UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

**JEAN POSSEBON WAGHETTI**

# PROJETO DE DIPLOMAÇÃO

# WEB SERVICES PARA SISTEMAS EMBARCADOS

Porto Alegre
2011

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

# WEB SERVICES PARA SISTEMAS EMBARCADOS

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para Graduação em Engenharia Elétrica.

ORIENTADOR: Prof. Dr. Marcelo Götz

Porto Alegre
2011

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

JEAN POSSEBON WAGHETTI

# WEB SERVICES PARA SISTEMAS EMBARCADOS

Este projeto foi julgado adequado para fazer jus aos créditos da Disciplina de "Projeto de Diplomação", do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____

Prof. MARCELO GÖTZ, UFRGS

Doutor em Engenharia Elétrica/Informática (UFRGS – Brasil/Paderborn, Alemenha)

Banca Examinadora:

Prof. Dr. Marcelo Götz, UFRGS

Doutor pela Universität Paderborn – Paderborn, Alemanha

Eng. Murilo Larroza Fonseca, UFGRS

Engenheiro pela Universidade Federal de Santa Catarina – Florianópolis, Brasil

Prof. Dr. Walter Fetter Lages, UFRGS

Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Porto Alegre, julho de 2011.

# DEDICATÓRIA

Dedico esse trabalho à minha avó Thalita, que, apesar de não estar mais entre nós, é uma das melhores lembranças que tenho, e à minha namorada Mariana, que me faz enxergar o horizonte da vida muito mais bonito.

# AGRADECIMENTOS

Agradeço aos meus pais Norberto e Rita, por sempre acreditarem em mim, e ao meu irmão Victor, por sempre se mostrar disponível em me ajudar no possível. Ao resto de minha família, pelo carinho durante todos esses anos.

À minha namorada Mariana, uma inspiração para eu seguir em frente.

Ao Brasil, país onde nasci e cresci, para qual o conhecimento adquirido nesses últimos anos quero aplicar visando seu desenvolvimento.

Aos colegas pelo companheirismo, dentro e fora da vida acadêmica, e auxílio nas tarefas desenvolvidas durante o curso.

À Universidade, professores e funcionários, todos responsáveis pelo excelente ensino ao qual tive acesso.

Aos verdadeiros amigos, nem sempre presentes devido a distância, mas com quem sempre pude contar.

# RESUMO

Este documento relata as atividades desenvolvidas para a realização do projeto “Web services para sistemas embarcados”, o qual foi também utilizado para obter os créditos referentes à disciplina Projeto de Diplomação, do curso de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul (UFRGS).

O projeto consiste em permitir, com a utilização de um sistema embarcado, que um dispositivo ofereça suas funcionalidades através de uma interface orientada a serviço. Os processos utilizados para alcançar isso são descritos, bem como os passos para que novos serviços sejam criados com a utilização desta plataforma.

**Palavras-chaves: Engenharia Elétrica. Orientação a serviço. Sistemas embarcados. Interoperabilidade.**

# ABSTRACT

This document reports the developed activities for the implementation of the Project “Web services for embedded systems”, which was also used to obtain the credits for the discipline Projeto de Diplomação, of the Electrical Engineering course of Universidade Fedaral do Rio Grande do Sul (UFRGS).

The project consists in allowing, with the utilization of an embedded system, a device to offer its functionalities through a service oriented interface. All the steps to this achievement are described, as the steps to the creation of new services and the utilization on the platform.



**Keywords: Electrical Engineering. Service orientation. Embedded systems. Interoperability.**

# SUMÁRIO

# LISTA DE ILUSTRAÇÕES

# LISTA DE ABREVIATURAS

DPWS: Devices Profile for Web-Services

GPIO: General Purpose Input Output

HAVi: Home Audio/Video interoperability

HTTP: Hypertext Transfer Protocol

I/O: Input/Output

IDE: Integrated Development Environment

IP: Internet Protocol

IRI: International Resource Identifier

JINI: Java Intelligent Network Infrastructure

JMEDS: Java Multi Edition DPWS Stack

JRE: Java Runtime Environment

JTAG: Joint Test Action Group

LAN: Local Area Network

LCD: Liquid Crystal Display

MTOM: Message Transmission Optimization Mechanism

OASIS: Organization for the Advancement of Structured Information Standards

OSGi: Open Service Gateway initiative

SD: Secure Digital

SO: Sistema Operacional

SOA: Service Oriented Architecture

SOAP: Simple Object Access Protocol

UDP: User Datagram Protocol

UFRGS: Universidade Federal do Rio Grande do Sul

URI: Universal Resource Identifier

URL: Universal Resource Locator

USB: Universal Serial Bus

W3C: World Wide Web Consortium

QoS: Quality of Service

WAN: Wide Area Network

WS4D: Web Services For Devices

WSA: Web Service Architecture

WSD: Web Service Description

WSDL: Web Service Description Language

XML: Extensible Markup Language

# 1 INTRODUÇÃO

Os dispositivos eletrônicos, cada vez mais, possuem capacidade de processamento e de interoperabilidade. Equipamentos eletrônicos são desenvolvidos por fabricantes diferentes ao redor do mundo, que geralmente desenvolvem seus próprios protocolos de comunicação e mantê-los fechados. Isso impossibilita o desenvolvimento, por parte de outros fabricantes, de produtos capazes de operar em conjunto, trocando informações coerentes, com os que utilizam esses protocolos fechados.

Orientação a serviços apresenta-se como uma abordagem que facilita essa interoperabilidade entre equipamentos, pois é possível a oferta de “serviços” entre esses dispositivos, mesmo que oriundos de diferentes fabricantes, através de um padrão de alto nível e independente de tecnologia.

Este projeto de diplomação foi idealizado, inicialmente, para que fosse utilizado em um projeto de mestrado, onde a intercomunicação em nível de dispositivos fosse possível. Em um processo inicial de avaliação, o protocolo *Devices Profile for Web Services* (DPWS) surgiu como uma alternativa, pois esse define um conjunto mínimo de restrições de implementação para habilitar a troca de mensagens, descoberta, descrição e eventos seguros de *web services* em dispositivos com recursos limitados.

A utilização de um protocolo aberto, possivelmente padrão industrial no futuro, em redes de dispositivos heterogêneos, tanto em suas funções como em seus fabricantes, garantiria completa integração e a abertura de muitas possibilidades em relação à forma com que os dispositivos se comunicariam.

A meta do projeto é obter uma plataforma embarcada que utilize o protocolo para fornecer algum serviço e facilitar a oferta de novos serviços. Para alcançar este objetivo, a metodologia a seguir foi adotada. Em uma primeira etapa, foram estudados a orientação a serviços, o conceito de serviço e a definição dos requisitos. Esta etapa encontra-se descrita no

Capítulo 2. Já no capítulo 3, são apresentadas e analisadas as alternativas que atenderiam aos objetivos. Para se obter familiaridade com as ferramentas de software utilizadas, e com o ambiente de desenvolvimento, foram definidos testes preliminares, descritos no Capítulo 4. A implementação do uDPWS, a plataforma sobre a qual o protocolo é utilizado, no sistema embarcado são apresentados nos Capítulos 5 e 6. O Capítulo 7 apresenta testes mais completos, com o sistema embarcado escolhido sendo utilizado. O capítulo 8 apresenta a definição para a criação de um novo serviço e os passos utilizados para que seja utilizado, juntamente com outros serviços, em um dispositivo. As sugestões de melhoria e continuação dos trabalhos são comentadas no Capítulo 9. Por fim, o Capítulo 10 traz as considerações finais e comenta as conclusões obtidas com a realização deste trabalho.

# 2 CONTEXTO DO PROJETO

Com o avanço da tecnologia, dispositivos com maior capacidade de processamento e interoperabilidade se tornam mais facilmente disponíveis. A comunicação entre esses dispositivos é feita através de um protocolo, que define a forma, ou as regras, de como isso ocorre.

O projeto “Web wervices para sistemas embarcados” surgiu a partir da observação das tendências tecnológicas da chamada *internet of things* (ASHTON, 2009), que se refere a objetos unicamente identificados e sua representação em uma estrutura de rede, como a *internet*. Essas redes onde dispositivos inteligentes conversam entre si pode ter processamento descentralizado e maior integração entre as tarefas desempenhadas por cada um desses dispositivos (ZEEB).

## 2.1 SERVIÇO

Conforme a *Organization for the Advancement of Structured Information Standards* (OASIS), um serviço consiste em “um mecanismo para habilitar o acesso a uma ou mais capacidades, onde o acesso é provido utilizando uma interface descrita e executado de forma consistente com as restrições e política como especificadas pela descrição do serviço” (OASIS, 2006). Dessa forma, um serviço não passa de uma funcionalidade, possuindo regras que bem o definem.

A Figura 1 é um diagrama onde alguns dispositivos trocam serviços. O computador, através de uma interface com o usuário, por exemplo, pode configurar o ar-condicionado através do serviço que esse oferece, de refrigeração. Esse serviço possui a funcionalidade de ser informada qual a temperatura ambiente desejada (chamada de temperatura alvo na figura). Assim, o ar-condicionado provê um serviço que, para o exemplo, o computador é o cliente. O fabricante do ar-condicionado utilizou um sensor de temperatura de outro fabricante, que

possui uma interface que se comunica através de serviços. O serviço oferecido pelo termômetro é informar a temperatura, que será requisitado pelo ar-condicionado, para que este verifique o seu funcionamento – se deve aumentar a refrigeração, caso a temperatura ambiente seja maior que a configurada, por exemplo.


Figura 1 Diagrama exemplo de utilização de serviços.

Com interfaces bem definidas para os serviços de cada dispositivo, os clientes podem receber essa “prestação” de serviços dos outros dispositivos, sem serem necessários detalhes de como é feita a implementação por cada fabricante.

## 2.2 MOTIVAÇÃO

O projeto insere-se no contexto de um projeto de mestrado onde um protocolo de comunicação em nível de dispositivos é necessário. Vários dispositivos ligados à rede elétrica deveriam se intercomunicar para que sejam analisadas questões relativas à gestão de energia.

Surgiu a necessidade de integrar os dispositivos através de um padrão de alto nível, que permitisse a interoperabilidade. Então, a orientação a serviços surgiu como a principal candidata para promover essa interoperabilidade.

O desenvolvimento de uma aplicação que utilize *web services* foi motivada por ser executado sobre padrões de internet, altamente aceitos e consolidados, definidos pela *World Wide Web Consortium* (W3C), assim como *web services*.

Levando em consideração o tipo de aplicação onde o projeto de diplomação seria utilizado, surge a alternativa do protocolo *Devices Profile for Web Services* (DPWS), pois esse define um conjunto mínimo de restrições de implementação para habilitar mensagens, descoberta, descrição e eventos seguros de *web services* em dispositivos com recursos limitados.

## 2.3 OBJETIVOS

Os objetivos desse projeto são implementar o protocolo DPWS em uma plataforma embarcada, utilizando ferramentas já desenvolvidas para isso, que possa então fornecer algum serviço, descrevendo os processos utilizados para isso e avaliar suas características.

Havendo essa implementação executando no dispositivo, os passos para a criação de novos serviços serão descritos, para que a implementação utilizada seja mais facilmente compreendida, assim como a tarefa de criação de novas aplicações, contribuindo para pesquisas e desenvolvimentos futuros na área de integração de produtos diferentes.

## 2.4 WEB SERVICES

*Web services* são definidos pela W3C como “um padrão de meios de interoperação de diferentes aplicações de software, que podem ser executados em diversas plataformas diferentes. Tem uma interface descrita em um formato processável por máquina (especificamente *Web Service Description Language* (WSDL)). Outros sistemas interagem como o *web service* de uma maneira ditada pela sua descrição utilizando mensagens do tipo *Simple Object Access Protocol* (SOAP), tipicamente transportada através de *Hipertext*

*Transfer Protocol* (HTTP) com serialização de *Extensible Markup Language* (XML) em conjunto com outros padrões relacionados à rede".

SOAP é um protocolo para troca de informação estruturada entre serviços, HTTP é um protocolo em nível de aplicação, leve e robusto, para sistemas de informação (BERNERS-LEE et al, 1996) e XML é um conjunto de definições para organizar informações em um documento de forma processável por máquina.

O padrão não especifica exatamente como os *web services* são implementados, nem impõe restrições sobre isso, mas especifica um conjunto mínimo de características comuns à todos *web services*, e um conjunto de características necessários para um conjunto de *web services*.

Um *web service* é uma noção abstrata, que representa a capacidade de realização de tarefas coerentes. É uma aplicação que aceita solicitação de outros sistemas através de padrões de internet. Precisa ser implementado por um agente concreto, que é um programa computacional ou *hardware* que trabalha em nome de uma pessoa ou organização, recebendo e enviando mensagens. A funcionalidade que representa o *web service* não é atrelada ao agente físico que o disponibiliza. Por exemplo, um *web service* implementado em determinada linguagem de programação, sob uma determinada plataforma, pode ter a mesma funcionalidade se esse fosse implementado com outra linguagem de programação e/ou sendo executado sob uma plataforma com, por exemplo, um processador diferente.

Agentes podem ser tanto solicitadores como provedores de serviços, não sendo essas duas classes exclusivas entre si. Cada agente pode solicitar zero ou mais serviços bem como disponibilizar zero ou mais serviços.

Para que os serviços disponibilizados possam ser solicitados, a mecânica como as mensagens são trocadas é documentada pela descrição do serviço, ou *Web Service Description* (WSD), escrita em WSDL. A WSD pode conter a semântica do serviço, que

descreve o comportamento esperado em relação às mensagens enviadas. É o significado e a finalidade do serviço.

Muitas outras definições envolvem *web services*, entre elas a descoberta de serviços por parte de agentes, políticas (que impõem restrições no comportamento de pessoas ou organizações, definindo quais são suas capacidades, ou obrigações e permissões) etc. A Figura 2, mostra um caso de uso da Arquitetura Orientada a Serviço (*Service Oriented Architecture*, SOA), onde um provedor de serviço (*Service Provider*) disponibiliza tal com sua descrição, que é utilizado por um agente solicitador (*Service Requester*), que fez a descoberta do provedor através de uma agência de descoberta (*Discovery Agencies*). Essa última, nem sempre é necessária. É possível que a descoberta seja solicitada através de mensagens enviadas pelos provedores através da rede, para que haja a solicitação por parte de algum outro agente.


Figura 2 Diagrama de *Arquitetura Orientada a Serviço* (SOA) (adaptado BERNERS-LEE, 2002).

