4

Arquitetura Proposta

Este capítulo apresenta a arquitetura empregada na construção do OpenDevice. A arquitetura proposta será apresentada em uma visão top-down (de cima para baixo), onde partiremos de uma abordagem em alto nível até um nível mais baixo, apresentando detalhes de implementação. Na primeira seção (4.1), será apresentado a visão geral da arquitetura, permitindo o entendimento de suas camadas, modelos de comunicação, requisitos e extensibilidade. Na seção 4.2, será apresentada uma visão mais detalhada dos componentes da arquitetura, detalhando os módulos que compõe a arquitetura e responsabilidades. As seções 4.3, 4.4 4.5 e 4.12 apresentam os detalhes de como é realizada a abstração dos dispositivos e nas seções 4.7 e 4.11, são apresentados os recursos que permitem que as aplicações clientes se comuniquem com os dispositivos físicos, utilizando por exemplo o protocolo MQTT. A seção 4.6, apresenta os mecanismos de extensibilidade que a plataforma oferece, em complemento ao framework de conexões que é apresentado na seção 4.6.1. Na seção 4.13 são apresentado os detalhes da arquitetura utilizada na construção do firmware, um componente importante que permite a criação de dispositivos (sensores e atuadores) para internet das coisas, utilizando componentes de baixo custo, de maneira simplificada e extensível, e com suporte a várias tecnologias de comunicação. Um protocolo simples e fácil de ser implementado é proposto na seção 4.15 , para permitir a integração entre software e hardware com baixo poder de processamento.

4.1 Visão Geral

O OpenDevice é uma plataforma aberta (*open source*) que tem como objetivo fornecer uma solução completa para a criação de projetos baseados na Internet das Coisas. Suas ferramentas abrangem todas as plataformas envolvidas no ecossistema de IoT: (1) Hardware, (2) Desktop, (3) Cloud, (4) Mobile, (5) Web, promovendo uma infraestrutura de comunicação entre todas essas camadas, bem como serviços de: (1) Armazenamento, (2) Controle, (3) Configuração, (4) Visualização. A base da arquitetura foi construída usando a linguagem de programação Java, e por ser multi-plataforma, vários componentes podem ser reutilizados em várias plataformas. Devido à limitação de recursos de processamento e memória de alguns dispositivos embarcados alvos desse projeto, como no caso dos microcontroladores AVR/Arduino, um protocolo de nível de aplicação foi elaborado, implementado e disponibilizado através de bibliotecas em C/C++. Hardwares mais robustos, como no caso do Raspberry, Beaglebone ou outro dispositivo que tenha uma implementação da JVM disponível, podem executar as implementações em Java diretamente. O OpenDevice oferece mecanismos para implementar aplicações simples ou tratadores de eventos diretamente em JavaScript, que executam no lado do servidor, algo similar ao Node.js.

Nesta seção iremos apresentar uma visão macro da arquitetura e na seção seguinte (4.2) uma visão mais detalhada. Analisando a figura 4.1, podemos observar a integração dos componentes gerais do projeto, como veremos mais adiante na seção 4.1.2, vários modelos de comunicação (layouts) podem ser utilizados, de acordo com os requisitos de cada aplicação. Projetos para a Internet das Coisas têm por característica principal lidar com uma grande quantidade de atuadores e sensores heterogêneos, a camada do Firmware é responsável por oferecer um nível mais alto de abstração dos dispositivos (atuadores e sensores), já que esses podem utilizar diversos protocolos e tratamentos de dados variados. A integração do firmware com o middleware ou aplicações, ou seja, a integração entre software e hardware, é um desafio, pois os protocolos ainda estão em fase de desenvolvimento e validação. Devido a este problema a arquitetura tanto do middleware como do firmware foi projetada para atender novos requisitos e ter uma fácil extensibilidade. Apesar de não se ter uma padrão definido, os principais padrões e tecnologias de comunicação foram avaliados e implementados dentro da arquitetura.

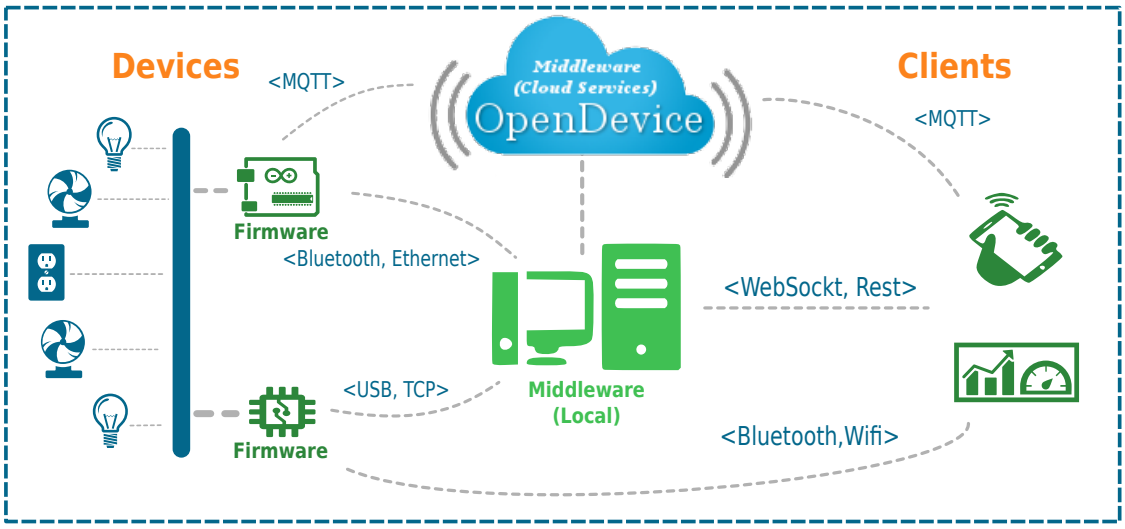

Figura 4.1 Visão Geral

4.1.1 Componentes da Visão Geral

- **Middleware (Servidor)** - O middleware tem um papel fundamental em projetos de IoT, como mencionamos na seção 2.8. No OpenDevice ele tem o papel de oferecer serviços para as aplicações clientes, fazer o gerenciamento dos dispositivos, gerenciar múltiplas conexões, fazer armazenamento e dispõe de módulos para visualização dos dados através de dashboards dinâmicos e gráficos em tempo-real ou históricos. Ele foi desenvolvido usando uma estrutura modular e extensível, permitindo a inclusão de novos componentes e plug-ins de maneira simplificada. Os desenvolvedores podem optar por utilizá-lo como um servidor à parte, de modo embarcado ou estendendo-o. Esse último é o modo aconselhado caso esteja desenvolvendo aplicações na linguagem Java, pois sua arquitetura foi pensada como um framework, de modo que os desenvolvedores de projetos de Internet das Coisas incluam apenas as regras de negócio, sem se preocupar com os detalhes de baixo nível. Projetos escritos em outras linguagens de programação podem utilizar as APIs REST, MQTT, Socket e WebSocket para se comunicar com os dispositivos através do middleware, que pode ser implantado em um servidor local (PC / Raspberry Pi) ou na nuvem.O OpenDevice suporta a criação de aplicações em JavaScript, que executam diretamente na JVM (Java Virtual Machine), o que permite acesso aos módulos Java e a criação de interfaces gráficas usando JavaFx.

- **Firmware** - O firmware tem o papel de implementar o protocolo do OpenDevice e

fazer o gerenciamento dos dispositivos físicos (atuadores e sensores) ligados a ele, criando uma abstração de alto nível, e facilitando a integração com a camada de software. Nele, os dispositivos (sensores e atuadores) são configurados e mapeados para um pino específico, de modo que apenas as informações do dispositivo (ID, Nome, Tipo) são expostas para as APIs externas, permitindo assim uma maior abstração dos detalhes do hardware, ou até mesmo a substituição do hardware por outro sem alterações na aplicação. Podemos pensar no firmware também como um *gateway,* responsável por manter a comunicação com o middleware ou aplicação e controlar os dispositivos físicos. É um componente de software projetado para rodar em dispositivos embarcados com baixo poder de processamento e memória, algo em torno de 2KB de RAM. O foco inicial do desenvolvimento foram os microcontroladores encontrados em plataformas de prototipação como o Arduino e similares. Com a expansão e popularidade da plataforma do Arduino inúmeros hardwares estão dando suporte as suas APIs[11, 12], ampliando a compatibilidade do firmware desenvolvido. Na seção 5.1, serão apresentados os hardwares testados. A arquitetura do firmware é extensível, permitindo incluir novos dispositivos, novos comandos, e suporte a novos tipos de conexões. Os detalhes do protocolo do OpenDevice serão apresentados na seção 4.15. O Firmware é um componente dispensável quando o projeto utilizar hardwares com um maior poder de processamento e que tenham suporte a GPIO, como no caso Raspberry ou BeagleBone.

- **Devices** - Representam os dispositivos físicos que podem ser atuadores (representado pela classe **Device**) ou sensores (representados pela classe **Sensor**). Estão organizados nos seguintes tipos: (1) ANALOG, (2) DIGITAL, (3) CHARACTER, que estão ligados ao modo de operação e tipo de dado suportado. Dentro do protocolo os dispositivos são identificados através de um código, denominado DeviceID. Os dispositivos do tipo ANALOG, podem receber uma faixa de valores numéricos, já os dispositivos do tipo DIGITAL trabalham com dois estados: 0 (desligado) e 1 (ligado), os dispositivos do tipo CHARACTER, estão aptos a receber uma String. O OpenDevice trabalha com um modelo orientado a objetos, permitindo que uma chamada dos métodos do *Device* na aplicação cliente, como por exemplo *Device1.on()*, acenda uma lâmpada real ou o realize fechamento de uma garra robótica. Eles podem ser independentes ou estarem conectados a um microcontrolador, executando o firmware, que faz o papel de Gateway. Os desenvolvedores podem estender essa abstração e adicionar novos comportamentos para os dispositivos e

sensores.

- **Clientes** - Representam as aplicações clientes, que podem ser aplicações Desktop, web ou *mobile* e podem ser desenvolvidas em qualquer linguagem. As aplicações clientes, podem se comunicar com o middleware, através das interfaces ofertadas (REST, MQTT, Socket, WebSocket e etc.) ou diretamente com os dispositivos físicos. Foram desenvolvidos módulos de bibliotecas clientes que permitem a integração com o middleware em linguagens Java e JavaScript e experimentos de integração usando a linguagem python. Mais detalhes sobre estas implementações serão vistas na seção 4.11.

4.1.2 Modelos de comunicação

A arquitetura planejada permite desenvolver projetos em vários Layouts e modelos de comunicação, permitindo atender desde aplicações locais, que se comunicam diretamente com os dispositivos, até aplicações distribuídas, com comunicação através da internet.

- **Comunicação direta** - É um modelo que permite a comunicação direta entre a aplicação final e os dispositivo físicos. Tem a característica de ser mais simples, pois neste modelo, não se faz necessária a presença do middleware (servidor). A aplicação neste cenário pode ser representada por: (1) um dispositivo mobile, (2) uma aplicação desktop ou (3) uma aplicação web. Representando o dispositivo físico, podemos ter um hardware que disponibilize um mecanismo de comunicação embarcado, por exemplo o ESP2866, que possui Wi-Fi, ou por exemplo um Arduino com um módulo Bluetooth acoplado. Estes dois componentes podem se comunicar usando diversas tecnologias, como: USB, Bluetooth, Ethernet, Wi-Fi. Mais detalhes sobre os meios de comunicação e como eles são implementados, serão vistos nas seções 4.13 e 4.13.3.

- **Comunicação local (Middleware)** - No modelo de comunicação local, entre em cena um novo componente do OpenDevice, o middleware. O middleware quando implantado dentro de um projeto pode ser visualizado como o servidor, e pode ser configurado tanto em um computador convencional, como em um mini-pc (ex.: Raspberry Pi). A vantagem da inclusão deste elemento é que ele permite que diversas aplicações se comuniquem com o mesmo dispositivo. Por exemplo, caso uma aplicação mobile esteja conectada via Bluetooth com um dispositivo,

outra aplicação será impedida de se comunicar com esse dispositivo, usando middleware, essa limitação é contornada, já que o middleware recebe os comandos das aplicações cliente e faz o redirecionamento para o dispositivo desejado. Outra vantagem é que o middleware pode gerenciar várias conexões com os dispositivos, utilizando vários meios de comunicação, liberando essa carga de gerenciamento das aplicações. Neste modelo de comunicação, os dispositivos podem operar no modo servidor, aguardando a conexão por parte do middleware, ou no modo cliente.

- **Comunicação pela Internet** - Neste modelo as aplicações podem se comunicar com os dispositivos através da internet. Para isso, os dispositivos devem possuir um hardware que suporte uma conexão usando o protocolo TCP/IP, necessitando apenas de um roteador convencional para interligação com a internet ou um modem GSM/GPRS. Neste cenário os dispositivos atuam no modo cliente e o middleware está implantado em um servidor na nuvem.

- **Comunicação pela Internet, usando um middleware local** - Neste modelo um middleware local é aplicado novamente, permitindo que as aplicações continuem funcionando caso a internet não esteja disponível ou quando é necessário integrar dispositivos que não possuem suporte o protocolo TCP/IP. Neste modelo, a mesma versão do middleware está rodando em um servidor local e em um servidor na nuvem, diferenciando apenas os módulos e infraestrutura utilizada, já que em um servidor local a memória e recursos são limitados e num servidor de nuvem é necessário uma performance e alta escalabilidade.

4.1.3 Requisitos

A arquitetura foi projetada para ser adaptável às capacidades e necessidades de casos de uso específicos. Esses podem ser categorizados como:

- **Requisitos de Comunicação** - O sistema deve ser apoiado em eventos e/ou comunicação autônoma. Os dispositivos devem ser configurados e controlados remotamente. O sistema deve suportar comunicação em tempo-real. O sistema deverá fornecer comunicação segura e confiável. O sistema deve prover recursos para extensão dos protocolos de comunicação.

- **Gerenciamento de dispositivos** - A API deve prover uma interface para o controle

dos dispositivos. Isso inclui a configuração do dispositivo bem como ativação, desativação e atualização remota.

- **Serviços de Descoberta** - O sistema deve oferecer mecanismos para descoberta e vinculação de novos dispositivos.
- **Capacidades do dispositivo** - A API deve prover informações sobre as capacidades dos dispositivos.
- **Requisitos de comunicação cliente/servidor** - A API deve prover suporte para operar os dispositivos tanto no modo cliente como no modo servidor.
- **Requisitos de monitoramento de status** - Status como nível de bateria, temperatura, estado de operação dentro da infraestrutura devem estar acessíveis.
- **Serviço de Armazenamento** - A API deve oferecer recursos para armazenamento das informações sobre os dispositivos, bem como mecanismo para obter o histórico dos dados do dispositivo.
- **Serviço de Visualização** - A API deve oferecer serviço para análise dos dados, recursos para consultas históricas, funções para agrupamento e agregação dos dados, bem como componentes visualização através de gráficos.
- **Orientado a Eventos** - O sistema deve oferecer um sistema para tratamento de eventos, notificando as partes interessadas quando alguma mudança de estado ocorrer nos dispositivos.
- **Interoperável entre várias redes (PAN, LAN e WAN)** - Precisa trabalhar através de uma variedade de redes e protocolos, tanto com redes IP e não-IP, incluindo dispositivos de baixa potência (low-power) em redes como Z-Wave, Zigbee e Bluetooth.
- **Extensível** - Deve fornecer recursos que permitam a inclusão de novos componente e funcionalidades através de extensões ou plug-ins.
- **Independência de Plataforma** - A API de serviços deve ser multi-plataforma, executando nos principais sistema operacionais encontrados no mercado.

4.1.4 Extensibilidade

Toda a arquitetura do OpenDevice foi pensada visando uma fácil extensibilidade, permitindo inclusão de novos protocolos, novos módulos de comunicação e integração com outras ferramentas. Devido à grande diversidade de áreas, requisitos e domínios variados que projetos de Internet das Coisas podem ser empregadas, e por ser uma área relativamente nova, é uma tarefa complexa desenvolver uma plataforma/Framework que atenda a todos os requisitos. Esse trabalho oferece contribuições com uma base sólida para criação de outros projetos especializados para outros domínios como automação residencial, saúde, cidades inteligentes e etc, porém sua estrutura generalista pode ser empregada para criar vários projetos sem necessidade de modificações, no capítulo 5 será realizada uma avaliação experimental, usando como base o domínio da automação residencial para validar a arquitetura, com base na sua aplicação generalista e nas suas capacidades de extensibilidade.

No contexto de extensibilidade, foram analisadas algumas estratégias de implementação de suporte a plug-ins e módulos dinâmicos, uma das soluções mais promissoras nesse campo é o OSGI (Open Services Gateway Initiative)[123], porém foi descartado por considerarmos que é uma ferramenta que adiciona uma complexidade extra no desenvolvimento das extensões e necessita de um gerenciamento complexo que é feito pelo contêiner de OSGI, consequentemente consumindo mais recursos. A solução adotada é baseada em uma ferramenta disponível pela própria linguagem Java, o SPI (Service Provider Interfaces), que tem como objetivo, de oferecer recursos de extensão de forma simples e leve, baseando-se apenas em interfaces que são definidas pelo próprio OpenDevice e um arquivo de configuração simples. Uma pequena desvantagem encontrada no mecanismo do SPI é que ele não permite o carregamento dinâmico de plug-ins em tempo de execução, apenas no carregamento da aplicação.

4.2 Arquitetura detalhada

Nesta seção, veremos os detalhes da arquitetura empregada na construção do projeto OpenDevice, analisando os módulos individualmente, os blocos funcionais e suas responsabilidades.

Uma das considerações importantes no desenvolvimento desse projeto é que o core

da arquitetura e os módulos servidores principais (MQTT, Rest e WebSocket) pudessem ser executados em hardwares de baixo poder de processamento, algo em torno de 512MB de RAM e 500Mhz de CPU. Com base nessa restrição, foram desenvolvidos módulos de servidores que executam em modo embarcado(*embedded*), já que as soluções disponíveis de servidores Java como: Tomcat , Jetty, JBoss ou GlassFish, iriam consumir muitos recursos da máquina. O mesmo critério foi aplicado na seleção do banco de dados, que é um componente opcional. Soluções embarcadas foram adotadas em relação à soluções instaladas separadamente (ex.: MySQL), facilitando o desenvolvimento e distribuição da aplicação.

A camada do middleware, das aplicações clientes e firmware são baseadas no modelo de desenvolvimento orientado eventos (*Event-Driven*), realizada através do envio e recebimento de comandos, usando comunicações em tempo-real. O modelo baseado em eventos facilita o desenvolvimento de aplicações de IoT, principalmente quando é necessária a interação com sensores, permitindo que os desenvolvedores registrem que tipo de evento ou os dispositivos que desejam monitorar, e quando o evento ocorrer, como por exemplo um sensor mudar seu valor, os interessados no evento serão notificados. Esse modelo também permite uma fácil extensão da ferramenta, pois tira a responsabilidade das extensões de conhecer fluxo de conexão e aspectos internos do funcionamento, podendo agregar funções mais elaboradas para lidar com os eventos, como por exemplo, um algoritmo de inteligência artificial (IA) que faça predição.

Na Figura 4.2 pode-se observar a arquitetura e, cada camada, assim como, cada elemento será explicado a seguir.

4.2.1 Camadas da Arquitetura

- **Camada de Aplicação (User Application APIs)** - Constituem os módulos e bibliotecas disponibilizados para utilização pelas aplicações clientes, permitindo a integração com o OpenDevice. A maioria dos módulos são projetados para que as aplicações se comuniquem com os dispositivos físicos (hardware) através do middleware, porém estão disponíveis módulos que permitem a comunicação direta entre a aplicação cliente e o hardware. Foram desenvolvidos módulos clientes para Web, Desktop e Android, os detalhes da implementação e tecnologias suportadas serão abordados na seção 4.11.

