

      #endif


      /******************************************************************************/

      #if REST_RES_PUSHING

      /*

       * Example for a periodic resource.
```

```
 * It takes an additional period parameter, which defines the interval to call
[name]_periodic_handler().

 * A default post_handler takes care of subscriptions by managing a list of
subscribers to notify.

 */

PERIODIC_RESOURCE(pushing, METHOD_GET, "test/push", "title=\"Periodic
demo\";obs", 5*CLOCK_SECOND);



void

pushing_handler(void* request, void* response, uint8_t *buffer, uint16_t
preferred_size, int32_t *offset)

{

  REST.set_header_content_type(response, REST.type.TEXT_PLAIN);



  /* Usually, a CoAP server would response with the resource representation
matching the periodic_handler. */

  const char *msg = "It's periodic!";

  REST.set_response_payload(response, msg, strlen(msg));



  /* A post_handler that handles subscriptions will be called for periodic
resources by the REST framework. */

}



/*

 * Additionally, a handler function named [resource name]_handler must be
implemented for each PERIODIC_RESOURCE.

 * It will be called by the REST manager process with the defined period.

 */

void

pushing_periodic_handler(resource_t *r)

{

  static uint16_t obs_counter = 0;

  static char content[11];
```

```
  ++obs_counter;

  PRINTF("TICK %u for /%s\n", obs_counter, r->url);

  /* Build notification. */
  coap_packet_t notification[1]; /* This way the packet can be treated as pointer as usual. */
  coap_init_message(notification, COAP_TYPE_NON, REST.status.OK, 0 );
  coap_set_payload(notification, content, snprintf(content, sizeof(content), "TICK %u", obs_counter));

  /* Notify the registered observers with the given message type, observe option, and payload. */
  REST.notify_subscribers(r, obs_counter, notification);
}
#endif




/*****************************************************************************/
#if REST_RES_TOGGLE
/* A simple actuator example. Toggles the red led */
RESOURCE(toggle, METHOD_POST, "actuators/toggle", "title=\"Red LED\";rt=\"Control\"");
void
toggle_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)
{
  leds_toggle(LEDS_RED);
}

#endif /* PLATFORM_HAS_LEDS */
```

```
/*---------------------------------------------------------------------------*/

#if REST_RES_LIGHT && defined (PLATFORM_HAS_LIGHT)

/* A simple getter example. Returns the reading from light sensor with a simple etag */

RESOURCE(light, METHOD_GET, "sensors/light", "title=\"Photosynthetic and solar light (supports JSON)\";rt=\"LightSensor\"");

void

light_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{

  SENSORS_ACTIVATE(light_sensor);

  //uint16_t light_photosynthetic = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);

  uint16_t val = light_sensor.value(LIGHT_SENSOR_TOTAL_SOLAR);

  //float s = (float)(val * 0.4071);


  uint16_t decli = 0;

  if (val>0){

 decli = (val*4)/10;

 }


  uint16_t light_solar = "";


  const uint16_t *accept = NULL;

  int num = REST.get_header_accept(request, &accept);


  if ((num==0) || (num && accept[0]==REST.type.TEXT_PLAIN))

  {

    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);

    //snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "Light: %d.%02u lux",dec, (unsigned int)(frac * 100));

    snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "Light: %u lux",decli,val);
```

```
            REST.set_response_payload(response,     (uint8_t     *)buffer,     strlen((char
*)buffer));

           }

           else if (num && (accept[0]==REST.type.APPLICATION_XML))

           {

            REST.set_header_content_type(response, REST.type.APPLICATION_XML);

            snprintf((char   *)buffer,   REST_MAX_CHUNK_SIZE,   "<light   solar=\"%d\"/>",
decli);


            REST.set_response_payload(response, buffer, strlen((char *)buffer));

           }

           else if (num && (accept[0]==REST.type.APPLICATION_JSON))

           {

            REST.set_header_content_type(response, REST.type.APPLICATION_JSON);

            snprintf((char   *)buffer,   REST_MAX_CHUNK_SIZE,   "{'light':{'solar':%d}}",
decli);


            REST.set_response_payload(response, buffer, strlen((char *)buffer));

           }

           else

           {

            REST.set_response_status(response, REST.status.NOT_ACCEPTABLE);

            const char *msg = "Supporting content-types text/plain, application/xml, and
application/json";

            REST.set_response_payload(response, msg, strlen(msg));

           }

           SENSORS_DEACTIVATE(light_sensor);

          }

          #endif /* PLATFORM_HAS_LIGHT */
```

```
/*---------------------------------------------------------------------------*/

#if REST_RES_TEMP && defined (PLATFORM_HAS_SHT11)

/* A simple getter example. Returns the reading from TEMPERATURE AND
HUMIDITY sensor with a simple etag */

