View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Universidade Estadual de Campinas
Instituto de Computação

Jeferson Rech Brunetta

PROST: Um arcabouço para o desenvolvimento de dispositivos programáveis para a IoT

CAMPINAS
2017

Jeferson Rech Brunetta

PROST: Um arcabouço para o desenvolvimento de dispositivos programáveis para a IoT

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Edson Borin
Coorientadora: Prof. Dra. Juliana Freitag Borin

Este exemplar corresponde à versão final da Dissertação defendida por Jeferson Rech Brunetta e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): FUNCAMP; CNPq, 132788/2015-2
ORCID: http://orcid.org/0000-0001-5660-8267

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

B835p

Brunetta, Jeferson Rech, 1993-
PROST : um arcabouço para o desenvolvimento de dispositivos programáveis para a IoT / Jeferson Rech Brunetta. – Campinas, SP : [s.n.], 2017.

Orientador: Edson Borin.
Coorientador: Juliana Freitag Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Internet das coisas. 2. Redes de computadores - Protocolos. 3. Dispositivos embarcados da Internet. I. Borin, Edson,1979-. II. Borin, Juliana Freitag,1978-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: PROST : a framework for the development of programmable devices for IoT
Palavras-chave em inglês:
Internet of things
Computer network protocols
Embedded Internet devices
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Sandro Rigo
Ricardo Menotti
Data de defesa: 17-04-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Jeferson Rech Brunetta

PROST: Um arcabouço para o desenvolvimento de dispositivos programáveis para a IoT

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC/Unicamp
- Prof. Dr. Ricardo Menotti
  DC/UFSCar
- Prof. Dr. Sandro Rigo
  IC/Unicamp

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 17 de abril de 2017

Resumo

Incontáveis tipos de dispositivos embarcados conectados à internet compõem a chamada Internet das Coisas, a qual promoverá a geração de um fluxo massivo de dados e o controle dos dispositivos, muitas vezes, de forma centralizada. Entretanto, as soluções existentes para este tipo de cenário estão sujeitas a falhas de conexão e requerem elementos externos para o monitoramento e controle dos dispositivos. A fim de mitigar essas limitações, este trabalho propõe uma plataforma para execução de código de usuário como serviço em dispositivos da Internet das Coisas. O principal objetivo da solução apresentada é permitir a execução de código como serviço em qualquer dispositivo conectado na internet e promover a interação transparente entre dispositivos. Algumas vantagens dessa abordagem são a capacidade de evitar a propagação de dados não essenciais na rede, a possibilidade de controlar dispositivos internamente e a compatibilidade com dispositivos computacionalmente restritos. O arcabouço PROST (PROgrammable Smart Things) proposto combina: (I) um protocolo de comunicação referência para descoberta e descrição de dispositivos conectados em uma rede de computadores e (II) uma plataforma compacta que executa aplicações compiladas para uma arquitetura de fácil emulação. Finalmente, para validação da plataforma projetada, são apresentados estudos de caso em diferentes cenários, corroborando a eficácia da solução.

Abstract

Innumerable types of embedded devices connected to the internet make up the so-called Internet of Things, which will promote the generation of a massive flow of data and the control of the devices, often centrally. However, the existing solutions for this type of scenario are subject to connection failures and require external elements for the monitoring and control of the devices. In order to mitigate these limitations, this work proposes a platform for executing user code as a service in devices of the Internet of Things. The main goal of the presented solution is to enable the execution of code as a service on any device connected to the Internet and to promote transparent interaction between devices. Some advantages of this approach are the ability to avoid spreading nonessential data on the network, the ability to control devices internally, and compatibility with computationally constrained devices. The proposed PROgrammable Smart Things framework combines: (I) a reference communication protocol for discovery and description of connected devices in a computer network and (ii) a compact platform that runs compiled applications for an easy-to-emulate architecture. Finally, to validate the projected platform, case studies are presented in different scenarios, corroborating the effectiveness of the solution.

Lista de Figuras

2.1 Exemplo de máquina virtual, adaptado de [61] 18
2.2 Taxonomia de máquinas virtuais, adaptado de [61] 19
2.3 Arquitetura AllJoyn, adaptado de [63] 24
2.4 Abstração do barramento AllJoyn, adaptado de [6] 24
2.5 Clientes leves conectados ao barramento AllJoyn, adaptado de [6] 25
2.6 Bloco funcional OCF, adaptado de [23] 29
2.7 Exemplo de dispositivo OCF, adaptado de [23] 30
2.8 Arquitetura do arcabouço Soletta, adaptado de [19] 32

3.1 Arquitetura libWeave, adaptado de [29] 40

4.1 Arquitetura da COISA-VP em um microcontrolador ATmega328 43
4.2 Programa de usuário na interface Blockly 46

5.1 Visão geral da libWeave 50
5.2 Diagrama de sequência da instalação e execução de código de usuário . . . 56
5.3 Diagrama de inserção de eventos na PROST 56

6.1 Programa de usuário executado na sala inteligente 59
6.2 Programa de usuário executado no robô 61
6.3 Programa de usuário executado no carro inteligente 63
6.4 Gráfico de chamadas de funções e métodos modificados e adicionados à libWeave 64
6.5 Alocação da seção BSS 65

Lista de Tabelas

2.1 Tabela comparativa entre as principais máquinas virtuais para sistemas embarcados [53] . . . 22
2.2 Tabela com as principais URIs utilizadas pelo padrão OCF . . . 30
2.3 Tabela de descrição da plataforma OCF . . . 31
2.4 Tabela comparativa entre os protocolos de comunicação . . . 33

3.1 Argumentos adicionais no SSID . . . 36
3.2 Resumo das regras de acesso Weave . . . 37
3.3 Invólucro libWeave . . . 41
3.4 Requisitos libWeave + invólucro Linux . . . 41

6.1 Tamanho de cada seção e diferenças, em *bytes*, entre a libWeave e a PROST 64
6.2 Análise dos programas de usuário dos casos de uso . . . 65

Lista de Listagens

4.1 Início do programa de usuário gerado pelo Blockly 46
4.2 Procedimentos de tratamento de eventos do programa de usuário gerado pelo Blockly 47
4.3 Função principal do programa de usuário gerado pelo Blockly 47
4.4 Seção de dados do programa de usuário gerado pelo Blockly 48
5.1 JSON de descrição do *Trait* _coisa 53
5.2 Registrando o procedimento ao tratador de comandos da libWeave 53
5.3 Procedimento para instalação do código na máquina virtual e início da execução 54
5.4 Procedimento para execução da máquina virtual e consumo de eventos . . 54
6.1 JSON de descrição do *Trait _room* 58
6.2 JSON de descrição do *Trait _robot* 60
6.3 JSON de descrição dos *Traits* do carro inteligente 62
A.1 Exemplo de descrição de componente 74
B.1 Mensagem do tipo ssdp:alive 76
B.2 Mensagem do tipo ssdp:byebye 76
B.3 Mensagem do tipo ssdp:update 76
B.4 Mensagem do tipo ssdp:discover 77
B.5 Breve descrição do dispositivo e seus serviços no formato XML 77
B.6 Descrição detalhada do dispositivo e seus serviços no formato XML 78
C.1 Programa de usuário para a sala inteligente em linguagem de montagem . . 80
C.2 Programa de usuário para o robô em linguagem de montagem 82
C.3 Programa de usuário para o carro inteligente em linguagem de montagem . 83

Lista de Abreviações e Siglas

ABI Application Binary Interface

AJSL AllJoyn Standard Library

AJTCL AllJoyn Thin Core Library

API Application Program Interface

BLE Bluetooth low energy

CBOR Concise Binary Object Representation

CIL Common Intermediate Language

CoAP Constrained Application Protocol

COISA-VP Compact OpenISA Virtual Platform

CPU Central Processing Unit

CRUDN Create Read Update and Delete

DHCP Dynamic Host Configuration Protocol

DNS-SD DNS Service Discovery

DTN Delay Tolerant Networking

GATT Generic Attribute Profile

GCD Google Cloud Devices

GENA General Event Notification Architecture

GPIO General-purpose input/output

HLL-VM High-Level Language Virtual Machine

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

HVAC Central Heating Ventilation and Air-Conditioning

IoT Internet of Things

ISA Instruction Set Architecture

JSON JavaScript Object Notation

LWM2M Lightweight Machine to Machine

MAC Media Access Control

mDNS multicast Domain Name System

MQTT Message Queuing Telemetry Transport

NFC Near Field Communication

OCF Open Connectivity Foundation

OIC Open Interconnect Consortium

P2P peer-to-peer

PROST PROgrammable Smart Things

QoS Quality of Service

RAM Random Access Memory

RMI Remote Method Invocation

ROM Read Only Memory

RPC Remote Procedure Call

SIL Mote Runner intermediate language

SOAP Simple Object Access Protocol

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

SSDP Simple Service Discovery Protocol

SSID Service Set Identifier

TCP Transmission Control Protocol

TLS Transport Layer Security

UDA UPnP Device Architecture

UDP User Datagram Protoco

UPnP Universal Plug and Play

URI Uniform Resource Identifier

URL Uniform Resource Locator

UUID Universal Unique Identifier

XML eXtensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Sumário

1 Introdução **14**
1.1 Hipótese 15
1.2 Objetivos 15
1.3 Contribuições 16
1.4 Organização do texto 16

2 Trabalhos relacionados **17**
2.1 Mecanismos para execução de código de usuário em dispositivos embarcados 17
2.1.1 Máquinas virtuais compactas 18
2.2 Plataformas de interoperabilidade para dispositivos IoT 22
2.2.1 AllJoyn 23
2.2.2 UPnP+ 25
2.2.3 IoTivity 28
2.2.4 Weave 31
2.2.5 Soletta 31
2.3 Considerações 32

3 Weave **34**
3.1 Descoberta de dispositivos 34
3.1.1 Descoberta Local 35
3.1.2 WiFi 35
3.1.3 BLE 36
3.1.4 Nuvem 36
3.2 Provisionamento 36
3.3 Controle de acesso 37
3.4 Esquemas 37
3.4.1 *Traits* 38
3.4.2 Componentes 38
3.5 Comandos 39
3.6 Implementações 39
3.6.1 Cliente 39
3.6.2 Dispositivo 40

4 COISA **43**
4.1 Camada de abstração de *hardware* 44
4.2 Máquina Virtual 44
4.3 Gerenciador de Eventos 44
4.4 Monitor COISA-VP 45
4.5 Programação 46

5 **PROST - PROgrammable Smart Things** **49**
5.1 LibWeave . . . 49
5.2 Introspecção . . . 52
5.2.1 Geração de eventos . . . 54
5.2.2 Execução de comandos do dispositivo a partir de código de usuário 55
5.2.3 Visão geral . . . 55

6 **Resultados experimentais** **57**
6.1 Materiais e métodos . . . 57
6.2 Casos de teste . . . 57
6.2.1 Caso 1: Sala inteligente . . . 57
6.2.2 Caso 2: Robô . . . 59
6.2.3 Caso 3: Carro . . . 61
6.3 Avaliação do consumo de memória . . . 64
6.3.1 Tamanho das aplicações . . . 65

7 **Considerações finais** **67**
7.1 Trabalhos futuros . . . 68

A **Anexo Weave** **74**

B **Anexo UPNP** **76**

C **Anexo Experimentos** **80**

Capítulo 1

Introdução

Avanços em tecnologia da informação, sensores, comunicação sem fio e processadores embarcados estão propiciando o desenvolvimento de sensores e atuadores de baixo custo que conectarão o mundo de uma forma inédita. Lâmpadas, torneiras, unidades de ar condicionado, roupas, relógios, celulares, medidores inteligentes, meios de transporte, servidores na nuvem e inúmeros outros sistemas estarão conectados permitindo um uso mais inteligente dos nossos recursos e uma melhor interação com o ambiente ao nosso redor. Neste novo mundo cheio de possibilidades, conhecido como internet das coisas, ou IoT[1] [9], bilhões de dispositivos estarão conectados, compartilhando informação continuamente sensoreada e permitindo controle e monitoramento próximo ou remoto de ambientes.

Sob a perspectiva do usuário, um dos benefícios do ecossistema da internet das coisas é a habilidade de entrar em um ambiente (um escritório, por exemplo) e controlar dispositivos ou fazer uso de sensores próximos de maneira fácil. Para permitir isso, dispositivos como *smartphones* e *smartwatches* terão que operar em um modo ciente de contexto e adaptar seu comportamento baseado na localização, objetos próximos e características do usuário [46].

Apesar de ser fácil imaginar um *smartphone* ou um *smartwatch* com algumas dezenas de aplicativos de controle de dispositivos instalados, este cenário pode se tornar problemático com milhões de dispositivos. Portanto, se faz necessário o desenvolvimento de tecnologias para quando qualquer usuário se aproximar de um dispositivo da IoT (ou entrar em um ambiente) possa controla-lo. Com isso, seria possível o usuário interagir com esse dispositivo sem a necessidade de instalar um aplicativo especializado ou realizar procedimentos de configuração sofisticados [64].

Para contornar este desafio, protocolos de comunicação definem mecanismos que possibilitam a interação entre os dispositivos, permitindo a descoberta e descrição dos serviços. Desta maneira, novos dispositivos são conectados e configurados sem qualquer tipo de intervenção externa.

Em função de restrições como consumo de energia e custo, muitos dos dispositivos da IoT são construídos com sistemas computacionais bastante restritos, muitas vezes com microcontroladores que possuem no máximo 2 Kb de RAM. Para satisfazer estas restrições, o *software* desses sistemas é geralmente fixo e possui poucas funcionalidades.

[1] Do inglês: *Internet of Things*.

Entretanto, a execução de código como serviço em dispositivos na IoT pode permitir a personalização e o desenvolvimento de novos serviços neste ecossistema. Incontáveis dispositivos conectados permitiriam o processamento de quantidades massivas de dados, muitas vezes produzidos por eles mesmos.

As tecnologias de descoberta e auto-descrição tornam possível o desenvolvimento de mecanismos para migração de aplicações entre os dispositivos IoT. Um código de usuário que possa ser executado exatamente da mesma maneira em qualquer plataforma diminui a necessidade de uma transmissão massiva de dados na rede e permite a reconfiguração dos dispositivos de forma genérica e unificada. Dessa forma, faz-se necessário o desenvolvimento de plataformas que permitam tal serviço de execução de código em dispositivos IoT restritos.

Propomos então desenvolver uma plataforma capaz de proporcionar o envio e a execução de aplicações de usuário, de tal forma que o processo seja exatamente o mesmo para qualquer tipo de dispositivo.

Tendo em vista a necessidade de comunicação dos dispositivos, a empresa Google, que domina mais de 85% do mercado de Sistemas Operacionais para *smartphones* [44], esta desenvolvendo um protocolo de comunicação chamado Weave [30]. Este protocolo permite a comunicação e o gerenciamento de dispositivos IoT de maneira local e remota, incluindo mecanismos de autenticação e criptografia.

Já para a execução de código, temos o COISA-VP[2] [10, 53], que é uma plataforma virtual para permitir a execução de código descrito em uma linguagem de fácil emulação, voltada para dispositivos na Internet das Coisas, inclusive aqueles com poucos recursos de memória e capacidade de processamento.

A COISA-VP [53] é uma ferramenta que se demonstrou robusta para execução de código portátil, utilizando um pequeno consumo de recurso computacional.

Neste trabalho, propomos desenvolver uma plataforma para migração e execução de código em dispositivos IoT restritos. Para isto, combinamos o protocolo de comunicação Google Weave e a plataforma para execução de código COISA-VP.

Com a combinação destas ferramentas, é perfeitamente possível a implementação de um serviço de execução de código que se integre ao ambiente de maneira autônoma e possibilite a execução de aplicações, permitindo de maneira dinâmica a alteração e o carregamento destas aplicações.

1.1 Hipótese

Dado um código descrito em uma linguagem qualquer, é possível executa-lo em diferentes plataformas conectadas, incluindo plataformas computacionalmente restritas, por meio de um protocolo de comunicação e descrição de dispositivos.

1.2 Objetivos

De forma sucinta, com o desenvolvimento da plataforma pretendemos:

[2]Do inglês: *Compact OpenISA Virtual Platform.*

- Erradicar o uso de aplicações especializadas e permitir a interação direta entre os dispositivos e aplicações;
- Escrever aplicações genéricas que possam ser executadas independente da plataforma de *hardware* utilizada nos dispositivos IoT;
- Promover a redução do tráfego de dados pela rede e aumentar a precisão da automação dos dispositivos ao permitir a execução de código local que realize o monitoramento e controle internamente no dispositivo.

1.3 Contribuições

Este trabalho contribui com um arcabouço que permite o desenvolvimento de dispositivos inteligentes capazes de prover serviços para execução de aplicações independente de plataforma de *hardware*.

1.4 Organização do texto

O restante dos capítulos está organizado da seguinte maneira. No Capítulo 2, discutimos os trabalhos relacionados, analisando máquinas virtuais compactas e plataformas de interoperabilidade para dispositivos IoT. Nos Capítulos 3 e 4, apresentamos as tecnologias base utilizadas no desenvolvimento deste trabalho: Google Weave e COISA-VP, respectivamente. O arcabouço proposto, denominado PROST, é apresentado no Capítulo 5. Resultados experimentais da plataforma aplicada em cenários modelo são exibidos no Capítulo 6. Por fim, no Capítulo 7, discutimos os resultados obtidos e apresentamos possíveis extensões do trabalho desenvolvido.

Capítulo 2

Trabalhos relacionados

O mundo da IoT é novo e possui desafios que precisam ser vencidos para sua expansão. Este cenário se assemelha muito ao de redes de sensores sem fio e conforme destacado por Koshy e Pandey [47] tais desafios são:

- Heterogeneidade dos sistemas finais;
- Permitir a atualização dinâmica de *software*;
- Explorar uma rica interface de programação.

Neste mundo, os sistemas são compostos pelos mais variados tipos de dispositivos, com diferentes tipos de arquiteturas e capacidade computacional, incluindo em muitos dos casos microcontroladores com apenas 2 ou 4 kB de RAM.

Desta maneira, os trabalhos relacionados podem ser classificados em duas principais categorias: as máquinas virtuais compactas - para permitir a execução de programas de usuário em dispositivos embarcados; e arcabouços que permitem a interoperabilidade de comunicação entre os diferentes dispositivos. Estas duas categorias de trabalhos estão descritas ao longo das próximas seções.

2.1 Mecanismos para execução de código de usuário em dispositivos embarcados

Para facilitar o desenvolvimento de aplicações são desenvolvidos mecanismos que proporcionam uma rica interface de programação. Seja através do uso de bibliotecas, pela flexibilidade da forma de programação ou ainda do uso de sistemas operacionais que definam interfaces para generalizar funções características de plataformas. A programação em linguagem Java para microcontroladores facilita o desenvolvimento para programadores habituados a esta linguagem bem como permite a abstração de componentes específicos de plataforma.

A abordagem utilizada por HaikuVM [2] permite a compilação de aplicações escritas em linguagem Java para o uso em microcontroladores. Assim é possível fazer uso de grande parte das funcionalidades Java, traduzindo os *bytecodes* para estruturas da linguagem C. Esta abordagem permite a execução em dispositivos restritos, como o Atmega8 (1 kB de

RAM). Uma abordagem similar é utilizada em SimpleRTJ [3], porém requerendo entre 18 e 24 kB de memória extra. VM$\star$ [47] é um arcabouço para compilação de aplicações escritas em linguagem Java, otimizando sua execução para dispositivos restritos. Este tipo de abordagem facilita o desenvolvimento de aplicações para tais dispositivos, porém, a aplicação precisa ser recompilada para cada diferente plataforma alvo.

Uma vez que uma aplicação é compilada, ela pode ser executada da mesma forma em qualquer plataforma. Isto torna possível a redução de transmissão de dados na rede. Para que isso seja viável, é necessário que existam mecanismos para execução de aplicações independente da arquitetura do dispositivo que a execute. Assim, é fundamental o uso de máquinas virtuais capazes de fornecer este suporte, tornando possível a execução em diferentes plataformas de *hardware*, evitando a recompilação para cada plataforma alvo e facilitando a distribuição da aplicação.