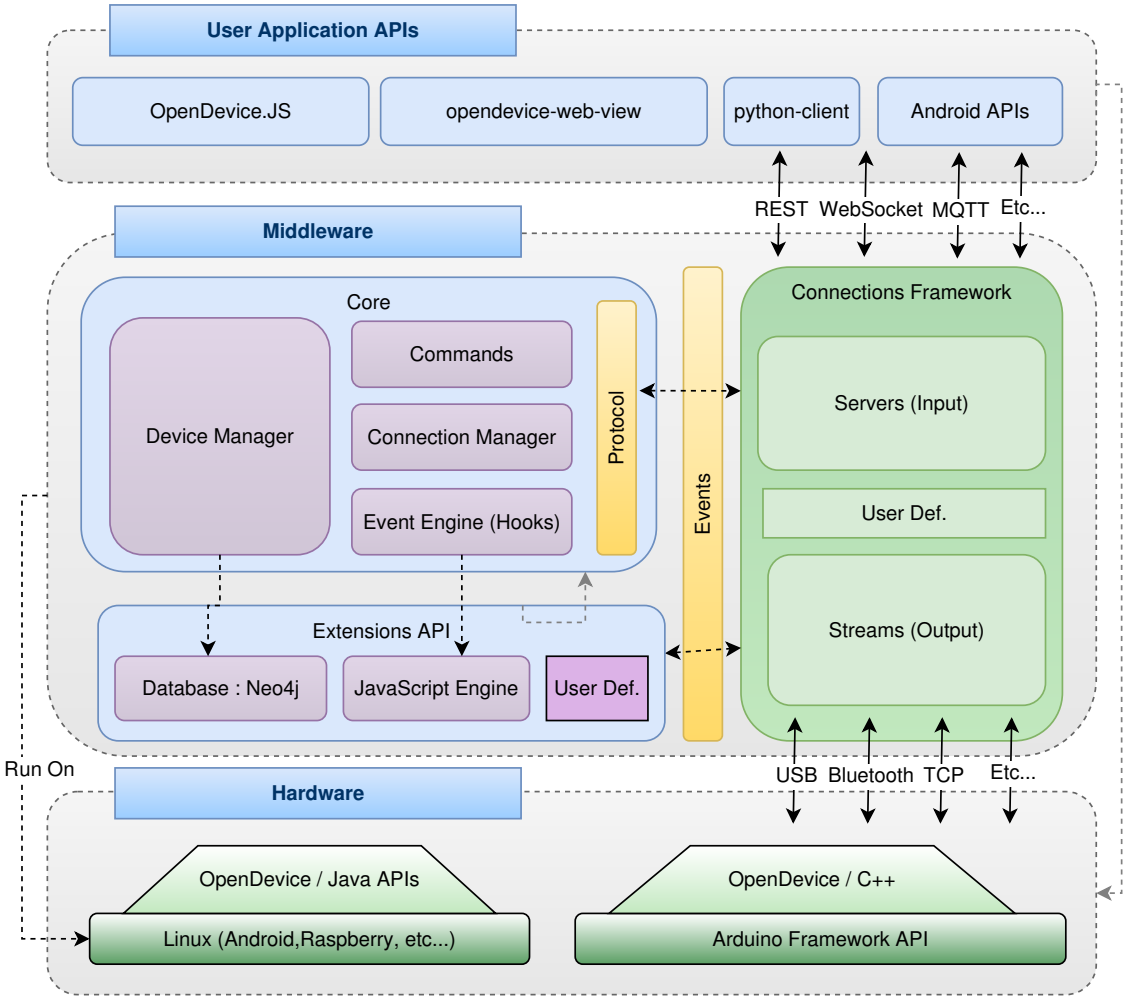

Figura 4.2 Arquitetura detalhada

- **Middleware** - O middleware é uma camada altamente modular e customizável, que oferece uma série de serviços para as aplicações, como mencionamos na seção 4.1.1 (Componentes da Visão Geral). Ele é a peça central que permite a comunicação das aplicações com os hardwares utilizando uma linguagem de alto-nível. O middleware foi desenvolvido para uma fácil extensão, devido a essa característica, ele se torna um framework para criação de projetos de Internet das Coisas. Conta com um poderoso framework de conexões, responsável pelas definições do protocolo, comunicação com os dispositivos físicos e integração com as aplicações. Mais detalhes e sub-componentes serão abordados a seguir.

- **Hardware** - Os hardwares podem ser classificados em microcontroladores e Mini PCs. Para os microcontroladores são disponibilizadas bibliotecas, que chamamos nesse trabalho de firmware, que permitem uma fácil integração com o middleware (servidor) e facilitam a criação de objetos inteligentes. Elas dão suporte a utilização de várias tecnologias de comunicação, como: Usb, Bluetooth, Ethernet e Wi-Fi. Essas bibliotecas são baseadas no framework do Arduino, o que as tornam compatíveis com uma séries de Hardwares e plataformas de prototipação, inclusive que não fazem parte do projeto do Arduino[11, 12, 13]. Quando se trata de Mini PCs, que envolvem hardwares de maior poder de processamento e memória, como por exemplo, Raspberry Pi e BeagleBone, é possível executar o middleware diretamente neles, desde que se tenha disponível uma implementação da JVM para esses dispositivos. Nos Mini PCs, o acesso aos pinos de GPIO ainda é um problema, pois cada hardware possui suas especificações. O projeto Device I/O [39], mantido pela comunidade do OpenJDK, tem a proposta de criar uma implementação para o acesso aos periféricos desses dispositivos, porém ainda está em fase de desenvolvimento. Para hardwares não suportados pelo projeto, existem três alternativas: (1) criar adaptadores/Wrapper para bibliotecas já existentes, (2) implementar chamadas JNI ou (3) usar o drivers baseados em *Sysfs* que alguns Kernels disponibilizam para acessar a GPIO como fossem simples arquivos[124].

4.2.2 Módulos

Nesta seção, abordaremos os módulos que compõe a arquitetura. Na Figura 4.3 pode-se observar os módulos, assim como, cada elemento será explicado a seguir.

1. Módulos Gerais

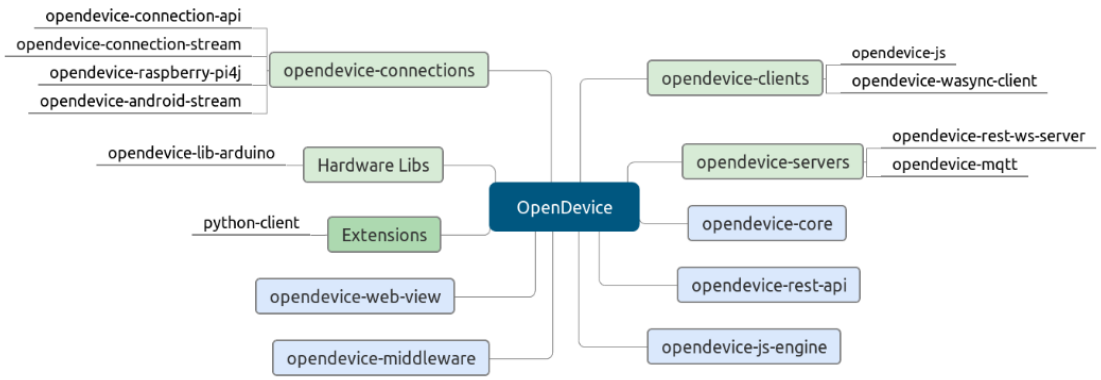

Figura 4.3 Módulos

(a) CORE: Módulo base da arquitetura, com o sistema de gerenciamento de dispositivos, sensores, conexões, eventos, armazenamento, API de comandos e implementação do protocolo. Esse módulo pode ser usado no desenvolvimento de aplicações Desktop, Web ou Mobile;

(b) REST-API: Definições das interfaces REST para controle dos dispositivos e sensores;

(c) JS-ENGINE: Implementação do suporte a execução de JavaScript no lado do servidor;

(d) WEB-VIEW: Interface HTML/5 + AngularJS + OpenDeviceJS;

(e) MIDDLEWARE: Aplicação de gestão, controle e monitoramento, que usa a maior parte dos módulos do OpenDevice, usando o banco de dados Neo4J + Hibernate OGM (JPA).

2. Módulos do framework de conexões

(a) CONNECTION-API: Especificação das interfaces de conexão cliente/servidor;

(b) CONNECTION-STREAM: Implementações de conexões USB, Bluetooth, TCP (PC/RaspPI);

(c) ANDROID-STREAM: Implementação de conexões USB, Bluetooth para Android[1];

1

i. Demais conexões (Rest, WebSocket, MQTT) podem ser utilizada no Android através dos outros módulos.

   (d) RASPBERRY-PI4J: Comunicação com a GPIO do Raspberry usando PI4J.

3. Módulos Cliente

   (a) OPENDEVICE-JS: Biblioteca JavaScript com suporte a WebSocket e REST;

   (b) OPENDEVICE-WASYNC-CLIENT: Biblioteca WebSocket para Android e PC;

   (c) PYTHON-CLIENT: Biblioteca em Python com suporte a TCP.

4. Módulos Servidores

   (a) REST-WS-SERVER: Servidor REST e WebSocket;

   (b) OPENDEVICE-MQTT: Servidor MQTT;

5. Bibliotecas para hardware

   (a) opendevice-lib-arduino: Bibliotecas em C++ baseadas na API do Arduino, que implementa o protocolo do OpenDevice e são usadas para criação do firmware. Provê suporte ao gerenciamento de dispositivos e conexões: USB, Bluetooth, Wi-Fi, Ethernet. Veja a lista de placas testadas na seção 5.1.

4.3 Gerenciamento de Dispositivos

Um dos principais requisitos de uma arquitetura de Internet das Coisas é realizar a abstração dos dispositivos, permitindo ligar com a sua grande heterogeneidade. No OpenDevice as abstrações base são implementadas através das classes Device e Sensor. Algumas implementações de clientes sofrem algumas variações nessa abstração, como por exemplo no cliente JavaScript OPENDEVICE-JS, onde existe apenas a classe *Device* e a identificação, se é um sensor ou atuador, é feita através de um atributo, já que nessa linguagem não temos suporte a orientação a objetos.

O OpenDevice permite a conexão com vários hardwares ao mesmo tempo, cada hardware (ex.: Arduino) pode gerenciar vários sensores e atuadores, cada sensor e atuador é interpretado como um "*Device*" e recebe um ID (DeviceID) único na plataforma, que pode ser codificado manualmente ou dinamicamente. Quando o módulo cliente ou o middleware estabelece uma conexão com o hardware (ex.: Arduino), ele solicita as definições dos dispositivos que foram configurados. A biblioteca (firmware) instalada

no hardware, cuida de todo o processo de negociação. A configuração de dispositivos no hardware pode ser feita de forma estática, através de uma pre-configuração, ou dinâmica, através de comandos. No hardware, os dispositivos são mapeados de forma a vincular o pino do microcontrolador com um ID (DeviceID), de modo que as aplicações externas conheçam o apenas ID, criando uma abstração do hardware final, permitindo mudanças sem afetar a aplicação. Os hardwares, atuariam como um Gateway, podendo ser identificados através de um nome ou ID, e seriam responsáveis por controlar sensores e atuadores, e integra-los às aplicações.

A listagem 4.1 apresenta um exemplo (em Java) da configuração dos dispositivos e conexão. Ao instanciar uma classe Device dentro de uma classe que estende *LocalDeviceManager,* eles passam a ser gerenciados pelo OpenDevice, e qualquer alteração dos valores dos dispositivos (ex.: *led.on()* e *led.off()*), resulta em um envio de um comando para o hardware, que verifica o ID do dispositivo e faz o mapeamento para o pino correspondente. No lado do hardware/firmware (listagem 4.2), a configuração dos dispositivos foi realizada de forma estática, onde foram adicionados dois dispositivos: (1) um atuador digital (led), conectado no pino 5 do Arduino e (2) um sensor digital (switch), conectado no pino 3. A associação do ID para cada dispositivo foi realizada de forma automática e sequencial, porém é possível especificar um ID manualmente.

Ao pressionar o sensor físico (ID=2), o firmware reconhece a alteração no seu estado e envia uma notificação para a aplicação (ou middleware), que chama o evento "onChange" do dispositivo especificado e notifica outros componentes (incluindo extensões) que foram registrados para esse evento. No evento disparado, no exemplo na listagem 4.1, ele verifica o status atual do botão (linha 11), se estiver ligado/pressionado, ele chama o método *"on()"* do led. O OpenDevice detecta essa alteração e envia o comando para o hardware (firmware) e este faz o acionamento do pino correspondente ao dispositivo. Mais detalhes sobre os fluxos de execução de comandos são apresentados na seção 4.14.

O exemplo apresentado demonstra a integração entre uma aplicação e um hardware baseado em um microcontrolador, que tem um recursos extremamente limitados. Em hardwares com maior poder de processamento, denominados Mini-PCs (ex.: Raspberry), é possível executar a aplicação e fazer o controle dos dispositivos diretamente, pois ele permite acesso aos periféricos (pinos GPIO). A listagem 4.3, apresenta um exemplo resumido de como realizar o mapeamento dos dispositivos para os pinos correspondentes do RaspberryPi.

Listagem 4.1 Configuração dos dispositivos - Java

```
1  // alguns trechos de código foram omitidos
2  public class BlinkButtonDemo extends LocalDeviceManager{
3
4     public BlinkButtonDemo() {
5
6        final Device led = new Device(1, Device.DIGITAL);
7        final Device btn = new Sensor(2, Device.DIGITAL);
8
9        connect(out.bluetooth("00:13:03:14:19:07"));
10
11       btn.onChange(device -> {
12          if(btn.isON()){
13             led.on();
14          }else{
15             led.off();
16          }
17       });
18    }
19 }
```

Listagem 4.2 Configuração dos dispositivos no Arduino - Firmware/C

```
// alguns trechos de código foram omitidos
void setup(){
   ODev.name("ModuleName");
   ODev.addDevice(5, Device::DIGITAL); // ID:1 - led
   ODev.addSensor(3, Device::DIGITAL); // ID:2 - button
   ODev.begin(Serial1, 9600);
}

void loop(){
  ODev.loop();
}
```

Listagem 4.3 Configuração dos dispositivos no Raspberry - Java

```
// alguns trechos de código foram omitidos
Device led = new Device(1, DeviceType.DIGITAL).gpio(1);
// ...
connect(new RaspberryGPIO());
```

4.4 Modelo Orientado a Eventos

O design da arquitetura segue um modelo orientado a eventos, ou *Event-Driven*, que permite desacoplar os componentes da arquitetura e aplicações. Este desacoplamento pode ser de tempo, espaço ou sincronização[125].

O mecanismo de eventos é importante na construção do framework, pois ele é considerado mais eficiente e escalável do que o modelo baseado em *Polling*, que é um mecanismo síncrono de requisição e resposta, que pode introduzir uma latência na comunicação e no tempo de resposta[126]. Esse mecanismo também permite a isolação das aplicações, de saber qual a frequência com que os dispositivos geram os dados, passando apenas a utilizar um mecanismo de "observar" os dispositivos, e reagir aos eventos quando eles acontecem.

Os eventos gerados pelas aplicações clientes, são direcionados para o middleware ou diretamente para os dispositivos, de forma automática e transparente. Os principais eventos gerados pela arquitetura são: mudança do estado do dispositivo, associação de novos dispositivos, mudança no estado das conexões, porém exitem outros e novos podem ser criados. È possível monitorar eventos gerados por dispositivos individuais, registrado ouvintes (listeners) para as instâncias especificas, ou para todos os dispositivos, registando ouvintes (listeners) no gerenciador de dispositivos (*DeviceManager*).

Um exemplo foi apresentado na seção 4.3, listagem 4.1, onde é adicionado um "ouvinte" no dispositivo e quando o valor dele mudar, o "ouvinte" é executado. No exemplo citado, o "ouvinte" ao ser executado, faz a chamada do método "led.on()", gerando um evento (comando) que é despachado para os componentes interessados. Um dos interessados é o *CommandDelivery*, que cuida do envio e monitoramento da entrega do comando, através das conexões de saída. Outro interessado é o serviço de armazenamento que, se habilitado, registra o histórico de alterações dos valores dos

dispositivos.

Algumas bibliotecas disponibilizadas para construção de aplicações clientes se baseiam no sistema Publish/Subscribe, que permite um baixo acoplamento e uma alta escalabilidade. Um dos exemplos é a biblioteca JavaScript para desenvolvimento de aplicações WEB, OPENDEVICE-JS, que utiliza o procolo de comunicação WebSocket e consegue interagir (envio e recebimento) com os dispositivos praticamente em tempo-real.

4.5 API de Comandos

As informações trocadas entre hardware, middleware e aplicações clientes são baseadas em mensagens, que são chamadas de comandos e são representadas pela classe base *Command*. Esses comandos são convertidos no protocolo do OpenDevice (mais detalhes serão vistos na seção 4.15), através dos serializadores. O framework permite a comunicação com os hardwares e controle dos dispositivos, utilizando apenas os comandos, sem utilizar as abstrações obtidas com os dispositivos (classe *Device* e *Sensor*). A listagem 4.4 mostra um exemplo da equivalência da operação usando a classe *Device* e usando os comandos. A classe DeviceCommand, permite o envio de comandos do tipo DIGITAL e ANALÓGICO. O recebimento de informações geradas pelo hardware ou de outro componente (ex.:aplicação cliente), é realizada através de comandos. Para realizar o monitoramento e recebimento desses comandos é necessário adicionar os ouvintes (listeners) nas conexões, um exemplo simplificado é apresentado na listagem 4.5. Ao enviar um comando para o hardware, ele irá responder com uma mensagem de status, que é representada pela classe *ResponseCommand*, permitindo identificar se os comandos foram recebidos corretamente ou não. O envio das mensagens é gerenciado pelo *CommandDelivery*, que permite direcionar a mensagem para a conexão certa e monitorar a entrega. Caso ela não seja feita, devido a um delay ou falha de comunicação, ele notifica a aplicação com um erro de *timeout* detalhes desse fluxo serão apresentados na seção 4.14. A figura 4.4 mostra um diagrama de classe simplificado da API de comandos. Essa API promove mais uma camada de abstração, permitindo que, caso as implementações atuais dos dispositivos (Device e Sensor) não sejam suficientes, elas possam ser substituídas.

Listagem 4.4 Comparação da API de comandos e Devices

```
// Usando a API de comandos
DeviceConnection conn = Connections.out.usb();
conn.send(DeviceCommand.ON(1)); // '1' is DeviceID

// Usando a abstração
Device led = new Device(1, Device.DIGITAL);
led.on();
```

Listagem 4.5 Monitorando recebimento de comandos

```
DeviceConnection conn = Connections.out.usb();
conn.addListener(new ConnectionListener() {
   public void onMessageReceived(Message message,
      DeviceConnection connection) {
      String type = message.getClass().getSimpleName();
      System.out.println("onMessageReceived("+type+"): "+
         message);
   }
});
```

4.6 Mecanismo de extensão

Como mencionado na seção 4.1.4, o mecanismo de extensão da arquitetura é baseado no SPI (Service Provider Interface), um recurso simples e leve, disponível no Java 6 e posteriores. Embora a arquitetura tenha sido projetada como um framework, para ser usado como base para criação de projetos mais especializados, existe uma implementação padrão, denominada *middleware*, que permite customizações através de extensões/plug-ins. Os mecanismos de extensão padrão estão voltados para: (1) conexões, permitindo adicionar novas conexões ou substituir por implementações mais eficientes, (2) tratadores de eventos, que permitem plugar estratégias de tratamento de eventos, e (3) sistema de armazenamento. As extensões são implementadas através da interface *OpenDeviceExtension*, que são inicializadas durante o carregamento da aplicação. Para que as

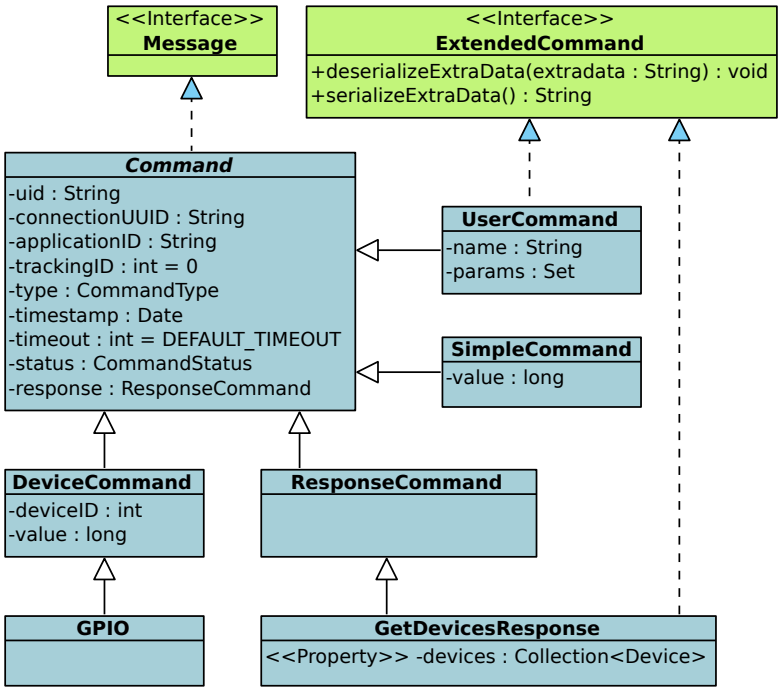

Figura 4.4 Diagrama de classe simplificado dos Comandos

extensões sejam carregadas corretamente é necessário que no módulo (.jar), seja incluído o arquivo de configuração na pasta: *resources/META-INF/services,* com o nome: *br.com.criativasoft.opendevice.engine.js.ExtensionPoint*, seguindo as especificações do SPI. As conexões seguem um mecanismo similar, porém possuem seus pontos de extensão individualizados para cada tipo de conexão, como foi visto na figura 4.5.