RESOURCE(sht11, METHOD_GET, "sensors/sht11", "title=\"Temperature and
Humidity\";rt=\"SHT11\"");

void

sht11_handler(void* request, void* response, uint8_t *buffer, uint16_t
preferred_size, int32_t *offset)

{

  SENSORS_ACTIVATE(sht11_sensor);

  uint16_t temperature = sht11_sensor.value(SHT11_SENSOR_TEMP);

  uint16_t humidity = sht11_sensor.value(SHT11_SENSOR_HUMIDITY);

  uint16_t dec=0;

  if (temperature!=-1){

dec = (temperature/100 - 40);

}

  int p2 = (humidity*2)/1000;

  int hum = 0;


  hum = ((humidity*4/100) - 4) - (28*p2)/10;




  //s= (((0.0405*val) - 4) + ((-2.8 * 0.000001)*(pow(val,2))));


  const uint16_t *accept = NULL;

  int num = REST.get_header_accept(request, &accept);


  if ((num==0) || (num && accept[0]==REST.type.TEXT_PLAIN))

  {

    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
```

```
            snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "Temperature: %u C, Humidity: %d ( % ) ",dec,hum);


            REST.set_response_payload(response, (uint8_t *)buffer, strlen((char *)buffer));

          }

          else if (num && (accept[0]==REST.type.APPLICATION_XML))

          {

            REST.set_header_content_type(response, REST.type.APPLICATION_XML);

            snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "<temperature=\"%u\" humidity=\"%u\"/>", temperature, humidity);


            REST.set_response_payload(response, buffer, strlen((char *)buffer));

          }

          else if (num && (accept[0]==REST.type.APPLICATION_JSON))

          {

            REST.set_header_content_type(response, REST.type.APPLICATION_JSON);

            snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "{'SHT11':{'temperature':%u,'humidity':%u}}", temperature, humidity);


            REST.set_response_payload(response, buffer, strlen((char *)buffer));

          }

          else

          {

            REST.set_response_status(response, REST.status.NOT_ACCEPTABLE);

            const char *msg = "Supporting content-types text/plain, application/xml, and application/json";

            REST.set_response_payload(response, msg, strlen(msg));

          }

         SENSORS_DEACTIVATE(sht11_sensor);

        }

        #endif /* PLATFORM_HAS_SHT11 */
```

```
/*****************************************************************************/

#if REST_RES_BATTERY && defined (PLATFORM_HAS_BATTERY)

/* A simple getter example. Returns the reading from light sensor with a simple etag */

RESOURCE(battery, METHOD_GET, "sensors/battery", "title=\"Battery status\";rt=\"Battery\"");

void

battery_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{

  int battery = battery_sensor.value(0);

  int battery2 = 100*battery / 2600; //Battery Max Value


  const uint16_t *accept = NULL;

  int num = REST.get_header_accept(request, &accept);


  if ((num==0) || (num && accept[0]==REST.type.TEXT_PLAIN))

  {

    REST.set_header_content_type(response, REST.type.TEXT_PLAIN);

    snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "Battery %d ( % )", battery2);


    REST.set_response_payload(response, (uint8_t *)buffer, strlen((char *)buffer));

  }

  else if (num && (accept[0]==REST.type.APPLICATION_JSON))

  {

    REST.set_header_content_type(response, REST.type.APPLICATION_JSON);

    snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "{'battery':%d}", battery);


    REST.set_response_payload(response, buffer, strlen((char *)buffer));

  }
```

```
  else

  {

    REST.set_response_status(response, REST.status.NOT_ACCEPTABLE);

    const char *msg = "Supporting content-types text/plain and application/json";

    REST.set_response_payload(response, msg, strlen(msg));

  }

}

#endif /* PLATFORM_HAS_BATTERY */




#if defined (PLATFORM_HAS_RADIO) && REST_RES_RADIO

/* A simple getter example. Returns the reading of the rssi/lqi from radio sensor */

RESOURCE(radio, METHOD_GET, "sensor/radio", "title=\"RADIO: ?p=lqi|rssi\";rt=\"RadioSensor\"");



void

radio_handler(void* request, void* response, uint8_t *buffer, uint16_t preferred_size, int32_t *offset)

{

  size_t len = 0;

  const char *p = NULL;

  uint8_t param = 0;

  int success = 1;



  const uint16_t *accept = NULL;

  int num = REST.get_header_accept(request, &accept);



  if ((len=REST.get_query_variable(request, "p", &p))) {

    PRINTF("p %.*s\n", len, p);

    if (strncmp(p, "lqi", len)==0) {
```

```
      param = RADIO_SENSOR_LAST_VALUE;
    } else if(strncmp(p,"rssi", len)==0) {
      param = RADIO_SENSOR_LAST_PACKET;
    } else {
      success = 0;
    }
  } else {
    success = 0;
  }

  if (success) {
    if ((num==0) || (num && accept[0]==REST.type.TEXT_PLAIN))
    {
      REST.set_header_content_type(response, REST.type.TEXT_PLAIN);
      snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "%d",
radio_sensor.value(param));