O uso de *web services* permitiria, de um ponto de vista de automação industrial uma adequação mais rápida à dinâmica do mercado, pois as reconfigurações das linhas de produção seriam mais rápidas para atender as demandas, e a falta de interoperabilidade, causadas por *ilhas de tecnologia*, seria extinta, pois não haveria mais o uso de padrões sem vasta aceitação. As vantagens seriam dispositivos mais inteligentes, que possibilitariam maior modularização, reconfigurabilidade e facilidade na manutenção (JAMMES et al, 2009).

## 2.5 DEVICES PROFILE FOR WEB SERVICES (DPWS)

*Device Profile for Web Services* (DPWS) define um conjunto mínimo de restrições de implementação para habilitar mensagens, descoberta, descrição e eventos seguros de *web services* em dispositivos com recursos limitados, para promover a interoperabilidade entre esse tipo de dispositivos e com clientes com recursos mais flexíveis.

Os objetivos do DPWS são similares ao *Universal Plug and Play* (UPnP, um conjunto de protocolos de rede que permite dispositivos de rede descobrirem uns aos outros automaticamente e estabelecerem serviços de rede funcionais), mas com a adição de utilizar a tecnologia de *web services* e permitir a integração de serviços de dispositivos em aplicações de maior porte, em relação às aplicações domésticas do UPnP.

Outros padrões parecidos com DPWS, que como o UPnP promovem comunicação dispositivo-dispositivo são: *Open Service Gateway initiative* (OSGi), *Home Audio/Video Interoperability* (HAVi) e *Java Intelligent Network Infrastructure* (JINI). A grande vantagem do DPWS em relação aos padrões citados é a utilização de *web services*, que implica em maior aceitação entre desenvolvedores e plataformas, assim como a independência de linguagem de programação (ZEEB et al).

O DPWS, cuja especificação foi inicialmente publicada em maio de 2004, foi aprovado como um padrão da *Organization for the Advancement of Structured Information*

*Standards* (OASIS), um consórcio mundial que trata do desenvolvimento, convergência e adoção de padrões para *e-business* e *web services*, em junho de 2009 (WALLENSTEIN; LEITE, 2008).

Os protocolos padrão de *web services* utilizados são SOAP (SOAP-*over*-HTTP, para troca de informação estruturada, utilizando XML, como explicado anteriormente), WSDL (para descrição dos *web services*) e XML-Schema (para expressar um conjunto de regras que um documento XML deve seguir para ser considerado válido de acordo com o padrão). Outros padrões são necessários para o DPWS:

- WS-Policy: pelo fato da *Web Service Architecture* (WSA) ser independente de plataforma, linguagem de programação e protocolo de transporte, os usuários que forem utilizar um serviço devem estar de acordo com a política (*policy*) desse serviço, que são definidas pelas capacidades, características e requisitos impostos. Define características de *Quality of Service* (QoS, termo em inglês para "qualidade do serviço, define a forma como os recursos são utilizados para atender a performance requerida) e de segurança necessárias para a comunicação;

- WS-Addressing: o principal objetivo do WS-Addressing é que o endereçamento para *web services*, bem como as mensagens, sejam neutras em relação ao meio de transporte dessas. Introduz o conceito de *endpoint reference*, estruturas definidas para endereçar *web services endpoints*. Podem ser um *Uniform Resource Locator* (URL), *Universal Resource Identifier* (URI) ou qualquer outro endereço lógico expresso por um *Internationalized Resource Identifier* (IRI), que são utilizados para identificar um nome ou um recurso na *internet*. Cabeçalhos são definidos para endereçar as mensagens de forma independente ao mecanismo de transporte;

- SOAP-over-UDP: mensagens SOAP sobre o *User Datagram Protcol* (UDP). Utiliza WS-Addressing para suportar o mesmo padrão para troca de mensagem que o SOAP-over-HTTP;

- MTOM: SOAP *Message Transmission Optimization Mechanism* (MTOM) prove uma recurso abstrato para otimizar a transmissão de mensagens SOAP, que deve ser adaptada ao protocolo utilizado para transmissão;

- WS-Discovery: protocolo baseado em endereço *Internet Protocol* (IP) *multicast* (para vários pontos) para habilitar a descoberta automática de serviços. Introduz três diferentes *endpoints: Target Services* (o próprio web service sendo oferecido à rede), *Client* (que procura *Target Services* e os descobre dinamicamente) e *Discovery Proxy*, que serve para habilitar a descoberta em redes estendidas, ou seja, permitir que uma rede local (*Local Area Network*, LAN) encontre serviços de outra LAN, caso as duas possuam interligação de alguma forma, como por exemplo, uma *Wide Area Network* (WAN).

- WS-MetadataExchange/WS-Transfer: definem tipos de dados e operações para adquirir metadados (no inglês *metadata*, que é a informação trocada pelos serviços) de *endpoints*, que definirá como interagir com o *endpoint* descrito.

- WS-Eventing: define um protocolo para administrar assinaturas (*subscriptions*) para *web services* baseado em eventos. Define três *endpoints*: *subscribers*, *subscription source* e *subscription manager*. *Subscribers* solicitam assinaturas como utilizadores de eventos de *event sources*. *Subscription managers* são responsáveis por guardar assinaturas de *event sources*. Essas devem ser requeridas e podem expirar, sendo necessária a revalidação ou finalização da assinatura. Não há restrições ou limitações em suporte a outros tipos de suporte a eventos definidos por usuário.

- WS-Security: Especificação que provê mecanismo para integridade das mensagens e confidencialidade de mensagens SOAP, de forma independente da tecnologia utilizada.

A Figura 3 mostra o protocolo DPWS como uma pilha de protocolos, onde na parte superior encontram-se os protocolos específicos da aplicação, que são definidos por requerimentos do serviço em si, na camada abaixo estão os protocolos citados anteriormente (representando todos a camada de aplicação do modelo ISO-OSI). Seguindo abaixo, há o protocolo HTTP, também da comada de aplicação no modelo ISSO-OSI, e os protocolos da camada de transporte: UDP, *Transmission Control Protocol* (TCP). Os protocolos TCP e UDP. Na camada de rede, temos o protocolo IP.

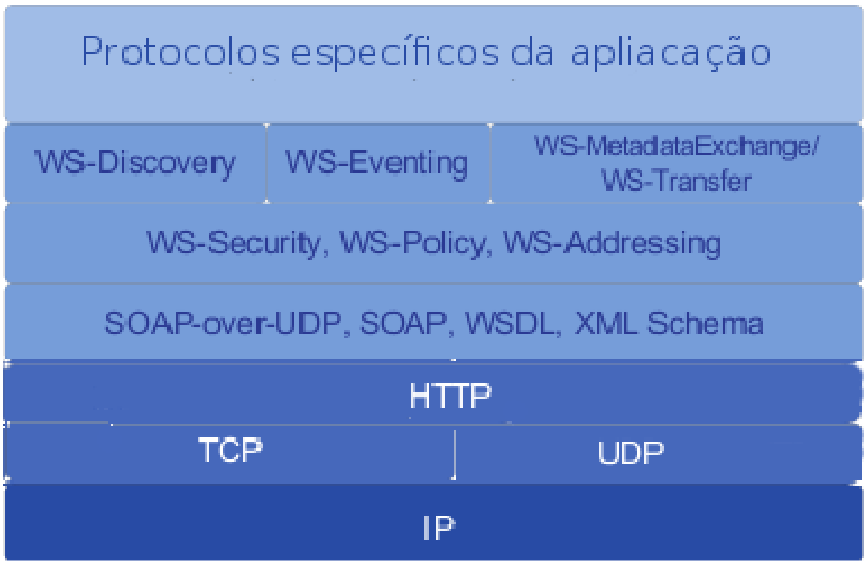


Figura 3 Diagrama de DPWS como uma pilha de protocolos (adaptado WS4D, Web Services For Devices).

A Figura 4 mostra um diagrama que exemplifica a interação entre um cliente que se comunica com um dispositivo que oferece um serviço. É o caso de uma impressora, onde o cliente troca dados com o dispositivo, para descoberta do dispositivo (utilizando o WS-Discovery) e de metadados, relativos ao próprio dispositivo ou ao serviço que ele oferece (WS-MetadataTransfer). A possibilidade de assinatura de eventos é feita através do WS-Eventing. MTOM é utilizado para otimizar a transmissão de mensagens, referente ao tamanho das mensagens transmitidas. Outras mensagens podem ser trocadas para iniciar o processo de impressão e enviar o estado da impressora e do serviço.

Descrevendo um caso de exemplo desse sistema como um todo, o cliente decide procurar por um serviço de impressão, enviando mensagens de *Probe* para serviços do tipo impressão. O dispositivo recebe essa mensagem e verifica que ele é possui esse serviço, enviando uma mensagem de *Probe Match* para o cliente. Isso é feito através do padrão WS-Discovery. Dispositivo e cliente, então, trocam metadados para que informações acerca do dispositivo (por exemplo: fabricante, número de série e outras informações referentes ao dispositivo, como os serviços disponíveis) sejam conhecidas pelo cliente. O cliente também receberá metadados do serviço, referentes às funcionalidades desse. Então toda a funcionalidade e a forma de interação já é conhecida pelo cliente, que pode requerer ações do serviço, como iniciar impressão (*Start Print Job*), se inscrever para receber notificações de eventos da impressora, como estado do processo (*Job Status*) etc.

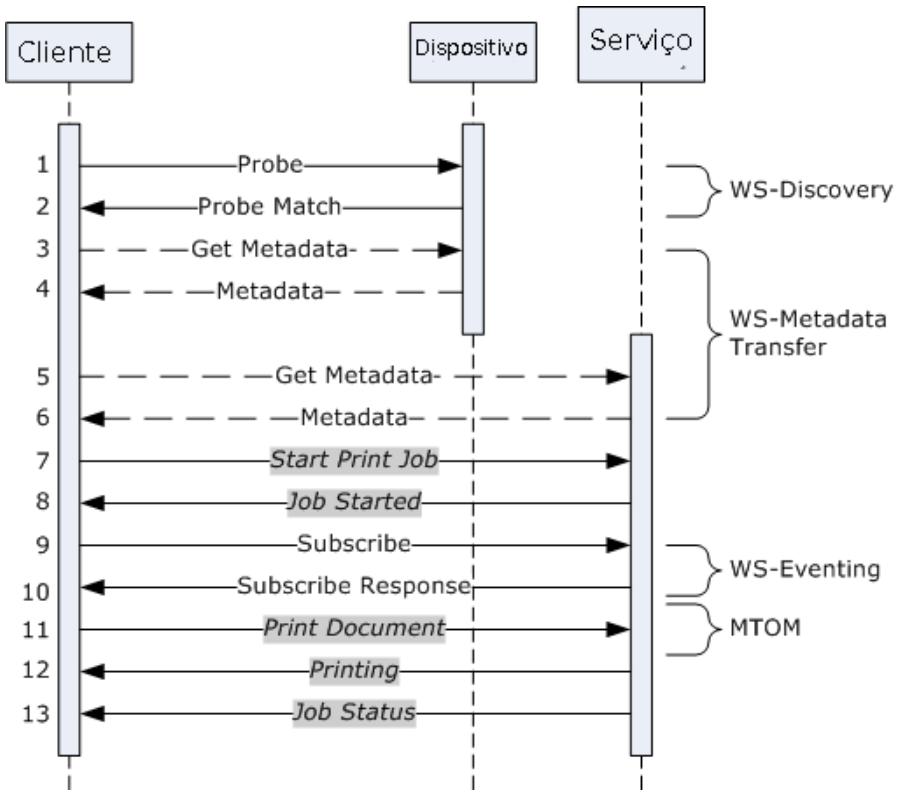


Figura 4 Exemplo de dispositivo cliente, dispositivo e serviço DPWS (adaptado, SCHLIMMER, 2004).

# 3 ANÁLISE DE ALTERNATIVAS

A escolha do uso de *web services* para o projeto se justifica pela interoperabilidade que os dispositivos têm com esse tipo de protocolo. Outro ponto a favor de seu uso é de ser a implementação mais difundida de SOA (DOHNDORF et al). Em vista do alvo do projeto ser sistemas embarcados, que possuem recursos bastante limitados em grande parte dos casos, DPWS foi escolhido por ser uma implementação de *web services* que visa esse tipo de dispositivos.

O protocolo não será desenvolvido para o projeto, mas sim uma implementação existente será utilizada e adaptada para ser executada no sistema embarcado escolhido, atendendo aos propósitos do projeto de mestrado no qual este projeto de diplomação está inserido.

## 3.1 PLATAFORMA EMBARCADA

A escolha da plataforma embarcada em que a aplicação seria executada teve como fator principal a adequação aos requisitos do projeto de mestrado. A facilidade de implementação da aplicação no sistema embarcado, considerando-se um ambiente de pesquisa, também foi considerada. A possibilidade de execução do sistema operacional (SO) GNU/Linux embarcado, para que o acesso aos periféricos fosse menos trabalhoso em termos de código foi um fator importante nesta tomada de decisão. GNU/Linux embarcado também seria interessante no caso de a aplicação ser escrita em Java, pois a instalação e configuração de um ambiente Java (*Java Runtime Environment*, JRE) seria bem mais simples. O SO Linux também possibilitaria, quando necessário, o desenvolvimento e alteração de códigos, uma vez que este sistema é de código aberto.

Uma plataforma Digi Connect ME 9210, que pode ser visualizada na Figura 5, foi avaliada. Mas, por ser uma plataforma restrita, não permitia a flexibilidade e o alto nível

desejados para o ambiente desta pesquisa, sendo mais adequada ao desenvolvimento de um produto de mercado.

Figura 5 Plataforma Digi Connect ME 9210.

A plataforma SunSpot, da empresa Sun, visível na Figura 6, tem como vantagem permitir a execução de código Java nativamente, mas tratam-se de dispositivos criados para "encorajar" desenvolvedores de Java a desenvolverem aplicativos em Java para sistemas embarcados. Esses dispositivos vêm com alguns sensores integrados (acelerômetros, sensor de temperatura, entre outros) e alguns pontos de entrada/saída (*Input/Output*, I/O), além de uma interface de rádio para comunicação sem fio – uma grande vantagem desses dispositivos. Apesar dos pinos de entrada e saída, a interface com outros dispositivos, que fossem sensores ou outros sistemas embarcados que forneceriam informações para os *web services*, seria um pouco trabalhosa, pois acreditou-se que o desenvolvimento em Linux seria mais fácil. A vantagem na utilização desses dispositivos seria que eles já estavam disponíveis. Além disso, esses dispositivos não se encontram para compra no mercado brasileiro.