4.6.1 Framework de Conexões

O framework de conexões foi projetado para ser usado de forma independente do restante da arquitetura. A figura 4.5 mostra a hierarquia base das conexões suportadas pela arquitetura. Essas interfaces são os pontos de extensão, permitindo que implementações possam ser utilizadas de acordo com a plataforma ou mesmo trocadas por uma implementação mais eficiente. A figura 4.6 mostra um exemplo de implementação da conexão Bluetooth. A implementação para aplicações Desktop é fornecida pelo módulo CONNECTION-STREAM, já a implementação para aplicações Android são fornecidas pelo módulo ANDROID-STREAM, dessa maneira é possível que uma aplicação possa ser portada sem modificações no seu código base para outras plataformas. No OpenDevice a implementação correta é escolhida pela fábrica de conexões, usando

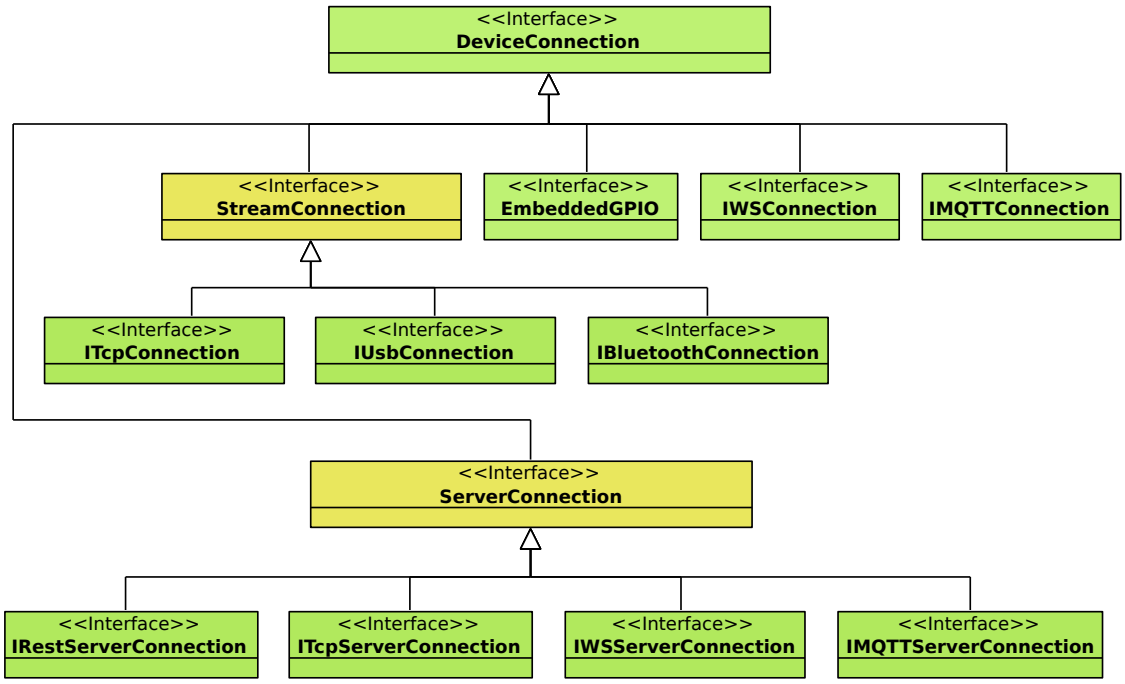

Figura 4.5 Diagrama de classe das conexões

Connections.out.bluetooth("..."). Esta fábrica está disponível no módulo core, e possui método para criação das principais tecnologias de comunicação suportadas, tanto clientes, como servidores. Os principais componentes desse framework e suas descrições são listadas a seguir.

- DeviceConnection - Interface base para todos as conexões do sistema, define o modo de operação geral das conexões e em conjunto com o *MessageSerializer* define o modelo de protocolo a ser utilizado. Possui uma implementação base, através da classe *AbstractConnection*, que facilita a criação de implementações finais.

- MessageSerializer - Como mencionado anteriormente, é o componente responsável pela serialização e desserialização das mensagens enviadas e recebidas pelas conexões. São as implementações que definem o tamanho e formato das mensagens. A implementação padrão, localizada no módulo core, é feita pela classe *CommandStreamSerializer*.

- ConnectionListener - Interface que permite monitorar de forma plugável os eventos ocorridos na conexão, como conexão, desconexão e recebimento de mensagens,

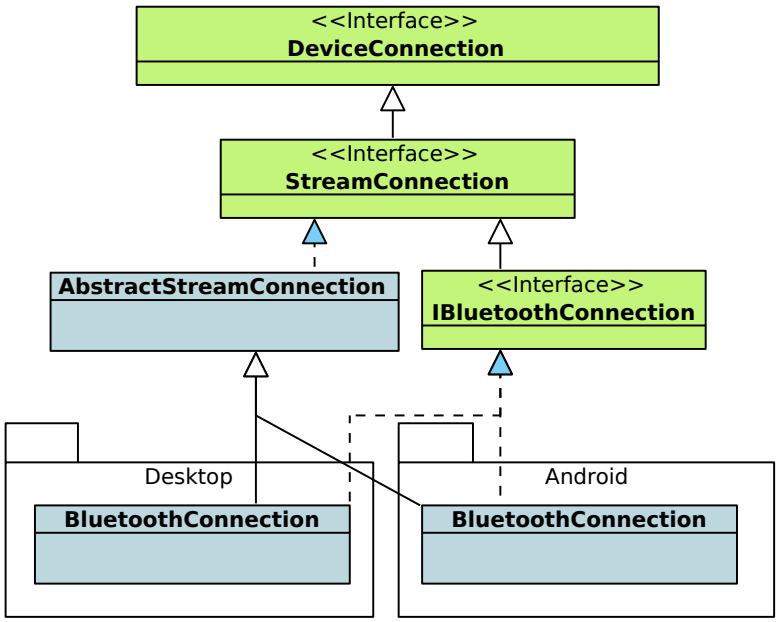

Figura 4.6 Exemplo de implementação da conexão bluetooth

permitindo às aplicações reagirem à esses eventos. Vários ouvintes de eventos (Listeners) podem ser adicionados àconexão.

- Message - Interface que encapsula os dados enviados e recebidos, exemplos de implementações são *ByteMessage*, *Request*, etc. A implementação base usada no OpenDevice é realizada pela classe *Command*, localizada no módulo core.

No OpenDevice, esse framework é utilizado em duas camadas: (1) conexões de entrada (Input), que são os servidores e têm por objetivo disponibilizar os serviços para as aplicações clientes, como por exemplo o servidor REST, e (2) conexões de saída (Output), denominados na maioria das vezes como streams, que são as conexões com os módulos físicos (hardware) e que implementam o protocolo de baixo nível, realizando a serialização e desserialização dos comandos enviadas pelas aplicações.

4.7 Módulos Servidores

Os módulos servidores, são responsáveis por fazer a interface com as aplicações clientes. Eles permitem a inclusão de novos mecanismos de conexão ou novos protocolos. Um exemplo de utilização de novo protocolo é encontrado no servidor REST, que expande o

protocolo do OpenDevice (4.15), permitindo criar interfaces mais simples e de mais alto nível para as aplicações cliente se comunicarem com os dispositivos.

O servidor WebSocket permite que aplicações (Web, Desktop, Mobile) se comuniquem em tempo real e de forma bidirecional com os dispositivos, usando middleware. O framework é projetado seguindo dois conceitos de conexões: (1) conexões de entrada (Input), que são os servidores, e (2) conexões de saída (Ouput), que são as conexões com os dispositivos físicos. O papel do middleware é realizar a tradução dos protocolos de entrada e converte-los para os protocolos de saída, específicos para cada dispositivo, confirme a figura 4.7. O módulo de visualização (com gráficos e dashboards), apresentado na seção 4.9, utilizam as conexões em WebSocket para permitir a comunicação em tempo-real com os dispositivos.

Os servidores permitem a abstração e desacoplamentos entre dispositivos e aplicações, de modo que uma aplicação pode se comunicar com vários dispositivos, ou permitir que varias aplicações se comuniquem com o mesmo dispositivo, contornando alguns problemas das conexões que suportam apenas um cliente, como no caso do Bluetooth e USB.

Os servidores foram projetados pensando na otimização de recursos do hardware. Como o objetivo é permitir que eles possam executar em hardwares, na categoria Mini-PCs, como o Raspberry e BeagleBone, as implementações dos servidores utilizados no middleware, foram escolhidas de modo que rodassem de forma embarcada e compartilhado o máximo de recursos possíveis. Devido a arquitetura estar projetada para suportar inicialmente os protocolos HTTP, Rest, WebSocket e MQTT, a implementações dos mesmos seriam um desafio, devido a variedade de requisitos, estaria fora do escopo da proposta deste trabalho. Foi então realizado um estudo que mapeou as implementações desses protocolos individualmente, e observou-se que as soluções mais maduras estavam baseadas em frameworks de rede. Os principais frameworks encontrados foram: Jetty (Eclipse)[2], Grizzly (GlassFish)[3] e Netty[4]. O Jetty é um contêiner de aplicações Java e servidor web, que tem uma estrutura modular e suporta protocolos como Http e WebSocket. Apesar ser possível executar de forma embarcada, sua estrutura foi planejada para executar aplicações Java web (.war), não como um framework genérico e nem com plataformas embarcadas em mente. O Grizzly por sua vez é projetado como um

[2]http://www.eclipse.org/jetty/
[3]https://grizzly.java.net
[4]http://netty.io

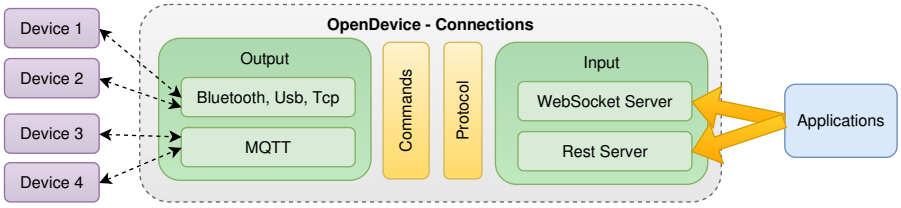

Figura 4.7 Fluxo de conexão e descoberta

framework, e utilizado como base na construção do servidor de aplicação GlassFish, suportando também HTTP, WebSocket e sendo de fácil extensão. Por fim, o Netty, é também um framework que tem uma estrutura simples, é utilizado para construção de projetos como Apache Spark, Elasticsearch, Neo4j (banco de dados), Minecraft e outros[5]. O Netty é framework utilizado como base das implementações dos servidores escolhidos para o projeto arquitetura.

A escalabilidade da arquitetura pode ser alcançada, substituindo os implementações dos servidores embarcados, por implementações mais robustas, que permitam escalonamento horizontal e balanceamento de carga. Isto pode ser alcançado, de forma transparente para a aplicação, utilizando os mecanismos de extensão (4.1.4, 4.6).

4.7.1 Servidor MQTT

A implementação de servidor MQTT escolhida, foi o "Moquette MQTT", projeto mantido pela fundação Eclipse, e que utiliza como base o framework de rede Netty.

Os servidores são em sua maioria destinados à comunicação com as aplicações clientes (conexões de entrada). O MQTT é uma exceção, pois permite que os dispositivos físicos também sejam conectados ao middleware. Para permitir um gerenciamento e integração com o middleware, e permitir a comunicação bidirecional entre aplicações e dispositivos físicos, algumas convenções foram adotadas na nomenclaturas dos tópicos.

Tanto as aplicações como os dispositivos físicos são "publish" e "subscriber". O middleware é o encarregado de monitorar as mensagens e fazer o direcionamento adequado, atuando como espécie de "subscriber" geral. Caso uma aplicação envie uma requisição para um dispositivo, é papel do middleware fazer o direcionamento para o tópico correto, e monitorar a resposta e envia-la de volta (publish) para a aplicação que

[5]http://netty.io/wiki/adopters.html

fez a requisição. A camada das aplicações não conhecem a estrutura de tópicos do borker MQTT e seu mapeamento para os dispositivos, elas apenas possuem as abstrações dos dispositivos, orientadas a objetos, e enviam os comandos para o middleware (ex.: device.on()). Isso mermite que aplicações clientes MQTT consigam se comunicar com dispositivos Bluetooth e MQTT de forma transparente.

A tabela 4.1, apresenta as nomenclaturas estabelecias para o nome dos tópicos. Quando uma aplicação realiza uma operação na abstração do dispositivo (ex.: device.on()), um comando é enviado para tópico "ProjectID/middeware/in", que é o canal que o middleware usa para o recebimento dos comandos das aplicações. Na implementação atual, por ser embarcada, o recebimento é feito diretamente sem necessidade do middleware se inscrever nos tópicos. O middleware identifica o dispositivo e determina qual conexão que o dispositivo em questão está vinculado, que pode ser uma conexão Bluetooth ou MQTT, nesse ultimo caso, o firmware envia (publish) o comando para seu respectivo tópico: "ProjectID/in/ModuleName".

O componente denominado "Firmware" é um hardware (ex.: arduino) que pode estar gerenciando um ou mais dispositivos (sensores e atuadores), onde internamente cada um recebe uma identificação (DeviceID). Cada conexão com um hardware recebe uma identificação chamada "ModuleName".

Componente	Operação	Tópico	Descrição
Firmware	Publish	ProjectID/out	Envio de Dados
Firmware	Subscribe	ProjectID/in/ModuleName	Recebimento de comandos
Middleware	Subscribe*	ProjectID/out	* Monitoramento (Listener)
Middleware	Subscribe*	ProjectID/middeware/in	* Monitoramento (Listener)
App	Publish	ProjectID/middeware/in	Envio de Comandos
App	Subscribe	ProjectID/middeware/out	Notificações Gerais
App	Subscribe	ProjectID/middeware/out/CID	Tópico de Resposta

Tabela 4.1 Nomenclatura de tópicos MQTT

4.7.2 Servidores WebSocket, Rest e Http

O WebSocket é o protocolo originalmente utilizado para permitir a comunicação em tempo-real com as aplicações Web, porém é possível a sua utilização em aplicações Desktop. A implementação desse protocolo é fornecida pelo projeto Nettosphere[6], também

[6]https://github.com/Atmosphere/nettosphere

baseado no framework Netty. O grande diferencial é que ele oferece a implementação dos três protocolos utilizando a mesma porta (ex.: 80), e consequentemente poupando muitos recursos do hardware. Isso é possível devido a estrutura de processamento do framework Netty, que permite identificar e processar cada protocolo separadamente. Teoricamente o mesmo poderia ser feito para o protocolo MQTT, mas seria um desafio atingir o nível de maturidade da implementação embarcada que adotamos.

O servidor HTTP, permite a configuração de pastas de recursos como HTML, CSS e imagens, permitindo a criação de interfaces gráficas.

O servidor REST, permite a criação de serviços que atendem ao protocolo REST. A integração com o Jesey[7], permite a implementação da especificação Java *(JAX-RS - The Java API for RESTful Web Services*), facilitando e padronizando a criação de novos serviços, estendendo as capacidades do *middleware*. Os serviços Rest criados, contam com suporte a injeção de dependências, seguindo a especificação JSR-330, que se utilizam das anotações *@Inject e @Named*, para injeção dos componentes.

4.8 Suporte a JavaScript

O suporte a JavaScript permite a criação de aplicações simples e tratadores de eventos, que rodam nativamente, e é integrado ao OpenDevice através do mecanismo de extensões. Também é possível criar aplicações Web, utilizando JavaScript, com auxílio da biblioteca OPENDEVICE-JS, que permite a abstração dos dispositivos e implementa os protocolos REST e WebSocket.

No primeiro caso, é possível criar aplicações que executa nativamente, ou seja, sem depender de um navegador, pois rodam diretamente na JVM, Este suporte é fornecido através do módulo JS-ENGINE, que usa os recursos da própria JVM implementados no projeto Nashorn [8]. Este recurso auxilia na sua utilização por desenvolvedores que não têm experiência com linguagem Java ou mesmo desenvolvedores experientes que precisam realizar uma prototipação mais rápida ou pela simplicidade da implementação de tratamento de eventos com uma linguagem de script. Devido a necessidade de recursos avançados da JVM, esse módulo (JS-ENGINE) depende da versão Java 8, que inclui várias melhorias de performance e integração com JavaScript. Os modos de desenvolvimento

[7] https://jersey.java.net
[8] http://openjdk.java.net/projects/nashorn/

serão detalhados a seguir.

4.8.1 Criação de aplicações nativas em JavaScript

As aplicações criadas em JavaScript, executam na JVM e têm interoperabilidade com as classes definidas em Java. Desse modo é possível realizar chamadas nas classes e métodos do OpenDevice. A listagem 4.6, demonstra um exemplo de criação de uma aplicação simples, que ao detectar uma mudança no botão (BUTTON), controla o estado da lâmpada (led). O módulo JS-ENGINE pode ser compilado para um executável (odevjs.exe ou odevjs.jar), destinado a execução de aplicações em JavaScript usando o comando: `> odevjs.exe myscript.js`.

Outro exemplo de aplicação, usando interface gráfica (GUI) em JavaFX, pode ser encontrado no Apêndice 6.2. Para habilitar o JavaFX é necessário informar o parâmetro "-fx", exemplo: `> odevjs.exe -fx myscript.js`. Os códigos JS podem ser executados diretamente de aplicações Java, através da classe *OpenDeviceJSEngine*, exemplo: `OpenDeviceJSEngine.run("myscript.js")`.

Listagem 4.6 Exemplo de Aplicação em JavaScript

```
var led = new Device(1, DIGITAL);
var button = new Sensor(2, DIGITAL);

button.onChange(function(){
   if(button.isON()){
      led.on();
   }else{
      led.off();
   }
});

connect(usb());
```

4.8.2 Tratadores de Eventos (EventHook)

Os tratadores de evento (EventHook), são pequenos trechos de código JavaScript que estão vinculados os dispositivos (Devices e Sensors) e são executados quando acontece alguma mudança do seu valor. Esse mecanismo é uma extensão para o EventManager, e implementada pela classe *JavaScriptEventHandler*. A implementação padrão faz o carregamento dos scripts através de arquivos, mas pode-se implementar outras formas de armazenamento/carregamento. Na implementação atual os eventos são mapeados para os dispositivos através de metadados incluídos no próprio script. Na listagem 4.7, é implementado a mesma lógica da listagem 4.6, entretanto, eles não são interpretados através do executável (odevjs.exe), eles são gerenciados pelo middleware e são executados quando os dispositivos mapeados através da anotação @DEVICES, sofrem alguma modificação. A variável "device", é injetada pelo framework e representa o dispositivo que sofreu a alteração.

Listagem 4.7 Exemplo do "EventHook" em JavaScript

```
/**
 * @name ButtonHookDemo
 * @devices 2
 * @description TestCase
 * @type JavaScript
 */
var led = findDevice(1);
if(device.isON()){
   led.on();
}else{
   led.off();
}
```

4.8.3 Criação de aplicações WEB em JavaScript

A biblioteca OPENDEVICE-JS foi desenvolvida para auxiliar no desenvolvimento de aplicações Web escritas em qualquer outra linguagem, e sua integração com os dispositivos físicos. Ela permite a comunicação em tempo real com os dispositivos graças ao suporte a WebSocket. Trabalha no modelo orientado a eventos, ou seja, quando ocorre

alguma mudança no estado no dispositivo, o evento "*onChange*" é chamado, conforme no exemplo na listagem 4.8. Ela é utilizada na construção do *Front-End Web* do middleware (4.9), que permite fazer o controle dos dispositivos, realizar a análise e visualização de dados em tempo-real ou de dados históricos.

Listagem 4.8 Exemplo de utilização da biblioteca OPENDEVICE-JS

```
<script>
   $(function(){ // JQuery ready()
      ODev.connect();
   });

   ODev.onChange(function(device){
      if(device.sensor){
         ODev.findDevice(1).setValue(device.value);
      }
   });
</script>
```

4.9 Visualização e controle dos dispositivos

O middleware conta uma uma interface Web (figura 4.8 e 4.9), desenvolvida em HTML5 e AngularJS, e é implementado pelo módulo OPENDEVICE-WEB-VIEW, que permite o monitoramento, controle dos dispositivos e visualização do dados através de gráficos e indicadores. A visualização pode ser em tempo-real ou através de consultas a dados históricos. É possível aplicar funções como: (1) média, (2) mínimo, 3 (máximo), 4 (soma), 5 (contagem) e 6 (desvio padrão), nos dados de um determinado intervalo que é configurado via a interface gráfica. Os gráficos implementados são: (1) gráfico de linha, (2) gráfico de pizza, (3) gauge e (4) indicador numérico. Os *dashboards* são altamente flexíveis, permitindo configurar o tamanho, adicionar e remover gráficos. Alguns gráficos permitem a inclusão de vários dispositivos, permitindo uma análise comparativa, como no exemplo da figura 4.9 (Luz Semana 1), foram incluídos três dispositivos em um gráfico de linha. O mesmo permite funções de zoom em determinado período de forma interativa. Os dashboards permitem também a inclusão de dispositivos e sensores digitais, permitindo o controle, ativação, desativação e visualização do status atual.