2.1.1 Máquinas virtuais compactas

Para permitir a interação dos mais variados sistemas são definidas regras para comunicação de seus componentes, seja de *hardware* ou de *software*. As interfaces bem definidas permitem o desenvolvimento tanto de *software* quanto de *hardware* de maneira independente. Assim, uma aplicação escrita em linguagem de alto nível pode ser compilada para uma determinada plataforma alvo - composta por *hardware* e o Sistema Operacional - e ser executada em qualquer plataforma que siga os padrões previamente acordados. Esta abordagem cria restrições de execução para a plataforma alvo. As máquinas virtuais (VMs[1]) servem para flexibilizar as restrições impostas pela padronização, permitindo que, por exemplo, aplicações compiladas tendo um alvo possam ser executadas em outra plataforma [61].

Um exemplo de máquina virtual é apresentado na Figura 2.1. O *hardware* desta máquina, denominada anfitriã, implementa uma arquitetura de conjunto de instruções. Sobre o sistema anfitrião, é executado um *software* de máquina virtual, que é responsável por intermediar a comunicação entre o anfitrião e um sistema que pode atender a um outro conjunto de instruções. O *software* que executa sobre a máquina virtual é denominado convidado.

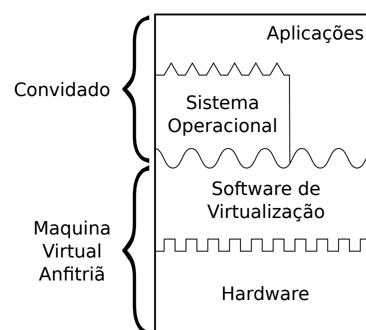

Figura 2.1: Exemplo de máquina virtual, adaptado de [61]

[1] Do inglês: *Virtual Machine*.

Conforme ilustrado na Figura 2.2, a taxonomia das máquinas virtuais é dividida em duas categorias principais. Uma com máquinas virtuais capazes de executar processos e outra com máquinas virtuais capazes de emular todo um sistema. As máquinas virtuais de processo foram divididas em duas subcategorias de acordo com seu conjunto de instruções da arquitetura (ISA[2]). Para as máquinas virtuais em que a máquina anfitriã e a convidada compartilham o mesmo ISA, são citados dois exemplos, os sistemas multiprogramados e a otimização dinâmica de binários [61]. Já as máquinas virtuais com diferentes ISAs possuem os exemplos de tradução dinâmica de binários e as máquinas virtuais de linguagens de alto nível.

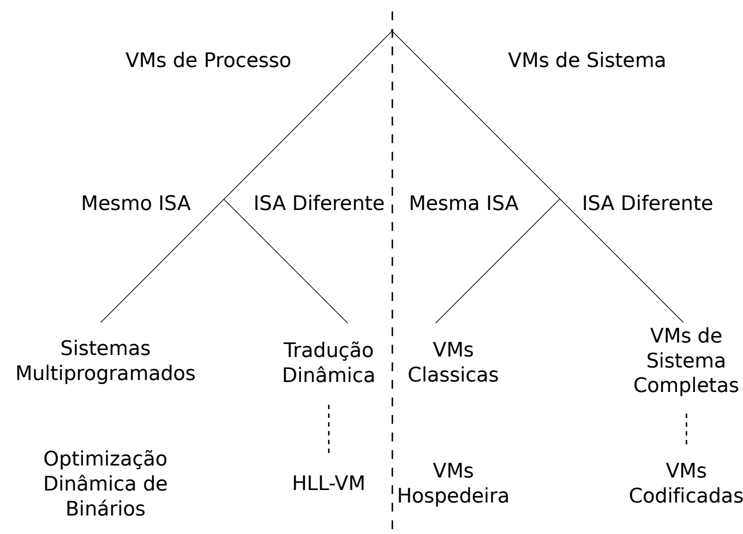

Figura 2.2: Taxonomia de máquinas virtuais, adaptado de [61]

Além disso, na Figura 2.2 estão representadas as máquinas virtuais de sistema. Estas também são divididas nas que compartilham o mesmo ISA, como por exemplo as máquinas virtuais clássicas que visam promover o isolamento de sistemas replicados e as anfitriãs que se diferenciam por possuírem um gerenciador de máquina virtual. As máquinas que não compartilham o mesmo ISA são exemplificadas com máquinas virtuais completas, cujo objetivo é geralmente a exatidão do sistema e máquinas codificadas, onde geralmente o desempenho é o alvo [61].

Máquinas virtuais de sistema completo

Utilizando microcontroladores, existem diversos projetos [4, 12, 41, 49, 56, 62] que visam emular outras plataformas. Tais projetos são voltados à emulação de sistemas completos e não possuem uma abstração dos periféricos de entrada e saída, tornando-os menos interessantes para o cenário de IoT.

[2]Do inglês: *Instruction Set Architecture*.

Máquinas virtuais de linguagens de alto nível

As máquinas virtuais de linguagens de alto nível (HLL-VM[3]) permitem, através de um conjunto padrão de bibliotecas, o acesso a componentes do sistema anfitrião [61]. Em um cenário restrito, como em microcontroladores, as HLL-VM muitas vezes são capazes de executar apenas um subconjunto da linguagem e consomem grande parte dos recursos do dispositivo.

A plataforma PyMite [5], por exemplo, permite a execução de programas escritos em linguagem Python por microcontroladores. Esta solução suporta apenas um subconjunto da linguagem, mais especificamente 25 das 29 palavras-chave e 89 de 112 *bytecodes* do Python 2.6. Para inicialização da plataforma, são gastos aproximadamente 4 kB de RAM; para imprimir uma mensagem de "Ola Mundo", são necessários 5 kB; a quantidade mínima de memória recomendada é de 8 kB.

Máquinas virtuais de processos

As máquinas virtuais de processos devem seguir uma convenção de módulos de programas descritos por uma interface (ABI[4]). Sua virtualização emula tanto instruções da arquitetura quanto as chamadas de sistema [61]. As chamadas de sistema podem permitir a interação da aplicação com os periféricos presentes no dispositivo anfitrião.

O projeto NanoVM [43] apresenta uma solução para executar aplicações escritas em linguagem Java em microcontroladores, podendo ser executado até mesmo no ATmega8, que possui apenas 2 kB de RAM. Sua principal limitação se dá no suporte a plataformas, suportando apenas quatro modelos de microcontroladores da Atmel.

O projeto uJ [42] também visa a execução de aplicações escritas em linguagem Java em microcontroladores. Permite a execução em diferentes plataformas restritas, como o ATmega64 (4 kB de RAM), porém suporta apenas um subconjunto das funcionalidades da linguagem.

O trabalho de Aslam e outros [8] apresenta uma abordagem para execução de aplicações Java. A máquina virtual chamada TakaTuka executa sobre o sistema operacional TinyOS [50] e é voltada para dispositivos com aproximadamente 100 kB de memória RAM.

Darjeeling [13] é uma máquina virtual Java também voltada para microcontroladores, ela executa sobre os sistemas operacionais de tempo real TinyOS [50], Contiki [20] e FOS [65]. Esta solução é voltada para microcontroladores de 2 kB a 10 kB de RAM, porém, é capaz de executar apenas um subconjunto da linguagem Java.

A máquina virtual Squawk [60] permite a execução de aplicações Java em dispositivos embarcados, porém, utiliza 80 Kb de RAM e 270 Kb de flash.

CILIX [66] é uma máquina virtual para executar CIL[5]. Possui suporte às linguagens C#, C++/CLI, Visual Basic, J++, F#, ou seja, linguagens suportadas pelo arcabouço .NET. Sua principal característica é o fato de poder operar em microcontroladores de 8

[3] Do inglês: *High-Level Language Virtual Machine*.
[4] Do inglês: *Application Binary Interface*.
[5] Do inglês: *Common Intermediate Language*.

bits com 4 kB de RAM. Porém, a CIL é uma linguagem com muita sobrecarga para ser utilizada em microcontroladores [14].

O IBM Mote Runner [14, 48] é um arcabouço e uma máquina virtual que permite a execução de código escrito em linguagem de alto nível, atualmente C# e Java. O código é compilado e então é traduzido para uma linguagem intermediária própria, denominada SIL[6]. O montador processa o código intermediário com suas respectivas bibliotecas transformando-o em arquivos binários para distribuição ou depuração. Tais binários podem ser então executados em dispositivos restritos. Características importantes do Mote Runner são o seu modelo de programação orientado a eventos e sua capacidade de gerenciamento remoto.

Os trabalhos supracitados, em alguns casos, atendem as restrições de recursos impostas, porém ficam limitados em relação à linguagem de programação utilizada.

COISA-VP[7] [10, 53] é uma plataforma que permite a execução de aplicações de usuário orientada a eventos, voltada para dispositivos restritos. Ela é capaz de executar aplicações compiladas para as arquiteturas OpenISA e MIPS-I, assim, é possível empregar qualquer compilador capaz de gerar código para uma destas arquiteturas, como por exemplo o GCC, que leva ao suporte de várias linguagens de programação.

Apresentamos na Tabela 2.1 um comparativo entre as principais máquinas virtuais compactas, contendo seus requisitos.

Apache Edgent

Dados de sensores vêm de qualquer nó da rede e cada vez mais dispositivos são conectados a ela. Os dados coletados por estes sensores são enviados para análises em abordagens centralizadas, geralmente em sistemas na nuvem computacional. Neste cenário, o tráfego de dados cresce muito com a quantidade de dispositivos conectados. No entanto, nem sempre todos os dados produzidos pelo dispositivos são necessários para tomada de alguma decisão.

O **Apache Edgent** é uma plataforma de código aberto que promove o envio de dados sob demanda. Esta plataforma permite a execução de código de usuário para processamento ou pré-processamento de dados no dispositivo, assim reduzindo a quantidade de dados transmitidos e armazenados.

O código de usuário é executado em uma máquina virtual Java e interage com as funções do **Apache Edgent** de forma similar a interação com bibliotecas. Esta aplicação, no próprio dispositivo IoT, analisa os dados para determinar quando eles precisam ser enviados para outros sistemas para análises, ações ou armazenamento. Por exemplo, um dispositivo poderia ser instalado em um vaso e, somente quando a planta está muito seca, são enviadas informações para se disparar um irrigador.

Caso o sistema esteja sendo executado conforme o esperado, não existe uma necessidade do envio constante dos dados. Porém, isto não descarta a possibilidade do envio de todos os dados quando uma anomalia é detectada, a fim de investigar as causas do problema.

[6] Do inglês: *Mote Runner intermediate language*.
[7] Do inglês: *Compact OpenISA Virtual Platform*.

Tabela 2.1: Tabela comparativa entre as principais máquinas virtuais para sistemas embarcados [53]

Máquina virtual	CPU (bits)	RAM (kB)	ROM (kB)	ISA	Foca na linguagem de programação
SimpleRTJ [3]	8/16/32	2-24	32-128	Java Bytecode	Java
Darjeeling [13]	8/16	2-10	32-128	Java Bytecode	Java
NanoVM [43]	8	1	8	Java Bytecode	Java
uJ [42]	8/16/32/64	4	80-60	Java Bytecode	Java
TakaTuka [8]	8/16	4	48	Java Bytecode	Java
Squawk [60]	32	512	4000	Squawk Bytecode	Java
IBM Mote Runner [14, 48]	8/16/32	4	32	SIL	C#,Java
CILIX [66]	8/16	4	32	CIL	C#
PyMite [5]	8/16/32	5	64	Python Bytecode	Python
COISA [10, 53]	8/16/32	0.35	6	OpenISA, MIPS-I	C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, Go

O Edgent permite com que um fluxo continuo de dados triviais não sejam enviados ao servidor, possibilitando com que apenas dados essenciais e significativos sejam transmitidos, na medida em que eles são produzidos.

Atualmente, ele é suportado nas plataformas Android, Java 7 e Java 8 (podendo ser utilizado em um Raspberry Pi). Além disso, esta plataforma utiliza MQTT para a propagação dos dados na camada de aplicação.

2.2 Plataformas de interoperabilidade para dispositivos IoT

É fundamental que um sistema na internet das coisas supere a heterogeneidade da infraestrutura e permita a integração dos dispositivos de forma transparente. Assim, existem diversas iniciativas para integrar dispositivos IoT através da definição de um conjunto de padrões que promovam a interoperabilidade. Neste contexto, os arcabouços são plataformas que fornecem um conjunto de mecanismos para comunicação e descrição dos dispositivos através da infraestrutura de rede. Alguns arcabouços também permitem o

desenvolvimento de aplicações para execução em diferentes tipos de dispositivos, como, por exemplo, o Soletta[19]. Dentre várias iniciativas de padronização, podemos destacar o AllJoyn, UPnP[8], IoTivity e Weave, descritas brevemente ao longo desta seção.

2.2.1 AllJoyn

O arcabouço AllJoyn [6] é uma plataforma de código aberto que promove um ambiente para aplicações distribuídas executarem entre diferentes classes de dispositivos. Seu foco principal é a mobilidade, a segurança e a configuração dinâmica de aplicações distribuídas.

Este arcabouço promove uma interface comum para o tratamento de heterogeneidade, endereçamento e mobilidade. Além disso, tem como principal conceito a proximidade e mobilidade dos dispositivos, permitindo a criação de aplicações *ad-hoc*, baseadas em proximidade e par-a-par (P2P[9]) independente da tecnologia de comunicação utilizada.

Segundo Villari e outros [63], o Alljoyn possui como principais mecanismos:

- A descoberta de dispositivos e aplicações próximas;
- Um arcabouço flexível para variados tipos de dispositivos;
- Um modelo capaz de abrigar diferentes tecnologias de comunicação (WiFi, *Bluetooth*);
- Uma maneira eficiente e segura de transporte de dados no barramento.

Atualmente, esse protocolo possui suporte somente para redes WiFi, porém, pode ser estendido para utilizar a tecnologia *Bluetooth* [6].

A arquitetura de um sistema AllJoyn é apresentada na Figura 2.3, onde esse arcabouço atua em uma camada superior ao sistema operacional, promovendo a abstração dos métodos de conexão e transporte de pacotes entre os diferentes componentes do sistema.

O arcabouço faz abstrações de seus conceitos para facilitar a compreensão sobre suas partes. Na sequencia são apresentados os principais conceitos do arcabouço AllJoyn.

Invocação Remota de métodos

Os sistemas distribuídos compreendem um conjunto de computadores operando sob uma infraestrutura de rede a fim de atingir um objetivo. Para que uma função de um programa seja executada total ou parcialmente em outra máquina da rede, geralmente é realizada uma RPC[10] ou RMI[11].

Para este tipo de interação, existe um cliente, que faz a invocação e um servidor, que provê a execução. Este é tido como um serviço no modelo AllJoyn. Em muitos casos mais de um servidor implementam as funcionalidades requeridas por um determinado cliente, assim, o arcabouço AllJoyn pode interpretar tanto um modelo cliente-servidor como uma topologia de rede P2P para atender a este cliente.

[8]Do inglês: *Universal Plug and Play*.
[9]Do inglês: *peer-to-peer*.
[10]Do inglês: *Remote Procedure Call*.
[11]Do inglês: *Remote Method Invocation*.

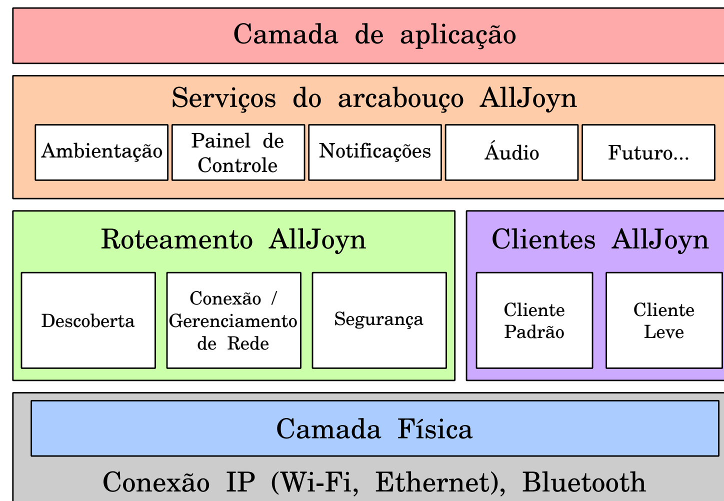

Figura 2.3: Arquitetura AllJoyn, adaptado de [63]

Barramento AllJoyn

O barramento é a abstração mais básica do arcabouço, de forma sucinta, é uma maneira rápida e leve de mover dados empacotados no sistema distribuído. Na Figura 2.4, temos a representação de um barramento no ecossistema AllJoyn. A linha horizontal representa o barramento e as linhas verticais representam a conexão dos módulos ao barramento. Os módulos são representados como hexágonos, numerados unicamente de acordo com a conexão.

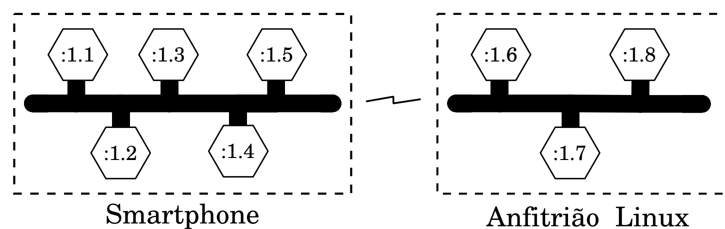

Figura 2.4: Abstração do barramento AllJoyn, adaptado de [6]

A ideia é que os módulos possam interagir de maneira transparente. Tanto os módulos de um mesmo processo como os de outros processos na mesma máquina ou em diferentes máquinas.

Dispositivos restritos

A implementação padrão do arcabouço, também conhecida como AJSL[12], é projetada para ser executada sobre um sistema operacional em sistemas computacionais de propósito geral, sendo tais sistemas capazes de sustentar o serviço de barramento. Existe uma variação da implementação principal do AllJoyn chamada AJTCL[13] [6, 7]. A AJTCL

[12]Do inglês: *AllJoyn Standard Library*.
[13]Do inglês: *AllJoyn Thin Core Library*.

implementada em dispositivos restritos permite a comunicação destes com demais que implementam a AJSL.

A AJTCL é uma versão leve focada em dispositivos com menos poder computacional, podendo executar sobre um pequeno sistema operacional de tempo real (por exemplo mbedRTOS[51] ou freeRTOS[11]). Esta implementação está disponível apenas para uma plataforma baseada na família de microcontroladores stm32f4 e para o Arduino DUE [7].

Dispositivos que trazem a implementação da AJTCL não possuem poder computacional suficiente para se conectar diretamente a um barramento. Para fazer parte do barramento, tais dispositivos restritos se conectam a anfitriões AJSL, conforme ilustrado na Figura 2.5.

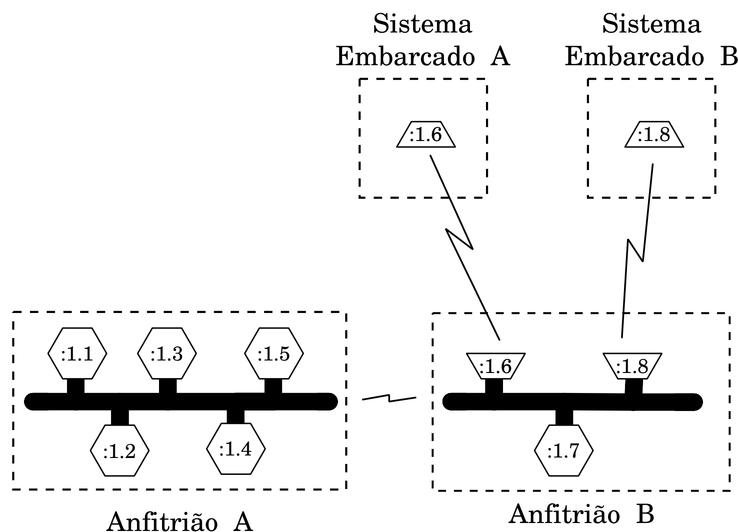

Figura 2.5: Clientes leves conectados ao barramento AllJoyn, adaptado de [6]

Conforme destacado por Villari e outros [63], o AllJoyn não possui um bom suporte para atuar fora do contexto de redes locais, o que faz com que ele não seja escalável o suficiente para o contexto de internet das coisas.

