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.

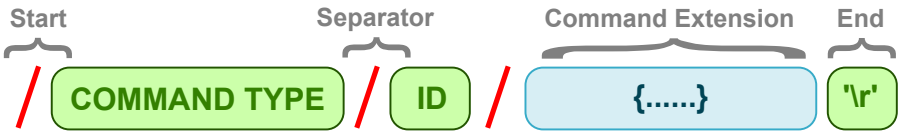

Figura 4.19 Formato do protocolo OpenDevice

- 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.

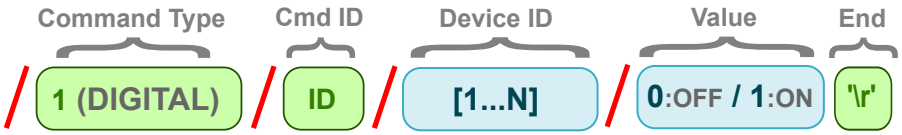

Figura 4.20 Comando: DIGITAL

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

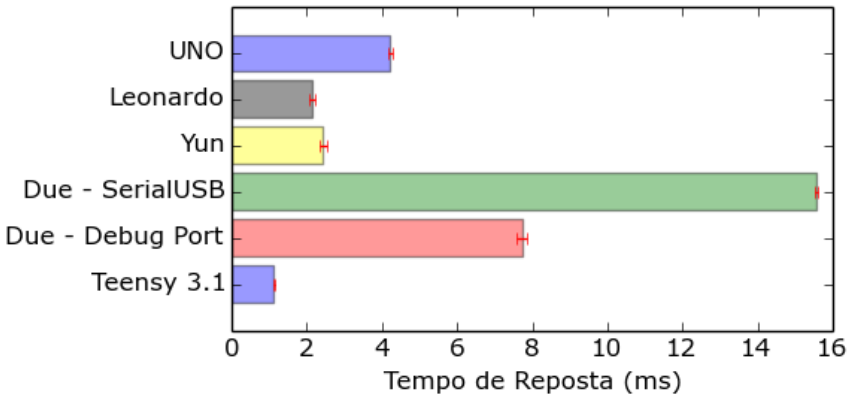

Figura 5.1 Teste de performance - USB