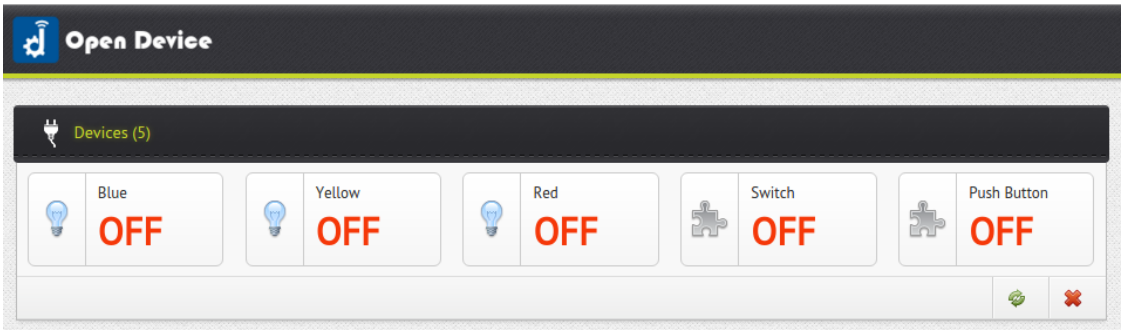

Figura 4.8 Interface de controle de dispositivos

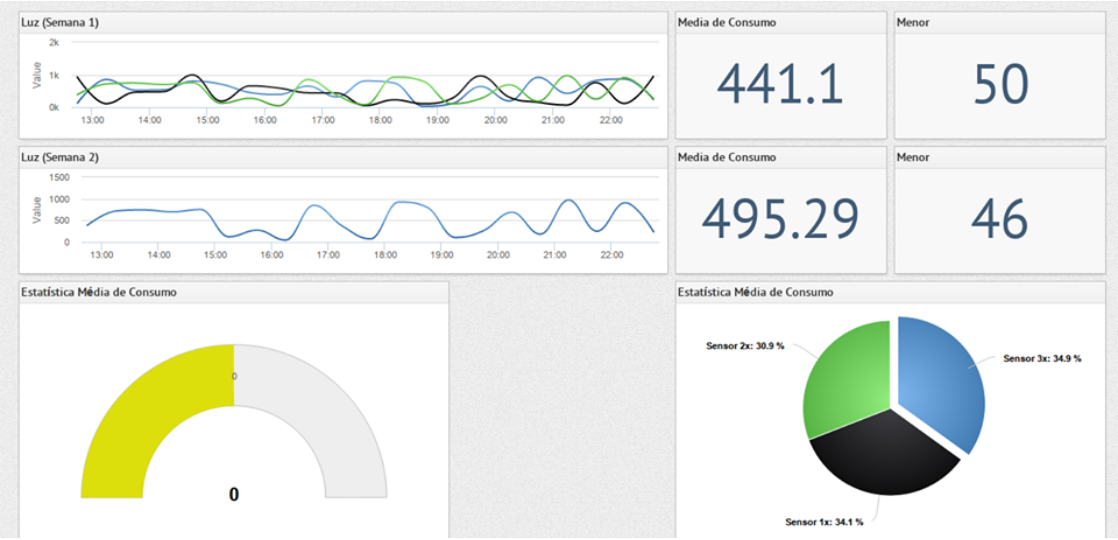

Figura 4.9 Interface de Dashboards Gráficos

4.10 Serviço de descoberta

Conectar e configurar um dispositivo (microcontrolador, sensor ou atuador), é uma tarefa relativamente simples. Fazer o mesmo para centenas ou milhares de dispositivos, não é uma tarefa fácil. Este é o cenário que os pesquisadores e desenvolvedores irão encontrar no ambiente de Internet das Coisas. Um mecanismo dinâmico, adaptável e utilizando um processo mais automatizado possível, é necessário para realizar a descoberta de dispositivos e registro de suas informações básicas. Além disso existe a necessidade de trabalhar de forma unificada com diferentes protocolos e dispositivos de rede.

O framework conta com serviços de descoberta de dispositivos, suportando as tecno-

logias Usb, Bluetooth, Ethernet e Wi-Fi. O middleware também disponibiliza um serviço de descoberta, permitindo que as aplicações clientes o localizem em uma rede local. São dois os possíveis cenários onde as aplicações e dispositivos IoT estarão executando: Local e Internet. No cenário onde os dispositivos estão conectados à Internet, eles devem ser configurados para atuarem como clientes, conectando-se no servidor em nuvem do OpenDevice em um endereço fixo. Já em um cenário local, focando-se em dispositivos Ethernet, no cenário de automação residencial, por exemplo, os dispositivos poderiam atuar em modo cliente ou como servidor, em ambos os cenários seria necessário configurar manualmente um IP fixo para cada dispositivo, uma tarefa relativamente trabalhosa dependendo da quantidade de dispositivos.

Uma solução promissora é o padrão DNS-SD (DNS Service Discovery, RFC 6763) [127], que permite a descoberta de serviços na rede e associação de um "DNS local" para o dispositivo, como por exemplo: *lampada1.local*. Devido às limitações de alguns hardwares alvos do estudo (microcontroladores), uma solução mais simples foi adotada. Trata-se do envio de mensagens UDP em broadcast na rede. Os dispositivos (hardware) monitoram a rede e ao detectar uma solicitação de descoberta (um comando do tipo *DISCOVERY_REQUEST*), enviam uma mensagem de volta contendo o seu nome, tipo, IP atual, e porta. Desse modo, todos os dispositivos podem ser configurados com IP dinâmico usando DHCP, incluindo o próprio servidor (middleware). As aplicações clientes podem localizar os dispositivos ou middleware utilizando o mesmo mecanismo.

Foi desenvolvida uma estratégia bastante simples, influenciada pelo DNS-SD e combinado com a técnica desenvolvida, para conexão com os dispositivos TCP/IP, usando um endereço com sufixo pré-definido (ex.: *lampada1*.local.opendevice). A listagem 4.9 apresenta um modelo tradicional de descoberta e o modelo usando os endereços de domínio local. No primeiro exemplo (modelo tradicional) é obtido o serviço de descoberta e iniciado a busca durante 5 segundos por dispositivos na rede, em seguida é feita a conexão. No segundo exemplo, ele permite a simplificação do processo. O endereço informado "lampada1.local.opendevice", na verdade não se trata de um domínio, serve apenas para marcar e iniciar o sistema de descoberta automático, permitindo uma fácil conexão com determinado dispositivo (hardware). A imagem 4.10 apresenta um diagrama de fluxo de como esse processo funciona.

As conexões USB e Bluetooth, possuem mecanismos de descobertas menos sofisticados, oferecidos pelas próprias implementações no S.O, que são usados pelo serviço de descoberta. No caso do USB, a classe que implementa essa conexão (*UsbConnection*),

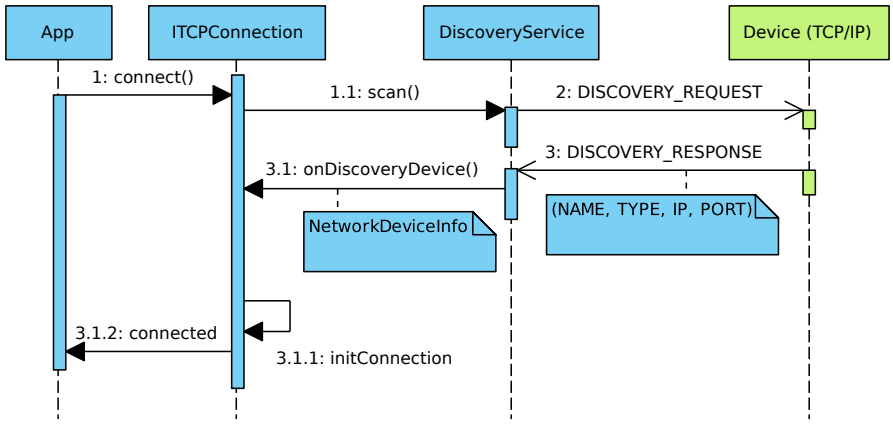

Figura 4.10 Fluxo de conexão e descoberta

possui o método "listAvailable", que lista todos os dispositivos USB-Serial conectados na maquina. É possível realizar facilmente uma conexão com o primeiro dispositivo encontrado usando: `connect(out.usb())`. A implementação para Bluetooth é similar, onde o método "listAvailable" da classe (*BluetoothConnection*), realiza a listagem dos dispositivos seriais SSP - (Serial Port Profile) disponíveis, também é possível conectar-se facilmente com o primeiro dispositivo encontrado, usando: `connect(out.bluetooth())` (observe que, antes é necessário realizar manualmente o pareamento dos dispositivos).

Listagem 4.9 Exemplos de descoberta de dispositivos

```
// Modelo tradicional
Set<NetworkDeviceInfo> devices =
   getDiscoveryService().scan(5000, null);
if(devices.size() > 0){
   NetworkDeviceInfo info = devices.iterator().next();
   if(info.getName().equals("lampada1")) {
      connect(out.tcp(info.getIp() + ":" + info.getPort()));
   }
}

// Modelo usando domínio local
connect(out.tcp("lampada1.local.opendevice"));
```

4.11 APIs para Aplicações Cliente

O conjunto de bibliotecas ou APIs para aplicações clientes, permitem a rápida construção de aplicações, sem que os pesquisadores ou desenvolvedores tenham que se preocupar com os detalhes de baixo nível do protocolo, adotando um modelo orientado a eventos (*Event-Driven*) e usando orientação a objetos na construção das aplicações, que podem ser aplicações Web, Desktop, Mobile ou uma interface simulação. O maior desafio encontrado na construção dessas aplicações é lidar com a heterogeneidade dos dispositivos (atuadores e sensores), fazer o gerenciamento das conexões e a integração com a aplicação. A arquitetura disponibiliza um middleware e um framework para construção aplicações ou middleware customizadas.

As bibliotecas disponibilizadas permitem uma comunicação bidirecional e são focadas na comunicação em tempo real, dessa maneira é possível construir gráficos para visualização das informações em tempo-real, aplicações de simulação, etc. As tecnologias empregadas que permitem essa comunicação, estão disponíveis para as plataformas Web, Desktop e Mobile, conforme a tabela 4.2.

Tipo	Web	Desktop	Mobile (Android)	Alvo de Comunicação
USB		X		Firmware
Bluetooth		X	X	Firmware
Socket (TCP)		X		Middleware/Firmware
WebSocket	X	X	X	Middleware
MQTT		X	X	Middleware

Tabela 4.2 APIs Clientes

4.11.1 Implementação

A maior parte das bibliotecas são implementados em linguagem Java com base no framework de conexões (4.6.1) e utilizam o módulo principal (core), que provê as APIs de comandos e abstração de dispositivos. Porém são disponibilizadas bibliotecas em JavaScript, apresentada na seção 4.8, e Python (experimental). As tecnologias de comunicação (ex.: USB), são implementadas com auxílio de bibliotecas externas, e suas considerações são listadas a seguir.

USB

A comunicação USB, por utilizar recursos do sistema operacional e ser dependente da arquitetura, não possui suporte nativo na JVM, apesar de ser umas das especificações iniciais da plataforma, registrada sobre a especificação Java USB API (JSR-80)[128].

Como alternativa, algumas bibliotecas de terceiros foram avaliadas. Uma das pioneiras e mais utilizadas é a biblioteca RXTX[9]. Esta biblioteca foi utilizada nas versões iniciais da plataforma, porém, pelo fato de não ter um desenvolvimento ativo, e nos testes realizados ter se mostrado instável, apresentando alguns erros (ex.: problemas de deadlocks e mal funcionamento em plataformas ARM), ela foi substituída.. A implementação de referência oficial, javax-usb [10], a julgar pelo site e documentação, estão abandonados há muito tempo.

A implementação utilizada, foi baseada na biblioteca JSSC (Java Simple Serial Connector)[11], que oferece suporte para várias plataformas[12], se mostrando uma alternativa promissora. Ela é a biblioteca utilizada na IDE do Arduino, substituindo a RXTX em versões anteriores.

Bluetooth

A comunicação Bluetooth é apoiada na especificação Java JSR-82[129], possui implementações bem estabelecidas e estáveis para o desenvolvimento para dispositivos móveis, usando JavaME. As implementações para ambiente desktop, sofrem com os mesmos problemas de plataforma do USB. Uma das poucas alternativas, é a biblioteca BlueCove[13], que tem implementações para os principais sistemas operacionais (Windows, Linux, MacOS). Para o suporte à plataformas ARM (no RaspberryPi), foi necessário a recompilação da mesma e ajustes, pois não foram encontradas versões disponibilizadas no site oficial nem de terceiros.

As implementações para aplicações mobile, estão disponíveis para o Android, e utiliza as APIs disponibilizadas pela própria plataforma. A arquitetura do OpenDevice, é

[9] https://github.com/rxtx/rxtx
[10] http://javax-usb.sourceforge.net/
[11] https://github.com/scream3r/java-simple-serial-connector
[12] Segundo o site: Windows(x86, x86-64), Linux(x86, x86-64, ARM soft & hard float), Solaris(x86, x86-64), Mac OS X(x86, x86-64, PPC, PPC64)
[13] http://bluecove.org/

projetada de maneira que a implementação utilizada é transparente para o desenvolvedor, sem precisar de modificações no código.

Socket (TCP)

Foi implementada usando as APIs nativas do Java, usando as classes da API de Socket. Devido o Java ser projetado para construção de sistemas em rede, as APIs de comunicação são suportadas em praticamente todas as plataformas.

Apesar de ser compatível com as plataformas mobile (Android), a implementação atual não é indicada, por não levar em consideração requisitos de consumo de bateria.

WebSocket

A implementação de WebSocket para plataforma Web utiliza as próprias APIs disponibilizadas pelos navegadores, através da implementação da especificação de WebSocket para o HTML5[130, 131].

A implementação para aplicações Desktop e Mobile(Android), são baseadas na biblioteca wAsync[14]. As especificações da API de WebSocket para plataforma Java, estão disponíveis através da especificação JSR 356[132], e uma implementação de referência chamada Tyrus[15], se mostra promissora, porém ainda conta com limitações para utilização no Android e por conta disso não foi utilizada.

4.12 Armazenamento

O sistema de armazenamento guarda informações sobre as conexões, dispositivos e histórico de dados. A implementação padrão é baseada em um banco de dados orientado a grafos, chamado Neo4j[16], baseado no conceito NoSQL, que permite ser executado de forma embarcada, junto com a aplicação. A base da arquitetura do Neo4j é construída usando o framework Netty, o mesmo utilizado nas implementações dos módulos de servidores (4.7), permitindo otimizando alguns recursos de espaço e consumo de memória.

[14] https://github.com/Atmosphere/wasync
[15] https://tyrus.java.net/
[16] http://www.neo4j.com

Ao incluir o middleware e o módulo de interface gráfica, as informações sobre os *dashboards* e configuração dos gráficos também são armazenadas. Na construção de aplicações e protótipos, o sistema de armazenamento pode ser dispensado, usando apenas o sistema de cache de dispositivos, porém este não permite a visualização de dados históricos.

4.13 Firmware

O firmware é um componente que permite a criação de dispositivos (coisas) para Internet das Coisas. Ele foi projetado para criação de sistemas embarcados para microcontroladores, e se baseia na API do Arduino para acesso aos periféricos do microcontrolador. Apesar de utilizar a API do Arduino, ele não está limitado apenas aos *hardwares* denominados Arduino. Várias outras plataformas vem implementando o suporte a sua API[11, 13], como por exemplo o ESP8266[10], um SoC (System-On-Chip) de 32 bits com Wi-Fi embutido. Hardwares com maior poder de processamento e que suportem a execução na JVM (Máquina Virtal Java), não farão utilização do firmware.

O firmware é flexível e pode ser utilizado como base para criação sistemas embarcados customizados, dando suporte a várias tecnologias de comunicação, como: Usb, Bluetooth, Ethernet e Wi-Fi, que pode operar tanto no modo cliente como no modo servidor (vide tabela 4.3). Dentro da plataforma do Arduino ele é disponibilizado como uma biblioteca escrita em C++, e é responsável pelo gerenciamento dos dispositivos, conexões e implementa o protocolo do OpenDevice.

4.13.1 Visão Geral

Na figura 4.11 é apresentada uma visão geral de como a arquitetura do *firmware* está estruturada. Podemos observar que ela é similar a arquitetura geral do projeto (Figura 4.2), a camada central, que compreende o firmware, é a junção das bibliotecas do Arduino, APIs do OpenDevice e o código do usuário (*User Code*), que auxiliam na criação dos sistemas embarcados. A camada superior, compreende as aplicações cliente, que implementam os protocolos de baixo nível (4.15), com auxílio das bibliotecas disponibilizadas pelo OpenDevice.

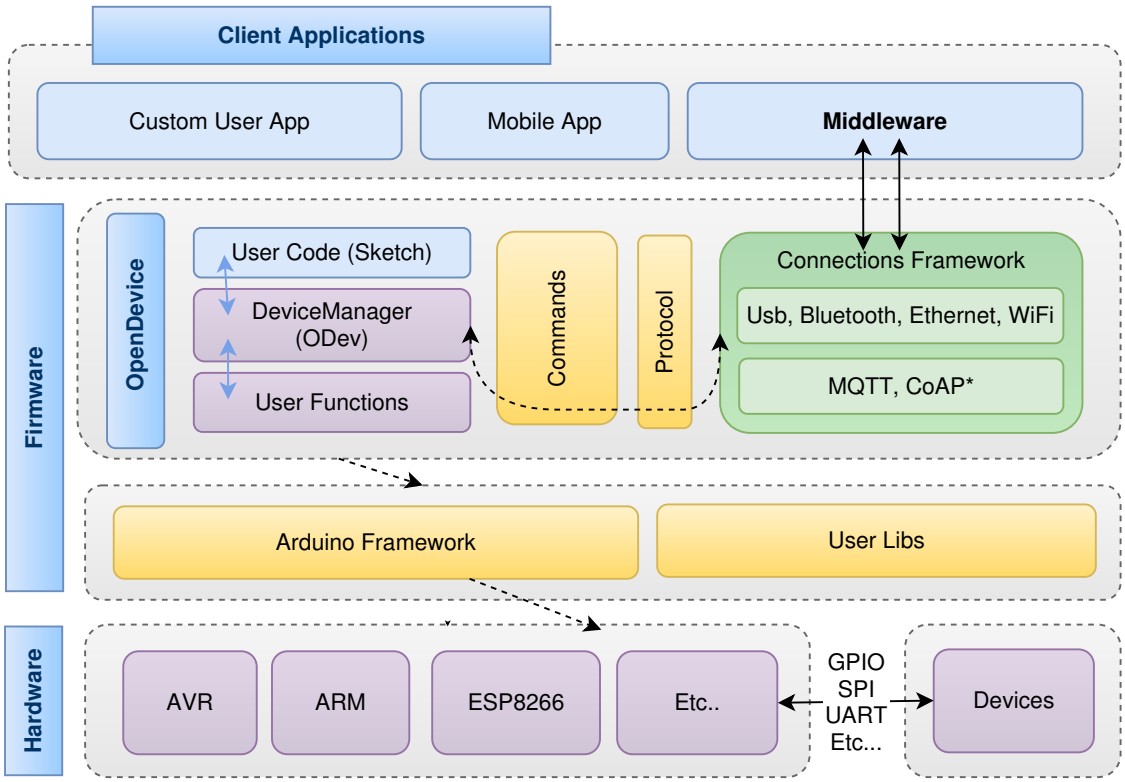

Figura 4.11 Arquitetura do Firmware

4.13.2 Meios de comunicação suportados

A tabela 4.3 apresenta as tecnologias de comunicação que são suportadas, destacadas juntos com os modelos de comunicação. Entende-se por modelo de comunicação a forma como o firmware irá operar, se é no modo cliente ou no modo servidor.

Tipo	Cliente	Servidor
Usb	Não	Sim
Bluetooth	Não	Sim
Ethernet	Sim	Sim
Wi-Fi	Sim	Sim

Tabela 4.3 Comunicação suportada pelo firmware

4.13.3 Gerenciamento de conexões

O bloco denominado “Connections Framework”, trata-se de adaptações das bibliotecas disponibilizadas pelo Arduino (“User Libs” e nativas) para implementação das conexões,

usando seus respectivos módulos módulos (shields), permitindo assim a comunicação com o middleware e aplicações. Por exemplo, o suporte a conexões Ethernet usando o módulo ENC28J60, necessita de uma biblioteca específica que realiza a implementação do protocolo TCP/IP via software, já o modulo(shield) Ethernet baseado no chip W5100 é suportado nativamente pelo framework do Arduino e OpenDevice. O Arduino disponibiliza uma API base para implementação das conexões Ethernet e Wi-Fi. Módulos (shields) que implementem essa API estariam compatíveis automaticamente com o OpenDevice.

As conexões USB e Bluetooth, são comunicações seriais, acessíveis através de portas UART, referenciadas geralmente pelas variáveis *Serial*, *Serial1*, etc. E que têm como implementação base a classe *Stream*(do Arduino). Outras tecnologias de conexão, que se baseiem nos mecanismos apresentados acima serão automaticamente suportados, ou necessitariam de pequenos ajustes.

As tecnologias implementadas estão listadas na tabela 4.3 e mais adiante na seção de hardwares testados (5.1);

4.13.4 Gerenciamento dos dispositivos

No firmware é realizado o mapeamento dos dispositivos e seus respectivos IDs para os pinos do microcontrolador. Ele interpreta os comandos enviados pelas aplicações e as transforma em ações. Um exemplo de mapeamento dos dispositivos é demonstrado da listagem 4.10, onde é feita a configuração de forma estática, adicionando atuadores e sensores. As classes que abstraem os dispositivos existem, e são baseados na classe Device, e conta com um atributo do tipo *booleano* chamado "sensor", para identificar se é um atuador ou sensor.

Listagem 4.10 Exemplo de configuração do firmware

```
#include <OpenDevice.h>
// Mapeamento dos Dispositivos
void setup(){
  ODev.name("ODev-Thing1");
   ODev.addCommand("alertMode", alertMode);
   ODev.addDevice(5, Device::DIGITAL); // ID:1
   ODev.addSensor(3, Device::DIGITAL); // ID:2
   ODev.addSensor(RFIDSensor(10,9)); // ID:1
   ODev.begin();
}

void loop(){
 ODev.loop();
}
// Comando de Usuario
void alertMode(){
 ODev.debug(ODev.readString());
 int count = ODev.readInt();
 // ....
}
```

Dispositivos suportados

A tabela 4.4, apresenta a lista de dispositivos suportados nativamente. A extensão de dispositivos permite a inclusão e suporte de novos dispositivos, sendo apresentado com mais detalhes na seção 4.13.9.