Figura 6 Plataforma Sunspot (THE SENSOR NETWORK MUSEUM).

Surgiu a sugestão da plataforma FriendlyArm, que suporta o núcleo (*kernel*) 2.6 do Linux. Tratando-se de uma plataforma relativamente barata por toda sua configuração (analisada mais detalhadamente a seguir), por executar Linux e a possibilidade de executar código em Java (o que não foi possível, conforme verificou-se em testes posteriores à decisão relativa ao hardware a ser utilizado), essa foi a plataforma embarcada escolhida para o projeto.

### 3.1.1 PLATAFORMA FRIENDLYARM

A Plataforma FriendlyArm MINI2440 utilizada possui um processador ARM9 da Samsung, com freqüência de 400 MHz, 256MB de memória Flash e 64MB de memória RAM. Essa é uma configuração de padrão bem elevado para um sistema embarcado, sendo que a placa vem com GNU/Linux instalado, podendo também executar outros SOs, como WinCE e Android, ou mesmo sem suporte algum de SO. Outros periféricos tornam a plataforma bastante atrativa, como a disponibilidade de quatro portas seriais, sendo uma com conector DB9, duas portas *Universal Serial Bus* (USB), interface de rede *Ethernet*, saída de áudio, entrada para cartão de memória *Secure Disk* (SD), um painel *Liquid Cristal Display* (LCD) sensível ao toque e acesso a pinos de I/O de propósito geral (*General Purpose Input Output*, GPIO), interface *Joint Test Action Group* (JTAG), para depuração (debug) reparo/modificação da *Basic Input Output System* (BIOS) do sistema. A Figura 7 mostra a plataforma conectada a um computador através de portas seriais e USB, executando GNU/Linux com um teclado em sua porta USB. Na Figura 1Figura 8 está a plataforma sem o LCD, mostrando os seus componentes internos.

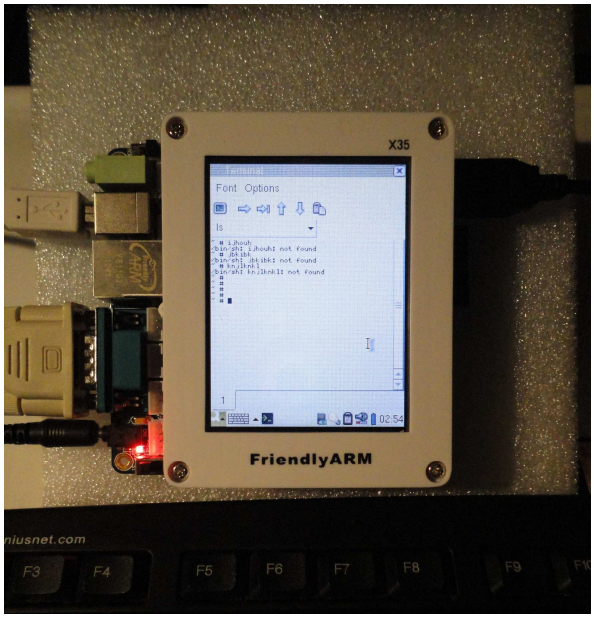


Figura 7 Plataforma FriendlyArm.

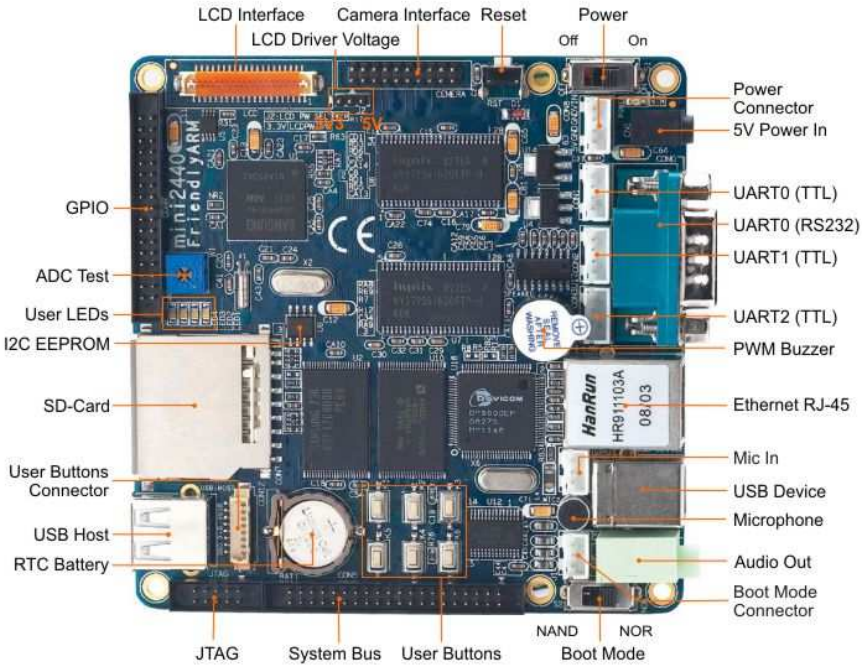


Figura 8 Plataforma FriendlyArm: componentes internos (FRIENDLYARM).

O kit que acompanha a placa possui um cabo serial, um cabo USB, um cabo *Ethernet*, e um cabo JTAG, sendo alguns desses necessários para a interface com um computador para acesso à BIOS e suas funcionalidades, como formatar a memória *flash*, instalação de novos

SOs ou aplicações simples para serem executadas diretamente no processador, executar o programa de teste da placa etc. O kit também inclui um DVD (FRIENDLYARM, DVD FriendlyArm) com a documentação, em chinês, um compilador para compilação cruzada (compilar em uma arquitetura de processador para executar em outra) aplicativos para o ARM (o *arm-none-linux-gnueabi-gcc*) e imagens prontas de SOs, *kernels* do Linux etc. e códigos fonte de aplicações para o FriendlyArm. Também estão presentes aplicativos para a comunicação através de portas USB com a plataforma, com uma boa velocidade de envio de dados.

## 3.2 IMPLEMENTAÇÃO DE DPWS

Algumas implementações para DPWS, ou também chamadas pilhas (no inglês, *stacks*), são apresentadas na página do *Web Services For Devices* (WS4D, Web Services For Devices), uma iniciativa mantida pelas Universidade de Rostock (Alemanha), Universidade de Dortmund (Alemanha) e MATERNA, companhia alemã de *Tecnologia da Informação* e aplicações móveis, que visa a aplicação de SOA e *web services* em dispositvos com recursos limitados, sem perder a interoperabilidade.

Dentre as plataformas encontradas na página, primeiramente foram consideradas as implementações em Java, devido à maior familiaridade com código nessa linguagem de programação. Essas são duas: *Java Multi Edition DPWS Stack* (JMEDS) e *WS4D Axis2 Stack*.

Com a JMEDS é possível implementar e executar serviços, clientes e dispositivos DPWS. Entre suas vantagens estão a de ser código aberto, facilitando o entendimento de seu funcionamento e permitindo assim alterações, e alta compatibilidade, testada com Windows Vista, o stack da Universidade de Rostock e a pilha da empresa Schneider.

O *WS4D Axis2 Stack* utiliza o *Apache Axis2*, sucessor do *Apache Axis*, que implementa SOAP. Comparando com o JMEDS, não apresenta nenhuma grande vantagem e possui configuração um pouco mais complicada.

Entretanto, ao iniciar as atividades com a plataforma de desenvolvimento já escolhida, não foi possível executar com sucesso o JRE específico para sistemas embarcados por falta de memória RAM na plataforma, a qual ainda devia ser compartilhada com o SO. Os requisitos mínimos para plataformas ARM executando Linux, disponíveis no site da Oracle (ORACLE, Java Embedded), sempre eram de 32MB ou mais de memória disponível apenas para o Java, chegando até em valores mínimos de 64MB ou mais.

Restaram, então, duas alternativas de implementações, ambas em linguagem C: WS4D-gSOAP, uma implementação para DPWS da gSOAP *web services toolkit*, que é um *toolkit* para *web services* em SOAP/XML, e o uDPWS (micro DPWS), uma implementação para DPWS para dispositivos com recursos muito restritos. Como o desenvolvimento em Java não foi possível pelo fato de não haverem recursos suficientes, a plataforma uDPWS foi escolhida. Uma justificativa foi utilizar uma plataforma que facilitasse o desenvolvimento, por possuir Linux, mas podendo ser implementada em plataforma de menor porte, possivelmente sem o Linux. Outro fator considerado na escolha do uDPWS foi a existência de exemplos já prontos para plataformas Linux em computadores.

### 3.2.1 uDPWS

O uDPWS é uma pilha, escrita completamente em C, é destinado a dispositivos com poucos recursos de memória e sensores sem fio. Foi desenvolvido de forma enxuta, implementado na dissertação de mestrado de Christian Lerche, levando em conta a criação de dispositivos DPWS que utilizam menos recursos. O uDPWS utiliza o SO de sistemas embarcados Contiki (DUNKELS) para que houvesse um ponto de partida para o

desenvolvimento. O uDPWS é executado em dispositivos Atmel AVR Raven (Figura 9), Crossbow TelosB (Figura 10), ambos sensores sem fio, e computadores ou sistemas embarcados com SO GNU/Linux (como o FriendlyArm), conforme especificado no site do desenvolvedor (WS4D, uDPWS: DPWS for highly resource-constrained devices).


Figura 9 Plataforma AVR Raven (ATMEL).

No caso do uDPWS para GNU/Linux, o SO Contiki é executado em cima desse, sendo criada uma interface de comunicação chamada *tap* para que o uDPWS se comunique com o Linux. A interface *tap* é uma interface de rede virtual, criada pelo *kernel* do sistema, operando complemente em *software*, ao contrário de interfaces de rede, como por exemplo, a *eth*, que representa uma interface de rede *Ethernet* para o Linux.

Figura 10 Plataforma TelosB (PLUMMER).

Os serviços e dispositivos são compilados junto com o núcleo do uDPWS e do Contiki, formando um grande módulo. Um diagrama para representar essa interação é mostrado na Figura 11.

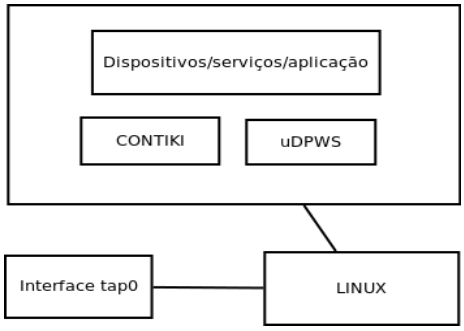


Figura 11 Diagrama da interação entre uDPWS, Contiki e Linux.

As características principais do uDPWS são uma interface para *web services* segundo padrões da W3C através de IP, independência da camada física (*Ethernet*, *ZigBee*, *Bluetooth* etc.), funcionalidade de serviços funcionarem automaticamente através dos métodos de descoberta do DPWS e possibilidade de roteamento, sendo que o serviço de um dispositivo pode ficar disponível em qualquer lugar do mundo se conectado à *internet*.

A Figura 12 representa os módulos do uDPWS. Nessa figura, pode-se dizer que o objetivo do projeto consiste em fazer todo o uDPWS ser executado em uma plataforma embarcada e focar na estruturação de serviços (*Serviço A, Serviço B e Serviço C*, na figura), descrevendo como criá-los.

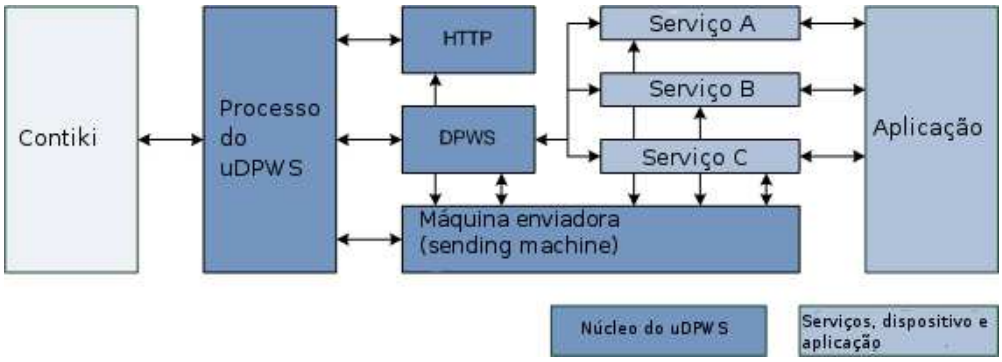


Figura 12 Módulos do uDPWS (adaptado, WS4D, uDPWS: DPWS for highly resource-constrained devices).

Por ser uma implementação que requer tão poucos recursos, o uDPWS não suporta algumas funcionalidades: WS-Eventing, *HTTP Chunked Mode* (transmissão de mensagens

em pedaços) e *HTTP Keep Alive* (mensagens enviadas para verificar se a conexão entre dois pontos está ativa ou para prevenir que ela acabe).

## 3.3 AMBIENTE DE DESENVOLVIMENTO

O kit FriendlyArm possui um compilador cruzado para gerar binários que possam ser executados na plataforma para ambiente Linux, o *arm-none-linux-gnueabi-gcc*. Dessa forma, para não ser necessário buscar outro compilador para ambiente Windows, o SO para desenvolvimento escolhidofoi a distribuição GNU/Linux Ubuntu, versão 10.10 (Maverick Meerkat).

Entretanto, para que erros em algum passo da configuração das ferramentas de desenvolvimento não comprometessem o sistema inteiro, foi utilizada a ferramenta de virtualização de SOs VirtualBox (ORACLE, VirtualBox), onde é possível executar um SO sobre o outro. Com a utilização dessa ferramenta, também seria possível criar “pontos de restauração” do sistema de desenvolvimento, já que ele é apenas um arquivo dentro do SO que está executando esse SO virtualizado. Caso ocorresse algum imprevisto, o problema estaria confinado no sistema virtual.

Para facilitar o manuseio de alguns componentes, o *Integrated Development Environment* (IDE) Eclipse (ECLIPSE FOUNDATION) foi instalado, junto com os *plugins* para trabalhar com a linguagem de programação C/C++. Além desse, ferramentas de linha de comando foram utilizadas para compilação de códigos.

O SO escolhido para executar o VirtualBox com o Linux (pela distribuição Ubuntu, o SO para ser utilizado no computador) foi o Windows 7. Isto pelo fato de o kit FriendlyArm dispor de uma interface gráfica para a comunicação através de USB entre computador e plataforma embarcada. Ferramentas com esse objetivo existem para GNU/Linux, mas são mais difíceis de depurar. O SO que será executado na plataforma embarcada, e a comunicação

com ela, não impuseram nenhuma restrição na escolha dos sistemas operacionais Windows e GNU/Linux.