5.2.4 Resultados: Conexão Bluetooth

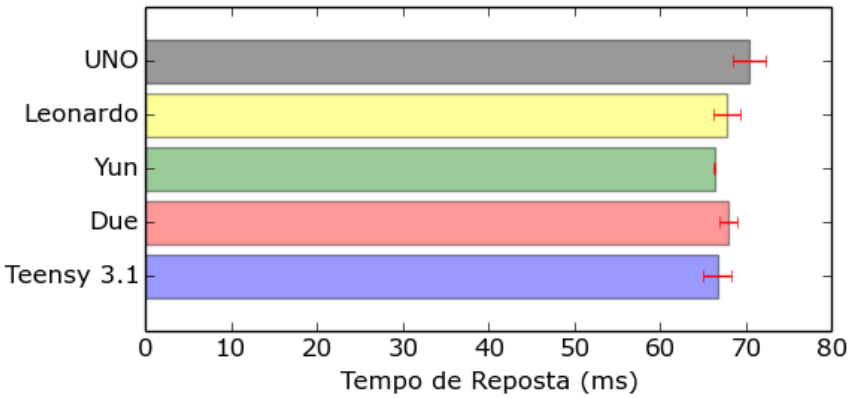

Figura 5.2 Teste de performance - Bluetooth

5.2.5 Resultados: Conexão Ethernet

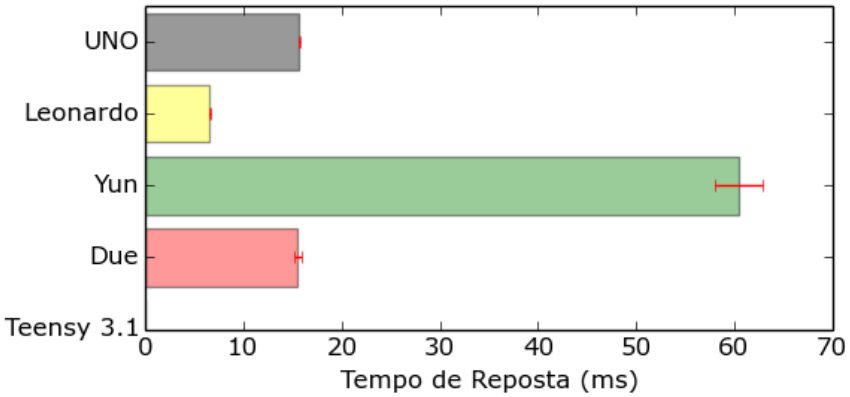

Figura 5.3 Teste de performance - Ethernet

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/*

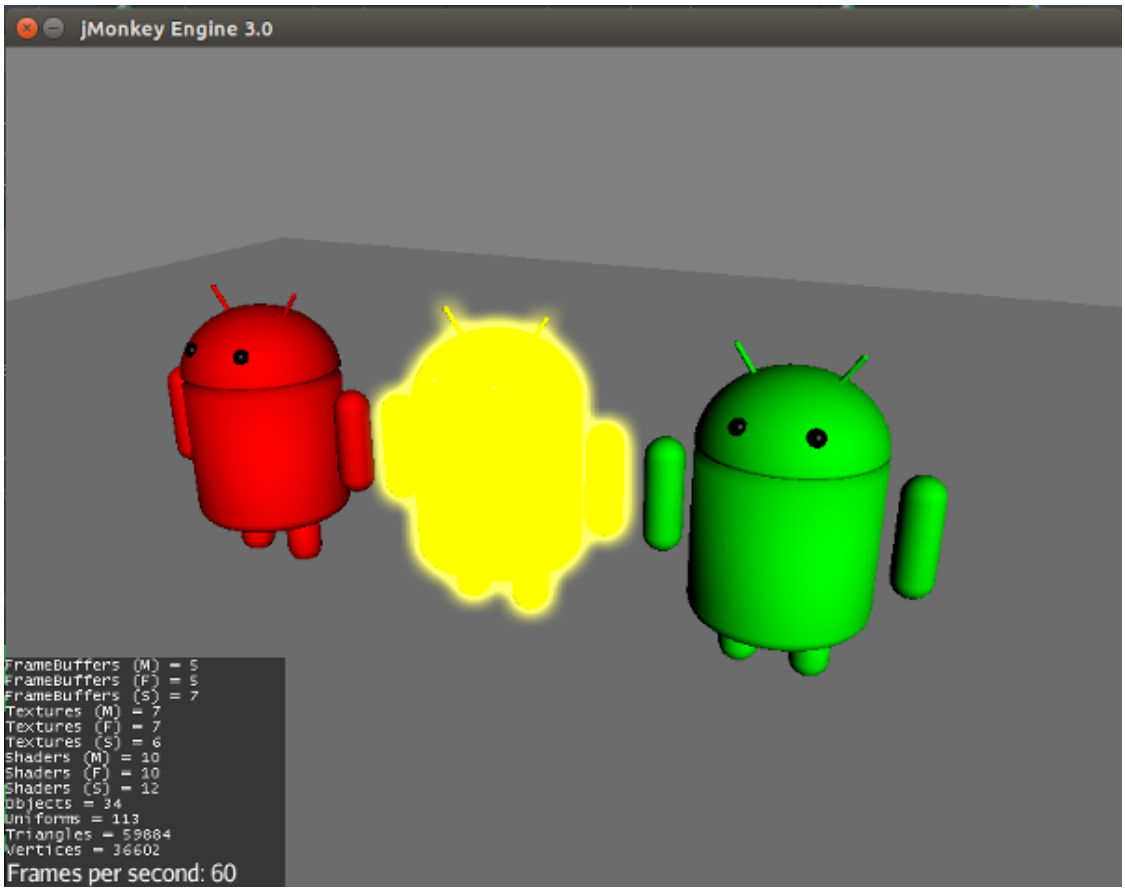

Figura 5.4 Integração com a JMonkeyEngine

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].