4.13.5 Comandos de Usuário

Os desenvolvedores podem criar novos comandos, estendendo o protocolo ou usando os recursos de comandos do usuário (*User Functions*, na figura 4.11). Esse recurso permite criar novos comandos e vinculá-los à funções definidas pelo próprio usuários. A listagem 4.10, apresenta um exemplo desse recurso. Os comandos são criados usando a função

Nome	Tipo	Biblioteca Extra
Genérico Digital (1 pino)	Atuador / Sensor	Não
Genérico Analógico (1 pino)	Atuador / Sensor	Não
RFID (MFRC522)	Sensor	Sim
Servo Motor	Atuador	Não
Temperatura (LM35)	Sensor	Não
Infra-Vermelho (Emissor)	Atuador	Sim
Infra-Vermelho (Receptor)	Sensor	Sim

Tabela 4.4 Dispositivos suportados nativamente

"`ODev.addCommand`", definindo o nome e a função que será executada ao receber esse comando. No lado da aplicação (Java), esse comando é executado através do método: "`sendCommand("alertMode","Your String", 5)`". Observe que também é possível passar parâmetros para a função, e no lado do firmware os parâmetros podem ser recuperados usando as funções "`ODev.read*()`".

4.13.6 Mecanismos de leitura de sensores

O mecanismo de leitura adotado, também é um sistema baseado em eventos, ou seja, as aplicações não precisam realizar consultas para obter os valores dos sensores. Quando houver alguma alteração no valor do sensor, automaticamente o firmware envia o valor atualizado para as aplicações. Internamente, o mecanismo de leitura opera de dois modos: síncrono e assíncrono, que serão apresentados a seguir.

Modo Síncrono (Polling)

O mecanismo síncrono ou "polling", é o mecanismo ativo por padrão, e por ser uma implementação mais simples, pode ser utilizado em qualquer microcontrolador, porém existem algumas limitações. Por se tratar de um mecanismo onde é preciso realizar uma leitura de todos os sensores(pinos) configurados, e comparar o valor lido com o valor atual, algum tempo será perdido lendo sensores que não alteraram seu valor, e consumindo recursos desnecessários CPU, que poderia estar desempenhando outras atividades. Dependendo do tempo da leitura dos sensores, alguma informação importante pode ser perdida. A leitura de pinos digitais e analógica é bem rápida, a leitura de um pino analógico por exemplo, demora cerca de 100 microssegundos (0.0001 s), em um

processador AVR 8-bits[17].

Modo Assíncrono (Interrupções)

As interrupções são sinais enviados para o microcontrolador com eventos que precisam de imediata atenção. A interrupção permite que o processador interrompa a tarefa atual , salve seu contexto, e execute um rotina especial de interrupção, conforme na figura 4.12. Quando um dispositivo precisa de atenção ele envia um sinal para o processador que desloca a execução para rotina de interrupção (ISR). O suporte a interrupção externas (mudança nos pinos de I/O), depende do microcontrolador, alguns suportam interrupções em todos os pinos e outros suportam interrupções apenas em alguns pinos predefinidos. Para citar exemplos, o Arduino DUE (SAM3X8E ARM) possui suporte a interrupções externas em todos os pinos, já o Arduino Uno (e similares com chip ATmega328p) suporta interrupções externas apenas nos pinos 2 e 3.

Para lidar com as limitações encontradas, o firmware utiliza uma biblioteca[18] que permite habilitar a associação de rotinas de interrupções individuais para todos os pinos do microcontrolador, fazendo o gerenciamento do estado dos pinos. Algumas limitações também são encontradas nessa alternativa. O tempo de interrupção pode sofrer atrasos na ordem de alguns micro-segundos para pinos que não suportem nativamente as interrupções externas. Dependendo da aplicação isso pode ser relevante. Uma análise realizada dos tempos de tratamento das interrupções, nessa implementação, é apresentada em [133].

Para habilitar o suporte a interrupções em um sensor é preciso habilitar o recurso nas configurações gerais e ativar os sensores que serão lidos com base nas interrupções. Com base no exemplo da listagem 4.11, quando ocorre alguma interrupção nos sensores configurados, os valores deles são lidos pela rotina *"OpenDeviceClass::onInterruptReceived()*", e marcados para sincronização, que irá ocorrer no ciclo de "loop". Um ponto importante é que os dados não podem ser enviados na rotina de interrupção, pois ela deve ocorrer o mais rápido possível, evitando problemas na leitura de outras interrupções e podem ocasionar conflitos com as interrupções de leituras das portas seriais. Detalhes de execução desse fluxo serão apresentados na seção 4.14.4.

[17] https://www.arduino.cc/en/Reference/analogRead
[18] https://github.com/GreyGnome/EnableInterrupt

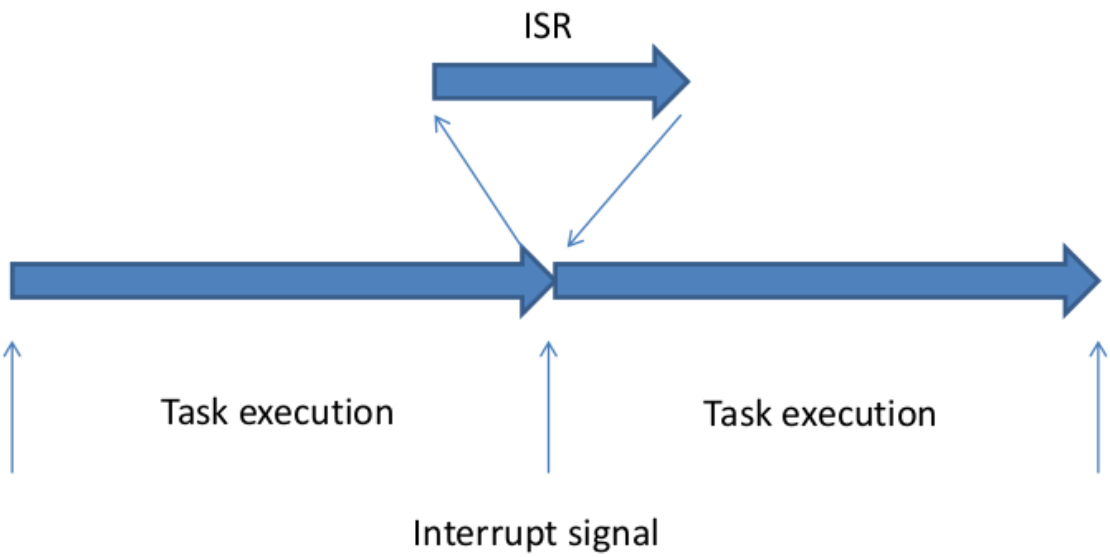

Figura 4.12 Execução de uma interrupção

Listagem 4.11 Leitura usando interrupções (Arduino/C++)

```
#include <EnableInterrupt.h>
#include <OpenDevice.h>

void setup() {
  // Modo 1
  ODev.addSensor(3, Device::DIGITAL)->enableInterrupt(CHANGE);
     // ID:1
  // Modo 2
  Device *s2 = ODev.addSensor(4, Device::DIGITAL); // ID:2
  *s2->enableInterrupt(CHANGE);

  ODev.begin();
}

void loop() {
  ODev.loop();
}
```

4.13.7 Configuração dinâmica e parametrizações

O firmware tenta detectar o hardware e as bibliotecas utilizadas e realizar as parametrizações necessárias, necessitando o mínimo de configurações possíveis. Um exemplo da técnica é apresentado na listagem 4.12, onde a implementação da Ethernet é alterada apenas escolhendo o *"include"* correspondente, sem necessidade de alterações no código. As duas implementações são totalmente diferentes, porém essas diferenças são abstraídas pelo firmware, permitindo que o desenvolvedor foque na lógica do sistema.

Algumas parametrizações e valores padrão, como velocidade, portas, tamanhos dos *buffers* de recepção de dados, podem ser ajustados no arquivo: "config.h", bem como é possível habilitar o modo de depuração (debug), para auxiliar na detecção de algum erro que esteja ocorrendo. As informações de depuração podem ser direcionadas para a conexão atual ou para uma porta serial do microcontrolador.

Listagem 4.12 Exemplo de configuração do firmware

```
//#include <UIPEthernet.h> // ENC28J60
#include <Ethernet.h>
#include <OpenDevice.h>

void setup(){
  ODev.addDevice(13, Device::DIGITAL);
  ODev.begin();
}

void loop(){
  ODev.loop();
}
```

4.13.8 Monitoramento

O sistema de monitoramento permite informar às aplicações a utilização da memória RAM e EPROM do microcontrolador e monitorar os estados das conexões, utilizando um mecanismo de Keep-Alive / Heartbeat. Em algumas situações pode ocorrer que algumas das partes da comunicação fique fora de sincronia, devido a problemas no link, falhas de

software ou hardware. Esse estado é geralmente chamado conexão semi-aberta (half-open connection). É importante que o lado da conexão que está funcionando corretamente seja notificado ou detecte a falha da outra ponta, e tente uma reconexão ou fecha a conexão semi-aberta. Mesmo existindo mecanismos similares em alguns protocolos como o TCP/IP, ele pode não ser ideal para alguns tipos de aplicações, pois em média ele é executado em um intervalo de duas horas e não permite ajustes a nível de aplicações, necessitando de ajustes no sistema operacional[134]. Por outro lado, outras conexões (ex.: USB) não suportaram este mecanismo. Para contornar esses problemas, o recurso de Keep-Alive / Heartbeat foi implementado no firmware. Quando é habilitado o suporte ao protocolo MQTT, o mecanismo de Keep-Alive do próprio protocolo são utilizados.

Este mecanismo é implementado através do envio de comando do tipo PING, em um intervalo ajustável e aguardando seu retorno através do comando do tipo PING_RESPONSE, podendo ser habilitado e desabilitado conforme a necessidade. Ao detectar uma falha ou estouro do tempo determinado (time-out), a aplicação ou middleware decide o que fazer com a conexão em estado inválido, se tenta uma reconexão ou finaliza a conexão, liberando os recursos alocados.

4.13.9 Extensibilidade

O firmware é projetado como uma biblioteca, permitindo aos desenvolvedores facilmente customizar e criar seu próprio firmware através da inclusão do código de usuário (Sketch, vide figura 4.11). O mecanismo de conexões (Connections Framework) também é flexível, permitindo plugar novas conexões facilmente, através da extensão da classe *DeviceConnection*, ou através do mecanismo de "Custom Connections", que permite a criação de novas conexões, sobrescrevendo métodos predefinidos usando apenas arquivos de cabeçalho (.h). Mais detalhes serão apresentados a seguir.

A classe Device pode ser estendida para dar suporte a dispositivos (sensores ou atuadores) mais complexos, que utilizem mais de um pino ou trabalhem com um protocolo específico (ex.: SPI, OneWire). Um exemplo de especialização desta classe é realizado pele classe *RFIDSensor*, disponível nas bibliotecas do firmware, que permite a integração de um sensor RFID de proximidade (baseado no chip MFRC522). A inclusão de novos dispositivos, é realizada estendendo a classe Device e implementando os métodos "setValue" e "hasChanged".

Outro método de inclusão de novos dispositivos, mais especificamente sensores, permite a integração com sensores mais complexos (que utilizem um protocolo específico), de uma forma simplificada e sem a necessidade de estender a classe Device, facilitando assim a integração e testes. A listagem 4.13 apresenta um exemplo deste recurso, onde a leitura do sensor é implementada por um método definido pelo usuário no programa principal (Sketch), onde o método deve retornar a leitura do sensor. O firmware é responsável por executar esse método e verificar se houve alguma mudança no valor do sensor, caso exista alguma alteração, as aplicações são notificadas.

Listagem 4.13 Suporte a novos sensores

```
void setup(){
   ODev.addSensor(readRfid); // ID:1
   // ... sensor setup ...
   ODev.begin();
}

unsigned long readRfid(){
  // sensor logic
}
```

Mecanismo "Custom Connections"

Nativamente o *firmware* suporta qualquer conexão que estenda a classe *Stream* (do Arduino). Caso seja necessário a inclusão de outro mecanismo de comunicação, onde a biblioteca projetada para o mesmo não implemente a classe Stream, um "adaptador" pode ser criado para essa conexão. Exemplos dessa implementação são encontrados no próprio firmware (arquivo: EthernetServerConnection.h) e a estrutura básica é apresentada na listagem 4.14. A vantagem é que não é necessária a criação de classes (C++), necessitando apenas de um arquivo de cabeçalho (.h). Ao realizar a inclusão deste cabeçalho no programa, o firmware automaticamente detecta que deve ser usado essa conexão e realiza as chamadas dos métodos implementados. É importante que a classe retornada pelo método "_loop()", seja uma instância e estenda a classe "Stream" para que o mecanismo funcione.

Listagem 4.14 Exemplo do mecanismo “Custom Connections”

```
#define USING_CUSTOM_CONNECTION 1
#define CUSTOM_CONNECTION_CLASS YourClassExtendStream

void custom_connection_begin(){

}

CUSTOM_CONNECTION_CLASS
   custom_connection_loop(DeviceConnection *conn){
  return // return instance of YourClassExtendStream;
}
```

4.14 Fluxo de Mensagens

Nesta seção abordamos os principais fluxos de execução, encontrados no firmware, middleware e aplicações, auxiliando a entender o processo interno de execução e quais componentes são utilizados em cada etapa.

4.14.1 Envio de Comandos

A figura 4.13, apresenta o fluxo de envio de comandos entre uma aplicação, que se comunica diretamente com os dispositivos (firmware). A primeira etapa inicia com a abstração do dispositivo e a execução do seu método “on” ou “setValue” (ex.: led.on()). Esse método gera um evento que é capturado pelo framework (fluxo 1), através da classe *DeviceManager*, que cria o comando apropriado, no caso o *DeviceCommand*, e inicializa com as informações do ID do dispositivo, tipo (ANALOG ou Digital) e valor e repassa para o CommandDelivery (fluxo 1.2), que cuida do envio para as conexões de saída de modo assíncrono, usando a *SendTask*. As mensagens são serializadas para o formato do protocolo (4.15) do OpenDevice usando o *MessageSerializer*. Cada comando enviado, recebe um ID (TrackingID), que é gerenciado pelo CommandDelivery, e é usado para fazer o mapeamento dos comandos enviados e comandos recebidos (fluxo 2 e 3). O ID do comando é gerado de forma sequencial, até um limite estabelecido pelo protocolo, depois

é reiniciado a contagem. A SendTask, é registrada para receber os eventos das conexões. Quando a resposta é recebida pela conexão, o método “onMessageReceived” (fluxo 3) é executado, e a resposta é mapeada para o comando enviado.

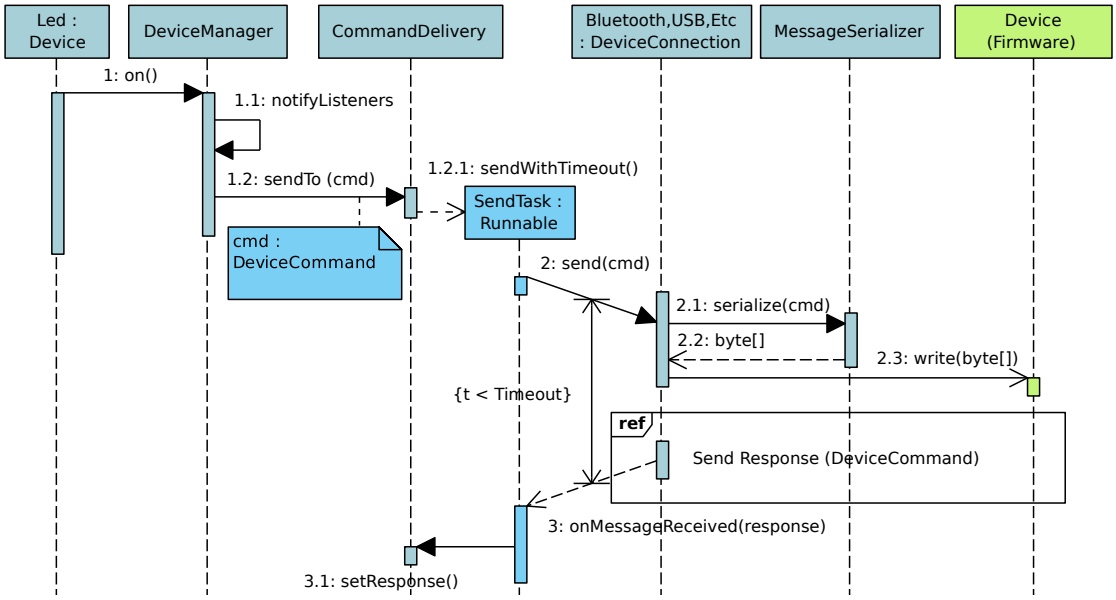

Figura 4.13 Envio de Comandos

4.14.2 Processamento dos Comandos (Dispositivo)

A figura 4.14, demonstra o fluxo de execução do firmware, para realizar a leitura dos comandos das aplicações (ou do middleware), transforma-lo em ações reais.

Os comandos recebidos são tratados por uma implementação da classe *Stream* (fluxo 1), que pode ser uma implementação nativa do Arduino, como a *Serial* ou *EthernetClient*, ou uma implementação disponibilizada por uma biblioteca de terceiros, que implementa outra tecnologia de comunicação. Geralmente esses dados são armazenados em um buffer, de software ou de hardware, e serão lidos no ciclo de “loop” do programa principal, através da chamada do “checkDataAvailable” (fluxo 2), que ao identificar o final da mensagem especificado no protocolo, faz a desserialização através do método “parseCommand” (fluxo 2.3), e repassa para a classe principal da biblioteca do OpenDevice (fluxo 2.4), que verifica que tipo de comando foi recebido e faz o tratamento adequado. Caso a mensagem recebida (fluxo 2.4), seja um *DeviceCommand* (Ex.: Analogic ou Digital), o dispositivo relacionado é localizado através do “DeviceID”, e em seguida seu valor é alterado (fluxo 3), conforme o valor recebido pelo comando. Em seguida a implementação do Device

determina como será o tratamento para o valor recebido. Por exemplo, se o dispositivo for um dispositivo do tipo DIGITAL, a implementação chama o método “digitalWrite” da API do Arduino, caso seja um Device do tipo ANALOG, a implementação chama o método “analogWrite”.

Quando um comando é recebido com sucesso e o dispositivo relacionado é encontrado, uma resposta é enviada para a aplicação (fluxo 4.1.1), informando o estados da execução. Essa resposta é encapsulada através da classe *ResponseCommand*, podendo ter vários status, conforme a tabela 4.9, na seção referente ao protocolo (4.15).

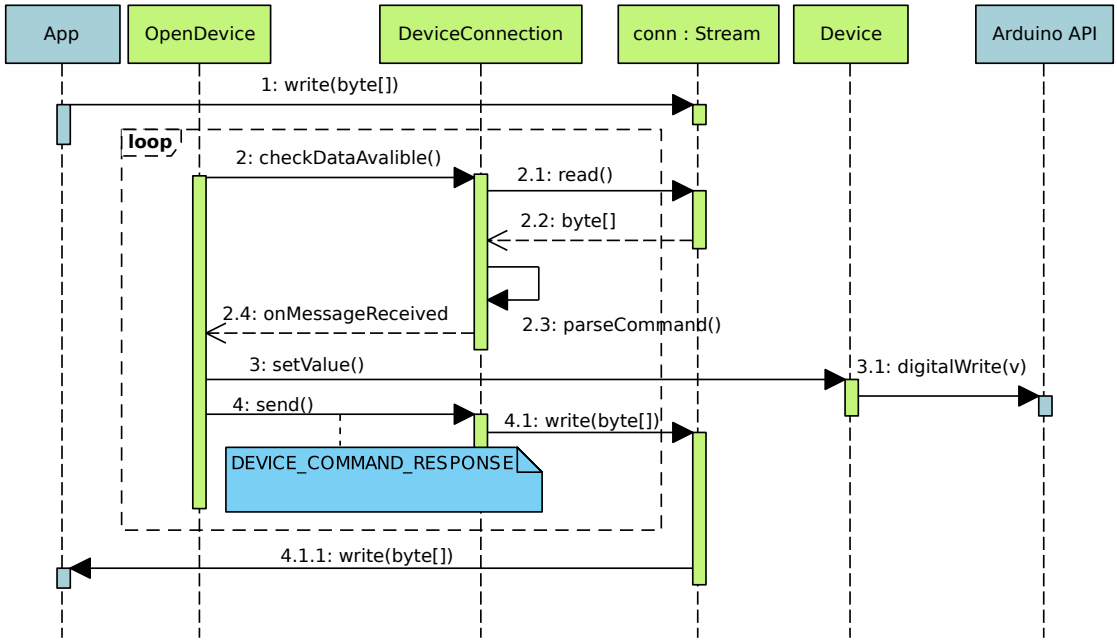

Figura 4.14 Processamento dos Comandos

4.14.3 Recebimento de Comandos

O diagrama de sequência apresentado na figura 4.15, trata-se da continuação do fluxo, quando o dispositivo (firmware) devolve a resposta com o status da execução do comando para a aplicação ou middleware. O componente que fará a leitura dos dados brutos, depende da implementação da conexão. No exemplo da figura, foi utilizando um *StreamReader*, mais especificamente *CommandStreamReader*, que é a implementação base para lidar com as conexões implementados no módulo CONNECTION-STREAM (USB e Bluetooth por exemplo). Essa implementação em específico, utiliza uma “thread” para leitura dos dados de cada conexão, e ao identificar o recebimento do pacote completo,

ele notifica a conexão (fluxo 1.3), que por sua vez notifica os componentes interessados (fluxos 2 e 3).