      REST.set_response_payload(response, (uint8_t *)buffer, strlen((char
*)buffer));
    }
    else if (num && (accept[0]==REST.type.APPLICATION_JSON))
    {
      REST.set_header_content_type(response,
REST.type.APPLICATION_JSON);

      if (param == RADIO_SENSOR_LAST_VALUE) {
        snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "{'lqi':%d}",
radio_sensor.value(param));
      } else if (param == RADIO_SENSOR_LAST_PACKET) {
        snprintf((char *)buffer, REST_MAX_CHUNK_SIZE, "{'rssi':%d}",
radio_sensor.value(param));
      }
```

```
      REST.set_response_payload(response, buffer, strlen((char *)buffer));
    }
    else
    {
      REST.set_response_status(response, REST.status.NOT_ACCEPTABLE);
      const char *msg = "Supporting content-types text/plain and application/json";
      REST.set_response_payload(response, msg, strlen(msg));
    }
  } else {
    REST.set_response_status(response, REST.status.BAD_REQUEST);
  }
}
#endif



PROCESS(rest_server_example, "Erbium Example Server");
AUTOSTART_PROCESSES(&rest_server_example);

PROCESS_THREAD(rest_server_example, ev, data)
{
  PROCESS_BEGIN();

  PRINTF("Starting Erbium Example Server\n");

#ifdef RF_CHANNEL
  PRINTF("RF channel: %u\n", RF_CHANNEL);
#endif
#ifdef IEEE802154_PANID
```

```
  PRINTF("PAN ID: 0x%04X\n", IEEE802154_PANID);
#endif

  PRINTF("uIP buffer: %u\n", UIP_BUFSIZE);
  PRINTF("LL header: %u\n", UIP_LLH_LEN);
  PRINTF("IP+UDP header: %u\n", UIP_IPUDPH_LEN);
  PRINTF("REST max chunk: %u\n", REST_MAX_CHUNK_SIZE);

  /* Initialize the REST engine. */
  rest_init_engine();

  /* Activate the application-specific resources. */
#if REST_RES_HELLO
  rest_activate_resource(&resource_helloworld);
#endif
#if REST_RES_MIRROR
  rest_activate_resource(&resource_mirror);
#endif
#if REST_RES_CHUNKS
  rest_activate_resource(&resource_chunks);
#endif
#if REST_RES_PUSHING
  rest_activate_periodic_resource(&periodic_resource_pushing);
#endif
#if defined (PLATFORM_HAS_BUTTON) && REST_RES_EVENT
  rest_activate_event_resource(&resource_event);
#endif
#if defined (PLATFORM_HAS_BUTTON) && REST_RES_SEPARATE && WITH_COAP > 3
  /* No pre-handler anymore, user coap_separate_accept() and coap_separate_reject(). */
```

```
  rest_activate_resource(&resource_separate);
#endif
#if defined (PLATFORM_HAS_BUTTON) && (REST_RES_EVENT || (REST_RES_SEPARATE && WITH_COAP > 3))
  SENSORS_ACTIVATE(button_sensor);
#endif
#if defined (PLATFORM_HAS_LEDS)
#if REST_RES_LEDS
  rest_activate_resource(&resource_leds);
#endif
#if REST_RES_TOGGLE
  rest_activate_resource(&resource_toggle);
#endif
#endif /* PLATFORM_HAS_LEDS */
#if defined (PLATFORM_HAS_LIGHT) && REST_RES_LIGHT
  //SENSORS_ACTIVATE(light_sensor);
  rest_activate_resource(&resource_light);
#endif
#if defined (PLATFORM_HAS_SHT11) && REST_RES_TEMP
  //SENSORS_ACTIVATE(sht11_sensor);
  rest_activate_resource(&resource_sht11);
#endif
#if defined (PLATFORM_HAS_BATTERY) && REST_RES_BATTERY
  SENSORS_ACTIVATE(battery_sensor);
  rest_activate_resource(&resource_battery);
#endif
#if defined (PLATFORM_HAS_RADIO) && REST_RES_RADIO
  SENSORS_ACTIVATE(radio_sensor);
  rest_activate_resource(&resource_radio);
#endif
```

```
  /* Define application-specific events here. */
  while(1) {
    PROCESS_WAIT_EVENT();
#if defined (PLATFORM_HAS_BUTTON)
    if (ev == sensors_event && data == &button_sensor) {
      PRINTF("BUTTON\n");
#if REST_RES_EVENT
      /* Call the event_handler for this application-specific event. */
      event_event_handler(&resource_event);
#endif
#if REST_RES_SEPARATE && WITH_COAP>3
      /* Also call the separate response example handler. */
      separate_finalize_handler();
#endif
    }
#endif /* PLATFORM_HAS_BUTTON */
  } /* while (1) */

  PROCESS_END();
}
```