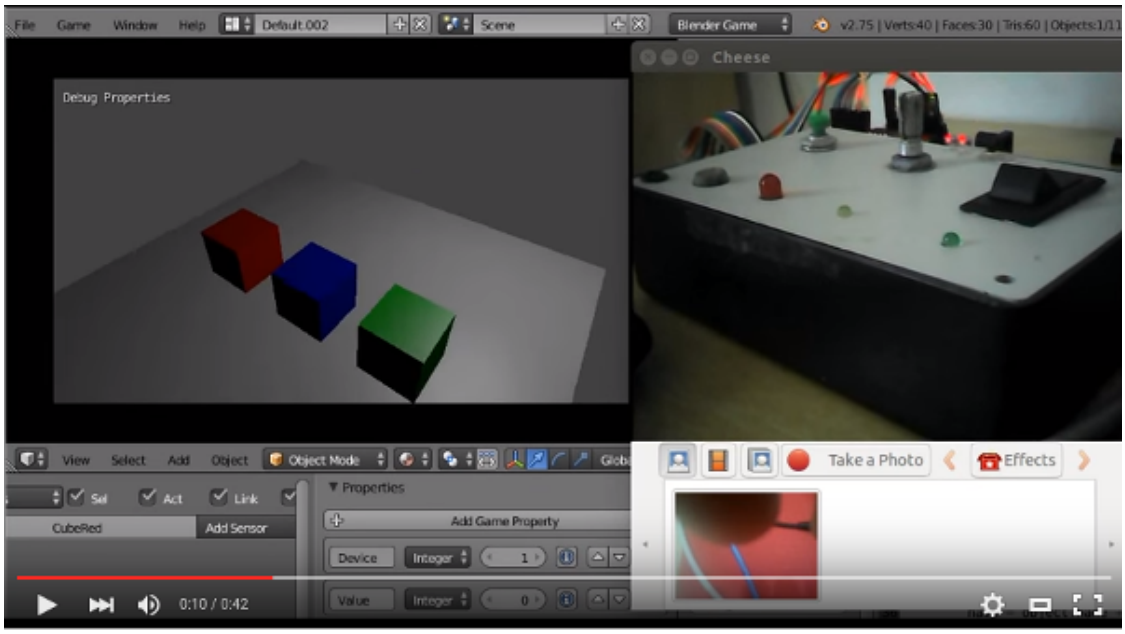

Figura 5.5 Integração com o Blender

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

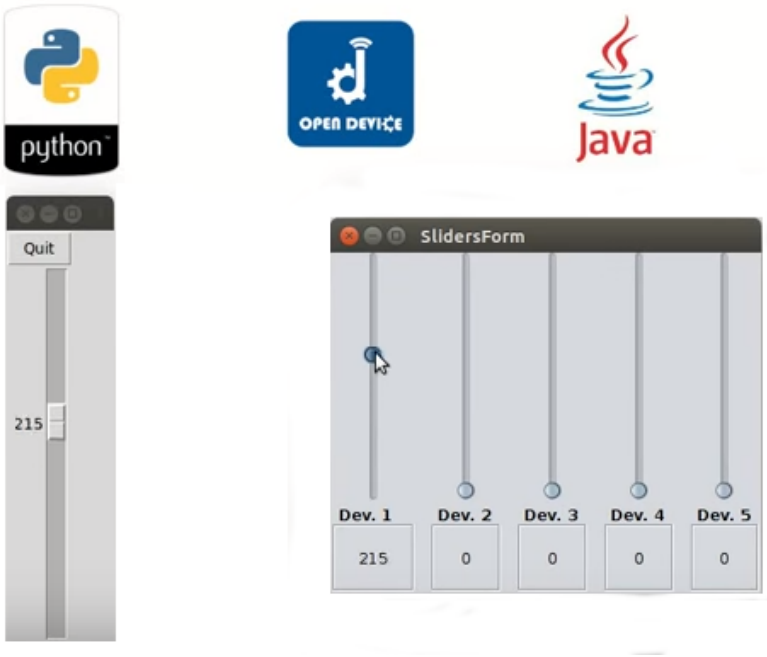

Figura 5.6 Teste de desempenho do cliente Python

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].