Quando se trata de uma resposta de um comando, por exemplo DeviceCommand, a SendTask, criada e gerenciada pelo CommandDelivery, é notificada e atualiza o status do comando que foi enviado, com base do ID do comando (TrackingID) e faz a liberação (remove a trava) do comando.

Caso não seja uma resposta de um comando, por exemplo a leitura de um sensor, o CommandDelivery não é utilizado . O responsável por tratar esse comando é o DeviceManager, como será visto na seção seguinte (4.14.4).

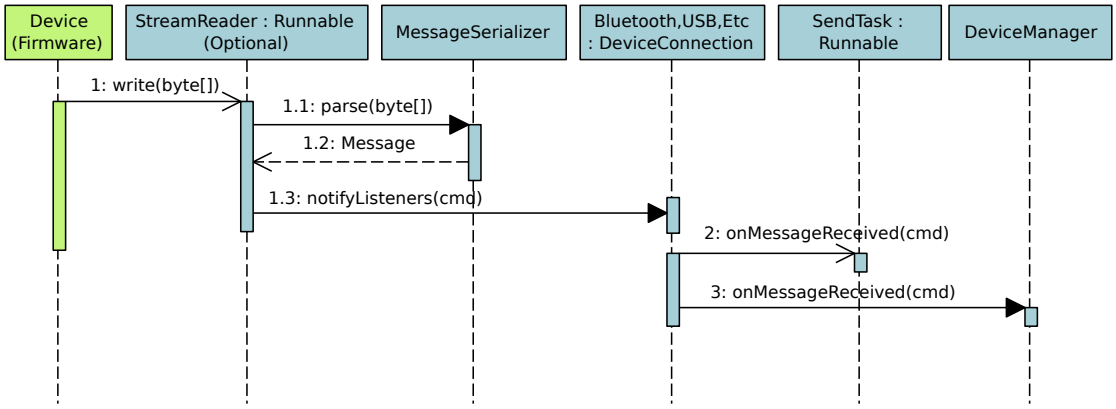

Figura 4.15 Recebimento de Comandos

4.14.4 Leitura de Sensores

O fluxo da leitura de sensores, apresentados nas figuras 4.16 e 4.17, trata-se também de um fluxo de recebimento de comandos, é similar ao da figura 4.15. Como abordamos na seção 4.13.6, o sistema de leitura de sensores é implementado de dois modos: síncrono (polling) e assíncrono (interrupções). O modo utilizado depende do suporte que o *hardware*. Como exemplo, utilizamos um microcontrolador AVR 8Bits (ex.: Arduino), executando o firmware, e enviando os dados dos sensores para a aplicação (ou middleware).

Leitura Síncrona (Polling)

No modo síncrono, a leitura dos sensores é feito no "loop" principal, e é realizada pela classe *OpenDevice*, através do método "checkSensorsStatus()" (fluxo 1). Neste método

todos os sensores serão lidos de forma sequencial, através do método “hasChanged()” da classe Device. Ao executar esse método o valor do dispositivo é atualizado, e caso tenha sofrido alguma alteração (fluxo 1.1.2), um comando (*DeviceCommand*) com o ID do dispositivo e valor lido é enviado para ao conexão (fluxo 1.2), que cuida de serializar e enviar os dados para a aplicação.

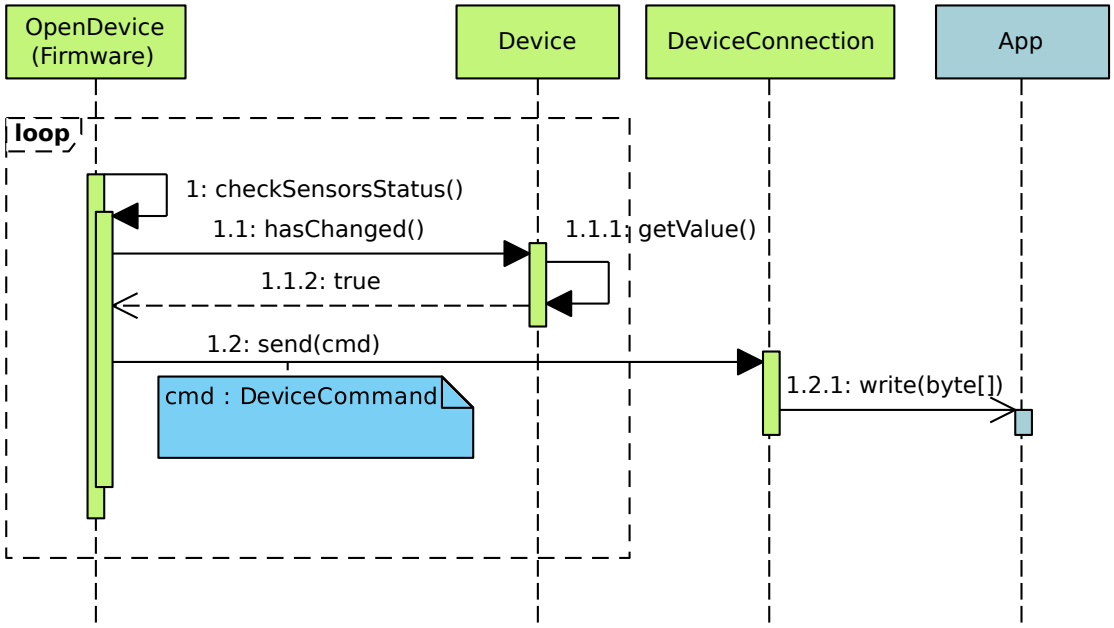

Figura 4.16 Leitura de Sensores no modo Polling

Leitura Assíncrona (Interrupções)

O método de leitura assíncrona, apresentado na figura 4.17, é realizado através de interrupções, como mencionamos na seção 4.13.6. Quando habilitado o suporte a interrupções, a classe OpenDevice é registrada para receber as interrupções através do método “onInterruptReceived()”. Neste método o dispositivo correspondente ao pino que sofreu a interrupção é localizado, e seu valor é atualizado. Os dispositivos que sofrerem alterações nos seus valores durante a interrupção, são marcados para sincronização (fluxo 1.2), que irá ocorrer na próxima execução do “loop” principal do programa. Similar à leitura síncrona, o método “checkSensorsStatus()” é chamada no “loop”, porém neste caso não é feito nenhuma leitura dos pinos, apenas o envios das informações para dos dispositivos que sofreram alterações para a aplicação (fluxo 1.2 e 2.1). A vantagem neste caso é que enquanto estão sendo enviadas as informações para aplicação, caso algum dispositivo tenha seu valor alterado, a interrupção cuida de deslocar o fluxo de execução, salvar esse

valor e retornar para a serialização dos dados. Na implementação no modo síncrona (polling), esse valor seria perdido.

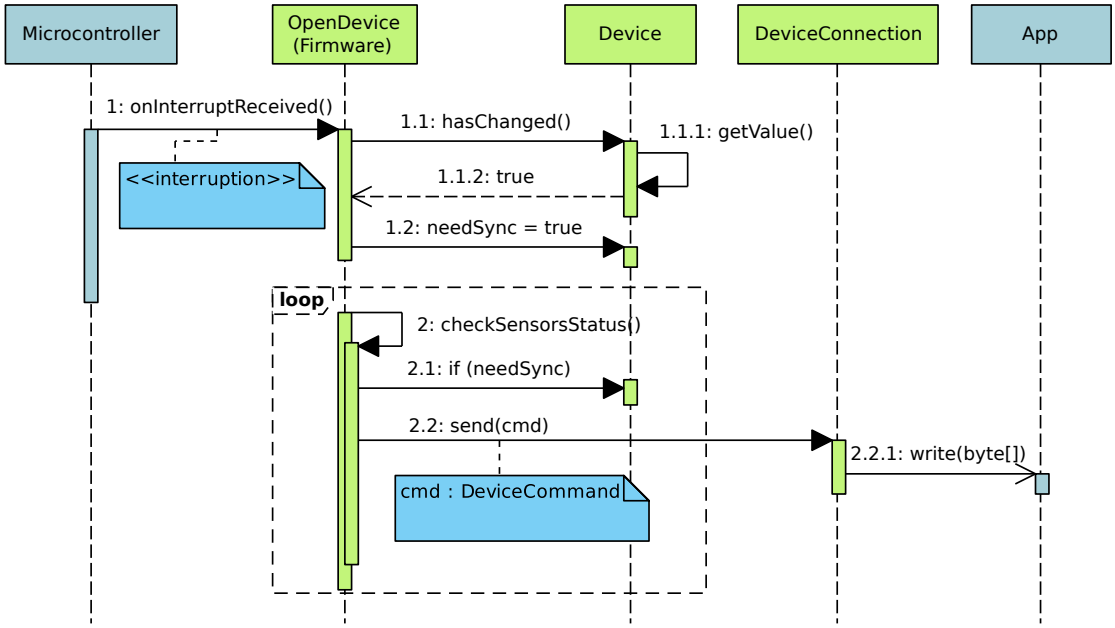

Figura 4.17 Leitura de Sensores usando Interrupções

4.14.5 Envio de Comandos (Aplicação - Middleware - Dispositivo)

A figura 4.18, apresenta o fluxo de execução de um comando enviado por uma aplicação cliente (WebApp) para os dispositivos, através do middleware. O framework trabalha com dois conceitos de conexões: (1) conexões de entrada, ou seja, os servidores e (2) conexões de saída, geralmente as conexões com os dispositivos físicos. O componente representado em "WSServerConnection", é um servidor WebSocket disponibilizado pelo módulo REST-WS-SERVER, que atua como uma conexão de entrada. A aplicação web no exemplo, está utilizando a biblioteca OPENDEVICE-JS, uma das implementações de cliente usando WebSocket, permitindo a conexão de forma simples com o servidor, e oferecendo a abstração dos dispositivos para a camada Web.

Ao alterar o valor de algum dispositivo na camada Web, que pode ser através dos métodos nos objetos JavaScript (fluxo 2), ou através de métodos disponibilizados no OPENDEVICE-JS, um comando é enviado via WebSocket para o Middleware (fluxo 2.1). A representação da conexão com o cliente "WSResource" recebe o comando (fluxo 2.1.1) e notifica para o DeviceManager (fluxos 2.1.2 e 2.1.2.1), que faz a atualização do dispositivo

relacionado ao comando recebido, salva no histórico de alterações, e repassa o comando para os dispositivos físicos através das conexões de saída. O procedimento de envio, representado na imagem pelo bloco “Ref: Send Command”, é o mesmo procedimento realizado no fluxo apresentado na seção Envio de Comandos (4.14.1).

O recebimento da resposta do dispositivo físico (fluxo 3), é redirecionado para a conexão de origem (WSResource). O redirecionamento é feito baseado na identificação da conexão (UUID), que é gerado no momento de sua criação (fluxo 1.1.1), e o mesmo é associado ao comando quando ele é enviado ou recebido, sendo então possível enviar a resposta para o cliente correto. Deste modo é possível permitir que múltiplas aplicações acessem o mesmo dispositivo, contornando as limitações do USB e Bluetooth que permitem apenas um cliente, ou mesmo de conexões Ethernet ou Wi-Fi, que dependendo do hardware, podem suportar apenas um cliente ou um número limitado.

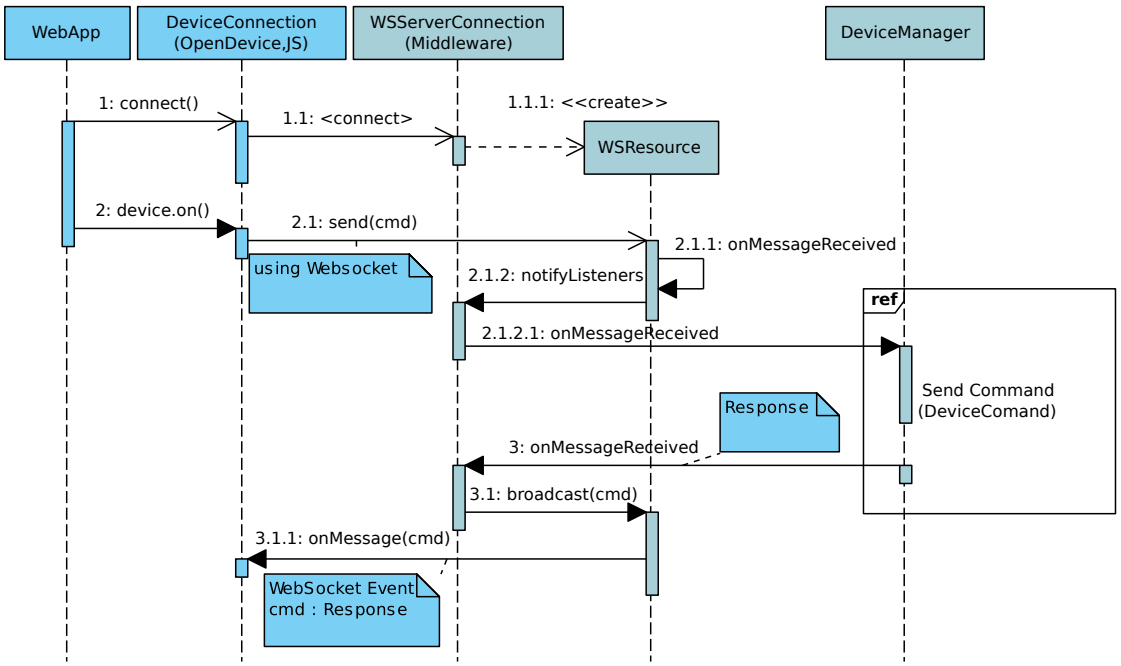

Figura 4.18 Envio de comandos

4.15 Protocolo

Esta seção apresenta o protocolo do OpenDevice. As principais características do protocolo é que ele foi projetado para ser um protocolo aberto, leve, simples, de fácil

implementação, legível para humanos e voltado para dispositivos com restrições de memória e processamento, como por exemplo, microcontroladores (AVR 8-bits, 2Kb de RAM).

O protocolo é baseado em ASCII (assim como o HTTP), orientado a mensagens/comandos e assíncrono. Surgiu de algumas influências do protocolo MIDI (formato usado para comunicação com instrumentos musicais) e do protocolo REST na sua estrutura básica.

O protocolo foi projetado para permitir a abstração, controle de dispositivos (atuadores), realizar a leitura de sensores e ser de fácil extensão, mas não limitado a isto. Ele pode ser utilizado em conjunto com outros protocolos, como WebSocket e MQTT, e com outras tecnologias de comunicação, como: USB, Bluetooth, Ethernet, Wi-Fi, etc.

4.15.1 Formato da Mensagem

Os comandos do OpenDevice possuem um "cabeçalho" fixo, contento o tipo do comando (*CommandType*) e o ID do comando, em seguida, o bloco *"Command Extension*", que varia de acordo com o tipo de comando.

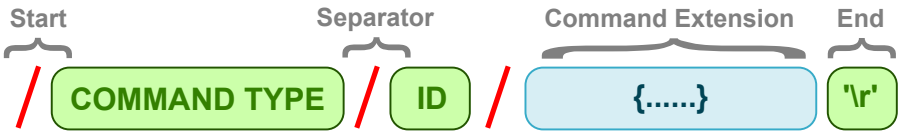

Figura 4.19 Formato do protocolo OpenDevice

- O **tipo do comando** define a estrutura do bloco "*Command Extension*". Os tipos suportados estão definidos na tabela 4.5.
- O **ID** do comando é numérico, sequencial e gerenciado pela aplicação cliente. Ele serve para a aplicação identificar o retorno(resposta) de algum comando enviado. Quando atinge seu valor máximo a contagem reinicia.
- Os blocos do comando são separados através do caractere "/".
- O comando finaliza com o terminador "\r" (Carriage return).

OBSERVAÇÃO: Na implementação atual do firmware, tanto o "tipo do comando" quanto o "id", são armazenados em varáveis de 8bits (uint8_t / byte), limitando a valores na faixa

de 0 a 255. Porém, os mesmos são parametrizados podendo escolher outros tipos como: uint16_t ou uint32_t.

Convenções e pontos em aberto

Algumas convenções foram adotadas para determinar observações, e destacar pontos em aberto. Caso o texto apresente "[*Numero]", significa que deve ser verificado as observações abaixo:

- Observação [*1]: Na implementação atual do firmware, o "DeviceID" é armazenado em uma variável de 8bits (byte), porém, o tipo é parametrizável.
- Observação [*2]: Significa que o parâmetro ou tipo é configurável.
- Observação [*3]: Significa que a versão atual do firmware não implementa o recurso.

4.15.2 Tipos de Comandos

A tabela 4.5, apresenta os tipos de comandos suportados.

Código	Tipo de Comando	Ref.	Formato
1	DIGITAL	4.15.3	[#/#]/{DeviceID}/{value}
2	ANALOG	4.15.4	[#/#]/{DeviceID}/{value}
3	NUMERIC	4.15.5	[#/#]/{DeviceID}/{value}
4...9	[Reservado]		–
10	COMMAND_RESPONSE	4.15.6	[#/#]/{status}/{DeviceID?}
11	SET_PROPERTY	4.15.7	[#/#]/{DeviceID}/{property}/{value}
12	GET_PROPERTIES	4.15.8	[#/#]/{DeviceID}
13	GET_PROPERTIES_RESPONSE	4.15.9	[#/#]/{DeviceID}/[{property1:value},...,N]
14	ACTION	4.15.10	[#/#]/{DeviceID}/{action}/[{value1?},...,{valueN}]
15	GET_ACTIONS	4.15.11	[#/#]/{DeviceID}
16	GET_ACTIONS_RESPONSE	4.15.12	[#/#]/{DeviceID}/[{action1},{actionN}]
16...19	[Reservado]		–
20	PING_REQUEST	4.15.13	[#/#][somente cabeçalho]
21	PING_RESPONSE	4.15.14	[#/#][somente cabeçalho]
22	DISCOVERY_REQUEST	4.15.15	[#/#][somente cabeçalho]
23	DISCOVERY_RESPONSE	4.15.16	[#/#]/{name}/{port}/{deviceLength}
24...29	[Reservado]		–
30	GET_DEVICES	4.15.17	[#/#]/{DeviceID?}/{value?}
31	GET_DEVICES_RESPONSE	4.15.18	[#/#]/[{DeviceInfo1},{DeviceInfoN?]
32	DEVICE_ADD	4.15.19	[#/#]/{DeviceInfo}
33	DEVICE_DEL	4.15.20	[#/#]/{DeviceID?}
41..100	[Reservado]		

Tabela 4.5 Tipos de comandos do protocolo

4.15.3 Comando: DIGITAL

Este comando permite controlar os dispositivos digitais (atuadores) ou notificar alguma mudança no estado/valor de um sensor. O comando adiciona dois blocos adicionais: DeviceID e Value (Figura 4.20).

Ao enviar um comando para um dispositivo, por exemplo, ligar lâmpada, o dispositivo deve retornar uma resposta (geralmente assíncrona), com o status do comando.

O formato da resposta é definida em "Reposta: COMMAND_RESPONSE", onde o ID da resposta será o mesmo ID do comando enviado.

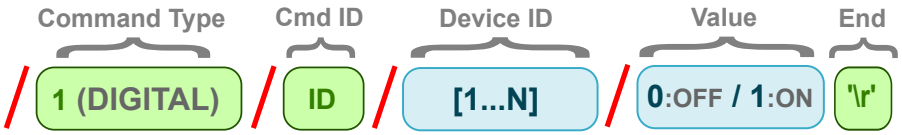

Figura 4.20 Comando: DIGITAL

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo
Value	bool	0 (Desligar / Desligado) ou 1 (Ligar / Ligado)

Tabela 4.6 Parâmetros - Comando: DIGITAL

4.15.4 Comando: ANALOG

Obedece o formado do "Comando: DIGITAL", mudando apenas o tipo do comando, no caso 2, e o tipo de dado do bloco "*Value*", que pode aceitar valores do tipo *"unsigned long*", com tamanho de 32 bits (4 bytes).

Este comando pode ser utilizado tanto para sensores como para atuadores, no caso de sensores, ele é utilizado para notificar um mudança de estado (já que sensores são apenas para "leitura").

O formato da resposta é definida em "Reposta: COMMAND_RESPONSE", onde o ID da resposta será o mesmo ID do comando enviado.

4.15.5 Comando: NUMERIC

Obedece o formado do "Comando: ANALOG". Este comando é utilizado em conjunto com os dispositivos (principalmente sensores) do tipo *NUMERIC*, que permitem o envio de informações repetidas. Um exemplo, o sensor RFID, que gera um evento toda vez que uma etiqueta é lida, mesmo sendo a mesma etiqueta.

4.15.6 Reposta: COMMAND_RESPONSE

Representa a resposta de um determinado comando. As respostas são associadas ao comando que a originou (requisição), através do atributo "ID" do cabeçalho fixo. Adi-

cionalmente os comandos que referenciam um dispositivo, na resposta, será incluído o mesmo “DeviceID” do comando da “requisição”.