Instaladas as ferramentas para compilação e compilação cruzada no Ubuntu, *drivers* e aplicativos no Windows, o ambiente de desenvolvimento estava configurado para gerar o necessário para ser executado no sistema embarcado. A Figura 13 mostra o Windows executando a ferramenta VirtualBox, com o SO GNU/Linux Ubuntu, um terminal para acesso a porta serial (para comunicação com o FriendlyArm) e o aplicativo para comunicação através de porta USB (também como o FriendlyArm). No Ubuntu, está sendo executado apenas um editor de texto.

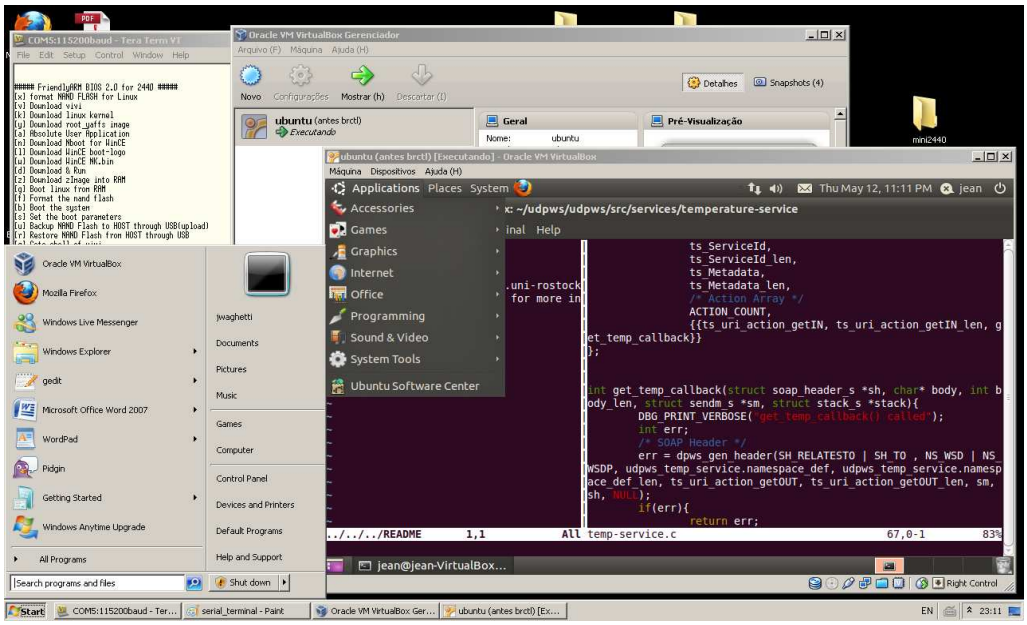


Figura 13 Ubuntu executado através do VirtualBox, no Windows.

# 4 TESTES PRELIMINARES

Nesta seção serão relatados os testes preliminares realizados com o uDPWS, sendo uma aplicação em um computador de mesa e os testes para a plataforma FriendlyArm, para que a instalação de um novo SO (no caso analisado, o Linux já compilado disponibilizado com o kit da plataforma FriendlyArm), seu núcleo (*kernel*) e *bootloader* (software responsável pelo carregamento do sistema quando o hardware é inicializado) sejam possíveis.

A instalação de um SO customizado (um kernel compilado novamente, com algumas funcionalidades extras habilitadas) é necessária devido a requisitos do desenvolvedor do uDPWS para criação de interfaces virtuais de rede, que não estão disponíveis no SO que vem instalado por padrão na plataforma FriendlyArm.

## 4.1 uDPWS

O uDPWS possui um tutorial de como executar um serviço-exemplo onde se lê a temperatura de um dispositivo virtual, para ser executado no SO GNU/Linux Ubuntu. Os passos são descritos na página do uDPWS, onde também se adquire o código fonte que, por sua vez, é livre. Possuindo ferramentas básicas de compilação em C, para a mesma arquitetura onde será executada a aplicação, torna-se uma tarefa bem simples compilar o código. Com a ferramenta GNU Make, que cria executáveis e outros arquivos a partir do código fonte, compila-se o uDPWS e a chamada plataforma *minimal-net*, presente no código fonte, que é a utilizada para que o uDPWS seja executado no Ubuntu.

Com os arquivos, pela linha de comando executa-se a aplicação. É necessário garantir que os privilégios necessários para a criação de uma interface virtual, chamada *tap0*. Isso pode ser feito, por exemplo, através da elevação dos privilégios do usuário para *root* (administrador em sistemas Linux).

Com o uDPWS executando o serviço de temperatura, é necessário um cliente de DPWS para haver a interação. Para tanto, foi utilizado o DPWS Explorer (WS4D, Tool: DPWS Explorer), um software com interface gráfica escrito em Java, que utiliza o DPWS *stack* JMEDS, que interage com dispositivos habilitados com DPWS.

O DPWS Explorer, não recebe automaticamente os pacotes enviados pelo uDPWS (para a descoberta de serviços), devido a interface padrão definida não ser a mesma que a do DPWS Explorer, conforme o desenvolvedor do uDPWS. Executando o seguinte comando, cria-se a rota necessária para que os pacotes cheguem ao DPWS Explorer:

```
$ sudo ip route add 224.0.0.0/4 dev tap0
```

Então é só fazer com que o DPWS Explorer procure por dispositivos e ele achará o dispositivo de temperatura, conforme mostra a Figura 14, onde a descrição do serviço está sendo exibida.

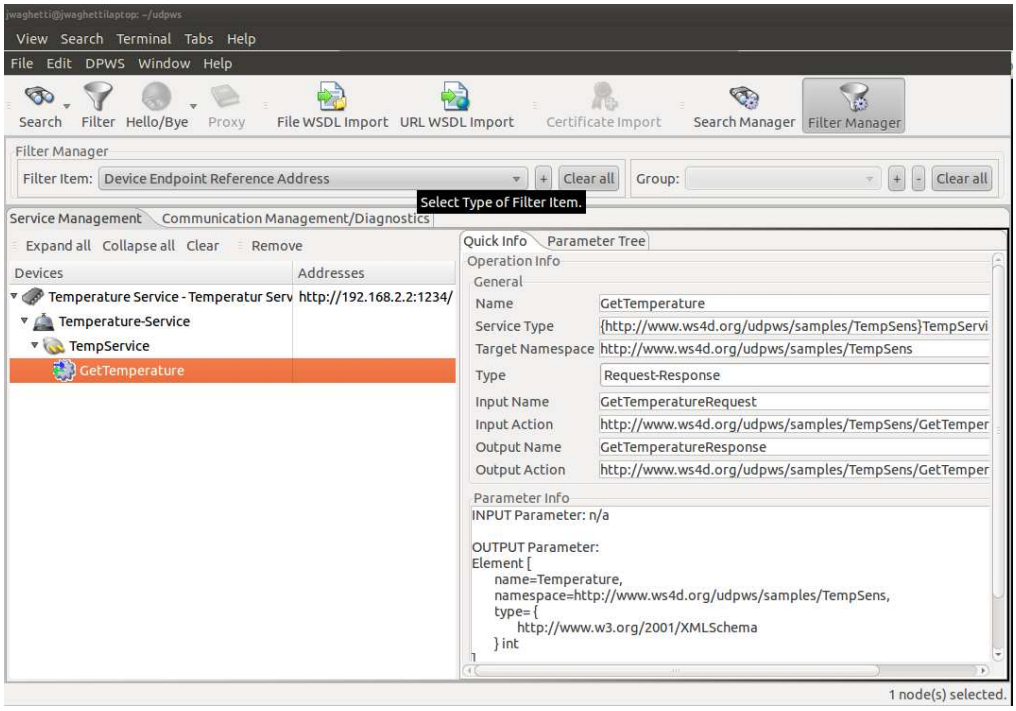


Figura 14 DPWS Explorer interagindo com o uDPWS.

Pode-se observar na Figura 14 um dispositivo chamado *Temperature Service*, que possui um serviço chamado *Temperature-Service*, que utiliza a porta *TempService*, e uma ação chamada *GetTemperature*. A descrição do serviço através de URIs, presentes na WSD, que o definem estão à direita na figura. Invocando essa ação, o dispositivo retorna o número 23, que representa a temperatura definida no código do uDPWS. Isso é acessível através da aba *Parameter Tree*, do painel do lado direito, onde está selecionada a aba *Quick Info*, que mostra as informações do serviço.

Uma análise mais detalhada do serviço será feita ao se criar um serviço, que consiste em um dos objetivos desse projeto.

## 4.2 FRIENDLYARM

Nesta seção serão descritos os passos utilizados para que um SO seja instalado na plataforma FriendlyArm.

A plataforma FriendlyArm possui duas memórias flash: NAND e NOR. A memória NAND é utilizada para gravar o SO ou aplicação que será executada. A memória NOR possui a BIOS da placa, acessível através da porta serial por um computador, onde existem várias ferramentas para gerenciamento da memória *flash*, *download* de *bootloader*, SO etc., *backup* e recuperação da memória.

Através de um terminal serial no Windows, com um cabo serial e um cabo USB ligados entre o PC e o FriendlyArm, e a memória selecionada para NOR, através de uma chave específica para isso, a BIOS do FriendlyArm foi acessada. Para a transferência das imagens, a aplicação de transferência USB que acompanha a plataforma, chamada DNW, foi utilizada. O acesso à BIOS é feito pela porta serial. A porta USB é utilizada apenas para transferência de dados com maior velocidade entre o computador e a plataforma embarcada.

Para instalar as imagens de Linux que são distribuídas no DVD que acompanha o kit, foram utilizados os passos descritos a seguir, todos disponíveis na BIOS do sistema mostrada na Figura 15. As imagens enviadas para o FriendlyArm estavam todas localizadas no diretório *"images/Linux"* no DVD do kit.

A compreensão dessa instalação é necessária, pois uma imagem nova do *kerne*l do Linux deve ser compilada, habilitando funcionalidades ausentes na imagem padrão distribuída com a plataforma.

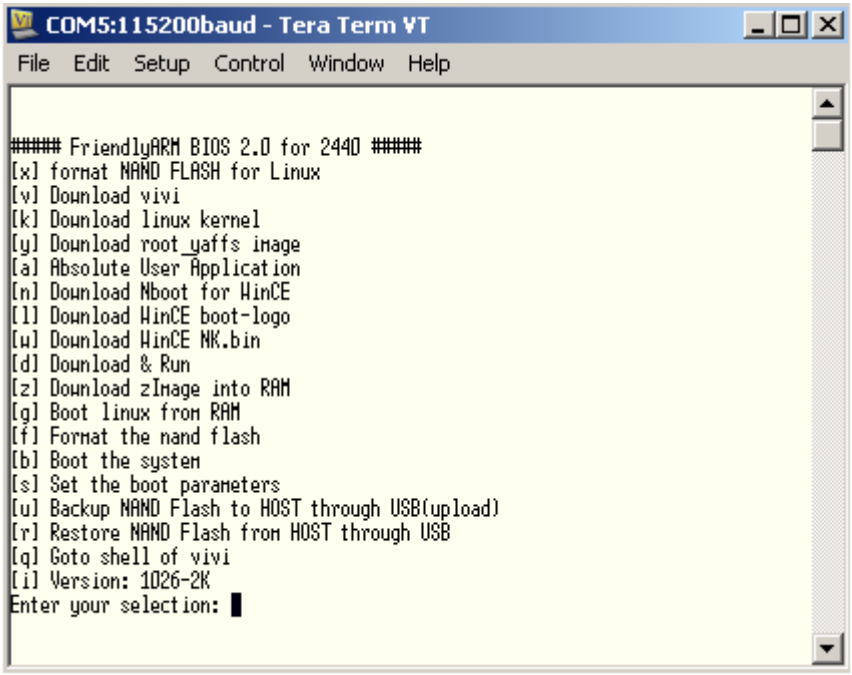


Figura 15 Terminal serial acessando a BIOS da plataforma FriendlyArm.

- formatação da memória NAND: opção [f] da BIOS. Após selecionar essa opção, é apenas necessário esperar a formatação e a mensagem de término da operação.

- criar partição para Linux: opção [p] da BIOS. Da mesma forma que no passo anterior, é apenas necessário esperar a operação ser completada.

- instalar o bootloader: opção [v] da BIOS. Agora é necessário fazer o *download* da imagem do *bootloader* através da porta USB para a placa. Após selecionar essa opção, a

BIOS informará que está esperando a transferência. Através do software DNW, a imagem *supervivi-64M* foi transferida por USB para o FriendlyArm.

- instalar o *kernel* do Linux: opção [k] da BIOS. Da mesma forma que o *bootloader*, é necessário enviar o arquivo pelo software DNW. Existem várias imagens do *kernel* do Linux no DVD que acompanha o a plataforma FriendlyArm, cada uma referente a um tipo de LCD utilizado (outras versões do FriendlyArm possuem *displays* diferentes). A imagem utilizada foi *zImage_x35*, que corresponde à imagem do *kernel* do Linux referente ao LCD da placa adquirida.

- instalar o SO: opção [y] da BIOS. A imagem do Qtopia, o ambiente gráfico do SO (QTOPIA), chamada *rootfs_qtopia_qt4.img*, também foi enviada via USB. Isso é necessário pois apenas o *kernel* do Linux não se constitui em um SO completo, faltando ainda uma interface com o usuário – nisso se constitui o Qtopia, além de uma interface gráfica que utiliza os recursos do LCD.

Após esses passos, o sistema foi desligado e a chave de seleção da memória foi reposicionada na posição NAND. Ligando o sistema, o Linux recém instalado estava sendo executado. A forma de instalação de um SO, com imagens já compiladas, já era entendida.

## 4.3 COMPILAÇÃO CRUZADA

Com o ambiente de desenvolvimento e a plataforma embarcada em funcionamento, algumas aplicações simples foram testadas. Pequenos códigos em C foram escritos, utilizando recursos como I/O em arquivos e linha de comando. Essas aplicações foram compiladas no ambiente de desenvolvimento com as ferramentas de compilação cruzada.

Os programas gerados para testes foram executados corretamente na plataforma embarcada, permitindo verificar, de forma indireta, que o compilador estava corretamente configurado.

# 5 EMBARCANDO O UDPWS

Após os testes preliminares, e entendida a forma de funcionamento do uDPWS em ambiente GNU/Linux, a tarefa de gerar executáveis para a plataforma FriendlyArm teve início. O uDPWS é apenas uma aplicação. Gerando-se seu arquivo, ele deve ser transportado para o sistema embarcado e inicializado como uma aplicação qualquer.