2.2.2 UPnP+

No fim da década de 90 o Forum UPnP foi criado com o propósito de definir um padrão que permitisse a comunicação de dispositivos multimídia, independente de fabricante, propósito de uso ou tecnologia de comunicação. No princípio era voltado exclusivamente para dispositivos multimídia, entretanto, com a evolução computacional e o advento da IoT, foi decidido propor uma nova arquitetura capaz de abranger tipos genéricos de dispositivos.

A Arquitetura de Dispositivo UPnP (UDA[14]) [22] versão 2.0 determina um conjunto de protocolos que pretendem promover a integração e configuração transparente e automática entre dispositivos. Para isto, são determinadas seis etapas distintas de configuração e interação com outros dispositivos UPnP. Tais etapas são: Endereçamento, Descoberta, Descrição, Controle, Evento e Apresentação. Estas etapas são descritas nas próximas seções.

[14]Do inglês: *UPnP Device Architecture*.

0 - Endereçamento

A comunicação entre os dispositivos pode utilizar TCP ou UDP, porém, sempre sobre a camada de endereçamento IP. Portanto, um dispositivo deve possuir um endereço IPv4 e, opcionalmente, um IPv6. A primeira etapa de configuração tem o objetivo de atribuir um ou mais endereços a este dispositivo. Por padrão, a primeira tentativa é encontrar um servidor DHCP, se ele falhar, o dispositivo deve se auto atribuir um endereço seguindo a RFC 3927 [15] e procurar um servidor DHCP a cada 5 minutos.

A faixa permitida para estes dispositivos é 169.254/16 e mensagens nesta faixa não serão encaminhadas à rede externa. É permitido que um dispositivo possua mais de um endereço IP, ele é chamado um dispositivo de *multi-homed* e atua como sendo mais de um dispositivo. Ainda neste passo, é permitida a atribuição de um apelido para o dispositivo, estabelecido com um acordo entre ele e o servidor DHCP, seguindo o RFC 1034 [54] e RFC 1035 [55].

Cada dispositivo deve possuir um identificador único (UUID[15]) de 128 *bits*, construído a partir do endereço MAC. Este identificador deve ser mantido de forma permanente, devendo ser armazenado em uma memória não volátil ou gerado novamente sempre da mesma maneira.

1 - Descoberta

Uma vez endereçado, o dispositivo precisa descobrir outros dispositivos à sua volta. Para tanto, são utilizadas mensagens multi-direcionais. Todos os dispositivos precisam sempre escutar mensagens vindas pela porta 1900 e responder à requisição se alguma de suas características casarem com o critério da mensagem de descoberta. Para evitar a sobrecarga da rede, cada dispositivo deverá acrescentar ao tempo de resposta um valor aleatório.

O padrão da mensagem de descoberta é baseado no SSDP[16]. O cabeçalho das mensagens deve estar em conformidade com o HTTP 1.1 e ser formatado como uma 'mensagem genérica' de acordo com a RFC 2616 [21]. Uma mensagem de descoberta deve conter no cabeçalho o tipo **ssdp:discover**. Um exemplo de mensagem de descoberta é apresentado na Figura B.4.

Quando um dispositivo é adicionado à rede, ele deve enviar uma mensagem de notificação (**ssdp:alive**) para avisar os outros dispositivos e serviços de sua existência. Esta mensagem, conforme ilustrado em B.1, contém um campo com o tempo em que o dispositivo permanecerá disponível, sendo responsabilidade do dispositivo espalhar mensagens para atualizar este tempo. É importante balancear este tempo de vida baseado nas características particulares da rede, não sendo interessante ser muito curto (menos de 1800 segundos) nem muito longo (um dia).

Para atualizar uma informação previamente divulgada, é usada uma mensagem de notificação com o campo **ssdp:update**, conforme a Figura B.3. Um dispositivo deve revogar sua mensagem de descoberta caso ele se torne indisponível ou troque seu endereço IP. Uma mensagem do tipo **ssdp:byebye**, conforme demonstrado na Figura B.2, deve ser enviada para informar tal alteração de estado.

[15] Do inglês: *Universal Unique Identifier*.
[16] Do inglês: *Simple Service Discovery Protocol*.

2 - Descrição

Nesta etapa, o dispositivo expõe suas características, tanto contêineres lógicos de seus componentes físicos (dispositivo) quanto suas capacidades (serviços) a partir de duas descrições, breve e detalhada. Na breve, os descritores são solicitados por métodos GET HTTP em uma URL recebida na etapa de descoberta. Enquanto que em ambas, no corpo da resposta estão contidos: os esquemas UPnP descritos em XML[17], e estes contêm apontadores (URLs) para os descritores de serviços e descritores de dispositivos. Um exemplo de descrição breve pode ser observado na Figura B.5.

Informações detalhadas dos recursos são requisitadas através dos apontadores fornecidos durante a descrição breve e as respostas variam de acordo com o tipo do recurso. Caso seja um serviço, são providos o nome da ação e seus argumentos. Caso sejam requisitadas informações a cerca do dispositivo, é provido o tipo deste dispositivo, seu nome, tipo de dados, faixa de valores, valor atual, o valor padrão, bem como a granularidade de incremento e decremento. A descrição detalhada do dispositivo é ilustrada na Figura B.6.

3 - Controle

Uma vez possuindo as informações acerca de um dispositivo, é possível invocar ações e receber os resultados destas ações. Estas ações se assemelham a chamadas remotas de procedimento; um ponto de controle envia uma solicitação a um dispositivo e quando este procedimento se finaliza são retornados seus resultados ou erros.

O comando é disparado através da URL (absoluta ou relativa) fornecida para uma dada ação de controle. É permitido que esta ação altere as variáveis que descrevem o estado do serviço em tempo de execução. É permitida, também, a definição de ações que retornem o valor atual de uma ou mais variáveis de ações da atual execução.

Tanto as mensagens de controle como as respostas devem ser codificadas usando a codificação UTF-8 utilizando o esquema SOAP[18] encapsulado em um *post* HTTP.

Para a transferência de uma quantia massiva de dados, não é aconselhado o envio como parte do argumento SOAP nem como um anexo, mas sim o envio de uma URL, onde o dado possa ser recuperado através de algum método HTTP.

Uma vez originada uma ação, uma mensagem de resposta deve ser retornada em no máximo 30 segundos, incluindo o possível tempo de atraso da rede. Caso a resposta não esteja pronta dentro de 30 segundos, uma mensagem de confirmação de recebimento é enviada e um evento é gerado quando a ação é completada.

4 - Eventos

Para propagação de eventos, é usado o padrão GENA[19]. Esta arquitetura implementa o modelo produtor-subscritor fazendo uso do protocolo HTTP. Com isso é possível que um dado dispositivo produza conteúdo e possa transmiti-lo a todos os consumidores interessados.

[17]Do inglês: *eXtensible Markup Language*.
[18]Do inglês: *Simple Object Access Protocol*.
[19]Do inglês: *General Event Notification Architecture*.

O dispositivo produtor de conteúdo é denominado raiz e deve informar que possui tal serviço de eventos. Os dispositivos interessados se inscrevem enviando uma mensagem "**subscription**" para o dispositivo raiz. Esta mensagem deve conter um campo para indicar o tempo de subscrição. O subscritor deve aguardar uma mensagem de confirmação por até 30 segundos, que virá com o atual valor do campo solicitado. O endereço de retorno da subscrição será armazenado no dispositivo raiz e para cada atualização do campo solicitado, uma nova mensagem será gerada, assim, sempre mantendo o subscritor atualizado. Uma vez que o tempo de subscrição acaba e o subscritor permanece interessado, ele deve enviar uma mensagem de renovação ("**renewal**") com um novo tempo de expiração. Um subscritor pode enviar uma mensagem de "**unsubscribe**" para cancelar o recebimento das mensagens **unicast**. É permitido a um dispositivo raiz propagar as atualizações através de mensagens de difusão seletiva (**multicast**) ao invés de mensagens individuais (**unicast**).

5 - Apresentação

A apresentação se dá por meio de uma página HTTP, totalmente definida pelo fabricante do dispositivo. Essa página pode prover uma interface alternativa de controle e visualização do dispositivo ou simplesmente mostrar mais informações sobre o dispositivo ou, ainda, redirecionar a uma loja de produtos do vendedor.

2.2.3 IoTivity

IoTivity [18] é um projeto patrocinado pela OCF[20] [23] (Antiga OIC[21]), um grupo de empresas, que lideram o desenvolvimento das especificações e do programa de certificações IoTivity. O projeto IoTivity visa abranger desde dispositivos restritos até sistemas mais completos e provê uma implementação referência de código aberto dos padrões e especificações da OCF. Deste modo é possível aproximar a comunidade de desenvolvimento de código aberto e acelerar o desenvolvimento do arcabouço e dos serviços requeridos para conectar bilhões de dispositivos.

A OCF adota uma arquitetura RESTFul definindo duas regras lógicas de dispositivos: o cliente inicia as transações, enviando requisições para acessar o servidor e o servidor, que abriga os recursos OCF, envia respostas ao cliente e provê serviços.

A Figura 2.6 representa o modelo funcional das especificações OCF.

De baixo para cima, o bloco funcional é estruturado da seguinte maneira:

- Conectividade L2: Provê funcionalidades requeridas para estabelecimento de conexões nas camadas de rede físicas e enlace de dados. Por exemplo, WiFi ou *Bluetooh*.

- Rede: Provê funcionalidades requeridas pela OCF para envio de dados na rede. Por exemplo, a internet.

[20] Do inglês: *Open Connectivity Foundation*.
[21] Do inglês: *Open Interconnect Consortium*.

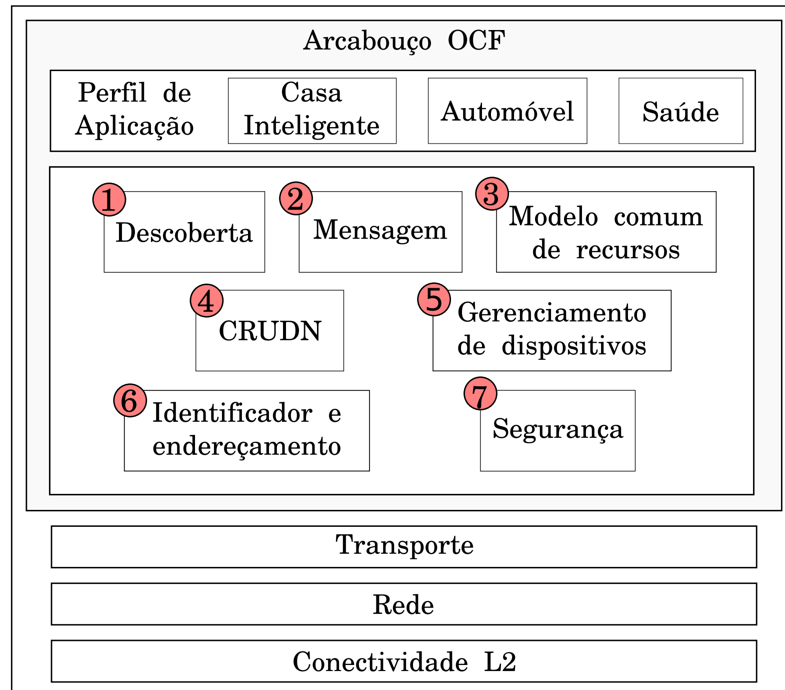

Figura 2.6: Bloco funcional OCF, adaptado de [23]

- Transporte: Abstrai o transporte fim-a-fim respeitando as restrições de QoS[22]. Por exemplo, usando TCP, UDP ou novos protocolos em desenvolvimento, como o DTN[23].
- Arcabouço OCF: Bloco funcional das requisições e respostas da comunicação entre dois dispositivos OCF.
- Perfil de aplicação: Modelos de segmento de mercado e especificação de funcionalidades. Por exemplo, o modelo de dados de uma casa inteligente para o segmento de mercado.

O arcabouço representa componentes chave para prover as funcionalidades de operação OCF. Tais componentes são:

1. Descoberta: Utilizando mecanismos padrões para a descoberta dos dispositivos (IETF CoRE [45]) e de recursos.
2. Mensagens: Protocolo para operações RESTFul tendo o CoAP [59] como principal protocolo para troca de mensagem. Caso necessário, traduções são feitas por intermediários.
3. Modelo de recursos: Representar entidades do mundo real e definir mecanismos para manipulação de recursos.

[22] Do inglês: *Quality of Service*.
[23] Do inglês: *Delay Tolerant Networking*.

4. CRUDN[24]: Mecanismo simples de requisições e respostas com comandos de criar, recuperar atualizar, deletar e notificar.

5. Gerenciamento de dispositivos: Configurações de rede e funções para monitorar, reiniciar o dispositivo e restaurar o padrão de fábrica.

6. Identificador e endereçamento: Etiquetamento das entidades OCF (dispositivos, clientes, servidores e recursos) seguindo o formato de URL.

7. Segurança: Autenticação, autorização e controle de acesso para garantir a segurança das entidades.

O grande diferencial das especificações OCF é o enfoque em dispositivos restritos. Para tanto, o protocolo CoAP é o padrão usado para comunicação e entidades intermediárias descentralizadas (**broker**) são usadas para reter informações de dispositivos.

A organização de um dispositivo é apresentada na Figura 2.7 e a Tabela 2.2 lista as principais URIs utilizadas por um dispositivo OCF. Neste exemplo, um dispositivo físico é listado pela URI "/oic/p" e abriga dois dispositivos OCF, cada um com suas devidas características. Os principais exemplos de características da plataforma estão listados na Tabela 2.3.

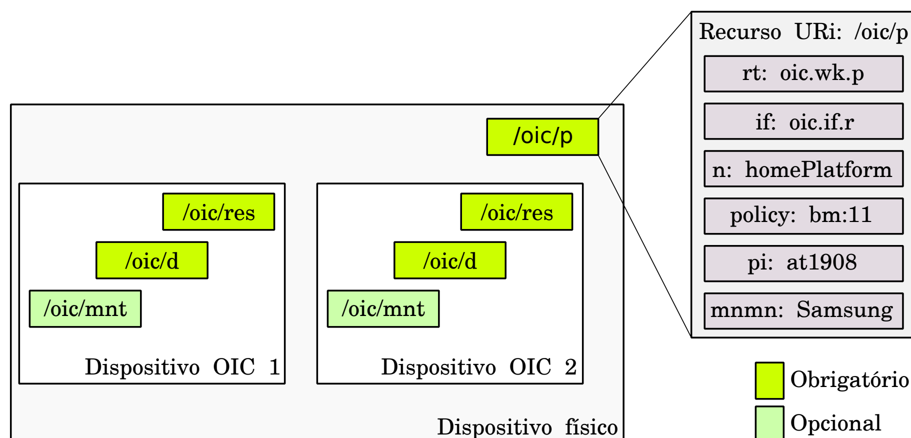

Figura 2.7: Exemplo de dispositivo OCF, adaptado de [23]

Tabela 2.2: Tabela com as principais URIs utilizadas pelo padrão OCF

Funcionalidade	URI
Descoberta	/oic/res
Dispositivo	/oic/d
Plataforma	/oic/p
Segurança	/oic/sec/
Manutenção	/oic/mnt
Monitoramento	/oic/mon
Configuração	/oic/con

[24]Do inglês: *Create, Read, Update, Delete, Notify.*

Tabela 2.3: Tabela de descrição da plataforma OCF

Recurso	Descrição
rt	Tipo do recurso
if	Interface do recurso
n	Nome do recurso
p	Política de acesso
pi	Identificador da plataforma
mnmn	Nome do fabricante
mnml	Link para detalhes do fabricante
mnmo	Número do modelo
mndt	Data de fabricação
mnpv	Versão da plataforma
mnos	Versão do sistema operacional
mnhw	Versão do *hardware*
mnvf	Versão do *firmware*
mnsl	Link para suporte
st	Tempo de sistema

2.2.4 Weave

O protocolo Weave é uma iniciativa da Google para a descoberta e gerenciamento de dispositivos IoT. Tendo como principais diferenças a segurança e a integração com o serviço da nuvem Google. A descoberta dos dispositivos é feita em uma rede local. Uma vez configurado, o dispositivo é vinculado a uma conta de correio eletrônico, podendo ser compartilhado com outras contas e aplicações. Os usuários do dispositivo IoT poderão então controlá-lo diretamente na rede local ou por intermédio da infraestrutura da Google. A nuvem agirá como intermediário, armazenando informações atualizadas dos estados do dispositivo e enfileirando comandos a serem executados. Os detalhes do funcionamento do Weave são abordados no Capítulo 3.

2.2.5 Soletta

Soletta é um arcabouço para dispositivos IoT projetado para os desenvolvedores acessarem facilmente sensores, atuadores e comunicação - sem possuírem um conhecimento específico sobre cada uma destas tecnologias. Ele utiliza tecnologias padrões de descoberta e identificação dos dispositivos - como exemplo, os padrões da OCF - para permitir o desenvolvimento de aplicações que possam controlar sensores e atuadores se comunicando com tecnologias referência. Além disso, o arcabouço Soletta tem aplicação em todos os tipos de dispositivos, até mesmo os mais restritos e é portátil e escalável, permitindo que desenvolvedores utilizem o mesmo programa em diferentes plataformas.

Foi desenvolvido para operar sobre um sistema operacional, possuindo suporte para os sistemas Linux, Zephyr, RIOT e Contiki. Projetado de forma modular, para que possa ser facilmente portado para um novo sistema operacional, pode ser aplicado em qualquer plataforma que possua suporte a algum destes sistemas.

Já foi testado em placas como Quark SE Dev Board (Zephyr), com 80 kb de SRAM

e 384 KB de Flash e o Atmel SAM R21 Xplained Pro (RIOT), com 32 kb de SRAM e 256 kb de flash.

A arquitetura do Soletta é apresentada na Figura 2.8. O arcabouço provê a abstração: dos módulos específicos da plataforma, como temporizadores SPI, GPIO; bem como implementa as camadas de comunicação, como o padrão da OIC, MQTT ou LWM2M.

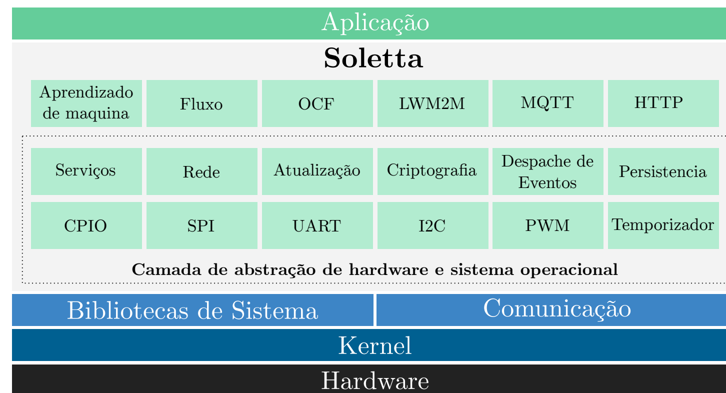

Figura 2.8: Arquitetura do arcabouço Soletta, adaptado de [19]

O código gerado, em conjunto com arquivos de configurações específicos de plataforma são compilados e ligados para gerar uma aplicação executável. Esta aplicação pode ser incorporada a um sistema operacional e é feita a geração de uma imagem gravável para uma plataforma restrita. Para sistemas menos restritos, aplicações geradas com base no arcabouço podem ser interpretadas.

2.3 Considerações

As principais características dos protocolos citados neste capítulo estão sumarizadas na Tabela 2.4. Além disso, esta tabela também aborda algumas questões sobre segurança e capacidade de descrição dos serviços.

Dos protocolos aqui abordados, o Soletta e o AllJoyn são protocolos que permitem a programação dos dispositivos, porém, focam principalmente em promover a troca de dados entre os dispositivos. Com estes protocolos, o desenvolvedor pode fazer a programação tendo um conhecimento prévio dos recursos presentes em cada dispositivo. Já os demais protocolos focam na descoberta dinâmica dos recursos de cada dispositivo além da troca de dados entre eles. Com base no estudo destes diferentes protocolos de comunicação e arcabouços analisados, que prometem promover a interoperabilidade entre os dispositivos, optamos pela utilização do Weave. Essa escolha é justificada pelo fornecimento de esquemas concisos e a disponibilidade de uma implementação referência de código aberto voltada para dispositivos restritos. O Capítulo 3 descreve as principais características do conjunto de protocolos e mecanismos de interoperabilidade utilizados pelo Weave.