HEADER	Status	DeviceID

Tabela 4.7 Reposta: COMMAND_RESPONSE

Parâmetro	Tipo	Descrição
ID	byte [*2]	Mesmo ID enviado na requisição
Status	byte	Obedece aos valores da tabela: 4.9
DeviceID	byte [*1] (Opcional)	Mesmo ID enviado na requisição

Tabela 4.8 Parâmetros - Comando: COMMAND_RESPONSE

Status	Código
SUCCESS	200
NOT_FOUND	404
BAD_REQUEST	400
UNAUTHORIZED [*3]	401
FORBIDDEN [*3]	403
PERMISSION_DENIED [*3]	550
INTERNAL_ERROR	500
NOT_IMPLEMENTED	501

Tabela 4.9 Status da resposta do comando

4.15.7 Comando: SET_PROPERTY

Comando utilizado para atualizar alguma propriedade do dispositivo. O formato da resposta é definido em “Reposta: COMMAND_RESPONSE”.

HEADER	DeviceID	property	value

Tabela 4.10 Comando: SET_PROPERTY

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo
property	String	Propriedade a ser alterada
value	String	Valor da propriedade

Tabela 4.11 Parâmetros: SET_PROPERTY

4.15.8 Comando: GET_PROPERTIES

Comando utilizado para recuperar a lista de propriedades do dispositivo e seus respectivos valores. O formato da resposta é definido em "Comando: GET_PROPERTIES_RESPONSE"

HEADER	DeviceID

Tabela 4.12 Comando: GET_PROPERTIES

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo

Tabela 4.13 Parâmetros: GET_PROPERTIES

4.15.9 Comando: GET_PROPERTIES_RESPONSE

Resposta do comando "*GET_PROPERTIES*", com a lista de propriedades do dispositivo. A lista é retornada no formato de um Array, e os elementos no formato: "Propriedade:Valor".

HEADER	DeviceID	[property:value,...,propertyN:value]

Tabela 4.14 Comando: GET_PROPERTIES_RESPONSE

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo
property	String	Propriedade
value	String	Valor da propriedade

Tabela 4.15 Parâmetros: GET_PROPERTIES_RESPONSE

4.15.10 Comando: ACTION

Comando usado para executar as ações definidas pelos dispositivos. As ações podem receber uma lista de valores, que são especificados em formato de Array. O formato da resposta é definido em "Reposta: COMMAND_RESPONSE".

HEADER	DeviceID	action	[{value1},...,{valueN}]

Tabela 4.16 Comando: ACTION

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo
action	String	Ação que deve ser executada
value	Array[String] (Opcional)	A lista de valores que a ação recebe

Tabela 4.17 Parâmetros: ACTION

4.15.11 Comando: GET_ACTIONS

Comando utilizado para recuperar a lista de ações definidas para o dispositivo. O formato da resposta é definido em "Comando: GET_ACTIONS_RESPONSE"

HEADER	DeviceID

Tabela 4.18 Comando: GET_ACTIONS

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo

Tabela 4.19 Parâmetros: GET_ACTIONS

4.15.12 Comando: GET_ACTIONS_RESPONSE

Resposta do comando "*GET_ACTIONS*", com a lista de ações do dispositivo. A lista é retornada no formato de um Array.

HEADER	DeviceID	[{action1},..,{actionN}]

Tabela 4.20 Comando: GET_ACTIONS_RESPONSE

Parâmetro	Tipo	Descrição
DeviceID	byte [*1]	Identificador único que representa o dispositivo
action	Array[String]	Lista de ações do dispositivo

Tabela 4.21 Parâmetros: GET_ACTIONS_RESPONSE

4.15.13 Comando: PING_REQUEST

Comando enviado pelo firmware para notificar que está em funcionamento e ao mesmo tempo monitora o status do middleware/aplicação.

Este comando não possui parâmetros adicionais, apenas cabeçalho. O formato da resposta é definido em "Resposta: PING_RESPONSE"

4.15.14 Resposta: PING_RESPONSE

Resposta para o comando "PING_REQUEST". Este comando não possui parâmetros adicionais, apenas cabeçalho.

4.15.15 Comando: DISCOVERY_REQUEST

Comando utilizado para realizar a descoberta de dispositivos (módulos) em uma rede. Geralmente é enviado via *broadcast*. Este comando não possui parâmetros adicionais, apenas cabeçalho.

4.15.16 Resposta: DISCOVERY_RESPONSE

Resposta com as informações de descoberta dos dispositivos (módulos) .

HEADER	name	port	deviceLength

Tabela 4.22 Comando: GET_ACTIONS_RESPONSE

Parâmetro	Tipo	Descrição
name	String	Nome do dispositivo (módulo)
port	Array[String]	Lista de ações do dispositivo
deviceLength	int	Quantidades de dispositivos que o módulo possui

Tabela 4.23 Parâmetros: GET_ACTIONS_RESPONSE

4.15.17 Comando: GET_DEVICES

Comando usado para obter informações de um ou mais dispositivos. Os campos 'DeviceID' e 'Valor' são opcionais e podem ser usados como filtro para o comando.

HEADER	DeviceID	Value

Tabela 4.24 Comando: GET_DEVICES

Parâmetro	Tipo	Descrição
DeviceID (Opcional)	byte [*1]	Informar o ID do dispositivo ou 0 para todos
Value (Opcional)	*unsigned long*	Permite filtrar por valor ou 0 para todos

Tabela 4.25 Parâmetros: GET_DEVICES

4.15.18 Resposta: GET_DEVICES_RESPONSE

Resposta do comando "GET_DEVICES", com a lista de dispositivos cadastrados. O bloco "*Command Extension*", consiste em um Array de objetos do "Tipo: DeviceInfo", separados por ",". A quantidade de dispositivos deve ser inferida automaticamente.

HEADER	[DeviceInfo,...,DeviceInfoN]

Tabela 4.26 Resposta: GET_DEVICES_RESPONSE

Parâmetro	Tipo	Descrição
ID	byte [*2]	Mesmo ID enviado da requisição
DeviceInfo	Array[DeviceInfo]	Lista dos atributos do Tipo: DeviceInfo

Tabela 4.27 Parâmetros - Resposta: GET_DEVICES_RESPONSE

4.15.19 Comando: DEVICE_ADD

Comando utilizado para configurar dinâmicamente os dispositivos de um módulo. O formato da resposta é definido em "Reposta: COMMAND_RESPONSE".

HEADER	DeviceInfo

Tabela 4.28 Comando: DEVICE_ADD

Parâmetro	Tipo	Descrição
DeviceInfo	DeviceInfo	Lista dos atributos do Tipo: DeviceInfo

Tabela 4.29 Parâmetros: DEVICE_ADD

4.15.20 Comando: DEVICE_DEL

Comando utilizado deletar o(s) dispositiv(o) de um módulo. O formato da resposta é definido em "Comando: GET_ACTIONS_RESPONSE"

HEADER	DeviceID

Tabela 4.30 Comando: DEVICE_DEL

Parâmetro	Tipo	Descrição
DeviceID	byte [*1] (Opcioanl)	Informar o ID do dispositivo ou 0 para todos

Tabela 4.31 Parâmetros: DEVICE_DEL

4.15.21 Tipo: DeviceInfo

Representa as informações e um dispositivo. É representado como um Array no seguinte formato: `[ID, PIN, VALUE, TARGET, SENSOR, TYPE]`.

Atributo	Tipo	Descrição
ID	byte [*2]	ID do dispositivo
PIN	byte	Pino que o dispositivo está vinculado
VALUE	unsigned long	Valor atual do dispositivo
TARGET	byte	Quando for sensor, representa outro dispositivo (Opcional)
SENSOR	bool	0 para atuador, 1 para sensor
TYPE	byte	Tipo de dispositivo (Tabela 4.33)

Tabela 4.32 Objeto: DeviceInfo

Código	Tipo
1	DIGITAL
2	ANALOG
3	NUMERIC

Tabela 4.33 Tipos de Dispositivos

5

Avaliação Experimental

Este capítulo apresenta avaliação experimental da implementação da arquitetura apresentada no capítulo 4. Os objetivos gerais desta avaliação e a estruturação da mesma são apresentados a seguir.

Na seção 5.1, foram realizados testes em diferentes hardwares, permitindo avaliar sua capacidade de adaptabilidade e compatibilidade tanto da perspectiva do middleware, quanto do firmware. Na seção 5.2, foram conduzidos testes de performance com diferentes tecnologias de comunicação e protocolos, permitindo avalizar a capacidade da arquitetura de lidar com tecnologias heterogêneas e avaliar o desempenho e latências de comunicação. A seção 5.3, apresenta experimentos de integração com ambientes de desenvolvimento de jogos 3D (*Game Engines*), *Blender* e *jMonkeyEngine.* Estas ferramentas podem ser utilizadas para criar ambientes de simulação, facilitando a criação de aplicações de IoT, sem depender de um ambiente ou hardware real. Por fim, a seção 5.4, apresenta um estudo de caso, que permite avaliar a arquitetura proposta em um contexto real. O contexto apresentado, está relacionado à automação residencial, focando, principalmente, no controle de acesso usando a tecnologia RFID.

5.1 Hardwares Testados

Os testes realizados com os hardware estão subdivididos nas categorias de microcontroladores (onde é executado o firmware) e os hardwares na categoria de mini PCs que possuem um poder de processamento maior e são destinados a executar o middleware ou aplicação construída utilizando o framework.

5.1.1 Microcontroladores (Firmware)

Apesar do firmware ser construído com base na API do Arduino, o que o tornaria automaticamente compatível com uma série de dispositivos [11, 12, 13], os testes demostraram que alguns microcontroladores possuem peculiaridades que têm que ser levadas em consideração, principalmente pelo nível de abstração que é proposto pelo sistema de configuração dinâmico de conexões que é implementado (seção 4.13.7). Como exemplo, o firmware deve ser capaz de identificar quando estiver rodando em um ESP8266, e realizar as configurações para conexão Wi-Fi sem necessidade de modificações na aplicação/firmware.

A tabela 5.1 apresenta a lista de hardwares dessa categoria que foram testados. Em seguida é apresentada a tabela 5.2, com os módulos de conexão testados.

Nome	Processador	Arquitetura
Arduino UNO	ATmega328P - 16 MHz	AVR 8-bit
Arduino Nano	ATmega328P - 16 MHz	AVR 8-bit
Arduino Leonardo	ATmega32u4 - 16 MHz	AVR 8-bit
Arduino MEGA	ATmega2560 - 16 MHz	AVR 8-bit
Arduino Yun	ATmega32U4 / AR9331 Linux	AVR 8-bit
Arduino Due	ATSAM3X8E - 84 MHz	ARM 32-bit
ESP8266	SoC (Tensilica's L106) - 80 MHz	RISC 32-bit
Stellaris Launchpad	LM4F120H5QR - 80 MHz	ARM M4F 32-bit
Digispark	ATtiny85 - 20MHz	AVR 8-bit
Teensy 3.1	MK20DX256 - 72 MHz	ARM M4 32-bit

Tabela 5.1 Hardwares Testados (Microcontroladores)

Módulo	Conexão
Arduino Ethernet Shield	Ethernet
Módulo ENC28J60	Ethernet
ESP8266 (AT Mode)	Wi-Fi
HC-05 / HC-06	Bluetooth SPP

Tabela 5.2 Hardwares Testados (Módulos)

5.1.2 Mini PCs

No trabalho, nomeamos de mini PCs, os hardware capazes de executar um sistema operacional completo, como o Linux, e que possuem 512MB de memória RAM ou

superior. Hardwares baseados na arquitetura ARM são elegíveis para suporte a máquina virtual Java (JVM), consequentemente tendo suporte para execução do middleware. Entretanto a forma de acesso aos periféricos dos mesmos, como os pinos de GPIO, são específicos de cada plataforma. Não existe uma especificação genérica como a disponível no framework do Arduino, que abstraia o acesso aos pinos de GPIO de uma forma unificado. Uma proposta é API *Device I/O*[39], porém, é um especificação nova e sua compatibilidade é limitada.

Nome	Arquitetura	RAM
Raspberry Pi Model B (Alpha)	ARM	256MB
BeagleBone Black	ARM	512MB

Tabela 5.3 Hardwares Testados (Mini PCs)

5.2 Testes de Performance

O objetivo deste experimento é avaliar a performance do middleware, implementações das tecnologias de comunicação e do firmware.

5.2.1 Procedimento de Medição

Os experimentos foram executados através da ferramenta JMH[1], que permite a criação de *micro-benchmarks* em Java, isolando alguns comportamentos da JVM, aproximando ao máximo de um ambiente real.

Os seguintes parâmetros de configuração foram utilizados na execução dos testes:

- Iterações de pré-execução: 2;
- Tempo de execução das iterações de pré-execução: 2s;
- Iterações de coleta de dados: 20;
- Tempo de cada iteração: 1s;
- Argumentos da JVM: “-server”.

[1] http://openjdk.java.net/projects/code-tools/jmh/

As iterações de “pré-execução” são utilizadas para garantir que a JVM, todas as classes da aplicação e bibliotecas, sejam carregadas completamente, evitando interferência de acesso ao disco.

Em cada iteração de coletas de dados são enviados um série de comandos (*Device-Command*) para os dispositivos, durante o período definido (1s). Em cada iteração, são enviados o máximo de comandos possíveis de forma sequencial e síncrona, permitindo calcular o tempo médio de resposta de cada comando.

Ao final de todas iterações definidas (20), as estatísticas são geradas pela ferramenta.

5.2.2 Métrica Utilizada

A métrica utilizada é o tempo de resposta fim-a-fim. Após todas iterações serem executadas, é calculado o tempo médio de resposta de cada iteração e em seguida, o tempo médio de reposta geral de todas iterações e o erro médio.

O tempo de resposta, inclui o tempo para geração da mensagem no middleware, transmissão, processamento e execução da ação no dispositivo e recebimento da resposta.

5.2.3 Resultados: Conexão USB

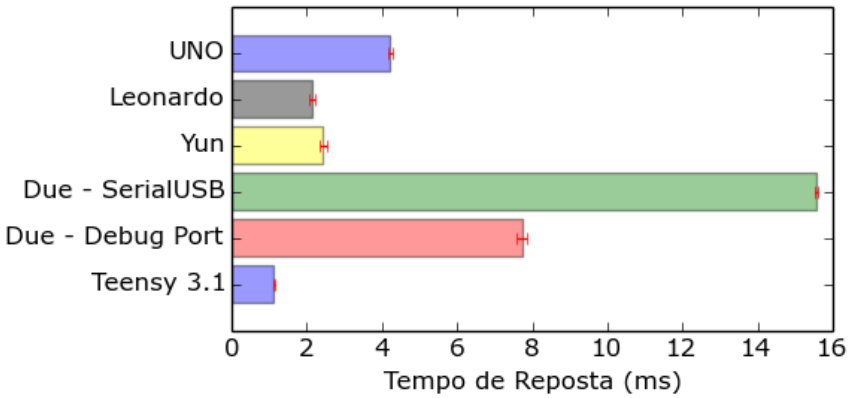

Figura 5.1 Teste de performance - USB

5.2.4 Resultados: Conexão Bluetooth

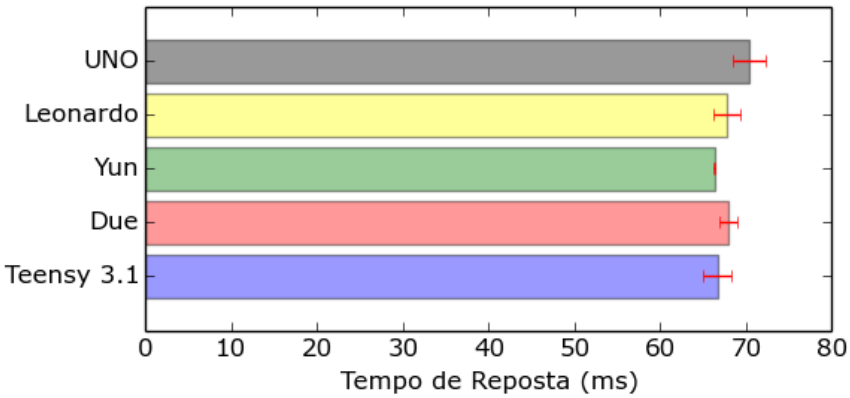

Figura 5.2 Teste de performance - Bluetooth

5.2.5 Resultados: Conexão Ethernet

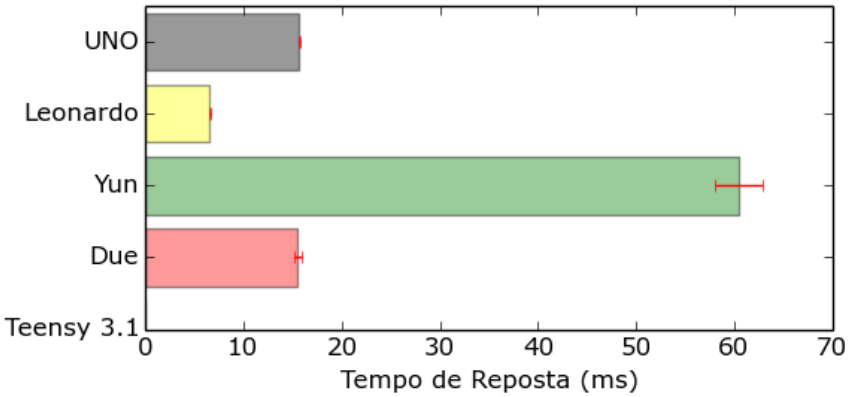

Figura 5.3 Teste de performance - Ethernet

5.3 Integração com Ambientes 3D

O desenvolvimento de aplicações para Internet das Coisas conta com o desafio de ter que lidar com vários dispositivos físicos. Como alternativa ambientes de simulação podem ser utilizados para executar os testes e experimentos, antes de testar com os dispositivos

reais. Outra possibilidade para utilização dos ambientes 3D, é permitir a integração entre um ambiente real e virtual.

Foram realizados testes de integração do OpenDevice com sistema de desenvolvimento de jogos (*Game Engine*), afim de validar uma possível utilização dos mesmos para criação de ambientes de simulação. Os experimentos realizados são detalhados a seguir.

5.3.1 Integração - jMonkeyEngine.

A *jMonkeyEngine*[2] é uma ferramenta gratuita de código fonte aberto, utilizada para construção de jogos 3D em Java. Possui uma boa documentação e ambiente de desenvolvimento integrado. Devido sua implementação ser baseada na linguagem Java, a integração com o OpenDevice se torna bastante simples e permite a construção de uma aplicação de simulação que utiliza as APIs e módulos do OpenDevice diretamente, sem a necessidade do middleware.

No teste realizado, foi construída uma aplicação simples para validar a integração, que consiste no mapeamento dos dispositivos físicos, no caso 3 LEDs, e sua vinculação com objetos virtuais 3D da *Game Engine*. Ao clicar em algum objeto 3D, como os representados na figura 5.4, o OpenDevice localiza o dispositivo vinculado e envia o comando para o acionamento do dispositivo físico.

5.3.2 Integração - Blender.

O *Blender*[3] é uma ferramenta livre e de código fonte aberto, escrita em sua maior parte utilizando a linguagem de programação Python. Ela possui ferramentas para modelagem, animação, renderização e desenvolvimento de Jogos. O diferencial em relação à *jMonkeyEngine,* é que o *Blender* possui uma série de facilidades para configurações de eventos e vinculações de scripts em Python utilizando apenas a interface gráfica.

Para permitir a integração com o OpenDevice, foi necessária a criação de uma biblioteca cliente em Python, que implementa o protocolo do OpenDevice e permite uma comunicação bidirecional através de uma comunicação TCP. O experimento realizado (Figura 5.5), é similar ao experimento realizado com a *jMonkeyEngine*, permitindo fazer

[2] *http://jmonkeyengine.org/*
[3] *https://www.blender.org/*

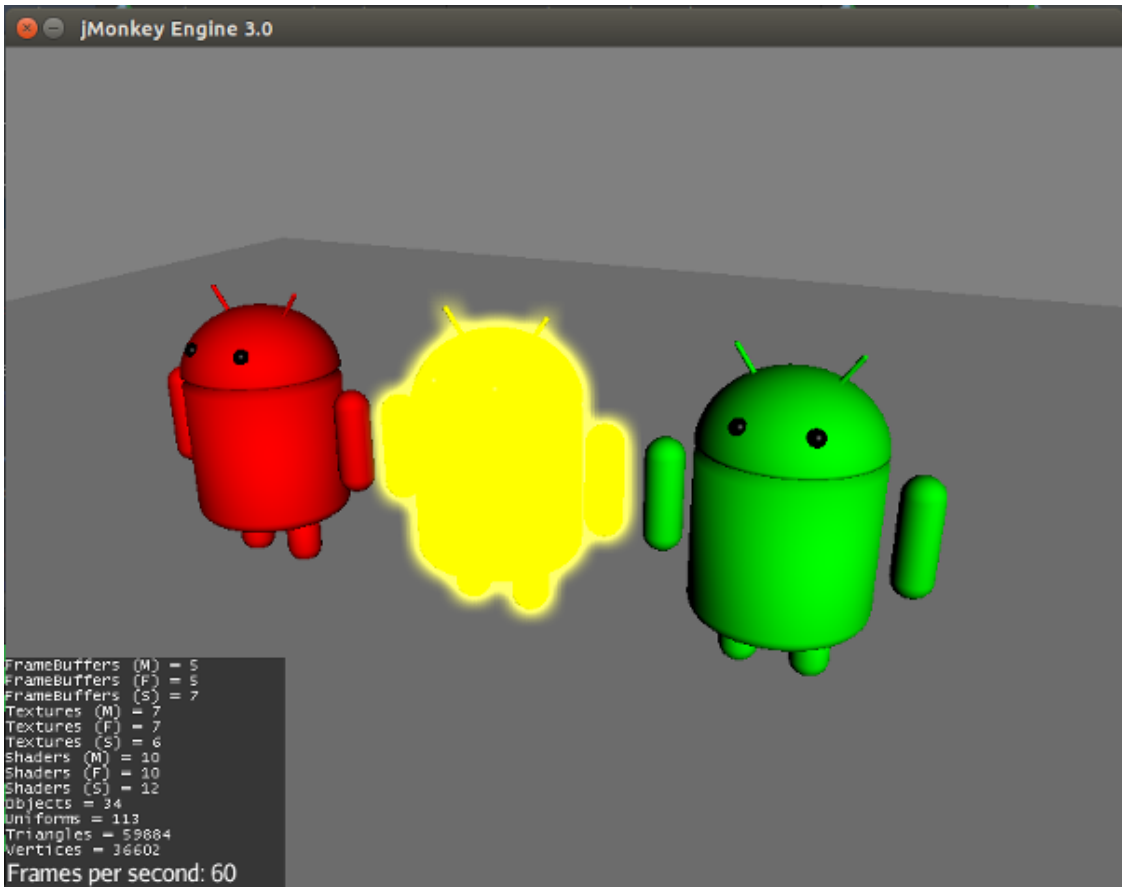

Figura 5.4 Integração com a JMonkeyEngine

a vinculação entre objetos virtuais 3D e os dispositivos físicos[4].