Três passos principais foram necessários para que a aplicação fosse executada na plataforma.

Primeiramente, o próprio uDPWS devia ser compilado, este já possuindo o serviço-exemplo de temperatura. Como este requer a criação de interfaces de rede virtuais, a *tap0*, o *kernel* do Linux teve que ser compilado para que essa fosse permitido à aplicação criar essa interface, e também para que as funcionalidades para criação de pontes (*bridges*) entre as interfaces virtual e física fosse possível, com o objetivo de fazer com que as interfaces física e virtual de rede se comportem como apenas uma interface (isso será analisado a seguir). Para as configurações de *bridge*, um software chamado Bridge-Utils (BUYTENHEK; HEMMINGER) foi utilizado.

## 5.1 COMPILAÇÃO DO UDPWS

O uDPWS tem seu código escrito em C. A árvore de diretórios tem uma parte destinada ao código específico de cada plataforma suportada. A de interesse é a chamada *minimal-net*, para ser utilizada em Linux. Como há o código do Contiki, do núcleo do uDPWS, dos dispositivos, dos serviços e das plataformas em locais diferentes, existem *makefiles* (arquivos de configuração para a ferramenta de compilação GNU Make) espalhados por toda essa árvore de diretórios.

Utilizou-se o IDE Eclipse, onde os códigos-fonte foram inseridos e o compilador configurado para o *arm-none-linux-gnueabi-gcc*, para efetuar a compilação cruzada.

Após ser compilado com sucesso, o arquivo binário gerado foi executado na plataforma FriendlyArm através de um cartão de memória do tipo SD. A alternativa do cartão SD foi escolhida apenas pela facilidade de trocar arquivos entre o sistema embarcado e o computador de desenvolvimento e também aumentar a capacidade de memória da plataforma. O sistema ainda não estava pronto para executar a aplicação de forma adequada, mas já era possível verificar se o binário gerado seria executado corretamente na arquitetura, caso contrário haveria uma simples mensagem de erro e ela se encerraria.

## 5.2 COMPILAÇÃO DO BRIDGE-UTILS

O Bridge-Utils é utilizado para conectar interfaces *Ethernet*, criando interfaces virtuais que englobam as interfaces conectadas através de pontes (*bridges*). Ou seja, as interfaces englobadas pela *bridge* tem uma forma de comunicação entre si, e não agem mais apenas como interfaces separadas. Os dados que circulam por uma, estão disponíveis para outra e vice-versa.

No caso da aplicação do uDPWS, servirá para conectar a interface Ethernet física da placa, chamada de *eth0* no Linux, com a interface virtual criada pelo uDPWS, chamada *tap0*. Isso pode ser melhor visualizado na Figura 16, onde as interfaces física e virtual são englobadas pela *bridge*, como se fossem uma interface de rede apenas, que se comunica externamente através da interface de rede física. Sem a criação dessa *bridge*, a aplicação não se comunicaria através da interface de rede da plataforma embarcada.

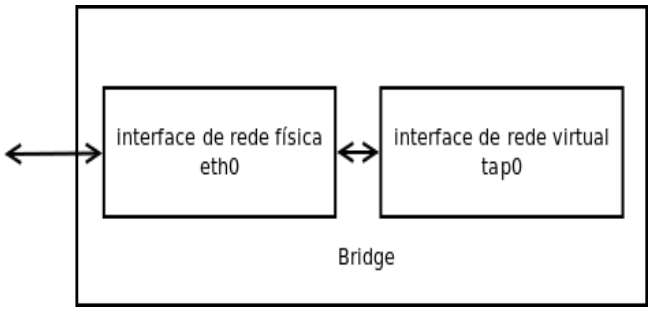


Figura 16 Diagrama da *bridge* criada.

Para a compilação do Bridge-Utils, a ferramenta *autoconf* é utilizada. Essa faz com que os *makefiles* sejam gerados de acordo com parâmetros do sistema onde haverá a compilação, como configurar o compilador de linguagem C, verificar quais comandos do sistema servem para determinadas funções etc.

Após executar o *autoconf*, os *makefiles* gerados estão configurados para serem executados na plataforma de desenvolvimento, não na plataforma alvo desse projeto. Então, os *makefiles* foram alterados para que o compilador utilizado fosse o *arm-none-linux-gnueabi-gcc*, ao invés do compilador padrão configurado. Ao contrário do uDPWS, onde o projeto foi compilado no IDE Eclipse, alteraram-se os *makefiles* por esse ser uma aplicação com bem menos arquivos de código-fonte e apenas dois *makefiles*, onde o compilador a ser utilizado estava sempre explícito no início do arquivo e a alteração era relativamente simples.

Com o arquivo binário gerado, da mesma forma que o uDPWS, esse foi executado na plataforma FriendlyArm através de um cartão SD, através do comando *brctl* (para utilizar o programa Bridge-Utils). Mensagens de erro coerentes foram mostradas pelo fato de o *kernel* do Linux que estava instalado não suportar as funcionalidades requeridas, faltando então habilitar no *kernel* as funcionalidades requeridas pelo Bridge-Utils.

## 5.3 COMPILAÇÃO DO KERNEL DO LINUX

Apesar de a plataforma embarcada não vir com um SO que tenha suporte à criação de interfaces de rede virtuais ou à configuração de *bridges*, o kit que a acompanha possui um DVD que tem o código fonte do *kernel* 2.6 do Linux, que pode ser também encontrado na *internet*. Com o código fonte em mãos, é possível criar uma imagem do *kernel* que possua essas funcionalidades.

Não é necessário modificar nada no código fonte em si, apenas no arquivo de configuração a ser utilizado para compilá-lo. Existem vários arquivos de configuração que são

referentes aos diversos tipos de FriendlyArms (com diferentes configurações de periféricos e LCD), que encontram-se na pasta raiz do código do *kernel*. Esses arquivos estão presentes no código distribuído com o kit do FriendlyArm, sendo que um código do *kernel* do Linux obtido por outras fontes não os possuiria. Escolhendo o correto (no caso do FriendlyArm de que se dispunha, o arquivo chama-se *config_mini2440_n35*), renomeia-se o arquivo para *.config* e com o comando "*make menuconfig"* altera-se as configurações de TUN e BRIDGE (para procurá-las no menu, se digita uma barra "/" e o nome da configuração. Com um editor de texto, verifica-se que, no arquivo *.config*, as linhas:

```
# CONFIG_TUN is not set
# CONFIG_BRIDGE is not set
```

foram alteradas para

```
CONFIG_TUN=y
CONFIG_BRIDGE=y
```

Com o arquivo de configuração devidamente acertado, a compilação do *kernel* se dá com o comando (na pasta raiz do código, onde se encontra o arquivo *.config*):

```
$ CROSS_COMPILE={CAMINHO}/arm-none-linux-gnueabi- ARCH=arm make
```

onde o parâmetro {CAMINHO} se refere ao caminho até o diretório onde se encontram os arquivos binários do compilador para arquitetura ARM. O processo de compilação do *kernel* é um pouco demorado. Ao finalizar, caso não haja nenhum erro, o arquivo *zImage* se encontra na pasta *"arch/arm/boot/*", relativo ao local onde se executa o comando *make*. Essa é a imagem do *kernel* que será enviada ao FriendlyArm, como descrito na seção 4.2, onde foram descritos os testes preliminares com a plataforma FriendlyArm. As imagens do *bootloader* e do Qtopia podem ser as pré-compiladas, e que estão presentes no DVD que acompanha o kit.

# 6 EXECUTANDO O UDPWS

Com o *kernel* e o Bridge-Utils configurados, é possível executar o uDPWS. Para isso, o primeiro passo é configurar a variável de ambiente PATH para que ela possua o caminho até os arquivos binários do uDPWS e do Bridge-Utils e esses possam ser executados apenas com os comandos *"udpws"* e *"brctl"*, sem ser necessário digitar o caminho completo até os executáveis. Isso é feito com o comando:

```
$ export PATH=$PATH:{CAMINHO}
```

onde *{CAMINHO}* representa o caminho na sistema de arquivos até os binários do uDPWS e do Bridge-Utils (esse comando deve ser executado para ambos programas).

Para executar o uDPWS, então, executa-se o comando referente ao nome do arquivo binário gerado. Assim, a interface *tap0* é criada e é necessário configurá-la como Bridge-Utils. Isso é feito em alguns passos:

- altera-se o endereço IP da interface de rede Ethernet para que fique no modo promíscuo (para receber todos pacotes da mesma rede, não apenas os endereçados à interface) e sem IP, com o comando:

```
# ifconfig eth0 0.0.0.0 promisc up
```

- da mesma forma, altera-se o endereço IP da interface *tap0*:

```
# ifconfig tap0 0.0.0.0
```

- e configura-se a *bridge*, criando a *bridge br0*, adicionando as interfaces *eth0* e *tap0* e então configurando o endereço IP:

```
# brctl addbr br0
# brctl addif br0 eth0
# brctl addif br0 tap0
# ifconfig br0 0.0.0.0 up
```

E assim, o uDPWS já está pronto para se comunicar com o computador.

# 7 TESTES POSTERIORES

Com o uDPWS sendo executado na plataforma embarcada, e pronto para comunicação com o computador, um cabo *Ethernet* foi utilizado para conectar as interfaces de rede dos dois dispositivos. O uDPWS possui aplicativos de teste, chamados *testclients*, que foram utilizados para testar a comunicação entre o computador e a plataforma embarcada, encontrados na pasta com mesmo nome nos arquivos fonte do uDPWS.

O computador foi configurado com IP fixo, no caso, o endereço foi 10.0.0.2, com máscara de rede 255.255.255.0, pois o IP do uDPWS sendo executado na plataforma embarcada foi 10.0.0.12 com a mesma máscara de rede.

Para os *testclients* serem executados, é necessário compilá-los. O *testclient* utilizado foi o *http4-client*, que acompanha o código do uDPWS. Para compilá-lo, o comando *make* deve ser executado no diretório referente a esse *testclient* da árvore de diretórios do uDPWS. A utilização do *testclient* é a seguinte:

```
$ ./main [IP] [PORT] [FILE]
```

onde *"./main"* faz o arquivo compilado do *testclient* ser executado com os argumentos entre colchetes. [IP] é o endereço IP do uDPWS (no caso do sistema embarcado, 10.0.0.12), [PORT] a porta utilizada (no caso, 1234), e [FILE] é o arquivo de teste utilizado que representa a mensagem que será trocada com o serviço. Esses arquivos encontram-se na pasta *"testfiles"*, presente na pasta dos *testclients*.

Executando o *testclient* com o arquivo de teste chamado *gettemp.xml*, obtém-se a seguinte resposta:

```
$ ./main 10.0.0.12 1234 ../testfiles/gettemp.xml
client started...

Waiting for answer...
Read 731 bytes:
HTTP/1.0 200 OK
```

```
Content-Type: application/soap+xml; charset=UTF-8
Content-Length: 640

<?xml version='1.0' encoding='UTF-8' ?><s:Envelope  xmlns:s=http://www.w3.org/2003/05/soap-envelope
xmlns:wsa="http://www.w3.org/2005/08/addressing" xmlns:wsd="http://docs.oasis-open.org/ws-
dd/ns/discovery/2009/01" xmlns:wsdp="http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01"
xmlns:t="http://www.ws4d.org/udpws/samples/TempSens"><s:Header><wsa:RelatesTo>urn:uuid:72816260-09a5-11df-
bf64-
9e807976fa2c</wsa:RelatesTo><wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To><wsa:Action>
http://www.ws4d.org/udpws/samples/TempSens/GetTemperatureOut</wsa:Action></s:Header><s:Body><t:Temperat
ure>23</t:Temperature></s:Body></s:Envelope>
socket closed... finish now
Roundtrip: 0 sec, 35245 usec
```

Que corresponde ao documento XML enviado pelo serviço com o valor fictício de temperatura lido pelo serviço, ou seja, 23.

Com isso, observou-se que a comunicação entre plataforma embarcada estava, de fato, ocorrendo.

Para facilitar a execução da aplicação, *Shell scripts* (*scripts* para o interpretador de comandos do SO) foram criados. Eles servem para configurar o SO, reconhecendo os arquivos do uDPWS e do Bridge-Utils e para configurar a *bridge* entre as interfaces de rede física e virtual. Esses *scripts* encontram-se nos apêndices.

# 8 CRIAÇÃO DE UM NOVO SERVIÇO E SUA DESCRIÇÃO

Após a plataforma executar o exemplo que acompanha o código do uDPWS com sucesso, a criação de um novo serviço foi iniciada

A compreensão da estrutura do código do uDPWS, e como se organizam os dispositivos e serviços foi o primeiro passo. Isso será descrito na próxima seção, seguida pela definição de o que será o serviço, especificando como ele foi implementado e, por último, os testes do serviço em funcionamento.

## 8.1 ESTRUTURA DOS DISPOSITIVOS E SERVIÇOS NO UDPWS.

Dentro da árvore de diretórios do código fonte do uDPWS, existem pastas específicas onde se dividem os serviços, dispositivos, o uDPWS em si e o código referente a cada plataforma onde o uDPWS pode ser executado.

A parte de interesse para a criação de novas aplicações se restringe aos serviços e dispositivos. Com o caminho relativo à raiz do código do uDPWS, essas partes encontram-se nas pastas "*src/devices/*" e *"/src/services/"*. Nestas pastas encontram-se as subpastas "*temp-device"* e *"temperature-service"*, que representam o dispositivo e o serviço, respectivamente, utilizados anteriormente.

Em cada uma dessas pastas, do serviço e dispositivo, há a subpasta *"gen"*, onde existem arquivos que representam as "partes" do serviço/dispositivo a ser criado. O uDPWS utiliza uma aplicação, que acompanha o código, chamada *file2c*, que serve para transformar esses arquivos de texto em estruturas (*structs*) da linguagem de programação C. Esses arquivos das pastas *"gen"* é que devem ser corretamente configurados para que os metadados sejam coerentes, pois a descrição em WSDL completa de todo o dispositivo e serviço é feita pelo uDPWS a partir desses arquivos.

O código fonte do dispositivo consiste, praticamente, apenas na criação da estrutura da linguagem C referente ao dispositivo. Já o código fonte do serviço possui a função que define a ação de requisição da temperatura, além da criação da estrutura da linguagem C referente ao serviço.