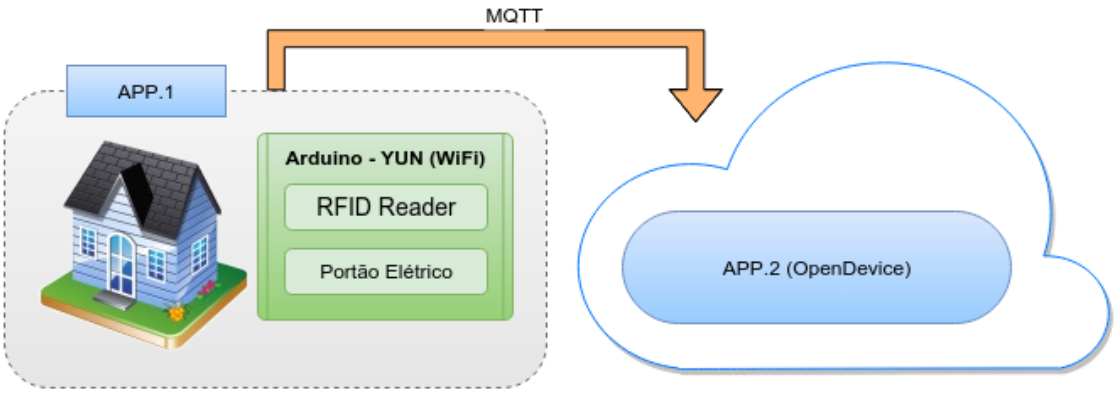

Figura 5.7 Diagrama - Cenário 1

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.

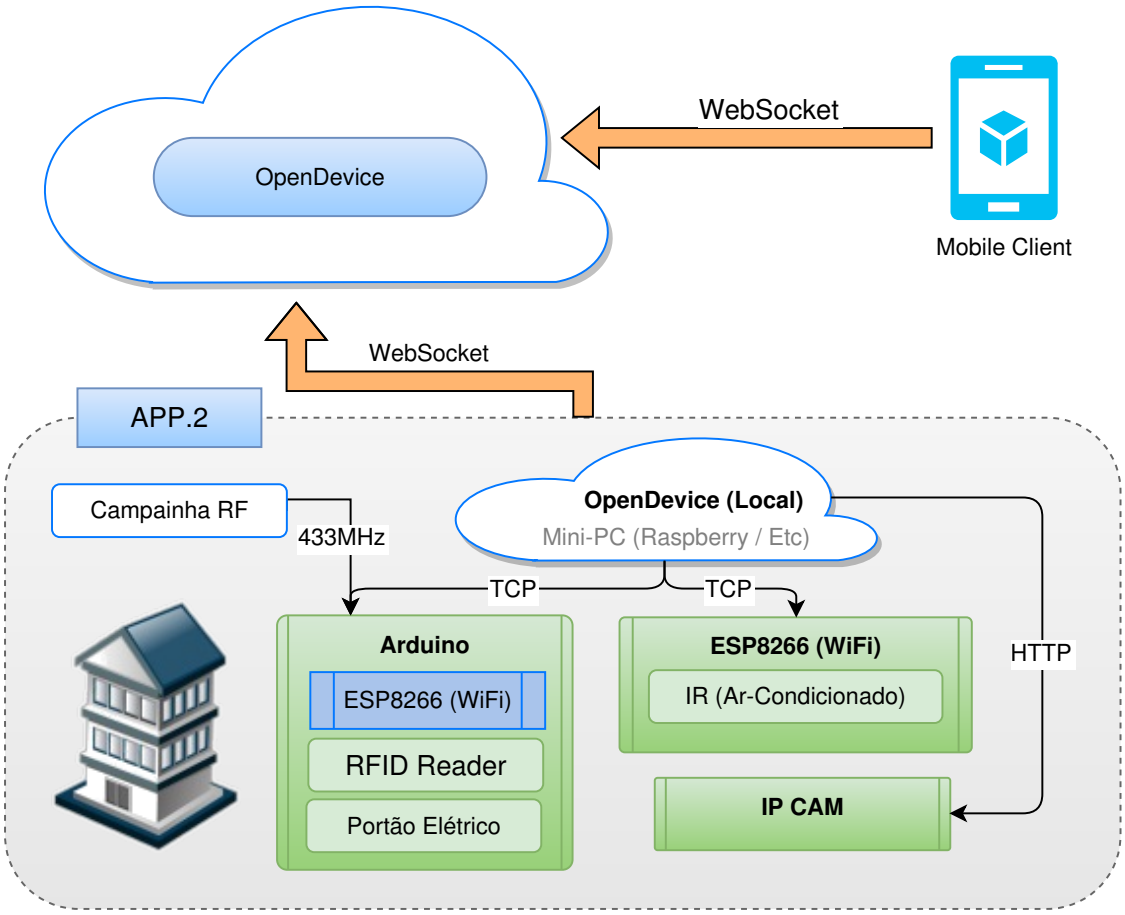

Figura 5.8 Diagrama - Cenário 2

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.