Tabela 2.4: Tabela comparativa entre os protocolos de comunicação

Protocolo		UPnP+	Weave	IoTvity	AllJoyn
Implementação de código aberto			x	x	x
Funcionalidades	Descoberta de dispositivos	x	x	x	x
	Descrição de serviços	x	x	x	
	Segurança	x	x	x	x
	Streaming	x		x	x
	Produtor/Subscritor	x	x	x	
	REST		x	x	
	Integração à nuvem		x	x	x
Camada de aplicação	Formatação	XML	JSON	JSON	
	Protocolo	HTTP	HTTP HTTPS	CoAP	
Camada de transporte	TCP	x	x		x
	UDP			x	x
Camada de rede	IP	x	x	x	x
	BLE		x		

Capítulo 3

Weave

Weave [30] é um protocolo de descoberta e gerenciamento de dispositivos elaborados pela Google.

Este protocolo define: mecanismos que permitem a identificação de novos dispositivos conectados em uma rede de computadores; métodos e normas para o estabelecimento de comunicação; mecanismos de segurança para troca de mensagens; critérios para identificação das capacidades dos dispositivos; e vias para propagação de atualizações dos estados destes dispositivos [36].

A nuvem é o melhor ponto de referência para um dispositivo, pois ela fornece uma rica quantidade de recursos que este dispositivo pode fazer uso e está sempre disponível, fazendo com que ela seja ideal para enfileirar as tarefas do dispositivo e guardar um histórico de seus estados, para quando o dispositivo estiver fora de alcance. Além disso, a nuvem permite o acesso e controle do dispositivo a partir de qualquer parte do globo.

A etapa inicial para permitir o acesso remoto do dispositivo é uma abordagem baseada em proximidade, fazendo a identificação de um dispositivo, conforme abordado na Seção 3.1. Depois ele deve ser correlacionado com uma conta com a qual ele poderá ser gerenciado e compartilhado, conforme descrito na Seção 3.2 e Seção 3.3.

A troca de mensagens deve seguir padrões para garantir a interoperabilidade e a manipulação de novos dispositivos, tais parâmetros são apresentados na Seção 3.4. Para o desenvolvimento, são disponibilizadas algumas implementações referência, conforme tratado na Seção 3.6.

3.1 Descoberta de dispositivos

A descoberta de dispositivos é feita por proximidade, podendo ser através de:

- Uma rede local, utilizando mDNS.
- Criando um ponto de acesso WiFi.
- Utilizando BLE.

3.1.1 Descoberta Local

Um dispositivo pode ser encontrado em uma rede local utilizando um protocolo baseado no protocolo **zeroconf** (mDNS + DNS-SD). Este protocolo é chamado **Privet** [34] e atua usando o protocolo TCP. Combinando mensagens mDNS [16] e DNS-SD [17] para explorar recursos na rede.

O formato para descrição do tipo de serviço segundo o DNS-SD é *"_<PA>._<PT>"*, onde <PA> é o protocolo de aplicação e <PT> é o protocolo de transporte. No caso do **Privet**, o serviço deve ser *"_privet._tcp"*.

O DNS-SD provê campos adicionais chamados de **TXT records**. No caso do **Privet** este campo é preenchido com as mesmas informações de identificação do dispositivo ("/privet/info"), conforme detalhado na Seção 3.4.

É recomendado que os campos possuam um tamanho de até 512 *bytes*. Os campos incluem a versão, porta HTTPS, um nome que o usuário possa reconhecer, um identificador do dispositivo, o modelo do manifesto, e algumas bandeiras (semelhante às utilizadas para WiFi). Opcionalmente um campo para um descrição textual do dispositivo e, caso o dispositivo possua, o identificador do dispositivo pela nuvem.

De acordo com as especificações do mDNS o dispositivo deve realizar notificações quando é ligado, desligado ou sofre uma mudança de estados. Este anuncio deve ser feito pelo menos duas vezes com o intervalo de pelo menos 1 segundo.

3.1.2 WiFi

Usando WiFi, um dispositivo Weave pode criar um ponto de acesso sem proteção. O SSID[1] deverá seguir a codificação [Nome do dispositivo].[Identificador do modelo][adicionais]prv.

O "Nome do dispositivo" é uma combinação de no máximo 20 caracteres em um formato facilmente interpretável por humanos combinado a um sufixo aleatório, para evitar colisões.

O "Identificador do modelo" é um conjunto de 5 caracteres definidos durante a criação do manifesto modelo e incluem 2 caracteres que representam a classe do dispositivo (por exemplo, impressora, câmera, brinquedo). O cliente usará este identificador para requisitar informações adicionais ao servidor.

Argumentos "adicionais" são passados como dois caracteres codificados em formato alfabético de base 64 do menos significativo para o mais significativo. Por exemplo, 'b' = 27 = 0x1B = 0b00011011, neste caso, os argumentos definidos são 0, 1, 3 e 4. Estes argumentos são utilizados para descrever algumas propriedades do dispositivo, conforme descrito na Tabela 3.1.

O cliente se conectará ao ponto de acesso criado e irá procurar por um servidor **Privet** utilizando mDNS. Na sequência, serão fornecidas as credencias para acesso da rede e, opcionalmente, as credenciais para o registro na nuvem.

[1]Do inglês: *Service Set Identifier*.

Tabela 3.1: Argumentos adicionais no SSID

Nome	Caractere	*Bit*	Descrição
Configuração WiFi	0	0	1 - Caso o dispositivo necessite de alguma configuração de WiFi para funcionar corretamente; 0 - Caso o dispositivo já tenha sido configurado ou possua outra interface de comunicação (LAN por exemplo)
Registro Weave	0	1	1 - Caso o Dispositivo precise ser registrado durante a configuração; 0 - Caso ele já tenha sido registrado.
Conta de usuário	0	2	Caso o dispositivo precise configurar uma conta de usuário
Reservado	0	3-5	Reservado para expansões futuras. Deve ser preenchido com 0.
Suporta WiFi 2.4Ghz	1	0	Caso o dispositivo suporte WiFi 2.4Ghz
Suporta WiFi 5.0Ghz	1	0	Caso o dispositivo suporte WiFi 5.0Ghz.
Bluetooth LE	1	0	Caso o dispositivo possa ser descoberto por BLE.
Reservado	1	2-5	Reservado para expansões futuras. Deve ser preenchido com 0.

3.1.3 BLE

A documentação carece de informações sobre a descoberta e o gerenciamento de dispositivos através da interface BLE[2], porém, a implementação da libμWeave trabalha apenas com esta tecnologia.

O dispositivo atua como um servidor GATT, utilizando um UUID específico para identificar um dispositivo Weave. A transmissão dos dados é feita utilizando duas características GATT, uma para transmissão e outra para recepção dos dados. Para criptografia é utilizada a encriptação AES e para autenticação SHA256. No processo de pareamento, é utilizado o SPAKE2 para troca de chaves utilizando criptografia de curva elíptica NIST P-224. Assim como através de uma comunicação IP, também é utilizado o protocolo **Privet** para descoberta de um dispositivo BLE.

3.1.4 Nuvem

Uma vez registrado, o usuário pode requisitar informações de seus dispositivos para infraestrutura da Google através do envio de uma requisição GET para o endereço `https://www.googleapis.com/weave/v1/device`.

3.2 Provisionamento

Uma vez que um novo dispositivo é retirado da caixa e descoberto por algum dos meios de comunicação, ele precisa ser configurado e atribuído a um dono. Este procedimento

[2]Do inglês: *Bluetooth low energy*.

tem o nome de provisionamento e a primeira etapa é estabelecer um canal seguro entre os dispositivos utilizando um código de 4 dígitos.

O código pode ser gerado por um periférico de entrada, como um teclado. Pode ser um número gerado de maneira aleatória e exibido no visor do dispositivo, ou ainda, pode ser um número previamente definido pelo fabricante e impresso na base do dispositivo.

Uma vez que o canal seguro é estabelecido, são enviadas informações relevantes ao dispositivo, como as credenciais da rede e informações sobre o registro na nuvem. Após isto, o dispositivo pode se conectar à rede e completar o registro na nuvem.

3.3 Controle de acesso

Uma vez que um novo dispositivo é atrelado a um usuário, este passa a ser seu dono e pode controlá-lo remotamente através da infraestrutura de nuvem da Google. O usuário, dono do dispositivo, pode compartilhar o dispositivo com usuários e aplicações, usando diferentes regras de acesso.

Normalmente durante o provisionamento são definidas as regras de acesso para o dono e para o tipo de usuário robô. O título de dono possui regras de acesso irrestritas sobre controles comandos e estados do dispositivo. Atualmente, este título não pode ser transferido. O dispositivo pode ser compartilhado com aplicativos, outros usuários, grupos e domínios. Além do dono existem mais quatro níveis de acesso (Gerente, robô, usuário e visualizador), detalhados na Tabela 3.2.

Um gerente tem os mesmos privilégios de acesso de um dono, com exceção de compartilhar o dispositivo com aplicativos. Um robô não pode compartilhar o dispositivo, porém possui prioridade de execução de comandos. Um usuário pode utilizar um subconjunto de comandos e pode gerenciar os comando enviados por ele. O visualizador, como o nome sugere, possui as regras mais básicas de acesso, podendo apenas visualizar os estados dos dispositivos, sem enviar comandos.

Tabela 3.2: Resumo das regras de acesso Weave

Regra de acesso	Restrição de acesso	Prioridade de execução	Detalhes
Dono	1	2	Acesso irrestrito ao dispositivo.
Gerente	2	3	Não pode compartilhar com aplicativos.
Robo	3	1	Não pode compartilhar o dispositivo com nenhum usuário ou aplicativo. Possui prioridade na execução de aplicações.
Usuário	4	4	Enviar um subgrupo de comandos, visualizar e manipular apenas comandos enviados por ele.
Visualizador	5	5	Apenas visualizar os estados do dispositivo.

3.4 Esquemas

O emprego de esquemas permite a padronização para criação de novos dispositivos. Assim, aplicações clientes podem interagir com cada dispositivo através do esquemático definido

em cada dispositivo. Desta maneira os esquemas promovem uma padronização da interface para os mais diferentes tipos de dispositivos [38].

Os esquemas Weave definem *Traits* de características especificas do dispositivo.

3.4.1 *Traits*

Os *Traits* descrevem as partes lógicas do dispositivo, abstraindo seus módulos e descrevendo seus respectivos comandos e estados.

Existe um conjunto pre definido de *Traits*. Sendo eles: base, brilho, temperatura de cor, cores no espaço *CIE 1931 xyY*, sensor de umidade, sistemas de controle de ventilação e condicionamento de ar (HVAC[3]), fechadura, liga/desliga, impressora, executor de rotina, escaneadora, sensor de temperatura, controlador de temperatura, unidade de configuração de temperatura e volume.

Caso o dispositivo possua alguma das características dos *Traits* padrões, estes devem ser descritos exatamente como na referência. Os *Traits* padrões não podem ser modificados de nenhuma maneira. Novos *Traits* podem ser criados pelos desenvolvedores, porém, tendo o nome começado com um sublinhado ("_") e nunca devem sobrepor as funcionalidades dos *Traits* de referência.

3.4.2 Componentes

Componentes são a exposição de um conjunto de *Traits*. Por exemplo, um condicionador de ar pode possuir os seguintes componentes: base, controle, ambiente e visor.

O componente base abriga o *Trait* base que engloba as características comuns para dispositivos Weave, como por exemplo, versão do *firmware*, e se o dispositivo esta visível para uma descoberta local.

O restante dos componentes são variáveis de acordo com cada dispositivo implementado. O componente controle conteria os *Traits* de liga/desliga controlador de temperatura e HVAC. Já o componente ambiente seria um conjunto dos *Traits* de: sensor de temperatura e sensor de umidade. O componente visor poderia conter *Traits* de liga/desliga, brilho, e contraste. Os componentes deste dispositivo de exemplo estão representados na Figura A.1.

Os componentes também possuem atributos chamados de estados.

Estados

Seguindo o exemplo anterior, o valor atual do Brilho seria um estado do componente Visor.

Caso um cliente esteja interessado constantemente nos estados de um determinado dispositivo, ele pode se registrar para receber notificações de mudança de estado. O próprio dispositivo fica encarregado de propagar as mudanças de estado a seus subscritores. Esta propagação é feita utilizando XMPP[4].

[3]Do inglês: *Central Heating Ventilation and Air-Conditioning*.
[4]Do inglês: *Extensible Messaging and Presence Protocol*.

3.5 Comandos

Os *Traits* podem ou não abrigar um conjunto de comandos. Os comandos servem para requisitar os serviços de um determinado dispositivo. Por exemplo, o componente visor, pode possuir os seguintes *Traits*, liga/desliga, brilho e contraste. O *Trait* de brilho, pode receber comandos de definir um brilho especifico ou aumentar e diminuir o brilho.

Os comandos podem ser executados rapidamente, tendo sua resposta devolvida ao usuário de forma imediata, como também podem levar um tempo considerável para serem executados. Para isso são definidos mecanismos de manipulação dos comandos, como o cancelamento de um comando e a representação do estado deste. Os estados válidos para comandos são abortado, cancelado, concluído, erro, expirado, enfileirado ou em progresso.

3.6 Implementações

São disponibilizadas, pela Google, algumas implementações tanto para um dispositivo provedor de serviços e conteúdo, quanto para dispositivos clientes.

3.6.1 Cliente

As aplicações clientes são capazes de interagir com a nuvem e/ou com dispositivos locais, disparando comandos e consultando estados de dispositivos. As implementações até então disponibilizadas são: uma aplicação para Android, uma implementação em Python, um aplicativo do Google Chrome, e uma página da Internet chamada Weave Console.

weave_client

A aplicação **weave_client** [26] é uma implementação em linguagem Python que permite a descoberta dos dispositivos na rede local. Um diferencial desta variação é a possibilidade do envio de comandos pela rede local com um dispositivo que não foi previamente pareado, informando apenas a chave do dispositivo. Esta aplicação também permite a listagem completa dos dispositivos conectados à nuvem, bem como o envio de comandos.

Weave Console

A gerência dos dispositivos é feita acessando a página do **Weave Console** [31]. Ela faz a listagem de todos os dispositivos atrelados à conta. Selecionando um dispositivo é possível visualizar todos os seu componentes e um registro completo do dispositivo, com os comandos enviados e alterações de estados, bem como a possibilidade de disparar novos comandos.

Chrome

A aplicação para Google Chrome expõe a interface de controle de forma semelhante ao Weave console e também, acrescenta a funcionalidade de descoberta de dispositivos na rede local ou através do BLE. Atualmente esta aplicação foi removida da loja de aplicações do Chrome e não pode ser mais obtida.

Android

São disponibilizadas duas aplicações de exemplo do uso da API Weave para Android, a **Weave LED Toggler Quickstart** [32] e a **Weave Playground Android app** [33].

Ambas aplicações interagem com um terceiro aplicativo (Weave [27]) solicitando a lista de dispositivos atrelados à conta do usuário. O aplicativo Weave estava disponível na loja de aplicativos, mas recentemente foi removido.

3.6.2 Dispositivo

Um dispositivo que provê estados e serviços, atuando como servidor deve utilizar a implementação de referência do Weave, chamada libWeave [29]. Uma versão alternativa a esta implementação é a libμWeave [28], sendo esta uma versão para trabalhar em dispositivos restritos.

libWeave

A libWeave provê uma implementação de código aberto do protocolo Weave a ser utilizado no dispositivo, sendo portátil e bem testada [35].

A Figura 3.1 apresenta a arquitetura da libWeave em um sistema Debian. A implementação desta biblioteca prevê o uso de invólucros para interagirem com os componentes de cada sistema operacional. Tais invólucros implementam funções de gerenciamento das interfaces de comunicação. Para Linux as implementações típicas de invólucros são apresentadas na Tabela 3.3.

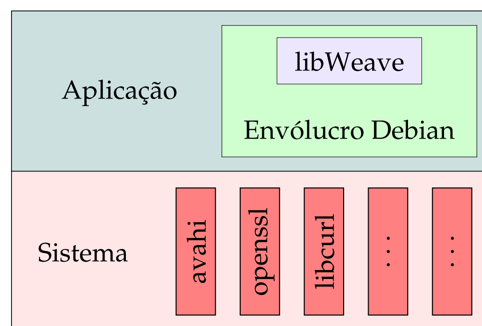

Figura 3.1: Arquitetura libWeave, adaptado de [29]

Na Tabela 3.4 estão apresentados os requisitos da atual implementação da libWeave com seus respectivos invólucros para um sistema Linux.

Brillo [25] é um sistema operacional para IoT desenvolvido para ser uma alternativa ao Android [24]. Ele incorpora de forma nativa a libWeave, trazendo uma implementação de código aberto de seus respectivos invólucros. Mesmo assim, o Brillo é baseado em Linux e tem requisitos semelhantes a solução implementada em outras distribuições Linux [39].

Tabela 3.3: Invólucro libWeave

Componente	Típica implementação Linux	Descrição
Cliente HTTP(s)	libcurl	Fazer requisições HTTP(s)
Servidor HTTP(s)	micro_httpd	Expõe a API de controle local
Gerenciamento de rede	wpa_supplicant, hostapd	Listar interfaces de rede, abrir sokets e criar pontos de acesso WiFi
Descoberta local	avahi	Descoberta DNS-SD (mDNS)
Suporte TLS	openssl	Gerenciar conexões TLS
Módulo BLE	bluez	Suporte para o modo periférico Buetooth

Tabela 3.4: Requisitos libWeave + invólucro Linux

Componente	Requisito
Arquitetura	x86, x64, ARMv7+, ARM64, MIPS ou MIPS64
Sistema Operacional	Linux
Memória RAM	pelo menos 32 MB
Armazenamento	Sistema atual + 8 MB
WiFi	Obrigatoriamente 2.4GHz e opcionalmente 5.0 GHz
BLE	Atualmente não é suportado

libμWeave

Conforme enfatizado na Tabela 3.4 a atual implementação da libWeave demanda um dispositivo com alta capacidade computacional. Logo, o emprego desta biblioteca em microcontroladores se torna inviável. A libμWeave é uma versão projetada para microcontroladores em contraste a libWeave que roda em um ambiente Linux completo [28, 39]. Esta versão trabalha apenas com a tecnologia *Bluetooth* assim, as requisições vindas da nuvem são entregues ao dispositivo somente de forma indireta. Os dados são codificados no formato CBOR. Este é baseado em JSON e uma codificação binária, assim, demanda menos recursos do dispositivo.

Atualmente a única plataforma suportada é Nordic SDK nrf51 e nrf52. Sendo esta equipada com um chip nRF52832, possuindo um processador ARM Cortex-M4 32-bit rodando a 64 MHz, tendo duas opções de capacidade de armazenamento: 512 kB flash/64 kB RAM; 256 kB flash/32 kB RAM. Esta plataforma de *hardware* também possui um módulo BLE e um receptor NFC[5] [57, 58].

Como os esforços para este projeto são relativamente novos, ele ainda se encontra em estágio muito inicial de desenvolvimento, não estando ainda praticável para produção [28]. Devido as restrições encobradas pela libμWeave, optamos por utilizar a implementação da libWeave. Uma vez que ambas implementações utiliza o mesmo protocolo de comunicação, seria relativamente fácil portar a plataforma desenvolvida para libμWeave.

A fim de complementar a funcionalidade dos dispositivos Weave e tendo como base o estudo sobre mecanismos de execução de código de usuário em dispositivos restritos

[5] Do inglês: *Near Field Communication*.

utilizamos a COISA-VP, a qual é discutida no Capítulo 4

Capítulo 4

COISA

A *Constrained OpenISA Virtual Platform* [10, 53], ou COISA-VP, é uma plataforma para execução de aplicações de usuário voltada para dispositivos com poder computacional e memória restritos, como microcontroladores. Ela faz uso de uma máquina virtual de processos em nível de ISA, sendo capaz de executar um arquivo binário OpenISA já compilado e otimizado em qualquer plataforma de *hardware*. Uma representação da implementação do COISA-VP para plataforma ATMega328 é apresentada na Figura 4.1.