## 8.2 NOVO SERVIÇO: CONTADOR

A partir do código do dispositivo e do serviço de temperatura, um novo serviço que implementa um contador foi idealizado. Esse serviço seria baseado no serviço de temperatura, com a diferença de que, ao invés de sempre retornar a temperatura fictícia 23, retornaria um número de 0 a 9, sequencialmente, a cada vez que a ação fosse requisitada – além da modificação da descrição do serviço. O serviço de temperatura não possui nenhum parâmetro de entrada. Dessa forma, uma funcionalidade que empregasse um valor passado ao serviço seria uma forma de conhecer e utilizar mais recursos do uDPWS. Então, através desse parâmetro, representado por um número inteiro, a funcionalidade do serviço ficou definida como:

- parâmetro maior ou igual a zero: contador é incrementado até 9, voltando para 0 na próxima ação;

- parâmetro menor que -100: contador é reinicializado e o parâmetro de retorno é 0.

- caso contrário: o parâmetro de retorno é o parâmetro de entrada acrescido de 5.

A criação desse serviço que implementa um contador utilizou o código fonte do serviço de temperatura como base. Apenas o conteúdo dos códigos fonte e dos arquivos das pastas “gen”, utilizados para criar as estruturas utilizadas pelo uDPWS, foram alterados.

Dessa forma, nos códigos fonte da linguagem C, foi apenas alterado o valor de retorno da função, que ao invés de ser um número constante, varia a cada ação e depende do parâmetro de entrada. O código do serviço, comentado, se encontra nos apêndices.

Os arquivos contidos nas pastas “gen” devem ser modificados conforme o serviço e o *namespace* definido (*namespace* é uma forma de “separar” um identificador para que ele seja único dentro daquele *namespace*, podendo haver um identificador com mesmo nome, mas completamente diferente, fora desse *namespace*). Os nomes dos arquivos contidos nessa pasta podem ser quaisquer, mas devem ser coerentes com os contidos no arquivo de código de linguagem C referente, onde se utiliza esses arquivos para a criação das estruturas utilizadas no uDPWS. Por conveniência, não se modificou os nomes. A descrição do dispositivo continuou sendo a mesma, focando-se apenas na alteração dos serviços. Nenhuma alteração foi necessária no dispositivo pelo fato de os serviços serem agregados ao dispositivo de maneira “automática” pelo uDPWS, estando esses definidos na lista de serviços do dispositivo.

Para as URIs específicas do serviço, onde eram utilizados, por exemplo *“http://www.ws4d.org/udpws/samples/TempSens”*, foi utilizido "jean/CounterDevice". O prefixo jean foi utilizado por se tratar apenas de um exemplo e não ser um serviço “disponibilizado” ao público.

Desses arquivos modificados, todos seguem um mesmo padrão, mas o mais importante é o *“cs_Metadata”* (cujo arquivo original chamava-se *“ts_Metadata”*, essa alteração foi feita apenas para que o nome ficasse coerente com o serviço, sendo *cs* proveniente de *counter-service*), que define os metadados que o serviço irá trocar com o cliente. A partir do entendimento da estrutura desse arquivo, a alteração, conforme a necessidade, dos outros arquivos é intuitiva. O conteúdo desse arquivo é analisado mais profundamente na próxima seção.

## 8.3 SERVIÇO CONTADOR: ANÁLISE DOS METADADOS

O arquivo *"cs_Metadata"*, referente ao serviço, possui o arquivo XML que será trocado contendo os metadados do serviço. Nele são definidos as URIs que representam o serviço e os padrões utilizados, os tipos de ações e os tipos de dados envolvidos etc.

Cada parte será analisada a seguir. O documento inteiro se encontra nos apêndices.

O arquivo é iniciado com:

```
<wsdl:definitions>
```

Que informa que essas são as definições da WSDL utilizada para o serviço. Logo após tem-se:

```
targetNamespace="jean/CounterDevice"
 xmlns:tns="jean/CounterDevice"
 xmlns:wsa="http://www.w3.org/2005/08/addressing"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
 xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">
```

onde se definem o namespace do document XML e as URIs para o WS-Addressing (*wsa*), WSDL (*wsdl*), WS-Eventing (*wse*) e o padrão SOAP utilizados (*wsoap12*). Essas URIs são definidas no padrão OASIS DPWS 1.1. Continuando no arquivo, encontra-se:

```
<wsdl:types>
        <xs:schema
                targetNamespace="jean/CounterDevice"
                xmlns:tns="jean/CounterDevice"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                elementFormDefault="qualified" blockDefault="#all">
               <xs:element name="Counter" type="xs:int" />
        </xs:schema>
</wsdl:types>
```

onde são definidos os tipos desse arquivo de descrição. Define-se que será utilizado o namespace definido já anteriormente, a URI do XML Schema (que define o que deve haver

em um documento XML para que ele seja considerado válido), que todos elementos de todos blocos são qualificados (ou seja, que pertencem a um namespace) e que há um elemento chamado *Counter* e outro do tipo *CounterInput*, ambos do tipo número inteiro (*int*). Após isso, os tipos de mensagens são definidos:

```
<wsdl:message name="GetCounterIn">
        <wsdl:part name="parameters" element="tns:CounterInput" />
 </wsdl:message>
<wsdl:message name="GetCounterOut">
        <wsdl:part name="parameters" element="tns:Counter" />
</wsdl:message>
```

Existem a mensagem que requisita o contador (*GetCounterIn*), que possui um parâmetro de entrada, chamado *CounterInput*, e é a ação do cliente para requisistar o serviço, e a mensagem de retorno do contador (*GetCounterOut*), a resposta do serviço ao cliente, que é o elemento *Counter* que, assim como *CounterInput*, foi definido anteriormente no documento. A porta utilizada pelo serviço é definida:

```
<wsdl:portType name="CounterService" wse:EventSource="false">
        <wsdl:operation name="GetCounter">
                <wsdl:input message="tns:GetCounterIn"
                        wsa:Action="jean/TempSens/GetCounterIn" />
                <wsdl:output message="tns:GetCounterOut"
                        wsa:Action="jean/TempSens/GetCounterOut" />
        </wsdl:operation>
</wsdl:portType>
```

A porta possui o nome *CounterService* e não é fonte de eventos (pelo WS-Eventing), já que o contador só é incrementado através da requisição do seu valor. Essa porta possui a operação *GetCounter*, que utiliza as mensagens definidas anteriormente no documento XML como entrada e saída. As URIs dessas mensagens são definidas para que sua implementação seja conhecida. O padrão DPWS 1.1 (OASIS, 2009), define que cada porta precisa uma

vinculação para que mensagens SOAP versão 1.2 sejam transmitidas sobre HTTP. Essa vinculação é definida nesta parte do documento XML:

```
<wsdl:binding name="CounterServiceSoap12Binding" type="tns:CounterService">
        <wsoap12:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="GetCounter">
                <wsoap12:operation />
                <wsdl:input>
                        <wsoap12:body use="literal" />
                </wsdl:input>
                <wsdl:output>
                        <wsoap12:body use="literal" />
                </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
```

Onde a vinculação (*binding*) da porta *CounterService* é feita à camada de transporte definida pela URI *http://schemas.xmlsoap.org/soap/http*. Por último, é definido o serviço em si:

```
<wsdl:service name="CounterService">
        <wsdl:port name="CounterPort" binding="tns:CounterServiceSoap12Binding">
                <wsoap12:address location="" />
        </wsdl:port>
</wsdl:service>
```

O nome do serviço é *CounterService*, que utiliza a porta *CounterPort*, vinculada ao “CounterServiceSoap12Binding”, ambos definidos anteriormente no documento. O endereço é vazio, pois esse serviço não requer o reencaminhamento para lugar algum. O processamento da informação é feito localmente. Assim, o serviço está com os metadados complemente descritos, e o documento se encerra com o fechamento do documento de WSDL:

```
</wsdl:definitions>
```

## 8.4 SERVIÇO CONTADOR EM FUNCIONAMENTO

Com os metadados alterados conforme a seção anterior, e os arquivos utilizados pelo uDPWS para criação das estruturas da linguagem C alterados coerentemente, a aplicação foi compilada novamente para o novo serviço ser validado. A seguir o DPWS Explorer foi utilizado novamente para a visualização do serviço. Na Figura 17, há uma captura de tela com as informações do serviço. Na Figura 18, as informações da porta e, na Figura 19, as informações da ação relativas ao serviço. Verificou-se que as informações estavam em conformidade com as alterações feitas nos arquivos.

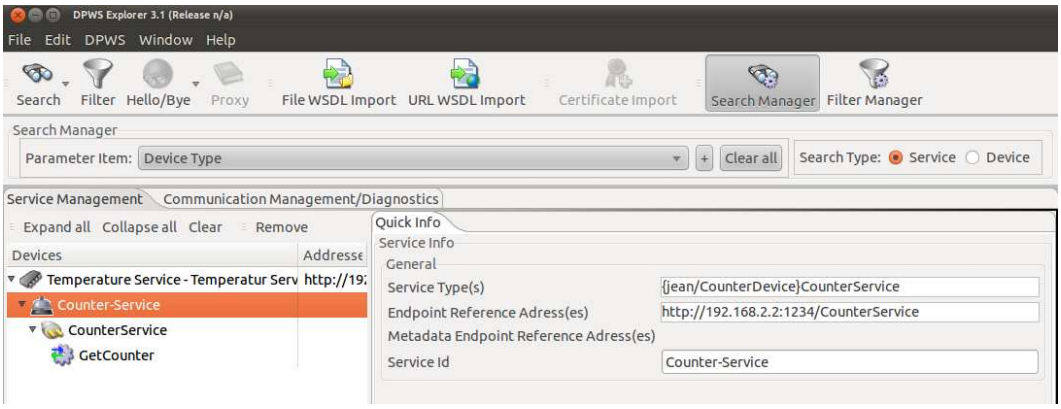


Figura 17 Serviço contador: serviço.

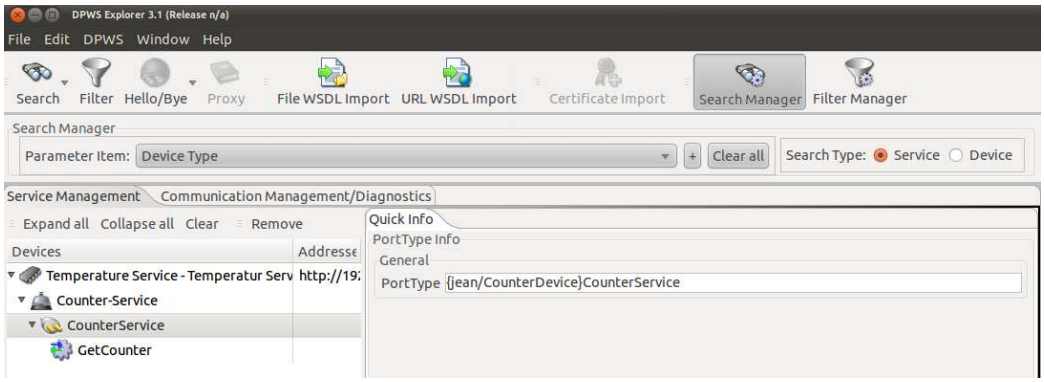


Figura 18 Serviço contador: porta.

O comportamento do serviço foi verificado primeiramente, com duas ações que possuíam um parâmetro de entrada positivo. Primeiramente com o valor de entrada arbitrário

21, cujo resultado pode ser visualizado na Figura 20 e, logo após, com o valor de entrada arbitrário 53, visualizado na Figura 21. Verificou-se que o contador estava sendo incrementado corretamente.

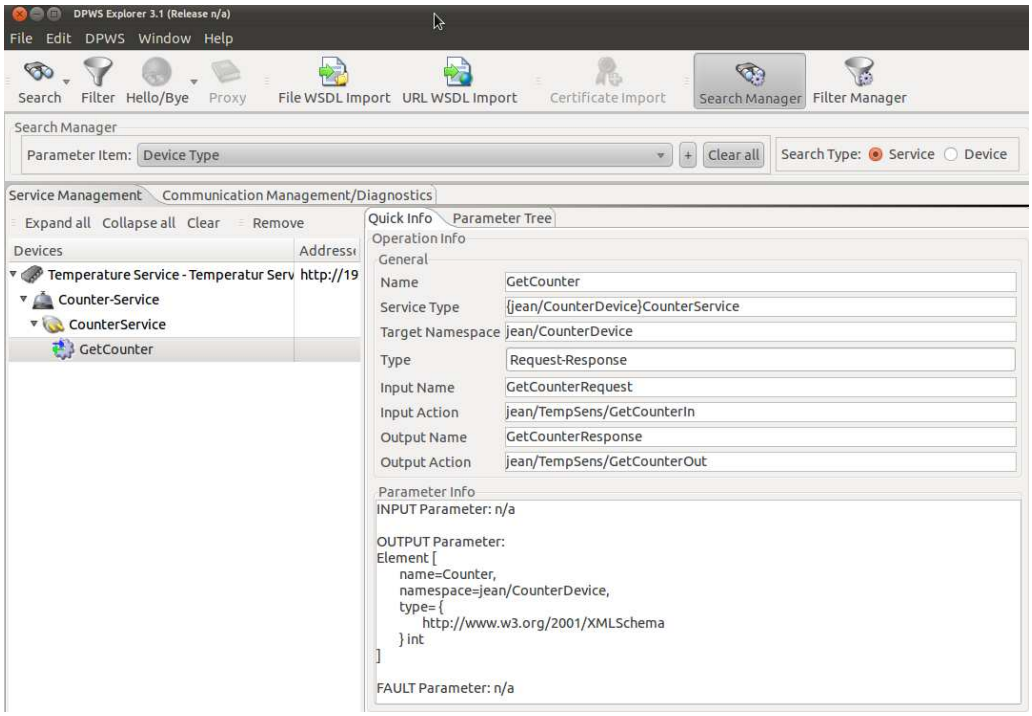


Figura 19 Serviço contador: ação.

Então se utilizou um valor negativo arbitrário de -127, menor que -100 para que o contador fosse zerado. O que, de fato, ocorreu, como pode ser verificado na Figura 22.