Para validar a performance da integração, foram realizados testes visuais do tempo de resposta entre duas aplicações gráficas (Figura 5.6), uma escrita em Java (executando o middleware) e outra Python. Verificou-se que as duas interfaces respondem em tempo real às modificações na barra de rolagem (slider) em ambas interfaces[5].

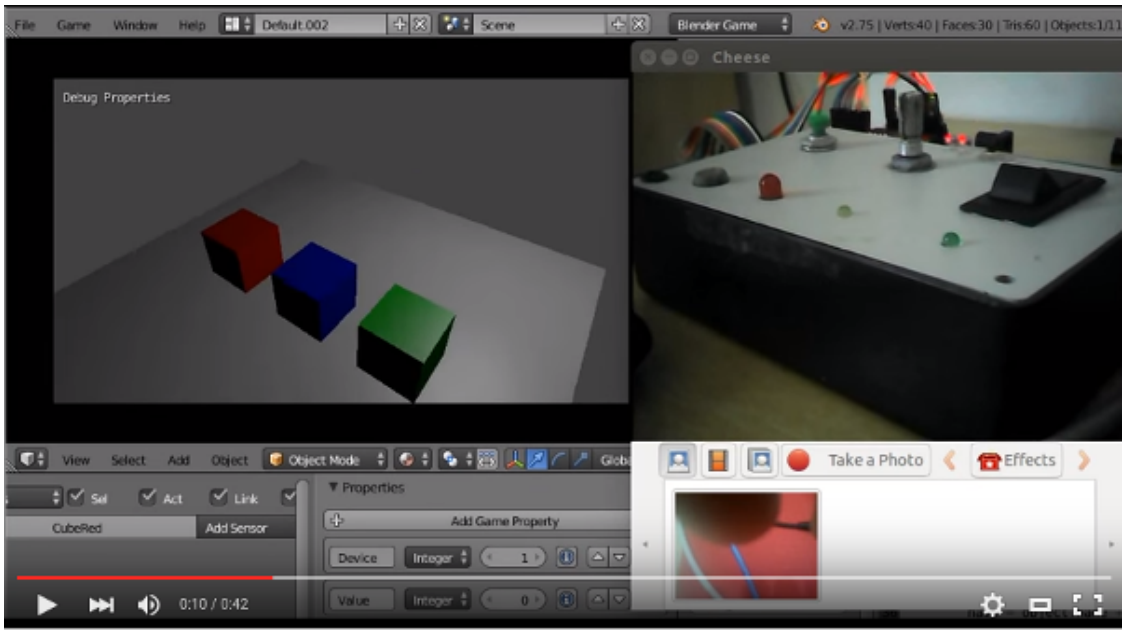

Figura 5.5 Integração com o Blender

5.4 Estudo de Caso

A arquitetura descrita conforme a seção 4, foi implementada vários testes foram conduzidos para validar os componentes da arquitetura em relação ao design, integração e performance. Nesta seção, faremos um estudo de caso baseado em um cenário real, permitindo avaliar a integração entre os componentes da arquitetura e as capacidades de evolução do framework proposto.

[4]Vídeo do experimento da figura 5.5: http://youtu.be/b3PbOPIMHmY
[5]Vídeo do experimento da figura 5.6: http://youtu.be/j4dMnAPZu70

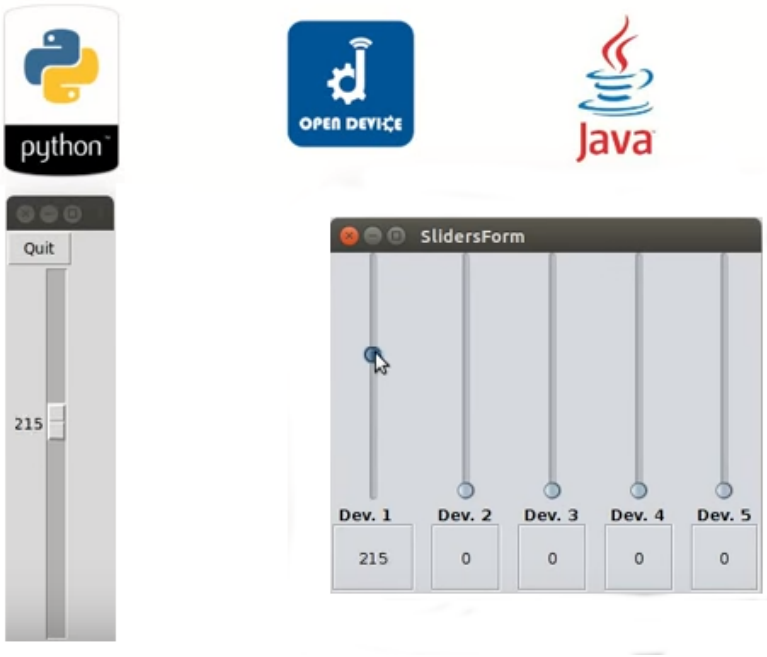

Figura 5.6 Teste de desempenho do cliente Python

5.4.1 Resumo

O cenário escolhido para validação da proposta consiste em um sistema de controle de acesso, usando a tecnologia RFID e alguns elementos de automação residencial. A estratégia utilizada consiste na elaboração de um cenário mais simples e sua posterior evolução para um cenário mais complexo, integrando outros dispositivos e plataformas: Hardware, Desktop, Web/Cloud e Mobile. O projeto será implantado nas instalações do prédio denominado “GEDAI”, onde está instalada a CriativaSoft (empresa do autor) e outras duas empresas, sendo utilizado para o controle de acesso dos funcionários.

Os principais objetivos são: (1) avaliar os componentes da arquitetura em conjunto, (2) validar os modelos de comunicação Cloud e Local, (3) avaliar a integração com novos dispositivos (sensores e atuadores), (4) avaliar a integração com dispositivos IP que utilizam outros protocolos, (5) avaliar a integração com aplicações cliente Mobile(Android) e (5) avaliar as capacidades de extensibilidade da plataforma.

5.4.2 Ambiente de Teste

Neste experimento serão utilizados hardwares de baixo custo, como o Arduino, ESP8266 e Raspberry Pi, os demais sensores e atuadores utilizados serão descritos nas seções seguintes. O middleware foi implantado em um servidor na Amazon EC2, permitindo que aplicações cliente controlem e recebam informações dos dispositivos pela Internet.

O experimento será avaliado em dois cenários, descritos e detalhados a seguir.

5.4.3 Cenário 1

Este cenário tem como objetivo avalizar a utilização da arquitetura do OpenDevice para criação de projetos simples para Internet das Coisas, utilizando componente e hardwares existentes no mercado para criação de projetos inovadores. Como mencionado, este cenário consiste na criação de uma aplicação para controle de acesso usando RFID, conforme apresentado na figura 5.7. Neste cenário, o hardware está conectada à Internet através de uma comunicação Ethernet ou Wi-Fi e se comunica com o middleware utilizando o protocolo do OpenDevice em conjunto com o protocolo MQTT. Este cenário tem como característica ser um cenário de fácil implantação.

Para implementação dessa aplicação, foi utilizado o Arduino Yún, uma versão do Arduino que possui conexão Ethernet e Wi-Fi já embutidas na própria placa, e executa uma versão customizada no Linux para roteadores, o OpenWrt-Yún[135], porém é possível utilizar qualquer outra versão do Arduino, com um módulo que forneça uma comunicação IP. A leitura dos cartões de acesso é feita por um leitor RFID de proximidade que trabalha na frequência 13.56 MHz. O leitor é baseado no processador MFRC522[136] da empresa NXP, compatível com cartões/tags RFID padrão ISO 14443A. A comunicação com este leitor é realizada através do protocolo SPI[137, 138].

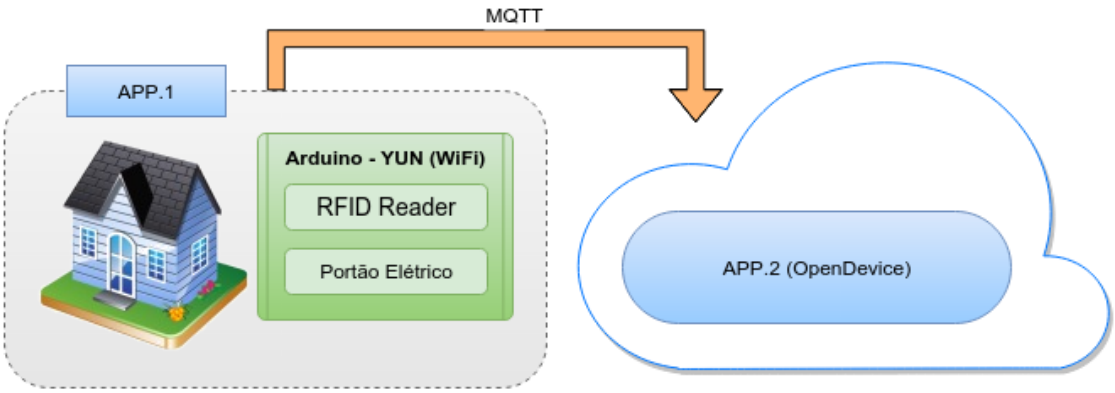

Figura 5.7 Diagrama - Cenário 1

Estruturação do projeto:

Neste cenário foram implementados duas aplicações: (1) uma aplicação embarcada (firmware), que faz o gerenciamento dos dispositivos físicos, e (2) uma aplicação Java que utiliza o framework do OpenDevice, que é a responsável pela validação dos cartões lidos. O firmware, utiliza as bibliotecas do OpenDevice, MQTT[139] e do leitor RFID (MFRC522)[140]. A aplicação Java, é uma aplicação bem simples, utilizando o módulo de servidor MQTT embarcado, o módulo core da plataforma e um banco de dados em memória chamado MapDB[141], com suporte a serialização em disco. O MapDB foi escolhido por ser o mesmo utilizado na implementação do servidor MQTT (baseado no Moquette).

Descrição básica de funcionamento

Ao detectar a presença de alguma etiqueta/cartão RFID, o firmware envia a notificação para aplicação Java, que verifica a existência do código lido no cache em memória (aumentando a performance) e envia a resposta de confirmação de volta para o firmware, através de um comando customizado (action), chamado *"onAuthFinish"*.

Na função "*onAuthFinish*", definida pelo firmware, ele verifica o parâmetro enviado pela aplicação, se a autenticação foi realizada, em caso positivo, é emitido um sinal sonoro e liberado a fechadura elétrica. Em caso negativo, é emitido um sinal sonoro longo e o acionamento de um LED vermelho, permitindo o usuário identificar a não autorização.

O firmware conta com uma função que emite um alerta (sonoro e visual), caso o

servidor (aplicação Java) não efetue a resposta até um tempo limite. É possível, também, a definição de chaves mestre, que não necessitam de autenticação “on-line”, permitindo a a liberação do acesso caso não exista conectividade com a Internet.

5.4.4 Cenário 2

O cenário 2, representado pela figura 5.8, pode ser considerado uma complementação do cenário 1, focado no controle de acesso, mas incluindo novos elementos de automação. Neste cenário é utilizado um servidor local, executando em um Raspberry Pi (ou BeagleBone), e o mesmo está sincronizado com o middleware na Internet.

O objetivo é avaliar o gerenciamento de vários dispositivos, a integração com uma câmera IP (que opera com protocolo próprio), a interface entre uma aplicação Mobile e dispositivos físicos pela Internet e a integração entre uma aplicação local e o middleware instalado em um servidor na nuvem.

Além do controle de acesso, usando RFID, este cenário integra uma campainha sem fio, operando na frequência 433Mhz, uma câmera IP (clone da Foscam) e um emissor de infra-vermelho para controle do ar-condicionado da sala de reunião.

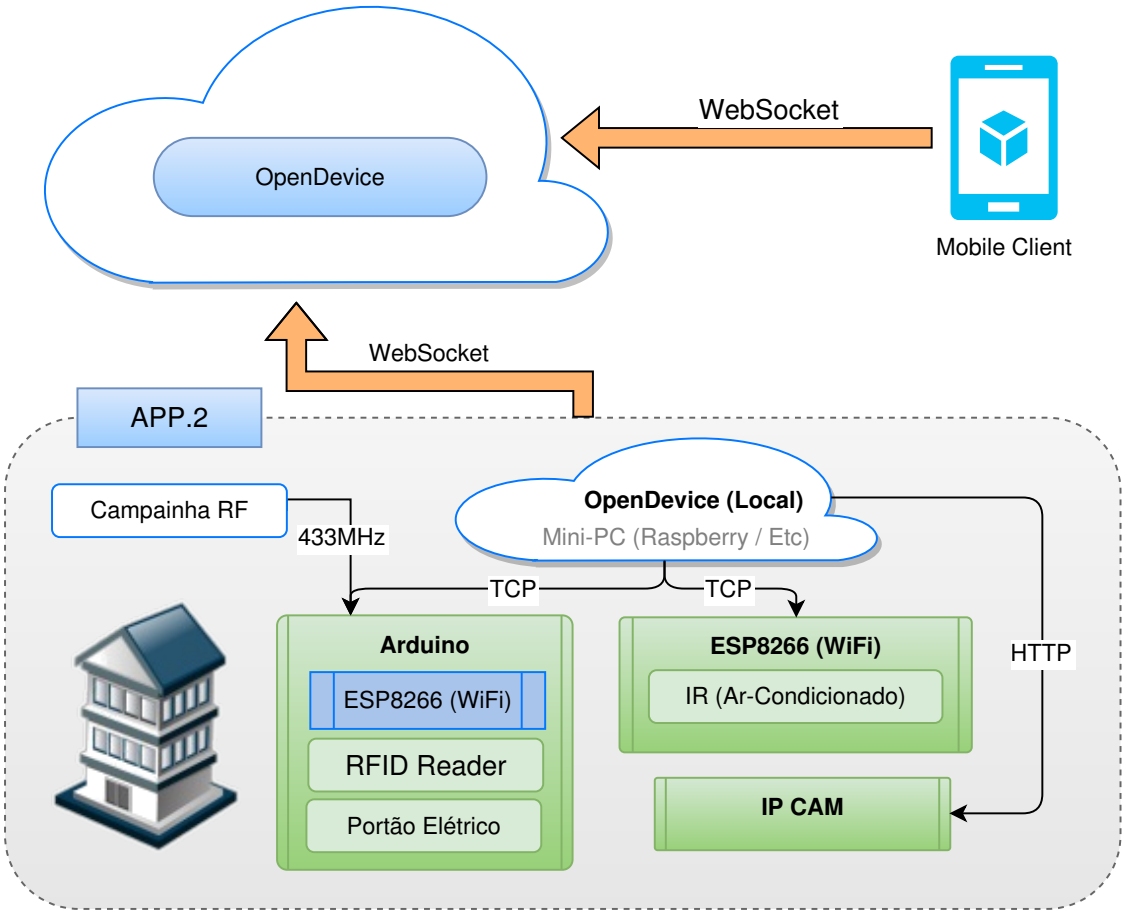

Figura 5.8 Diagrama - Cenário 2

Descrição básica de funcionamento

Quando a campainha é pressionada, o receptor RF 433Mhz acoplado ao Arduino detecta o sinal e o firmware envia a notificação, via conexão Wi-Fi, para a aplicação (ou middleware), que está executando no servidor local. Em seguida, a aplicação captura a imagem da câmera IP e envia uma notificação para as aplicações mobile, informando que uma visita está aguardando a liberação, que pode ser realizada pelo próprio aplicativo Mobile.

Estruturação do projeto:

Este cenário é composto por 5 aplicações/componentes, que serão detalhados a seguir:

- **Dispositivo 1 (Arduino)**: Similar à implementação do cenário 1, com algumas modificações no hardware. Foi utilizado o Arduino UNO e a conectividade Wi-Fi é fornecida pelo módulo ESP8266, utilizando o firmware AT, conectado na porta UART do Arduino. Também foi incluído um módulo receptor RF 433 Mhz que recebe o sinal da campainha.

- **Dispositivo 2 (ESP8266)**: O dispositivo que faz o controle do ar-condicionado. Opera no modo independente (standalone), ou seja, sem depender de outro componente. É programado com um firmware baseado no framework do Arduino[51], utilizando a biblioteca do OpenDevice e a biblioteca para operar o emissor de infra-vermelho[6]. A comunicação com a aplicação local, é feita usando o protocolo MQTT em conjunto com o protocolo do OpenDevice.

- **Middleware Local**: Aplicação que executa localmente no Raspberry Pi e responsável pelo gerenciamento de todos os dispositivos do projeto. É baseada na versão padrão do middleware, e as regras específicas para o projeto são implementadas através de extensões. As extensões incluídas neste cenário adicionaram novas entidades persistentes, novas interfaces Rest, suporte a novos dispositivos (câmera IP) e novas páginas na interface administrativa do middleware. O sistema de armazenamento utiliza a implementação padrão, baseada no Neo4J, com suporte a JPA (Java Persistence API). A aplicação local possui uma conexão com a versão do middleware que está implantado em um servidor na nuvem (Amazon), permitindo que aplicações clientes controlem os dispositivos pela Internet.

- **Middleware**: Implementação padrão disponibilizada pelo OpenDevice, o middleware foi implantado em um servidor na nuvem, permitindo a comunicação com dispositivos pela Internet, sem a necessidade de configurações de IP Fixo ou DDNS (Dynamic DNS).

- **Aplicação Mobile (Android)**: Permite a visualização da imagem (posteriormente vídeo) capturada pela câmera, liberação da fechadura elétrica e controle do ar-condicionado da sala de reunião. A comunicação é realizada com o middleware local, caso o dispositivo mobile esteja conectado na mesma rede do middleware, ou com o middleware na Internet, caso esteja usando uma rede 3G/4G.

[6] https://github.com/markszabo/IRremoteESP8266

Integração com Câmeras IP

A câmera IP utilizada, utiliza um protocolo HTTP próprio. Para realizar a integração, foi criado uma nova extensão que implementa a abstração para este dispositivo. A extensão consiste basicamente na implementação de duas classes: *IPCamConnection* e *IPCamGenericProtocol*.

A classe *IPCamGenericProtocol*, implementa a interface *MessageSerializer*, é responsável por converter os comandos *"ActionCommand"* e *"SetPropertyCommand"*, em requisições HTTP, seguindo as especificações da câmera. O protocolo da câmera foi obtida através de engenharia reversa e está disponível no website do autor[7].

5.5 Considerações Finais

Este capítulo apresentou uma avaliação experimental, que abordou vários aspectos e componentes da arquitetura proposta. Os testes de performance, permitiram analisar a sobrecarga no tempo de comunicação com os dispositivos, introduzido pelos componentes da arquitetura. Bons resultados foram obtidos, principalmente na comunicação USB, mesmo sem a arquitetura ter passado por nenhum processo de otimização.

Os experimentos conduzidos na seção 5.3, permitiram avaliar positivamente, a possibilidade de integração com plataformas 3D, que podem ser usadas para criar ambientes de simulação.

Os experimentos realizados no estudo de caso (seção 5.4), permitiram avaliar um amplo conjunto de componentes da plataforma. O estudo de caso, cenário 1, permitiu avaliar a plataforma como framework e a facilidade de implementação de projetos simples. O Cenário 2, permitiu avaliar as capacidades de extensibilidade do middleware, integração com vários dispositivos, utilização de várias tecnologias de comunicação e hardwares. A integração com a câmera IP, apresentou um real desafio, pois ela opera usando um protocolo próprio e é um dispositivo relativamente complexo, pois possui controle de movimentação, infra-vermelho, controle de brilho, saturação, tira fotos (snapshot), vídeo e etc. Porém a integração foi bem sucedida, de modo que a câmera e suas propriedades são totalmente acessíveis também da camada JavaScript / Web.

[7] Protocolo câmera IP Foscam: https://goo.gl/yDUjDI

Estes experimentos demonstraram o potencial e flexibilidade da arquitetura proposta.