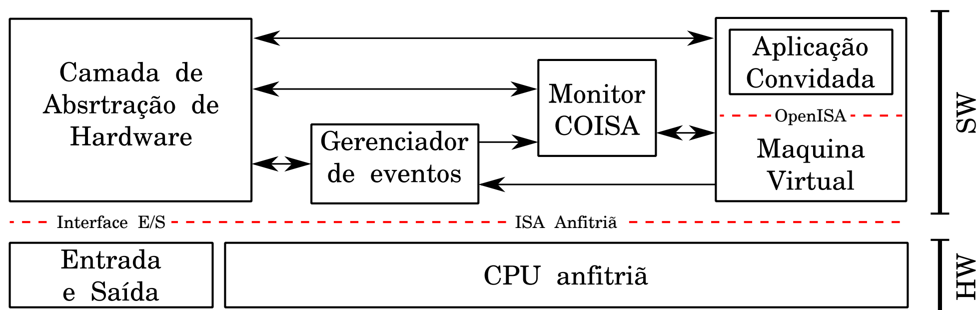

Figura 4.1: Arquitetura da COISA-VP em um microcontrolador ATmega328

Os principais componentes do COISA-VP são: O Monitor COISA-VP, responsável por gerenciar a plataforma, comandando a execução da máquina virtual e operando as chamadas de sistema; um Gerenciador de Eventos, responsável por gerenciar os componentes capazes de gerar eventos e adicioná-los na fila de eventos; uma Camada de Abstração de *hardware*, responsável por prover uma interface abstrata para interação com o *hardware* e o restante da plataforma; e uma Máquina Virtual, responsável por executar as aplicações do usuário.

Esta organização permite a construção de uma pilha de *software* compacta e que consuma pouquíssima memória. Quando executado no ATmega328 a COISA-VP consome apenas 348 *bytes* de memória volátil (RAM) e 6 kB de memória não volátil (*flash*), deixando aproximadamente 1.7 kB, dos 2 kB, da RAM para a aplicação do usuário [53]. As próximas seções descrevem os componentes do COISA-VP em detalhe.

4.1 Camada de abstração de *hardware*

A camada de abstração de *hardware* é o componente responsável por interagir com o *hardware* do dispositivo, como realizar a leitura de dados de sensores e controle dos atuadores. O código é escrito em linguagem C e não faz uso de nenhuma funcionalidade provida por bibliotecas. Além disso, todos os módulos dependentes de plataforma são agrupados em uma camada de abstração de *hardware*. Esta abordagem permite com que seja compilado facilmente para uma grande variedade de plataformas até mesmo as mais restritas. Conforme destacado por Millani e outros [10, 53] cerca de 10 linhas de código foram modificadas para execução em outra plataforma.

4.2 Máquina Virtual

A COISA-VP faz uso de uma máquina virtual de processo em nível de ISA. Tornando a plataforma capaz de executar programas compilados para arquitetura OpenISA e MIPS-I. Estes programas são chamados de programas convidados e executam na máquina virtual hospedeira do COISA-VP. Através deste componente o programa convidado pode executar instruções em nível de usuário, acessando registradores e memória reservados a este. Operações de entrada e saída são desempenhadas através de chamadas de sistema. De forma sucinta, este componente interpreta as instruções do programa convidado, detém os registradores da arquitetura e é responsável pelo gerenciamento da pilha do programa convidado.

4.3 Gerenciador de Eventos

O usuário pode desenvolver programas simples que executam uma computação única imediatamente após a instalação do programa na plataforma. Caso o programa do usuário tenha a necessidade de tomar uma ação baseada no constante monitoramento de um sensor ele pode entrar em um laço de repetição, fazendo uma espera ocupada. No entanto, este tipo de abordagem pode ocasionar um mal uso da unidade de processamento do dispositivo. Dessa forma, o COISA-VP foi desenvolvido para permitir que o usuário implemente programas orientados a eventos, ou seja, que reajam a eventos do sistema através da execução de rotinas de tratamento de eventos. Neste caso, a função principal do programa (`main`) registra, através de chamadas de sistemas, as rotinas de usuários que devem ser chamadas para tratar os respectivos eventos.

Os eventos podem ser gerados tanto pela máquina virtual quanto pela camada de abstração de *hardware*. A camada de abstração de *hardware*, por exemplo, pode gerar eventos em resposta à mudança de valores de sensores do *hardware*. Uma vez que um evento é gerado, ele é enviado para o gerenciador de eventos, que o armazena em uma fila e aciona a máquina virtual para que a mesma execute a rotina de tratamento registrada pelo usuário quando o evento estiver na cabeça da fila.

O gerenciador de eventos do COISA-VP também é responsável por descartar eventos que o usuário não tem interesse de tratar, ou seja, aqueles eventos para os quais não foram

registrados tratadores de eventos.

Para fazer uso do Gerenciador de Eventos do COISA-VP, o programa do usuário deve realizar chamadas de sistema para: (1) Solicitar a inicialização do Gerenciador de Eventos, para que os recursos necessários deste componente sejam inicializados; (2) Registrar as rotinas de tratamento para os eventos de interesse, informando a característica desejada - por exemplo, um botão seja pressionado - e o endereço da rotina responsável pelo tratamento do evento; (3) Opcionalmente, remover o tratador de eventos previamente registrado, quando não há mais interesse.

4.4 Monitor COISA-VP

O Monitor COISA-VP é o componente responsável por carregar os programas convidados na memória da máquina virtual e disparar a execução da rotina principal. Este componente também pode ser utilizado para controlar a execução da aplicação convidada, permitindo uma execução passo a passo. Este gerenciador permite que usuários externos façam o gerenciamento do sistema e atualizações nas aplicações convidadas.

Exemplificando, um usuário envia uma aplicação OpenISA a ser executada no dispositivo. O gerenciador do COISA-VP recebe o arquivo binário e o carrega na memória da máquina virtual. Com o programa carregado, o gerenciador invoca periodicamente uma rotina da máquina virtual para executar um conjunto de instruções da aplicação convidada. O gerenciador COISA-VP também invoca periodicamente o Gerenciador de Eventos para este (1) verificar os componentes do dispositivo e possivelmente enfileirar eventos e (2) definir os registradores da Máquina Virtual a fim de executar os eventos enfileirados.

Foram definidas chamadas de sistema para: indicar o término da execução de um procedimento convidado; inicializar a pilha de programa; configurar o Gerenciador de Eventos; e efetuar operações de entrada e saída em periféricos através da camada de abstração de *hardware* [52].

Quando um programa convidado faz o uso da pilha de programas, ele deve solicitar sua inicialização através de uma chamada de sistema. Foi definida uma chamada de sistema específica para a manipulação de periféricos de entrada e saída. Esta tem como parâmetros um vetor de caracteres que pretende identificar um determinado periférico e um identificador de 32 *bits* para o tipo de operação de cada periférico.

A máquina virtual orientada a eventos da COISA-VP permite, através de chamadas de sistema, registrar eventos de interesse para a chamada de procedimentos do programa de usuário na ocorrência do evento. Esta máquina implementa um modelo de execução colaborativa onde, através de uma chamada de sistema, o programa de usuário indica o término da execução de um procedimento.

O Gerenciador COISA-VP é responsável por invocar o Gerenciador de Eventos, este faz a checagem dos componentes capazes de gerar eventos

Na existência de algum evento, o Gerenciador de Eventos então, define as variáveis da máquina virtual para execução da rotina de usuário cadastrada. Após este passo, o controle é devolvido ao Gerenciador COISA-VP que coloca novamente a máquina virtual

em estágio de execução, mas desta vez o procedimento de usuário executado é a rotina de tratamento do evento.

4.5 Programação

A programação do COISA-VP pode ser feita em qualquer linguagem capaz de gerar código para a arquitetura OpenISA ou MIPS-I. Além das linguagens de programação padrões como C/C++, sua programação também tem sido feita pela linguagem de blocos, através de uma extensão da biblioteca Blockly [1]. A Figura 4.2 exibe a interface de programação do Blockly.

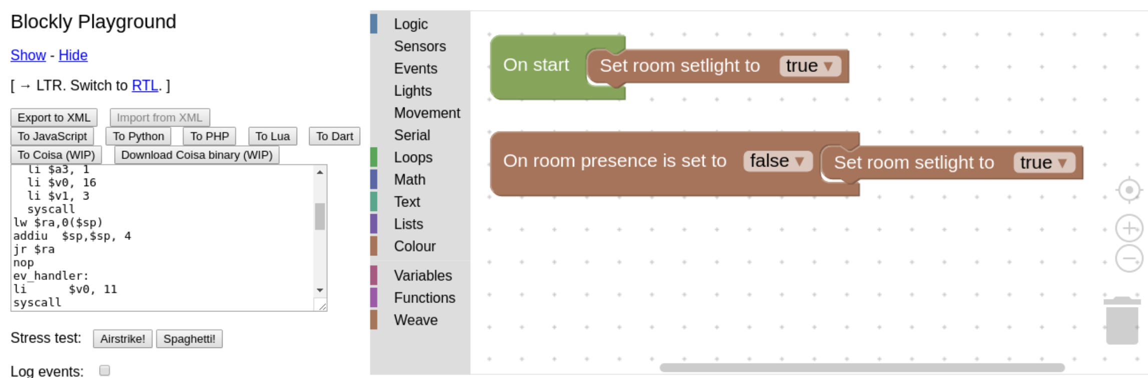

Figura 4.2: Programa de usuário na interface Blockly

O infraestrutura do Blockly foi modificada para gerar automaticamente código em linguagem de montagem e linguagem de máquina OpenISA a partir dos programas feitos com blocos.

A aplicação de usuário gerada a partir do programa da Figura 4.2 é apresentada em linguagem de montagem nas Listagens 4.1, 4.2, 4.3 e 4.4. A Listagem 4.1 mostra o código da função de inicialização (*start*), que invoca a chamada de sistema número onze para inicializar a pilha do programa, invoca a chamada de sistema número nove para registrar o endereço do tratador de eventos do programa de usuário e, finalmente, chama a função principal do programa do usuário (*main*).

Listagesm 4.1: Início do programa de usuário gerado pelo Blockly

```
1  start:
2  // Inicializacao da pilha
3  li $v0, 11
4  syscall
5  // Registro da rotina a ser executanda antes de todo evento
6  li $v0, 9
7  li $v1, ev_handler
8  syscall
9  // Devio para a funcao principal
10 jal main
11 nop
12 // Indica, para a plataforma COISA, o termino do procedimento
13 li $v0, 10
14 syscall
```

A Listagem 4.2 apresenta os procedimentos para tratamento de eventos. A rotina *ev_handler* é responsável por iniciar o ambiente virtual, invocar a rotina de tratamento apropriada para tratar o evento e, por fim, invocar a chamada de sistema número dez para indicar o término do tratamento do evento para o COISA-VP.

Listagesm 4.2: Procedimentos de tratamento de eventos do programa de usuário gerado pelo Blockly

```
1  // Rotina de tratamento do evento
2  on_room_presence_procedure:
3  // Empilha o endereco de retorno
4  addiu $sp,$sp,-4
5  sw $ra,0($sp)
6  // Chamada de sistema de comunicacao, com a referencia para o comando
7    la $a0, room_set_light
8    li $a3, 0
9    li $v0, 16
10   li $v1, 3
11   syscall
12 // Restaura o endereco da pilha e retorna
13 lw $ra,0($sp)
14 addiu $sp,$sp, 4
15 jr $ra
16 nop
17
18 // Funcao a ser chamada antes do tratamento de todo evento
19 ev_handler:
20 // Reinicializacao da pilha
21 li $v0, 11
22 syscall
23 // Salva o endereco para o retorno e chama o procedimento que tratara este evento
24 li $ra, event_end
25 jr $a0
26 nop
27 // Indica, para a plataforma COISA, o termino do procedimento
28 event_end:
29 li $v0, 10
30 syscall
```

Depois do ambiente da máquina virtual ser inicializado pela função *start*, o fluxo de execução é desviado para a função *main*. Esta função, representada na Listagem 4.3, é responsável por registrar os tratadores dos eventos de interesse e executar os comandos listados dentro do bloco *onStart*.

Listagesm 4.3: Função principal do programa de usuário gerado pelo Blockly

```
1  // Funcao principal do usuario
2  main:
3  // Armazena o endereco de retorno na pilha
4  addiu $sp,$sp,-4
5  sw $ra,0($sp)
6  // Registra o evento e sua funcao de tratamento
7  li $a0, 10
8  la $a1, on_room_presence_procedure
9  la $a2, set_room_presence_false
10 li $v0, 14
11 syscall
12 // Restaura o endereco da pilha e retorna
13 lw $ra,0($sp)
14 addiu $sp,$sp, 4
15 jr $ra
16 nop
```

No final do arquivo, conforme representado na Listagem 4.4, é adicionada a seção de dados da aplicação.

Listagesm 4.4: Seção de dados do programa de usuário gerado pelo Blockly

```
1 // Constantes do programa
2 .data
3 set_room_presence_false: .asciiz "ROPN"
4 room_set_light: .asciiz "component._room.setLight.light"
```

Uma vez que o código é gerado, ele é montado (convertido para o formato binário) e então enviado ao COISA-VP para execução.

Capítulo 5

PROST - PROgrammable Smart Things

Com o objetivo de permitir a execução de código como serviço em dispositivos auto-descritivos da IoT, propomos a implementação de uma plataforma que combine tecnologias de alto padrão para entregar uma solução eficiente e que consuma poucos recursos computacionais. Nossa solução é voltada para dispositivos computacionalmente restritos, como microcontroladores com 4-8 kB de RAM. Atualmente a implementação da libμWeave está em estágio muito inicial, tendo o uso restrito à tecnologia *bluetooth* e possuindo implementação para somente duas plataformas da Nordic Semiconductor. Assim, optamos pelo desenvolvimento desta plataforma utilizando a libWeave como base por se encontrar em um estágio de desenvolvimento mais avançado, suportando as funcionalidades requeridas para o desenvolvimento da PROST. Na Seção 5.1 é apresentada uma visão geral sobre a libWeave e na Seção 5.2 são discutidas as adaptações feitas na biblioteca para promover a implementação da PROST.

5.1 LibWeave

O código da libweave pode ser obtido através do repositório `https://weave.googlesource.com/weave/libweave/`

A Figura 5.1 apresenta uma visão geral da libWeave. A aplicação de usuário, assim como os componentes base da libWeave interagem com as interfaces de comunicação através de uma camada denominada *provider*.

Os componentes internos da libWeave estão todos contidos no diretório `src` e seus subdiretórios - `commands`, `privet` e `states`. No diretório `src`, se encontram os seguintes arquivos:

base_api_handler.cc Configurações genéricas de dispositivos Weave, por exemplo, versão do *firmware*, controle de descoberta local, informações de rede. Também define os nível de privilégio para execução dos comandos referentes ao dispositivo, por exemplo, para executar o comando *reboot* o nível do usuário deve ser pelo menos administrador.

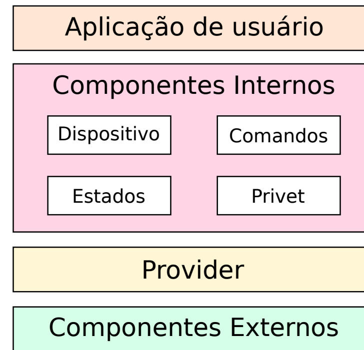

Figura 5.1: Visão geral da libWeave

access_api_handler.cc Parte análoga ao `base_api_handler.cc` mas para acesso, contendo uma lista de acessos revogados. Bloqueio do acesso em caso de tempo expirado ou revogação de acesso, função para listar comandos, funções que são chamadas quando os comandos terminam ou quando ocorre uma atualização de estados.

access_revocation_manager_impl.cc Manipula o arquivo de configuração (`/var/lib/weave/weave_settings_[model_id]_config.json`). Este arquivo contém o código responsável por manter de forma persistente as informações de acesso à nuvem. Faz a verificação se *tokens* estão expirados e permite a adição de *callbacks* quando uma nova entrada de acesso é adicionada.

backoff_entry.cc Implementa funções relacionadas a retransmissão de pacotes na rede (*backoff*).

component_manager_impl.cc Possui funções para remover componentes não acessíveis, procurar, adicionar e remover componentes. Registra *callbacks* para adição de componentes. Faz o *parser* dos comandos recebidos e verifica se ele corresponde a um comando válido.

config.cc Estrutura para armazenar chaves do dispositivo (identificador do cliente, versão, nome, descrição e outros)

data_encoding.cc Codificação e decodificação de arquivos binários.

device_manager.cc Expõe os componentes internos do dispositivo.

device_registration_info.cc Contém componentes para conexão e sincronismo com a nuvem da Google.

error.cc Decodifica e imprime mensagens de erro.

http_constants.cc Constantes HTTP (GET, POST e outros).

json_error_codes.cc Códigos de erro JSON.

registration_status.cc Constantes sobre os estados de registro.

streams.cc Leitura e escrita na memória. Manipulação do *buffer* interno.

string_utils.cc Manipuladores de *strings*.

utils.cc Conversão de tempo. Divisão de *strings* grandes. Opções do JSON.

Arquivos do diretório `src/commands` da libWeave:

cloud_command_proxy.cc Faz a intermediação para publicar atualizações na nuvem (utiliza funções do `device_registratio_info.cc` para despachar as mensagens).

command_instance.cc É instanciado por um nome de comando no formato "<NomeDoPacote>.<NomeDoComando>" e sua respectiva lista de parâmetros. Faz o parser do JSON criando e o armazenando em uma estrutura própria.

command_queue.cc Funções para tratar a fila de comandos, adicionando e removendo notificações de *callback* para quando um comando é adicionado ou removido, limpando a fila e adicionando o manipulador de comandos.

schema_constants.cc Algumas constantes referentes aos esquemas Weave.

notification_parser.cc Analisa as notificações do tipo "**created**" e "**deleted**" no formato JSON e invocam os métodos responsáveis pelo tratamento destas. Tais notificações são recedidas pela nuvem (GCD).

pull_channel.cc Disparado em um intervalo constante de tempo, é utilizado para enviar notificações de atualização para nuvem (GCD).

xml_node.cc Classe utilizada para representar a árvore de um documento XML. Utiliza a biblioteca "eXpat" para criar instâncias de *streams* XML.

xmpp_channel.cc Interface utilizada para abstrair o canal de envio de mensagens XMPP.

xmpp_iq_stanza_handler.cc Utilizado para processar informações de requisição e modificação de informações recebidas pelo canal XMPP.

xmpp_stream_parser.cc Utilizado para tratar o *stream* recebido, processando os dados na medida em que chegam.

Arquivos do diretório `src/privet` da libWeave:

state_change_queue.cc Utiliza *Macaroom* para criar e manipular *tokens* de acesso, confirmando o acesso a um cliente, definindo ou reiniciando uma chave de autorização.

cloud_delegate.cc Interface que promove acesso as funcionalidade GCD ao gerenciador Privet.

constants.cc Definição de constantes.

device_delegate.cc Interface para o *provider* acessar informações gerais do dispositivo. Recuperar informações sobre as portas padrões e *timeout* HTTP/HTTPS. Também permite agendar tarefas de *background* utilizando o *TaskRunner*.

device_ui_kind.cc Confere e retorna o tipo do dispositivo (AC: accessPoint AK: aggregator AM: camera AB: developmentBoard AH: acHeating AI: light AO: lock AE: printer AF: scanner AD: speaker AL: storage AJ: toy AA: vendor AN: video)

openssl_utils.cc Autenticador de mensagens.

privet_handler.cc Manipula requisições do tipo HTTP/HTTPS relacionadas ao protocolo *privet*.

privet_manager.cc Obtém o SSID ao qual o dispositivo está atualmente conectado. Tratador de requisições e respostas. Dispara uma *callback* na mudança do estado de pareamento. Caso o dispositivo esteja registrado, envia os valores de atualização.

privet_types.cc Declaração de alguns tipos de dados.

publisher.cc Publica e atualiza informações através do protocolo DNS-SD e remove o serviço se a rede local deixa de ser utilizada.

security_manager.cc Faz o controle de acesso do protocolo *Privet*.

wifi_bootstrap_manager.cc Interface para o manipuladores WiFi. Possui comandos como ligar e desligar o modulo WiFi, escanear dispositivos WiFi e gerenciar o SSID.

wifi_ssid_generator.cc Gerador de SSID randômico.

Arquivos do diretório `src/states` da libWeave:

state_change_queue.cc Mantém uma fila de requisições para mudanças de estado. Também armazena um histórico de atualização de estados que pode ser recuperada e esvaziada.