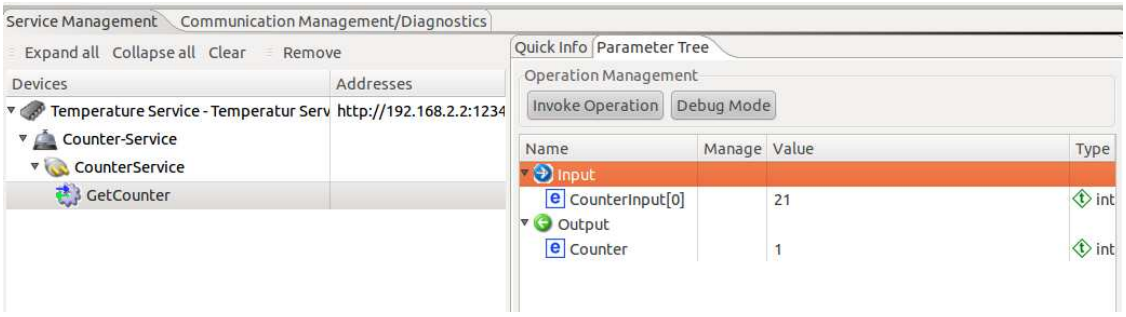


Figura 20 Serviço contador: parâmetro de entrada 21.

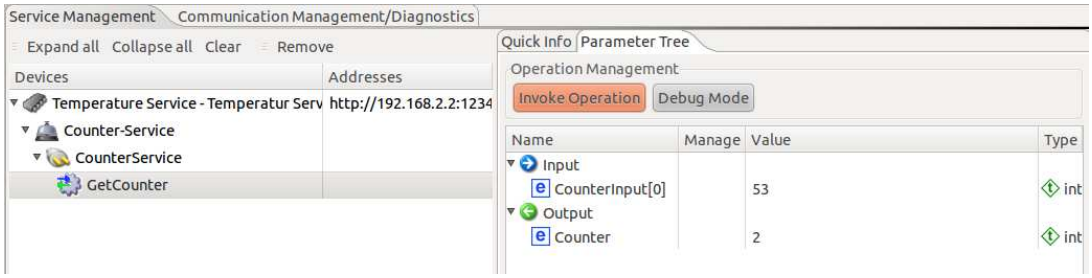


Figura 21 Serviço contador: parâmetro de entrada 53.

Finalmente, um valor negativo maior que -100, arbitrado em -9, foi utilizado como parâmetro de entrada. A resposta foi o número -4, conforme a Figura 23. Ou seja, a entrada acrescida de 5, o que permite concluir que o serviço criado tem as funcionalidades especificadas.

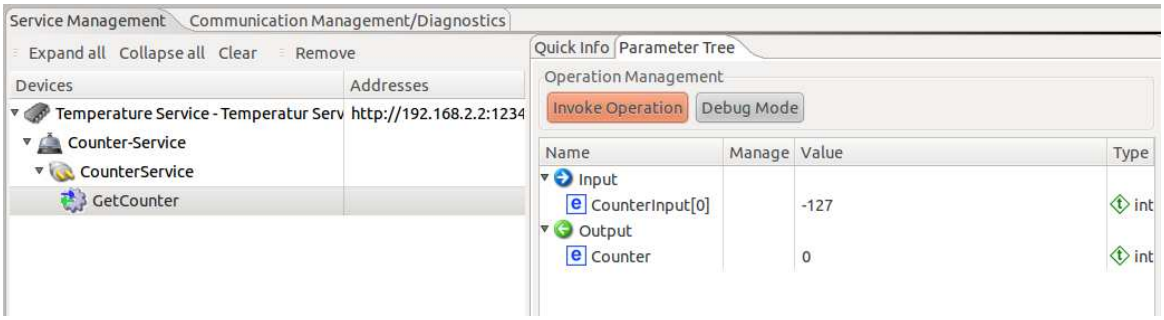


Figura 22 Serviço contador: parâmetro de entrada -127.

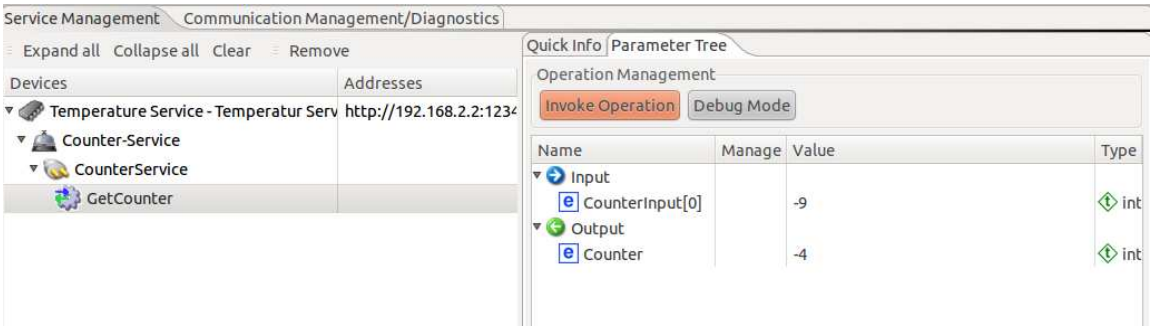


Figura 23 Serviço contador: parâmetro de entrada -9.

## 8.5 VÁRIOS SERVIÇOS EM UM DISPOSITIVO

Até então, apenas foi criado um serviço e este foi trocado no dispositivo pelo serviço exemplo que serviu de base para a sua construção. É possível que um dispositivo possa oferecer vários serviços distintos, e o uDPWS deve permitir isso de forma bastante modular.

O serviço criado se encontrava em uma subpasta da pasta *"services"* chamada *"counter-service"*, e o serviço de temperatura, que serviu de base para o novo serviço, na pasta *"temperature-service"*. Para que os dois serviços estivessem disponíveis no dispositivo, alteraram-se os arquivos referentes ao *makefile* e o código de linguagem C do dispositivo.

No arquivo do *makefile*, apenas inclui-se os diretórios referentes aos serviços e o *makefile* de cada serviço. No código C, deve-se configurar corretamente o número de serviços e ponteiro referente à estrutura de cada serviço, juntamente com a declaração destes.

Assim, obtém-se um dispositivo com dois serviços distintos, conforme pode se observar na Figura 24, onde o DPWS Explorer está mostrando os dois serviços, sendo que a ação do serviço de temperatura foi chamada.

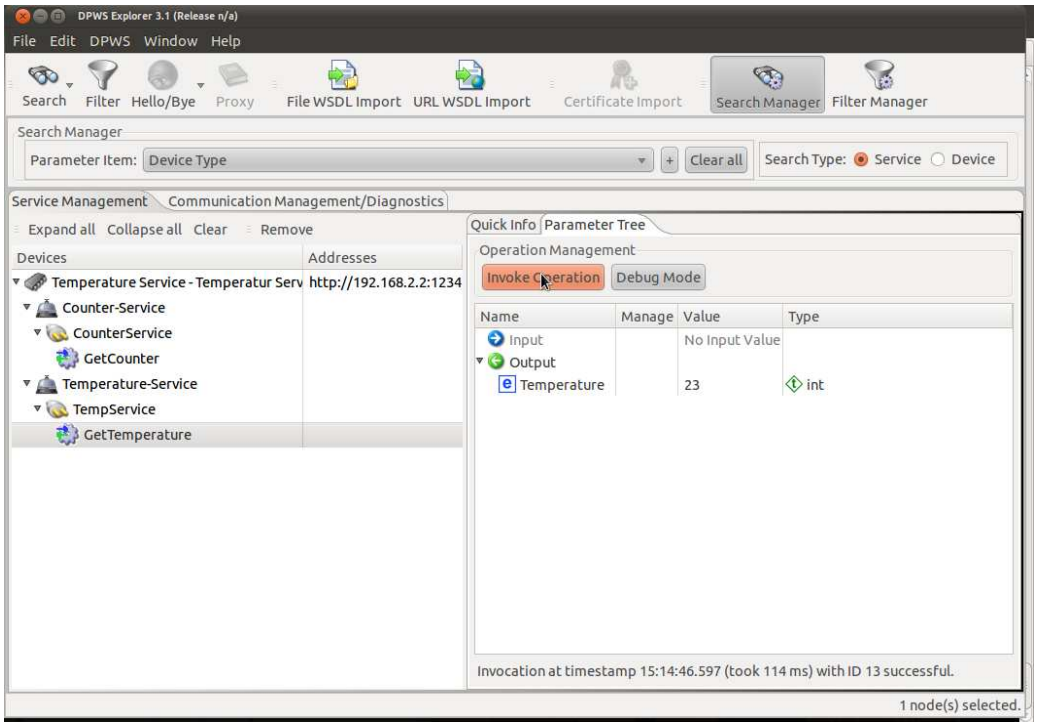


Figura 24 Serviços contador e temperatura em um único dispositivo.

O código de linguagem C e o arquivo que configura variáveis para o arquivo que configura variávies para o *makefile* referentes ao dispositivo encontram-se nos apêndices.

Assim como foi possível realizar-se a criação de dois serviços para um dispositivo, vários serviços para um mesmo dispositivo podem ser criados.

# 9 SUGESTÕES PARA TRABALHOS FUTUROS

O projeto foi concluído com uma plataforma embarcada que fornece serviços. Algumas sugestões facilitariam os testes a serem feitos com a plataforma ou tornariam a implementação na plataforma embarcada mais eficiente em relação aos recursos.

## 9.1 MÚLTIPLOS DISPOSITIVOS

A possibilidade de executar vários dispositivos virtuais em apenas uma plataforma embarcada tornaria possível a realização de testes para emular um sistema mais complexo com a utilização de menos recursos de *hardware*. Duas alternativas surgem em uma análise inicial: implementar mais de um dispositivo no código do uDPWS ou executar mais de um código compilado do uDPWS com apenas um dispositivo.

Analisando o código do SO Contiki e o código referente à plataforma utilizada para executar o uDPWS em Linux, verifica-se que a criação da interface de rede virtual (*tap*) cria sempre a mesma interface e, talvez, isso fizesse com que os dispositivos tivessem o mesmo endereço, não sendo possível diferenciá-los. Entretanto, é necessário verificar a possibilidade de se criar dois dispositivos antes de verificar se eles teriam o mesmo endereço.

Para que fossem executados vários uDPWS com apenas um dispositivo, seriam necessárias várias interfaces *tap* diferentes, o que demandaria alterações no código do próprio Contiki e sua interação com o Linux.

## 9.2 PORTAR O CONTIKI

A plataforma FriendlyArm, utilizada para o projeto, possui ports para alguns SOs. O escolhido para o projeto foi o Linux. O uDPWS, apesar de executar em Linux, precisa do SO Contiki, por ser desenvolvido a partir desse. O Contiki é um SO para sistemas embarcados, e o uDPWS já é executado nativamente apenas com este SO em algumas plataformas.

Para economizar recursos de processamento da placa, o SO Contiki poderia ser portado diretamente para ser executado sobre a plataforma FriendlyARM, sem a necessidade do Linux. Dessa forma, todos os recursos utilizados pelo Linux poderiam ser utilizados para a aplicação

A realização de um *port* do uDPWS para outras plataformas que rodem Linux sobre um processador ARM não seria diferente, caso essa placa possua os recursos de *bridge* e criação de interfaces virtuais no *kerne*l do Linux, pois o uDPWS é apenas uma aplicação executando sobre o SO.

Já essa realização de um *port* do uDPWS para ser executado diretamente sobre o Contiki sem a necessidade do Linux, faria com que *ports* para outras placas que não tem suporte à Linux fosse facilitada, pois então ter-se-ia uma idéia do que deve ser alterado para uma plataforma nova.

Essa abordagem não foi implementada, pois o objetivo é a inicial era obter uma plataforma que utiliza o conceito de orientação à serviços, mas possibilitando melhorias de performance e conhecimento para implementação em outras plataformas.

## 9.3 INTEGRAÇÃO EM UMA REDE SEM FIO

A utilização em uma rede sem fio *WiFi* (que utiliza o padrão IEEE 802.11) seria facilmente implementada nos dispositivos utilizados. Isso é muito interessante, pois facilitaria a instalação dos dispositivos na aplicação à que eles serviriam e iria ao encontro das tendências tecnológicas no sentido de maior mobilidade.

O próprio kit FriendlyARM possui placas de expansão *wireless* vendidas separadamente, que são interpretadas como interfaces de rede pelo Linux. No caso da interface de rede utilizada, ela se chamava *eth0*. Esse tipo de interface sem fio é normalmente

chamada de *wlan* no Linux, sendo que as formas de configurar endereços IP e *bridges* seria a mesma para os dispositivos de interfaces *Ethernet* cabeadas e interfaces *Ethernet* sem-fio.

Para o projeto de mestrado ao qual o contexto desse projeto se insere, foram adquiridas placas que utilizam o protocolo *ZigBee*, cuja interface é através de RS-232. O protocolo DPWS é independente da camada física, mas seria necessária juntar a interface de rede virtual (*tap*) criada pelo uDPWS com uma interface RS-232, para executar o uDPWS se comunicando através desses módulos.

A análise de como essa integração pode ocorrer é de grande importância para que esse tipo comunicação através desses módulos seja obtido.

# 10 CONCLUSÃO

O projeto obteve êxito na tarefa de obter um dispositivo embarcado que ofereça funcionalidades através de serviços. A comunicação do serviço disponibilizado com um computador foi realizada com sucesso, bem como a criação de novos serviços para serem oferecidos através da implementação do protocolo DPWS escolhida, o uDPWS.

Com essas ferramentas já compreendidas, a configuração de um ambiente para pesquisa, com dispositivos customizados para tal em uma rede de dispositivos inteligentes, é bastante facilitada. A compreensão do protocolo, da implementação do protocolo e as suas interações foram fundamentais para que a implementação de um novo serviço fosse possível, não apenas utilizar as ferramentas necessárias para executar o código em uma plataforma embarcada, que por sua vez, necessitou também de vários conhecimentos em programação e configurações de SOs GNU/Linux embarcados.

Uma grande quantidade de ferramentas é utilizada para a configuração do sistema onde a aplicação é desenvolvida e onde ela é executada. A descrição dos passos, neste projeto, para que essa configuração seja feita corretamente tem como intuito servir de guia para trabalhos futuros, seja para utilização da uma aplicação baseada na que foi desenvolvida, ou para as sugestões apresentadas, que visam melhor desempenho do sistema ou aumento de sua flexibilidade.

## 11 REFERÊNCIAS


ALBACORE. **Digi Connect ME® 9210**. Disponível em: <http://www.albacore.com.br/m/p/pg_prod/164>. Acesso em: 30 jun. 2011.

ASHTON, KEVIN. **That 'Internet of Things' Thing**. RFID Journal, Jun 2009. Disponível em: < http://www.rfidjournal.com/article/view/4986>. Visualizado em: 29 jun. 2011.

ATMEL. **Atmel Store**. Disponível em: <http://store.atmel.com>. Acesso em: 30 jun. 2011.

BERNERS-LEE, T. et al. **Hypertext Transfer Protocol**: HTTP/1.0. Disponível em: <ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt>. 1996. Acesso em 29 de jun. 2011.

BERNERS-LEE, T. **Web Services:** Program Integration Across Applications and Organizattion Boundaries. 2002. Disponível em: <http://www.w3.org/DesignIssues/WebServices.html>. Acesso em: 29 jun. 2011.

BOBEK, A. et al. **Device and Service Templates for the Devices Profiles for Web Services**. Rostock, Alemanha. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.3152&rep=rep1&type=pdf>. Acesso em: 29 jun. 2011.

BOOTH, D. et al. **Web Services Architecture**. 2003. Disponível em: <http://www.w3.org/TR/2003/WD-ws-arch-20030514/>. Acesso em: 29 jun. 2011.

BUYTENHEK, L.; HEMMINGER, S. **IEEE 802.1d Ethernet Bridging**. Disponível em: <http://sourceforge.net/projects/bridge/>. Acesso em: 29 de jun. 2011.

CÂNDIDO, G. et al. **Generic Management Services for DPWS-Enabled Devices.** Disponível em <http://www.socrades.eu/Documents/objects/file1259605647.07>. Acesso em: 29 jun. 2011.

DOHNDORF, O. et al. **Toward the Web of Things**: Using DPWS to Bridge Isolated OSGi Plataforms. Dortmund, Alemanha. Disponível em: <http://www.webofthings.com/wot/2010/pdfs/151.pdf >. Acesso em: 29 jun. 2011.

DUNKELS, A. **The Contiki Operating System**. Disponível em: <http://www.sics.se/contiki/>. Acesso em: 29 jun. 2011.

ECLIPSE FOUNDATION. **The Eclipse Foundation open source community website**. Disponível em <http://www.eclipse.org/>. Acesso em 29 jun. 2011.

FRIENDLYARM. **DVD FriendlyArm**. Conjunto de programas. 1 DVD.

FRIENDLYARM. **FriendlyARM**. Disponível em <http://www.friendlyarm.net>. Acesso em 29 jun. 2011.

HILBRICH, R. **An Evaluation of the Performance of DPWS on Embedded Devices in a Body Area Network**. Berlim, Alemanha. Disponível em: <http://robert.hilbri.ch/wp-content/uploads/2010/05/paper1.pdf >. Acesso em: 29 de jun. 2011.

JAMMES, F. et al. **Use of Web Services for next-generation automation systems.** 2009. Disponível em: <http://www.oasis-open.org/committees/download.php/33399/Use%20of%20Web%20Services%20for%20next-generation%20automation%20systems.pdf>. Acesso em: 29 jun. 2011.

OASIS. **Devices Profile for Web Services version 1.1**. 2009. Disponível em: <http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01>. Acesso em: 29 jun. 2011.

OASIS. **OASIS Reference Model for Service Oriented Architecture 1.0**. 2006 Disponível em: <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>. Acesso em 29 de jun. 2011.

ORACLE. **Java Embedded**. Disponível em: <http://www.oracle.com/technetwork/java/javame/embedded/overview/getstarted/index.htm> . Acesso em: 29 de jun. 2011.

ORACLE. **VirtualBox**. Disponível em: <http://www.virtualbox.org/>. Acesso em: 29 jun. 2011.

PLUMMER, A. **Bandwidth Scavaging in Sensor Networks (Hardware Project)**. Disponível em: <http://www.egr.msu.edu/~plumme23/whitespace_hardware.html>. Acesso em: 30 jun. 2011.

QTOPIA. **Qtopia**. Disponível em: http://qpe.sourceforge.net/. Acesso em: 29 de jun. 2011.

SCHLIMMER, J. **A Technical Introduction to Devices Profile for Web Services**. 2004. Disponível em: <http://msdn.microsoft.com/en-us/library/ms996400.aspx>. Acesso em 29 jun. 2011.

THE SENSOR NETWORK MUSEUM. **Sun SPOT**. Disponível em: <http://www.snm.ethz.ch/Projects/SunSPOT>. Acesso em: 30 jun. 2011.

WALLENSTEIN, H.; LEITE, T. **Análise de Vulnerabilidades em Ambientes *Triple-Play***. 2008. Disponível em: <http://student.dei.uc.pt/~how/mei/lc/S3P_HOME+ISP.pdf>. Acesso em: 29 jun. 2011.

WS4D. **Tool: DPWS Explorer**. Disponível em: <http://ws4d.e-technik.uni-rostock.de/dpws-explorer/>. Acesso em 29 de jun. 2011.

WS4D. **uDPWS: DPWS for highly resource-constrained devices**. Disponível em: <http://code.google.com/p/udpws/>. Acesso em: 29 jun. 2011.

WS4D. **Web Services For Devices**. Disponível em <http://www.ws4d.org/>. Acesso em: 29 de jun. 2011.

ZEEB, E. et al. **Service-Oriented Architecture for Embedded Systems using Devices Profile for Web-Services**. Rostock, Alemanha. Disponível em: <http://www.loms-itea.org/publications/p18-socne07.pdf>. Acesso em: 29 jun. 2011.

# APÊNDICE:

Alguns códigos utilizados no projeto

# APÊNDICE: ALGUNS CÓDIGOS UTILIZADOS NO PROJETO

Alguns códigos de linguagem C, arquivos que configuram variáiveis para *makefile* da aplicação, documentos XML que descrevem o serviço e scripts para a inicialização da aplicação na plataforma embarcada são apresentados a seguir. Comentários, conforme os padrões de cada linguagem, estão inseridos para facilitar seu entendimento, quando aplicável.

## APÊNDICE A: DOCUMENTO XML DA DESCRIÇÃO DO SERVIÇO EM WSDL

```
<wsdl:definitions
        targetNamespace="jean/CounterDevice"
        xmlns:tns="jean/CounterDevice"
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
        xmlns:wsoap12="http://schemas.xmlsoap.org/wsdl/soap12/">

    <wsdl:types>
        <xs:schema
            targetNamespace="jean/CounterDevice"
            xmlns:tns="jean/CounterDevice"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified" blockDefault="#all">
            <xs:element name="Counter" type="xs:int" />
            <xs:element name="CounterInput" type="xs:int" />
        </xs:schema>
    </wsdl:types>

    <wsdl:message name="GetCounterIn">
        <wsdl:part name="parameters" element="tns:CounterInput" />
    </wsdl:message>
    <wsdl:message name="GetCounterOut">
        <wsdl:part name="parameters" element="tns:Counter" />
    </wsdl:message>

    <wsdl:portType name="CounterService" wse:EventSource="false">
        <wsdl:operation name="GetCounter">
            <wsdl:input message="tns:GetCounterIn"
                wsa:Action="jean/TempSens/GetCounterIn" />
            <wsdl:output message="tns:GetCounterOut"
                wsa:Action="jean/TempSens/GetCounterOut" />
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="CounterServiceSoap12Binding" type="tns:CounterService">
        <wsoap12:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
```

```
                <wsdl:operation name="GetCounter">
                        <wsoap12:operation />
                        <wsdl:input>
                                <wsoap12:body use="literal" />
                        </wsdl:input>
                        <wsdl:output>
                                <wsoap12:body use="literal" />
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="CounterService">
                <wsdl:port name="CounterPort" binding="tns:CounterServiceSoap12Binding">
                        <wsoap12:address location="" />
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

## APÊNDICE B: CÓDIGO DE LINGUAGEM C DO SERVIÇO CRIADO

```
#include<stdlib.h>
#include<string.h>
#include "udpws.h"
#include "dpws.h"
#include "dpws-gen.h"

int get_counter_callback(struct soap_header_s *sh, char* body, int body_len, struct sendm_s *sm, struct stack_s *stack);

#define ACTION_COUNT  1

static char counter_str[] = "00000";
//string que é repassada como parametro de saida
int counter_str_len = 5;

int input_number = -1;

//criação da estrutura que representa o serviço
const struct hosted_service_s udpws_counter_service = {
                        cs_path,
                        cs_path_len,
                        cs_ns_def,
                        cs_ns_def_len,
                        cs_Type,
                        cs_Type_len,
                        cs_Type_def,
                        cs_Type_def_len,
                        cs_ServiceId,
                        cs_ServiceId_len,
                        cs_Metadata,
                        cs_Metadata_len,
                        /* Action Array */
```

```
                ACTION_COUNT,
                {{cs_uri_action_getIN, cs_uri_action_getIN_len, get_counter_callback}}
};

//função que é chamada quando a ação do serviço é requerida
int get_counter_callback(struct soap_header_s *sh, char* body, int body_len, struct sendm_s *sm, struct stack_s *stack){
        int err;

  char* body_end = body + body_len;
  char* p = body;
  uint8_t tag_type;
  /* lendo o arquivo XML recebido */
  if (xml_next_tag(p, body_end, NULL, &tag_type, XML_NEXT_TAG_ERR_IF_CLOSE | XML_NEXT_TAG_ERR_IF_EMPTY, &p))
    return DPWS_ERR_PARSE;
  /* procurando o parametro de entrada: */
  if (xml_next_tag(p, body_end, NULL, &tag_type, XML_NEXT_TAG_ERR_IF_CLOSE | XML_NEXT_TAG_ERR_IF_EMPTY, &p))
    return DPWS_ERR_PARSE;
  if (tag_type == XML_TAG_TYPE_EMPTY || (p = xml_cont_as_str(p, body_end)) == NULL) {
    //se não encontrou, parâmetro default é 0
                input_number = 0;
  } else {
                //se encontrou, retorna o numero inteiro representado pela string
    input_number = atoi(p);
  }

        //calculo do parametro de saída
        //0 se input<-100
        //counter++ se input>=0
        //input+5 se input <0 && input>=-100
        int num;
        if (input_number>=0) {
                num = atoi(counter_str);
                num++;
                if (num > 9 || num < 0) {
                num = 0;
                }
        } else if (input_number<-100) {
                num = 0;
        } else {
                num = input_number+5;
        }
        char buf[5];
        sprintf(buf, "%d", num);
        strcpy(counter_str, buf);


        /* SOAP Header */
        err = dpws_gen_header(SH_RELATESTO | SH_TO , NS_WSD | NS_WSDP, udpws_counter_service.namespace_def,
udpws_counter_service.namespace_def_len, cs_uri_action_getOUT, cs_uri_action_getOUT_len, sm, sh, NULL);
        if(err){
                return err;
        }
        //gerando saída
```

```
        sendm_add_end_rom(sm, cs_Counter, cs_Counter_len);
        sendm_add_end_ram(sm, counter_str, counter_str_len);
        sendm_add_end_rom(sm, cs_xCounter, cs_xCounter_len);
        DPWS_FINISH_SOAP(sm);
        UDPWS_PRINTF("Counter Service: GetCoutner Action called!\n");
        return 0;
}
```

## APÊNDICE C: CONFIGURAÇÃO DO *MAKEFILE* DO DISPOSITIVO COM DOIS SERVIÇOS

```
#must be set by every Service Makefile via +=
#MUST be "Simply expanded variables" (NOT recursively)
#http://www.gnu.org/software/automake/manual/make/Flavors.html#Flavors
UDPWS_SERVICE_SOURCE:=
UDPWS_SERVICE_GEN:=
UDPWS_SERVICE_DIRS:=

#aproach: set the UDPWS_SERVICE_DIR and include the makefile of every service
UDPWS_SERVICE_DIR:= $(UDPWS_DEVICE_DIR)/../../services/counter-service
include $(UDPWS_SERVICE_DIR)/counter-service.mk

UDPWS_SERVICE_DIR := $(UDPWS_DEVICE_DIR)/../../services/temperature-service
include $(UDPWS_SERVICE_DIR)/temp-service.mk

#just a helper
UDPWS_DEVICE_SOURCE=device.c
UDPWS_DEVICE_GEN=$(addprefix $(UDPWS_DEVICE_DIR)/gen/, $(shell ls $(UDPWS_DEVICE_DIR)/gen/))
UDPWS_DEVICE_DIRS=$(UDPWS_DEVICE_DIR)

UDPWS_DEVICE_SOURCE+=$(UDPWS_SERVICE_SOURCE)
UDPWS_DEVICE_GEN+=$(UDPWS_SERVICE_GEN)
UDPWS_DEVICE_DIRS+=$(UDPWS_SERVICE_DIRS)
```

## APÊNDICE D: CÓDIGO DE LINGUAGEM C DO DISPOSITIVO COM DOIS SERVIÇOS

```
#include "contiki.h"
#include "udpws.h"
#include "dpws.h"
#include "device.h"
#include "dpws-gen.h"

#define DEBUG               3  /* 0 - No Debug -> 5 - Full Debug */
#define UDPWS_DEBUG_MODULE    "TEMP"
#include "udpws-debug.h"

//declaração das estruturas de todos serviços
extern const struct hosted_service_s udpws_counter_service;
extern const struct hosted_service_s udpws_temp_service;
```

```
//numero de serviços
#define DEVICE_SERVICE_COUNT        2
//endereço das estruturas dos services
#define DEVICE_SERVICES             {&udpws_counter_service, &udpws_temp_service}

//os dois parametros acima são passados ao se construir a estrutura do dispositivo
//PROCESS(temp_process, "Temp_process");

const struct dpws_device_s udpws_temp_device = {
                        temp_UUID,
                        temp_UUID_len,
                        temp_ThisDevice,
                        temp_ThisDevice_len, /*TODO: temp_ThisDevice_len is not a constant*/
                        temp_ThisModel,
                        temp_ThisModel_len,
                        temp_MetadataVersion,
                        temp_MetadataVersion_len,
                        temp_ServiceId,
                        temp_ServiceId_len,
                        NULL,
                        DEVICE_SERVICE_COUNT,
                        DEVICE_SERVICES
};
```

## APÊNDICE E: *SHELL SCRIPT* PARA CONFIGURAÇÃO DE VARIÁVEIS DE AMBIENTE

```
#!/bin/sh
#estes são os lugares onde os arquivos do udpws e bridge-utils estão
echo ~Setting environment paths~
export PATH=$PATH:/sdcard/udpws_arm_jean/udpws/Release/
export PATH=$PATH:/sdcard/bridge-utils/brctl/
export PATH=$PATH:/sdcard/bridge-utils/libbridge/
```

## APÊNDICE F: *SHELL SCRIPT* PARA CONFIGURAÇÃO DE *BRIDGE*

```
#!/bin/sh
#executar após iniciar o udpws, senão interface tap0 não existe.
ifconfig eth0 0.0.0.0 promisc up
ifconfig tap0 0.0.0.0 promisc up
brctl addbr br0
brctl addif br0 tap0
brctl addif br0 eth0
ifconfig br0 0.0.0.0 up
```