5.2 Introspecção

Na classe *BaseApiHandler* definida no arquivo `src/base_api_handler.cc`, foi acrescentado um *Trait* para execução de código. Este é integrado ao componente "base", junto aos outros *Traits* "device" e "privet". Desta maneira a nova funcionalidade fornecida fica agrupada com as características básicas do dispositivo, permitindo que as aplicações já desenvolvidas para a libWeave possam executar sem nenhuma modificação na PROST. Um JSON que representa o *Trait* "_coisa" é apresentado na Listagesm 5.1.

Listagesm 5.1: JSON de descrição do *Trait* _coisa

```
{
  "_coisa": {
    "commands": {
      "setConfig": {
        "minimalRole": "user",
        "parameters": {
          "sendByteCode": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

Como os tipos de dados aceitos como argumentos não possuem nenhuma opção para o envio de arquivos binários, optamos por codificar os programas de usuário em uma *string* utilizando a codificação base64. Assim, promovemos uma extensão da aplicação em *python* para efetuar a codificação e implementamos um método para decodificação na libWeave.

Para o tratamento do comando recebido, foi adicionado ao tratador de comandos o procedimento "InstallCoisaCode", conforme ilustrado na Listagesm 5.2

Listagesm 5.2: Registrando o procedimento ao tratador de comandos da libWeave

```
device_->AddCommandHandler(kDeviceComponent,
                           "_coisa.setConfig",
                           base::Bind(&BaseApiHandler::InstallCoisaCode,
                           weak_ptr_factory_.GetWeakPtr()));
```

O procedimento "InstallCoisaCode" descrito na Listagesm 5.3 é disparado toda vez que um comando "_coisa.setConfig" é recebido, o procedimento então realiza a averiguação de seus parâmetros. Caso o parâmetro não seja recebido ou encontre algum erro em sua decodificação, o comando é finalizado informando uma mensagem de erro.

Listagesm 5.3: Procedimento para instalação do código na máquina virtual e início da execução

```
1  void BaseApiHandler::InstallCoisaCode
2                        (const std::weak_ptr<Command>& command) {
3      auto cmd = command.lock();
4      if (!cmd) return;
5      LOG(INFO) << "Received command: " << cmd->GetName();
6      const auto& params = cmd->GetParameters();
7      std::string requested_code_to_execution;
8      if (params.GetString("sendByteCode", &requested_code_to_execution)
9          && tm.InstallCode(requested_code_to_execution))
10     {
11         std::thread t1(&BaseApiHandler::RunCoisaCode, this);
12         t1.detach();
13         return;
14     }
15     weave::ErrorPtr error;
16     weave::Error::AddTo(&error, FROM_HERE, "invalid_parameter_value",
17                         "Invalid parameters");
18     cmd->Abort(error.get(), nullptr);
19 }
```

Uma nova *thread* é responsável pela execução de cada programa convidado. O procedimento executado na *thread* é apresentado na Listagesm 5.4. O procedimento executa um conjunto de instruções do programa convidado até este efetuar uma chamada de sistema que indica o retorno de uma função. Esta chamada faz com que o procedimento *ExecuteStep* retorne um valor diferente de 0. Então é verificado se existe algum evento na fila a ser consumido, caso sim, ele é consumido inteiramente até ceder a CPU. A *thread* aguarda a liberação de um semáforo que é incrementado quando um novo evento é inserido na fila de eventos e o processo se repete até que o comando seja cancelado pelo usuário ou por um gerenciador do dispositivo.

Listagesm 5.4: Procedimento para execução da máquina virtual e consumo de eventos

```
1  void BaseApiHandler::RunCoisaCode(
2                        const std::weak_ptr<weave::Command>& command) {
3      auto cmd = command.lock();
4      if (!cmd)
5          return;
6      while(cmd){
7          sem_wait(&mutex);
8          while (!tm.ExecuteStep());
9          while (consume_event()==1){
10             while (!tm.ExecuteStep());
11         }
12     }
13 }
```

5.2.1 Geração de eventos

O PROST gera eventos para a COISA-VP automaticamente sempre que há modificações nos estados do dispositivo IoT. Para isso, o arquivo `src/device_registration_info.cc` foi modificado de forma que ele encaminhe o conjunto de atualizações dos estados do dis-

positivo para outro procedimento. Esta função tem como parâmetro uma lista de valores representada de maneira análoga a um objeto JSON. A lista dos estados modificados é varrida e caso o usuário tenha registrado interesse em algum desses estados, um evento é inserido na fila de eventos. Posteriormente o evento é consumido na *thread* RunCoisa-Code. Para isso, também, foi alterado o tratador de eventos do COISA-VP (EH) para salvar os argumentos dos eventos registrados, como por exemplo o limiar superior ou inferior de algum estado.

A fim de tornar mais flexível a implementação e eliminar a necessidade da configuração manual de cada estado, seria interessante a migração deste procedimento para o EH.

5.2.2 Execução de comandos do dispositivo a partir de código de usuário

A fim de disparar comandos para o dispositivo, foi incorporada a implementação cliente desenvolvida em *python* na libWeave. Esta permite disparar comandos - manipulando estados - tanto para o dispositivo que utiliza esta implementação da biblioteca quanto para qualquer outro dispositivo com as devidas permissões registradas. O código de usuário permite a invocação dos comandos a partir de uma chamada de sistema, mais especificamente, a chamada de número 16. Esta chamada tem o propósito de gerenciar as operações da libWeave. A fim de generalizar a chamada são utilizados dois argumentos: (1) um código de operação, utilizado para determinar entre o envio de comandos ou recuperação de estados e (2) e um apontador para um vetor de caracteres, neste caso, utilizado para indicar o nome do componente, *Trait*, comando e seus argumentos.

5.2.3 Visão geral

Para ilustrar a implementação a Figura 5.2 traz a representação da sequência de instalação e execução do código de usuário. Iniciando-se com um usuário disparando um comando de instalação de código COISA-VP. O comando é encaminhado através da libWeave para o tratador base (o qual registrou o comando). Então o parâmetro contendo o binário na base64 é decodificado e carregado na memória principal da máquina virtual. O resultado deste procedimento é retornado ao usuário através da libWeave enquanto uma nova *thread* (representada em vermelho) é criada para execução do comando. Esta nova *thread* executa as instruções carregadas até a unidade de processamento ser cedida pelo programa convidado. Na sequência, o tratador de eventos tem a oportunidade de processar os eventos, definindo os parâmetros da máquina virtual para execução dos tratadores do usuário. Este ciclo então se repete até que o comando seja cancelado pelo usuário que instalou o código ou um administrador do dispositivo.

A Figura 5.3 apresenta a inserção de um evento na fila de eventos da COISA-VP. Primeiramente a libWeave chama o procedimento previamente registrado para ser executado na alteração de qualquer evento. Então a classe base da aplicação efetua a verificação do evento gerado e envia um comando para o Gerenciador de Eventos inserir este evento na fila.

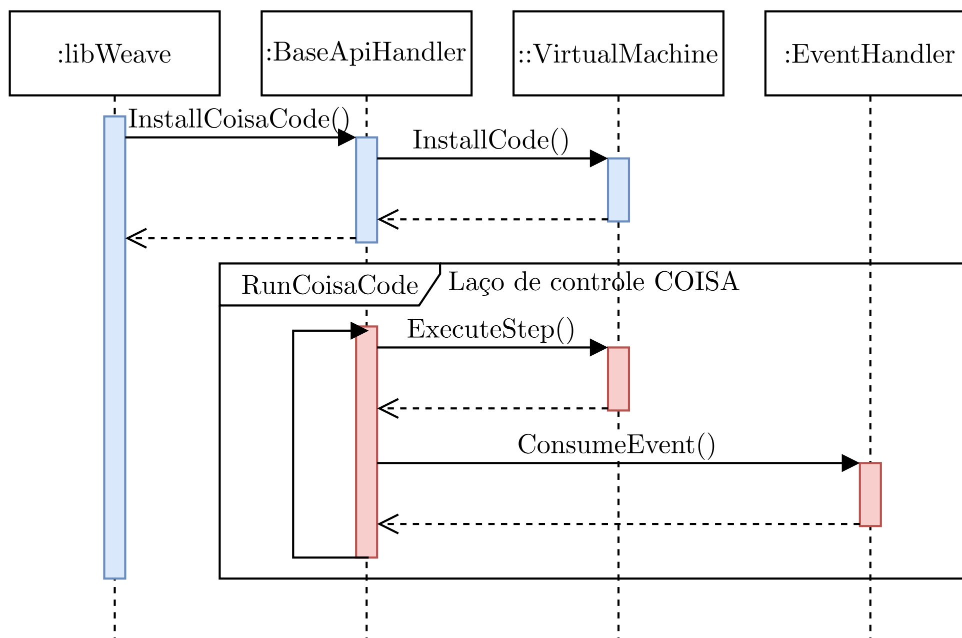

Figura 5.2: Diagrama de sequência da instalação e execução de código de usuário

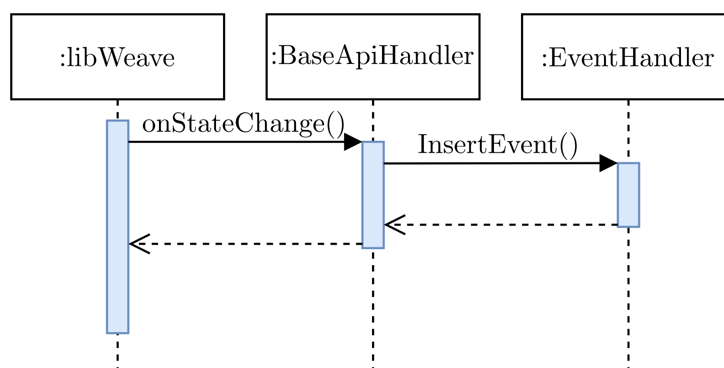

Figura 5.3: Diagrama de inserção de eventos na PROST

Capítulo 6

Resultados experimentais

A fim de validar a infraestrutura desenvolvida, são propostos alguns casos de teste. Estes compreendem a programação de uma aplicação para libWeave em um determinado cenário e a escrita de uma aplicação COISA-VP que interaja com os componentes deste cenário.

Além das modificações realizadas na libWeave, descritas no Capítulo 5, a interface *blockly* foi estendida para agregar os blocos destes casos de teste e, então, os programas de usuário foram codificados a partir desta interface.

6.1 Materiais e métodos

Para realização dos experimentos foi utilizado um laptop com o processador de arquitetura X86_64 (Intel® Core™ i7-2630QM), com 8 GB de RAM, executando o sistema operacional Linux Mint 17.3. Foi utilizado o compilador g++ na versão 4.8.5. Nenhuma dependência, além das já existentes pela libWeave, foi adicionada pela extensão do PROST.

6.2 Casos de teste

Propomos o desenvolvimento de cenários fictícios que ilustram possíveis aplicações do PROST. Os cenários são: uma sala inteligente, um robô e um carro inteligente; os quais são apresentados nas próximas subseções.

6.2.1 Caso 1: Sala inteligente

Imaginamos para este cenário uma sala que possui controle das lâmpadas do ambiente e do aparelho de ar condicionado. Esta sala inteligente possui um dispositivo central, responsável por controlar o aparelho de ar condicionado e as lâmpadas.

Descrição do dispositivo

Para o controle da sala inteligente foi definido um único *Trait*, denominado *_room*, que é apresentado na Listagesm 6.1. O *Trait _room* descreve comandos para ligar ou desligar tanto a lâmpada quanto o aparelho condicionador de ar. O aparelho condicionador de ar foi definido desta maneira a fim de controlarmos a temperatura da sala baseando-se

unicamente na leitura de estados da mesma. O dispositivo central conta com quatro estados dos quais "*AC*" e "*light*" representam um valor binário correspondente à situação do ar condicionado e da lâmpada. Já os estados "*temperature*" e "*presence*" representam estados de sensores instalados na sala para determinar a temperatura e indicar se existe ou não alguém na sala.

Listagesm 6.1: JSON de descrição do *Trait _room*

```
{
  "_room": {
    "commands": {
      "setAC": {
        "minimalRole": "user",
        "parameters": {
          "onOff": {
            "type": "boolean"
          }
        }
      },
      "setLight": {
        "minimalRole": "user",
        "parameters": {
          "light": {
            "type": "boolean"
          }
        }
      }
    },
    "state": {
      "AC":{
        "isRequired": true,
        "type": "boolean"
      },
      "temperature": {
        "isRequired": true,
        "maximum": 50,
        "minimum": -10,
        "type": "number"
      },
      "light": {
        "isRequired": true,
        "type": "boolean"
      },
      "presence": {
        "type": "boolean"
      }
    }
  }
}
```

Os comandos *setAC* e *setLight* são registrados no tratador de eventos da libWeave, para que, quando os comandos forem chamados por um cliente, os procedimentos referentes a seu tratamento sejam executados. Quando seus respectivos procedimentos são executados, eles verificam os argumentos recebidos pelo comando, alterando os estados do dispositivo Weave. Este é o procedimento que os atuadores são, de fato, acionados.

Para o teste do dispositivo foi escrito um programa de usuário que interage com os periféricos do dispositivo. Como o código de usuário reage a estímulos externos, como por exemplo à chegada de uma pessoa na sala, foram induzidas alterações nos estados do dispositivo para simular um ambiente real. As alterações de estados são feitas através da API padrão da libWeave, assim como outras aplicações que utilizam esta biblioteca.

Para exercitar as funcionalidades da plataforma, as mudanças de estados são feitas com o passar do tempo. Neste caso, a temperatura da sala sobe caso o aparelho condicionador de ar esteja desligado e cai quando ele está ligado e o sensor de presença inverte seu estado binário a cada 30 segundos.

Código de usuário

Utilizando a infraestrutura do Blockly, geramos um programa simples para testar a sala inteligente, representado na Figura 6.1. Este mesmo programa de usuário é apresentado, em linguagem de montagem, na Listagem C.1.

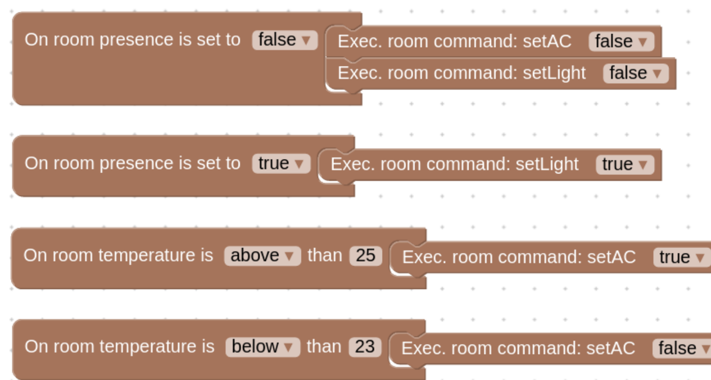

Figura 6.1: Programa de usuário executado na sala inteligente

O programa da Figura 6.1 trata quatro tipos de eventos:

- o primeiro tipo é gerado quando todos deixam a sala, ou seja, quando o sensor de presença muda seu estado para falso. Neste caso, o tratamento consiste em desligar o aparelho condicionador de ar e a lâmpada;
- o segundo tipo é gerado quando alguém entra na sala, ou seja, o sensor de presença muda seu estado para verdadeiro. Neste caso, o tratamento consiste em ligar a lâmpada da sala;
- o terceiro tipo é gerado quando o valor do estado temperatura é ajustado para qualquer valor acima de 25. Neste caso, o tratamento consiste em ligar o condicionador de ar;
- o quarto tipo é gerado quando o valor do estado temperatura é ajustado para qualquer valor abaixo de 23. Neste caso, o tratamento consiste em desligar o condicionador de ar.

6.2.2 Caso 2: Robô

Neste caso implementamos um robô fictício que possui um sensor de distância, instalado em sua parte frontal, um relógio interno e um conjunto de atuadores responsáveis por movimentá-lo. Estes atuadores permitem com que o robô ande para frente, quando sua

velocidade está definida como positiva, e para trás, quando sua velocidade é definida como negativa, e permitem que o robô gire em torno de seu eixo a quantidade de graus desejada em uma direção a escolha.

Descrição do dispositivo

Um único *Trait*, denominado *_robot*, define o dispositivo. O *Trait* é apresentado na Listagesm 6.2 e descreve comandos para ajustar a velocidade do robô, controlando se ele anda para frente, para trás ou pára; e para fazer o robô girar em torno de seu eixo com uma determinada angulatura.

O *Trait* também define como estados: uma variável retentora da velocidade atual, que é ajustada pelo robô à medida que a velocidade muda; o valor atual do sensor de proximidade, que é ajustado à medida que o robô se move; e uma variável para armazenar o relógio interno.

Listagesm 6.2: JSON de descrição do *Trait* *_robot*

```
1  {
2    "_robot": {
3      "commands": {
4        "setSpeed": {
5          "minimalRole": "user",
6          "parameters": {
7            "speed": {
8              "maximum": 100,
9              "minimum": -100,
10             "type": "number"
11           }
12         }
13       },
14       "turn": {
15         "minimalRole": "user",
16         "parameters": {
17           "direction": {
18             "type": "string",
19             "enum": [
20               "left",
21               "right"
22             ]
23           },
24           "degrees": {
25             "maximum": 0,
26             "minimum": 360,
27             "type": "number"
28           }
29         }
30       }
31     },
32     "state": {
33       "speed": {
34         "isRequired": true,
35         "maximum": 100,
36         "minimum": -100,
37         "type": "number"
38       },
39       "proximity": {
40         "maximum": 100,
41         "minimum": 0,
42         "type": "number"
43       },
44       "timer": {
```

```
45        "minimum": 0,
46        "type": "integer"
47      }
48    }
49  }
50 }
```

Código de usuário

A Figura 6.2 apresenta o programa desenvolvido com o Blockly para exercitar as funcionalidades do dispositivo. O programa, em linguagem de montagem, também é apresentado na Listagem C.2. No início do programa de usuário, a velocidade do robô é ajustada para 10. A partir dai, quando a proximidade do sensor atinge o valor 30, o robô gira 90° para esquerda e a cada 30 segundos ele vira 45° para a direita. É possível acionar ações em um intervalo constante de tempo realizando uma operação de módulo no valor numérico de tempo.

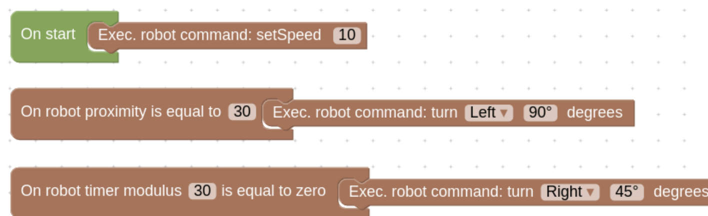

Figura 6.2: Programa de usuário executado no robô

No caso do robô, o valor do estado de proximidade é definido por uma variável aleatória de um número entre 0 e 90. O estado "*timer*" representa o tempo e é atualizado a cada segundo.

6.2.3 Caso 3: Carro

O carro inteligente possui é um dispositivo IoT que possui: um sensor para indicar se o motor principal do carro esta ligado ou desligado; um rádio capaz de emitir uma mensagem de chuva e com três estados de operação desligado, FM e DVD; e um exaustor, com a função de trocar o ar do interior do veículo. Aplicações externas são responsáveis informar ao carro dados sobre localização e a previsão do tempo

Descrição do dispositivo

Para descrever o carro são utilizados três *Traits* - *_radio*, *_engine* e *_general*. O *Trait* *_radio* é responsável por controlar o estado atual do rádio, podendo variar entre desligado, FM e DVD, armazenando a estação atual selecionada e também emitir um alerta sonoro para informar a previsão do tempo. O *Trait* *_engine* expõe de forma binária o estado atual do motor do carro. Já o *Trait* *_general* agrega os demais componentes do dispositivo. Este é composto por: um exaustor de ar; duas variáveis sobre a informações relativas ao

local (localização e previsão do tempo) providas por uma aplicação externa; uma variável que armazena a hora atual.

Listagesm 6.3: JSON de descrição dos *Traits* do carro inteligente

```
1  {
2    "_radio": {
3      "commands": {
4        "rainAlert": {
5          "minimalRole": "user"
6        },
7        "configureRadio": {
8          "minimalRole": "user",
9          "parameters": {
10           "state": {
11             "enum": [
12               "off",
13               "DVD",
14               "FM"
15             ],
16             "type": "string"
17           },
18           "station": {
19             "maximum": 108,
20             "minimum": 88,
21             "type": "number"
22           }
23         }
24       }
25     },
26     "state": {
27       "state": {
28         "isRequired": true,
29         "type": "string",
30         "enum": [
31           "off",
32           "DVD",
33           "FM"
34         ]
35       },
36       "station": {
37         "isRequired": true,
38         "maximum": 108,
39         "minimum": 88,
40         "type": "number"
41       }
42     }
43   },
44   "_engine": {
45     "state": {
46       "onOff": {
47         "isRequired": true,
48         "type": "boolean"
49       }
50     }
51   },
52   "_general": {
53     "commands": {
54       "setExhaustFan": {
55         "minimalRole": "user",
56         "parameters": {
57           "state": {
58             "type":"boolean"
59           }
60         }
61       },
62       "setIsGonnaRainToday": {
```

```
63          "minimalRole": "user",
64          "parameters": {
65            "isGonnaRainToday": {
66              "type":"boolean"
67            }
68          }
69        },
70        "setIsAtHome": {
71          "minimalRole": "user",
72          "parameters": {
73            "isAtHome": {
74              "type":"boolean"
75            }
76          }
77        }
78      },
79      "state": {
80        "hour": {
81          "isRequired": true,
82          "type": "number"
83        },
84        "isAtHome": {
85          "isRequired": true,
86          "type":"boolean"
87        },
88        "isGonnaRainToday": {
89          "isRequired": true,
90          "type":"boolean"
91        },
92        "isExhaustFanOn": {
93          "isRequired": true,
94          "type":"boolean"
95        }
96      }
97    }
98  }
```

Código de usuário

A Figura 6.3 apresenta o programa de usuário testado. A mesma implementação, porém em linguagem de máquina, é apresentada na Listagem C.1. Na logica da aplicação, às 16 horas liga o exaustor e às 18 desliga o mesmo. Quando o motor é ligado, o rádio é acionado no modo DVD e quando o motor é desligado o rádio também é desligado.

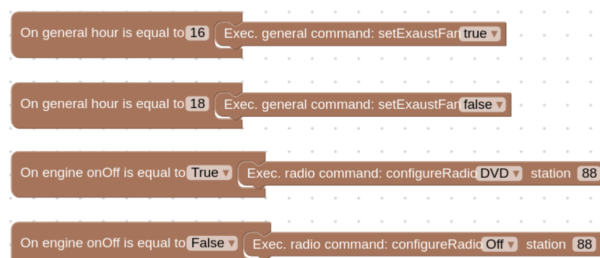

Figura 6.3: Programa de usuário executado no carro inteligente

No caso do carro, assim como no do robô, um estado que representa o tempo é incrementado de forma periódica, neste caso a cada hora. Para a simulação, o estado binário que representa o motor é alternado em um intervalo constante de tempo.

6.3 Avaliação do consumo de memória

Como um dos principais objetivo deste trabalho é a execução em dispositivos restritos, devemos considerar a quantidade de memória gasta pela implementação do *software*, de tal forma que seja viável sua aplicação neste tipo de dispositivo.

A implementação original do COISA-VP não faz uso de alocação dinâmica de memória na *heap*, o que foi seguido na implementação do PROST. O uso da pilha do programa também é baixo, pois as chamadas atingem o nível máximo de nove, conforme ilustrado no Figura 6.4, e cada função possui poucas e pequenas variáveis locais, o que garante que a quantidade máxima de dados alocados na pilha do programa seja pequena. Os retângulos ilustrados na Figura 6.4 representam métodos acrescentados em classes da libWeave, enquanto as elipses representam métodos do COISA-VP.

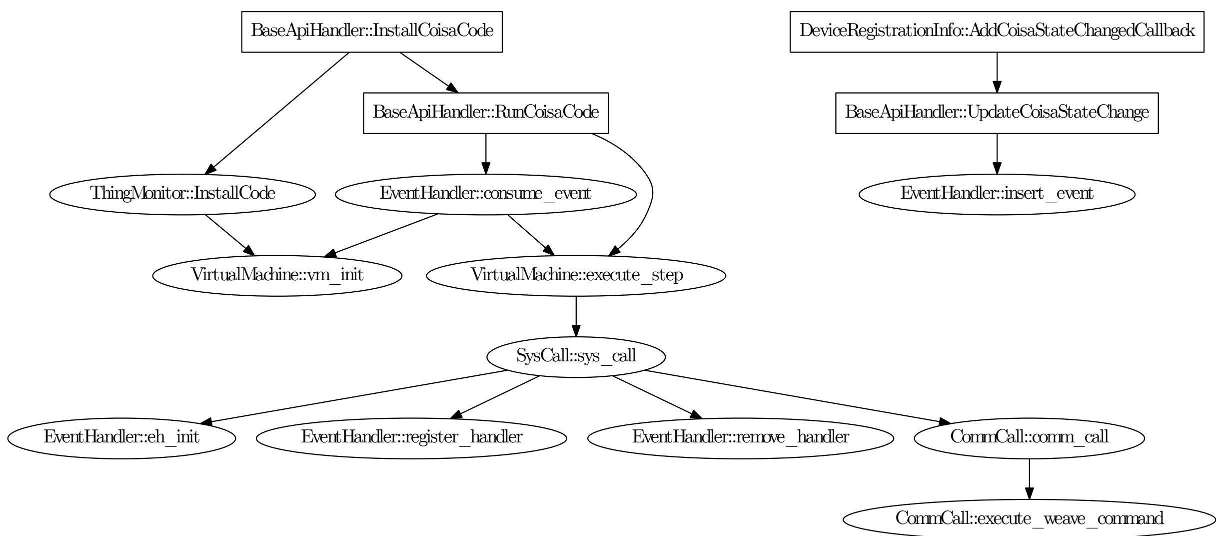

Figura 6.4: Gráfico de chamadas de funções e métodos modificados e adicionados à libWeave

Com o objetivo de investigar o consumo de memória para armazenar o código e a área estática de dados foram feitas análises nos arquivos binários da libWeave e da PROST. Sem qualquer modificação, a biblioteca compilada para arquitetura X86_64 fica com a soma das seções no tamanho de 352.80 kB, já a versão modificada, tem a soma em 367.72 kB. As modificações resultaram em um aumento de 14.92 kB, sendo que a diferença em cada seção é apresentada na Tabela 6.1.

Tabela 6.1: Tamanho de cada seção e diferenças, em *bytes*, entre a libWeave e a PROST

Seção	libWeave	PROST	Diferença
`.bss`	2936	4761	1825
`.text`	339956	352823	12867
`.rodata`	12276	12645	369
`.data.rel.ro`	5784	5984	200
`.data`	316	333	17
Total	361268	376546	15278

A seção `bss` contem variáveis que serão inicializadas com zero e ocupam a memória RAM. Os principais recursos desta seção são alocados para: a memória principal da máquina virtual, que é ajustável e ocupa nesta implementação 1400 *bytes*, o vetor de registradores, que ocupa 128 *bytes* e mais três variáveis do gerenciador de eventos, que são ajustáveis e ocupam nesta implementação 180 *bytes*. Estas variáveis são responsáveis por armazenar os eventos gerados e os tratadores de eventos registrados. A Figura 6.5 apresenta a distribuição do espaço extra necessário pelo PROST na seção *bss*

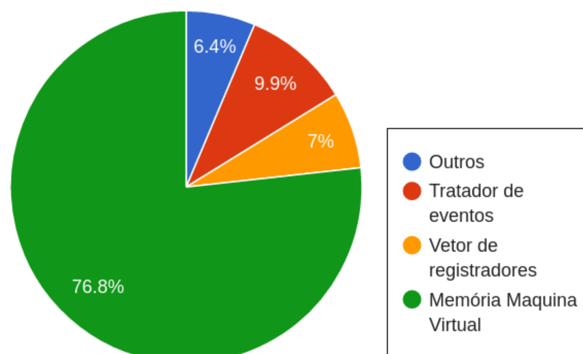

Figura 6.5: Alocação da seção BSS

A seção `data.rel.ro` é inteiramente ocupada por variáveis responsáveis pelo controle de *threads* e são necessárias em função do modelo de execução dentro da libWeave, entretanto, no caso de dispositivos restritos, estes recursos não seriam utilizados. A seção `rodata` apesar de ser pequena, contém apenas variáveis referentes a *threads* e ponteiros fracos, ou seja, algo também muito específico desta implementação utilizando a libWeave. Os elementos da seção `text`, `rodata` e `data.rel.ro` são alocados em memória *flash*, que geralmente é mais abundante do que RAM em dispositivos restritos, como por exemplo o ATmega328 que possui 32 kB de memória flash e apenas 2 kB de RAM.

6.3.1 Tamanho das aplicações

O cálculo do espaço utilizado pelo código de usuário foi feito a partir da análise do código em linguagem de montagem. Como a arquitetura utiliza instruções de 32 *bits*, o espaço necessário utilizado para armazenar instruções é obtido pela multiplicação da quantidade de instruções por este valor. A Tabela 6.2 apresenta a quantidade de instruções de cada programa de usuário testado, o espaço ocupado por essas instruções e o ocupado pela seção de dados. Todos os programas utilizam no máximo 4 *bytes* da pilha.

Tabela 6.2: Análise dos programas de usuário dos casos de uso

Caso	Número de instruções	Instruções (*bytes*)	Seção *.data* (*bytes*)	Pilha (*bytes*)	Total (*bytes*)
Sala	93	372	79	4	455
Robô	61	244	133	4	381
Carro	86	344	268	4	616

Podemos observar que o sistema precisa dedicar menos de 1 kB de espaço para o vetor que representa a memória da máquina virtual para permitir a execução de programas com três a quatro tratadores de eventos. Uma estimativa de tamanho do programa de usuário pode ser obtida em função de seus elementos, sendo aproximante: Inicialização: 36 *bytes* + *Main*: 24 *bytes* + Tratador de eventos: 30 *bytes* + M * (Novo evento registrado: 23 *bytes* + *String* do evento: 5 *bytes*) + 4 * Soma de todos os parâmetros dos eventos + K * Tratamento de cada evento: 24 *bytes* + N * Comando: 18 *bytes* + 4 * Soma de parâmetros dos comandos + *Strings* dos comandos.

Capítulo 7

Considerações finais

Foi proposto para este trabalho uma plataforma que permita a execução de aplicações de usuário, atendendo restrições computacionais do dispositivo alvo e independendo de sua arquitetura. O trabalho pode ser validado através dos casos de teste apresentados. Tal infraestrutura é capaz de fornecer a execução de aplicações de usuário como um serviço em dispositivos IoT, onde toda comunicação é feita utilizando o protocolo Weave e a execução de código de usuário utilizando a COISA-VP. Com base nas análises de recursos exigidos pela plataforma, podemos concluir que o projeto proposto é compatível com dispositivos restritos, contanto que exista um protocolo de comunicação capaz de atender a tais requisitos. É possível a implementação de um protocolo de comunicação compacto suficiente para atender os requisitos de dispositivos altamente restritos, conforme demonstrado pelo COISA-VP [53], que implementa um protocolo *ad-hoc* simples para isto. Uma aplicação que seja executada da mesma maneira em diferentes plataformas, em conjunto com mecanismos padrões de comunicação permite o desenvolvimento de aplicações que serão utilizadas da mesma maneira independente das características do módulo que a executa. O trabalho foi desenvolvido considerando plataformas computacionalmente restritas como alvo, porém, apenas a implementação da libWeave se mostrou estável suficiente para ser aplicada neste trabalho. Considerando a implementação deste mesmo protocolo em plataformas restritas, a prova de conceito realizada sugere que é possível implementar uma infraestrutura semelhante para plataformas de *hardware* computacionalmente restritas.

Também é importante destacar que a infraestrutura proposta - com a descrição dos *Trais* e a estruturação do COISA-VP - pode ser adaptada para outros arcabouços IoT, como o IoTivity ou UPnP. A documentação disponibilizada sobre o protocolo Weave se mostrou um pouco falha, muitas vezes fazendo referência a versões antigas do protocolo, bem como referenciando aplicações de exemplo inexistentes ou omitindo informações relevantes para o uso das plataformas. Em dezembro de 2016 foram anunciadas mudanças nas plataformas do Google voltadas para IoT [37] e uma das mudanças foi a descontinuidade da libWeve e da libμWeave, sendo substituídas pela biblioteca Iota [40]. As implementações são distintas, porém todas utilizam o mesmo protocolo de comunicação (Weave) e compartilham dos mesmos conceitos, logo, a portabilidade da infraestrutura para biblioteca Iota seria uma tarefa simples.

Na Seção 7.1 são apresentadas prováveis pontos de aperfeiçoamento e novas funcionalidades que podem ser desenvolvidas para a plataforma implementada.

7.1 Trabalhos futuros

Atualmente o PROST pode ser executada apenas em um dispositivo robusto, sob o sistema operacional Linux. Com a descontinuidade da libWeave pretendemos migrar a infraestrutura desenvolvida para outra implementação que também suporte o protocolo Weave, receba atualizações frequentemente e seja capaz de executar em plataformas restritas.

Como trabalho futuro pretendemos incorporar os mecanismos de descrição dos dispositivos Weave para geração automática de blocos do *blockly*.

Como a PROST foi desenvolvida focando em dispositivos restritos, não foram implementados mecanismos para o cancelamento de comandos e liberação dos recursos da máquina virtual.

Atualmente informações referentes um estado atual de um componente pode ser obtidas somente através de eventos registrados. A fim de permitir maior flexibilidade ao usuário pretendemos desenvolver mecanismos para leitura de um estado de forma direta.

Atualmente o procedimento "UpdateCoisaStateChange" da classe "BaseApiHandler" é responsável pela averiguação das mudanças de estados. Desta maneira, cada evento que pode ser gerado deve ser descrito de forma específica. Para isso, devemos incorporar mecanismos de descrição do evento no tratador de eventos COISA-VP (EH), de modo que qualquer modificação de estado seja encaminhado ao EH para uma avaliação e possível inserção do evento na fila.

Referências Bibliográficas

[1] Blockly COISA. github.com/cmillani/blockly. Acessado: 05-12-2016. 46

[2] HaikuVM - a java vm for arduino and other micros using the lejos runtime. haikuvm.sourceforge.net/. Acessado: 21-11-2016. 17

[3] SimpleRTJ a small footprint java vm for embedded and consumer devices. www.rtjcom.com/downfile.php?f=techpdf. Acessado: 21-11-2016. 18, 22

[4] Apple ii emulator. courses.cit.cornell.edu/ee476/FinalProjects/s2007/bcr22/final2007. Acessado: 31-10-2016. 19

[5] Pymite. wiki.python.org/moin/PyMite, 2014. Acessado: 3-11-2016. 20, 22

[6] AllSeen Alliance. Alljoyn framework. allseenalliance.org/framework, 2016. Acessado: 12-10-2016. 7, 23, 24, 25

[7] AllSeen Alliance. Alljoyn thin core library. github.com/allseenalliance/core-ajtcl, 2016. Acessado: 12-10-2016. 24, 25

[8] Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash A. Uzmi. *Optimized Java Binary and Virtual Machine for Tiny Motes*, pages 15–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. 20, 22

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. 14

[10] Rafael Auler, Carlos E. Millani, Alexandre Brisighello, Alisson Linhares, and Edson Borin. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform. *Concurrency and Computation: Practice and Experience*, 2016. 15, 21, 22, 43, 44

[11] Richard Barry. Soletta project. www.freertos.org/. Acessado: 10-12-2016. 25

[12] Brian Benchoff. C64 emulator for the arduino due. hackaday.com/2014/07/06/c64-emulator-for-the-arduino-due/, 2014. Acessado: 31-10-2016. 19

[13] Niels Brouwers, Peter Corke, and Koen Langendoen. Darjeeling, a java compatible virtual machine for microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, Companion '08, pages 18–23, New York, NY, USA, 2008. ACM. 20, 22

[14] Alexandru Caracas, Thorsten Kramp, Michael Baentsch, Marcus Oestreicher, Thomas Eirich, and Ivan Romanov. Mote runner: A multi-language virtual machine for small embedded devices. In *2009 Third International Conference on Sensor Technologies and Applications*, pages 117–125, June 2009. 21, 22

[15] Stuart Cheshire, Bernard Aboba, and Erik Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005. 26

[16] Stuart Cheshire and Marc Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), February 2013. 35

[17] Stuart Cheshire and Marc Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), February 2013. 35

[18] Linux Foundation Collaborative. Iotivity. www.iotivity.org/, 2016. Acessado: 10-10-2016. 28

[19] Intel Corporation. Soletta project. solettaproject.org/. Acessado: 10-12-2016. 7, 23, 32

[20] Adam Dunkels and outros. Contiki: The open source os for the internet of things. www.contiki-os.org/, 2006. Acessado: 17-01-2016. 20

[21] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. 26

[22] UPnP Forum. UPnP device architecture 2.0. upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf. Acessado: 07-09-2016. 25

[23] Open Connectivity Foundation. Open connectivity foundation. openconnectivity.org/, 2016. Acessado: 10-10-2016. 7, 28, 29, 30

[24] Google. Android open source project. source.android.com/. Acessado: 09-09-2016. 40

[25] Google. Brillo. developers.google.com/brillo/. Acessado: 09-09-2016. 40

[26] Google. Developer tools: weave_client. developers.google.com/weave/v1/dev-guides/devtools/weave_client. Acessado: 12-09-2016. 39

[27] Google. Google weave android appication. play.google.com/store/apps/details?id=com.google.android.weave. Acessado: 14-03-2016. 40

[28] Google. libμweave. weave.googlesource.com/weave/libuweave/. Acessado: 12-09-2016. 40, 41

[29] Google. libweave. weave.googlesource.com/weave/libweave/+/master. Acessado: 12-09-2016. 7, 40

[30] Google. Weave. developers.google.com/weave/. Acessado: 07-09-2016. 15, 34

[31] Google. Weave console. weave.google.com/console/#/devices/. Acessado: 09-09-2016. 39

[32] Google. Weave LED toggler quickstart. github.com/googlesamples/android-WeaveLedToggler. Acessado: 09-09-2016. 40

[33] Google. Weave playground android app. weave.googlesource.com/weave/ weave_playground_android. Acessado: 09-09-2016. 40

[34] Google. Privet local discovery. developers.google.com/cloud-print/docs/prive, 2014. Acessado: 07-09-2016. 35

[35] Google. Device development. developers.google.com/weave/v1/dev-guides/device-development/device-development-reference, 2015. Acessado: 09-09-2016. 40

[36] Google. Weave reference. developers.google.com/weave/v1/reference/, 2015/2016. Acessado: 07-09-2016. 34

[37] Google. Announcing updates to google's internet of things platform: Android things and weave. android-developers.googleblog.com/2016/12/announcing-googles-new-internet-of-things-platform-with-weave-and-android-things.html, 2016. Acessado: 05-01-2017. 67

[38] Google. Schema library. developers.google.com/weave/v1/dev-guides/device-behavior/schema-library, 2016. Acessado: 16-09-2016. 38

[39] Google. Weave compatibility definition document. developers.google.com/brillo/reference/program/compatibility-definition-document, 2016. Acessado: 13-09-2016. 40, 41

[40] Google. What is weave? developers.google.com/weave/guides/overview/what-is-weave, 2016. Acessado: 05-01-2017. 67

[41] Tom Gowing and Brian Pescatore. Nes emulation. courses.cit.cornell.edu/ee476/FinalProjects/s2009/bhp7_teg25/bhp7_teg25/, 2009. Acessado: 31-10-2016. 19

[42] Dmitry Grinberg. uJ - a java vm for microcontrollers. dmitry.gr/index.php?r=05.Projects&proj=12.%20uJ%20-%20a%20micro%20JVM. Acessado: 21-11-2016. 20, 22

[43] Till Harbaum. The NanoVM - java for the avr. www.harbaum.org/till/nanovm/index.shtml, 2005. Acessado: 21-11-2016. 20, 22

[44] Inc. IDC Research. Smartphone os market share, 2016 q3. www.idc.com/prodserv/smartphone-os-market-share.jsp, 2016. Acessado: 09-01-2017. 15

[45] Internet Engineering Task Force (IETF). Constrained RESTful environments. tools.ietf.org/wg/core/, 2016. Acessado: 09-09-2016. 29

[46] Nayeem Islam and Roy Want. Smartphones: Past, present, and future. *IEEE Pervasive Computing*, 13(4), 2014. 14

[47] Joel Koshy and Raju Pandey. VMSTAR: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 243–254, New York, NY, USA, 2005. ACM. 17, 18

[48] Thorsten Kramp, Michael Baentsch, Thomas Eirich, Marcus Oestreicher, and Ivan Romanov. The IBM mote runner. *ERCIM News*, 2009(76), 2009. 21, 22

[49] Louis Lecailliez. Toy virtual machine: modular compilation. netspring.wordpress.com/2015/05/10/toy-virtual-machine-modular-compilation/, 2015. Acessado: 31-10-2016. 19

[50] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005. 20

[51] ARM Ltd. mbed. www.mbed.com/en/. Acessado: 10-12-2016. 25

[52] Carlos E. Millani and Edson Borin. A compact OpenISA virtual platform for IoT devices. github.com/cmillani/COISA, 2016. Acessado: 10-10-2016. 45

[53] Carlos E. Millani, Alisson Linhares, Rafael Auler, and Edson Borin. COISA: A compact OpenISA virtual platform for IoT devices. *XVI Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2015)*, 2015. 8, 15, 21, 22, 43, 44, 67

[54] Paul V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. 26

[55] Paul V. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766. 26

[56] Jan Ostman. The nano VIC-20. www.hackster.io/janost/the-nano-vic-20-e37b39, 2014. Acessado: 31-10-2016. 19

[57] Nordic Semiconductor. nRF52 preview dk. www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF52-Preview-DK, 2016. Acessado: 14-09-2016. 41

[58] Nordic Semiconductor. nRF52832 product specification v1.1. infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.1.pdf, 2016. Julho de 2016. 41

[59] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014. Updated by RFC 7959. 29

[60] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java™on the bare metal. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 150–151, New York, NY, USA, 2005. ACM. 20, 22

[61] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005. 7, 18, 19, 20

[62] Mike Szczys. Emulating a z80 computer with an avr chip. hackaday.com/2010/04/27/emulating-a-z80-computer-with-an-avr-chip/, 2010. Acessado: 31-10-2016. 19

[63] Massimo Villari, Antonio Celesti, Maria Fazio, and Antonio Puliafito. AllJoyn lambda: An architecture for the management of smart environments in IoT. In *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*, pages 9–14, Nov 2014. 7, 23, 24, 25

[64] Roy Want, Bill N. Schilit, and Scott Jenson. Enabling the internet of things. *IEEE Computer*, 48(1), 2015. 14

[65] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009. 20

[66] Yanagisawa Yutaka, Kishino Yasue, Suyama Takayuki, Terada Tsutomu, Tsukamoto Masahiko, and Naya Futoshi. A CIL virtual machine for wireless sensor network applications. *IPSJ SIG Notes*, 2014(12):1–4, jul 2014. 20, 22

Apêndice A

Anexo Weave

Listagesm A.1: Exemplo de descrição de componente


```
1  {
2    "components": {
3      "base": {
4        "traits": [
5          "base"
6        ],
7        "state": {
8          "base": {
9            "firmwareVersion": "7077.10",
10           "localDiscoveryEnabled": true,
11           "localAnonymousAccessMaxRole": "user",
12           "localPairingEnabled": true
13         }
14       }
15     },
16     "controle": {
17       "traits": [
18         "onOff",
19         "hvacSubsystemController",
20         "tempSetting"
21       ],
22       "state": {
23         "onOff": {
24           "state": "on"
25         },
26         "hvacSubsystemController": {
27           "controllerMode": "auto",
28           "subsystemState": "off",
29           "supportsModeDisabled": true,
30           "supportsModeAlwaysOn": true,
31           "supportsModeAuto": true
32         },
33         "tempSetting": {
34         "degreesCelsius": 22,
35         "minimumDegreesCelsius": 10
36         }
37       }
38     },
39     "ambiente": {
40       "traits": [
```

```
41          "onOff",
42          "brightness"
43        ],
44        "state": {
45          "onOff": {
46            "state": "on"
47          },
48          "brightness": {
49            "brightness": 20
50          }
51        }
52      },
53      "_visor": {
54        "traits": [
55          "_contrast",
56          "brightness"
57        ],
58        "state": {
59          "_contrast": {
60            "contrast": 0.8
61          },
62          "brightness": {
63            "brightness": 0.5
64          }
65        }
66      }
67    }
68  }
```

Apêndice B

Anexo UPNP

Listagesm B.1: Mensagem do tipo ssdp:alive

```
1  NOTIFY * HTTP/1.1
2  HOST: 239.255.255.250:1900
3  CACHE-CONTROL: max-age = seconds until advertisement expires
4  LOCATION: URL for UPnP description for root device
5  NT: notification type
6  NTS: ssdp:alive
7  SERVER: OS/version UPnP/2.0 product/version
8  USN: composite identifier for the advertisement
9  BOOTID.UPNP.ORG: number increased each time device sends an initial announce or
       an update
10 message
11 CONFIGID.UPNP.ORG: number used for caching description information
12 SEARCHPORT.UPNP.ORG: number identifies port on which device responds to unicast
       M-SEARCH
```

Listagesm B.2: Mensagem do tipo ssdp:byebye

```
1  NOTIFY * HTTP/1.1
2  HOST: 239.255.255.250:1900
3  NT: notification type
4  NTS: ssdp:byebye
5  USN: composite identifier for the advertisement
6  BOOTID.UPNP.ORG: number increased each time device sends an initial announce or
       an update
7  message
8  CONFIGID.UPNP.ORG: number used for caching description information
```

Listagesm B.3: Mensagem do tipo ssdp:update

```
1  NOTIFY * HTTP/1.1
2  HOST: 239.255.255.250:1900
3  LOCATION: URL for UPnP description for root device
4  NT: notification type
5  NTS: ssdp:update
6  USN: composite identifier for the advertisement
7  BOOTID.UPNP.ORG: BOOTID value that the device has used in its previous
       announcements
8  CONFIGID.UPNP.ORG: number used for caching description information
9  NEXTBOOTID.UPNP.ORG: new BOOTID value that the device will use in subsequent
       announcements
```

```
SEARCHPORT.UPNP.ORG: number identifies port on which device responds to unicast
    M-SEARCH
```

Listagesm B.4: Mensagem do tipo ssdp:discover

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: seconds to delay response
ST: search target
USER-AGENT: OS/version UPnP/2.0 product/version
CPFN.UPNP.ORG: friendly name of the control point
CPUUID.UPNP.ORG: uuid of the control point
```

Listagesm B.5: Breve descrição do dispositivo e seus serviços no formato XML

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0"
configId="configuration number">
<specVersion>
<major>2</major>
<minor>0</minor>
</specVersion>
<device>
<deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
<friendlyName>short user-friendly title</friendlyName>
<manufacturer>manufacturer name</manufacturer>
<manufacturerURL>URL to manufacturer site</manufacturerURL>
<modelDescription>long user-friendly title</modelDescription>
<modelName>model name</modelName>
<modelNumber>model number</modelNumber>
<modelURL>URL to model site</modelURL>
<serialNumber>manufacturer's serial number</serialNumber>
<UDN>uuid:UUID</UDN>
<UPC>Universal Product Code</UPC>
<iconList>
<icon>
<mimetype>image/format</mimetype>
<width>horizontal pixels</width>
<height>vertical pixels</height>
<depth>color depth</depth>
<url>URL to icon</url>
</icon>
<!-- XML to declare other icons, if any, go here -->
</iconList>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:serviceType:v</serviceType>
<serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
<SCPDURL>URL to service description</SCPDURL>
<controlURL>URL for control</controlURL>
<C>URL for eventing</eventSubURL>
</service>
<!-- Declarations for other services defined by a UPnP Forum working committee
(if any) go here -->
<!-- Declarations for other services added by UPnP vendor (if any) go here -->
</serviceList>
<deviceList>
```

```
43 <!-- Description of embedded devices defined by a UPnP Forum working committee
44 (if any) go here -->
45 <!-- Description of embedded devices added by UPnP vendor (if any) go here -->
46 </deviceList>
47 <presentationURL>URL for presentation</presentationURL>
48 </device>
49 </root>
```

Listagesm B.6: Descrição detalhada do dispositivo e seus serviços no formato XML

```
1 <?xml version="1.0"?>
2 <scpd
3 xmlns="urn:schemas-upnp-org:service-1-0"
4 xmlns:dt1="urn:domain-name:more-datatypes"
5 <!-- Declarations for other namespaces added by UPnP Forum working committee (if
     any) go
6 here -->
7 <!-- The value of the attribute shall remain as defined by the UPnP Forum
    working
8 committee.
9 -->
10 xmlns:dt2="urn:domain-name:vendor-datatypes"
11 <!-- Declarations for other namespaces added by UPnP vendor (if any) go here -->
12 <!-- Vendors shall change the URN's domain-name to a Vendor Domain Name -->
13 <!-- Vendors shall change vendor-datatypes to reference a vendor-defined
    namespace -->
14 configId="configuration number">
15 <specVersion>
16 <major>2</major>
17 <minor>0</minor>
18 </specVersion>
19 <actionList>
20 <action>
21 <name>actionName</name>
22 <argumentList>
23 <argument>
24 <name>argumentNameIn1</name>
25 <direction>in</direction>
26 <relatedStateVariable>stateVariableName</relatedStateVariable>
27 </argument>
28 <!-- Declarations for other IN arguments defined by UPnP Forum working
29 Committee (if any) go here -->
30 <argument>
31 <name>argumentNameOut1</name>
32 <direction>out</direction>
33 <retval/>
34 <relatedStateVariable>stateVariableName</relatedStateVariable>
35 </argument>
36 <argument>
37 <name>argumentNameOut2</name>
38 <direction>out</direction>
39 <relatedStateVariable>stateVariableName</relatedStateVariable>
40 </argument>
41 <!-- Declarations for other OUT arguments defined by UPnP Forum working
42 committee (if any) go here -->
43 </argumentList>
44 </action>
45 <!-- Declarations for other actions defined by UPnP Forum working committee
```

```
46 (if any)go here -->
47 <!-- Declarations for other actions added by UPnP vendor (if any) go here -->
48 </actionList>
49 <serviceStateTable>
50 <stateVariable sendEvents="yes"|"no" multicast="yes"|"no">
51 <name>variableName</name>
52 <dataType>basic data type</dataType>
53 <defaultValue>default value</defaultValue>
54 <allowedValueRange>
55 <minimum>minimum value</minimum>
56 <maximum>maximum value</maximum>
57 <step>increment value</step>
58 </allowedValueRange>
59 </stateVariable>
60 <stateVariable sendEvents="yes"|"no" multicast="yes"|"no">
61 <name>variableName</name>
62 <dataType type="dt1:variable data type">string</dataType>
63 <defaultValue>default value</defaultValue>
64 <allowedValueList>
65 <allowedValue>enumerated value</allowedValue>
66 <!-- Other allowed values defined by UPnP Forum working committee
67 (if any) go here -->
68 <!-- Other allowed values defined by vendor (if any) go here -->
69 </allowedValueList>
70 </stateVariable>
71 <stateVariable sendEvents="yes"|"no" multicast="yes"|"no">
72 <name>variableName</name>
73 <dataType type="dt2:vendor data type">string</dataType>
74 <defaultValue>default value</defaultValue>
75 </stateVariable>
76 <!-- Declarations for other state variables defined by UPnP Forum working
       committee
77 (if any) go here -->
78 <!-- Declarations for other state variables added by UPnP vendor (if any) go
       here -
79 ->
80 </serviceStateTable>
81 </scpd>
```

Apêndice C

Anexo Experimentos

Listagesm C.1: Programa de usuário para a sala inteligente em linguagem de montagem

```
li $v0, 11
syscall
li $v0, 9
li $v1, ev_handler
syscall
jal main
nop
li $v0, 10
syscall
main:
addiu $sp,$sp,-4
sw $ra,0($sp)
li $a0, 10
la $a1, on_room_presence_procedure1
la $a2, room_presence_no
li $v0, 14
syscall
li $a0, 10
la $a1, on_room_presence_procedure2
la $a2, room_presence_yes
li $v0, 14
syscall
li $a0, 8
la $a1, on_room_temperature_procedure1
la $a2, room_temperature_above
li $a3, 25
li $v0, 14
syscall
li $a0, 8
la $a1, on_room_temperature_procedure2
la $a2, room_temperature_below
li $a3, 23
li $v0, 14
syscall
lw $ra,0($sp)
addiu $sp,$sp, 4
jr $ra
nop

on_room_presence_procedure1:
```

```
41 addiu $sp,$sp,-4
42 sw $ra,0($sp)
43   la $a0, room_set_ac
44   li $a3, 0
45   li $v0, 16
46   li $v1, 3
47   syscall
48   la $a0, room_set_light
49   li $a3, 0
50   li $v0, 16
51   li $v1, 3
52   syscall
53 lw $ra,0($sp)
54 addiu $sp,$sp, 4
55 jr $ra
56 nop
57
58 on_room_presence_procedure2:
59 addiu $sp,$sp,-4
60 sw $ra,0($sp)
61   la $a0, room_set_light
62   li $a3, 1
63   li $v0, 16
64   li $v1, 3
65   syscall
66 lw $ra,0($sp)
67 addiu $sp,$sp, 4
68 jr $ra
69 nop
70
71 on_room_temperature_procedure1:
72 addiu $sp,$sp,-4
73 sw $ra,0($sp)
74   la $a0, room_set_ac
75   li $a3, 1
76   li $v0, 16
77   li $v1, 3
78   syscall
79 lw $ra,0($sp)
80 addiu $sp,$sp, 4
81 jr $ra
82 nop
83
84 on_room_temperature_procedure2:
85 addiu $sp,$sp,-4
86 sw $ra,0($sp)
87   la $a0, room_set_ac
88   li $a3, 0
89   li $v0, 16
90   li $v1, 3
91   syscall
92 lw $ra,0($sp)
93 addiu $sp,$sp, 4
94 jr $ra
95 nop
96 ev_handler:
97 li $v0, 11
98 syscall
```

```
99  li $ra, event_end
100 jr $a0
101 nop
102 event_end:
103 li $v0, 10
104 syscall
105 .data
106 room_presence_no: .asciiz "ROPN"
107 room_presence_yes: .asciiz "ROPY"
108 room_temperature_above: .asciiz "ROTA"
109 room_temperature_below: .asciiz "ROTB"
110 room_set_ac: .asciiz "component._room.setAC.onOff"
111 room_set_light: .asciiz "component._room.setLight.light"
```

Listagesm C.2: Programa de usuário para o robô em linguagem de montagem

```
1  li $v0, 11
2  syscall
3  li $v0, 9
4  li $v1, ev_handler
5  syscall
6  jal main
7  nop
8  li $v0, 10
9  syscall
10 main:
11 addiu $sp,$sp,-4
12 sw $ra,0($sp)
13   la $a0, robot_set_speed
14   li $a3, 10
15   li $v0, 16
16   li $v1, 3
17   syscall
18 li $a0, 4
19 li $a3,30
20 la $a1, on_robot_on_proximity_procedure1
21 la $a2, robot_on_proximity
22 li $v0, 14
23 syscall
24 li $a0, 5
25 la $a1, on_robot_every_seconds_procedure
26 la $a2, robot_every_seconds
27 li $a3, 30
28 li $v0, 14
29 syscall
30 lw $ra,0($sp)
31 addiu $sp,$sp, 4
32 jr $ra
33 nop
34
35 on_robot_on_proximity_procedure1:
36 addiu $sp,$sp,-4
37 sw $ra,0($sp)
38   la $a0, robot_turn_left
39   li $a3,90
40   li $v0, 16
41   li $v1, 3
42   syscall
```

```
43 lw $ra,0($sp)
44 addiu $sp,$sp, 4
45 jr $ra
46 nop
47
48 on_robot_every_seconds_procedure:
49 addiu $sp,$sp,-4
50 sw $ra,0($sp)
51   la $a0, robot_turn_right
52   li $a3,45
53   li $v0, 16
54   li $v1, 3
55   syscall
56 lw $ra,0($sp)
57 addiu $sp,$sp, 4
58 jr $ra
59 nop
60 ev_handler:
61 li $v0, 11
62 syscall
63 li $ra, event_end
64 jr $a0
65 nop
66 event_end:
67 li $v0, 10
68 syscall
69 .data
70 robot_on_proximity: .asciiz "ROPR"
71 robot_every_seconds: .asciiz "RESE"
72 robot_set_speed: .asciiz "component._robot.setSpeed.speed"
73 robot_turn_left: .asciiz "component._robot.turn.direction.left.degrees"
74 robot_turn_right: .asciiz "component._robot.turn.direction.right.degrees"
```

Listagesm C.3: Programa de usuário para o carro inteligente em linguagem de montagem

```
1 li $v0, 11
2 syscall
3 li $v0, 9
4 li $v1, ev_handler
5 syscall
6 jal main
7 nop
8 li $v0, 10
9 syscall
10 main:
11 addiu $sp,$sp,-4
12 sw $ra,0($sp)
13 li $a0, 7
14 la $a1, on_car_on_hour_procedure1
15 la $a2, car_on_hour1
16 li $a3, 4
17 li $v0, 14
18 syscall
19 li $a0, 7
20 la $a1, on_car_on_hour_procedure0
21 la $a2, car_on_hour0
22 li $a3, 6
23 li $v0, 14
```

```
24 syscall
25 li $a0, 6
26 la $a1, on_car_engine_procedure1
27 la $a2, car_engine1
28 li $v0, 14
29 syscall
30 li $a0, 6
31 la $a1, on_car_engine_procedure0
32 la $a2, car_engine0
33 li $v0, 14
34 syscall
35 lw $ra,0($sp)
36 addiu $sp,$sp, 4
37 jr $ra
38 nop
39
40 on_car_on_hour_procedure1:
41 addiu $sp,$sp,-4
42 sw $ra,0($sp)
43   la $a0, car_set_exaust_fan_on
44   li $v0, 16
45   li $v1, 3
46   syscall
47 lw $ra,0($sp)
48 addiu $sp,$sp, 4
49 jr $ra
50 nop
51
52 on_car_on_hour_procedure0:
53 addiu $sp,$sp,-4
54 sw $ra,0($sp)
55   la $a0, car_set_exaust_fan_off
56   li $v0, 16
57   li $v1, 3
58   syscall
59 lw $ra,0($sp)
60 addiu $sp,$sp, 4
61 jr $ra
62 nop
63
64 on_car_engine_procedure1:
65 addiu $sp,$sp,-4
66 sw $ra,0($sp)
67   la $a0, car_set_radio_state_DVD
68   li $a3, 88
69   li $v0, 16
70   li $v1, 3
71   syscall
72 lw $ra,0($sp)
73 addiu $sp,$sp, 4
74 jr $ra
75 nop
76
77 on_car_engine_procedure0:
78 addiu $sp,$sp,-4
79 sw $ra,0($sp)
80   la $a0, car_set_radio_state_off
81   li $a3, 88
```

```
82    li $v0, 16
83    li $v1, 3
84    syscall
85  lw $ra,0($sp)
86  addiu $sp,$sp, 4
87  jr $ra
88  nop
89  ev_handler:
90  li $v0, 11
91  syscall
92  li $ra, event_end
93  jr $a0
94  nop
95  event_end:
96  li $v0, 10
97  syscall
98  .data
99  car_engine0: .asciiz "CAE0"
100 car_engine1: .asciiz "CAE1"
101 car_on_hour0: .asciiz "CAH1"
102 car_on_hour1: .asciiz "CAH2"
103 car_set_exaust_fan_on: .asciiz "component._general.setExhaustFan.state.true.arg0
      "
104 car_set_exaust_fan_off: .asciiz "component._general.setExhaustFan.state.false.
      arg0"
105 car_set_radio_state_off: .asciiz "component._radio.configureRadio.state.off.
      station"
106 car_set_radio_state_DVD: .asciiz "component._radio.configureRadio.state.DVD.
      station"
107 car_set_radio_state_FM: .asciiz "component._radio.configureRadio.state.FM.
      station"
```