Universidade de Brasília - UnB

Engenharia Elétrica

Solução de controle de patrimônio via Bluetooth

Autor: Luiz Felipe de Oliveira Campos

Orientador: Daniel Chaves Café

Brasília, DF

2017

Luiz Felipe de Oliveira Campos

Solução de controle de patrimônio via Bluetooth

Monografia submetida ao curso de graduação em Engenharia Elétricada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Elétrica.

Universidade de Brasília - UnB

Orientador: Daniel Chaves Café

Brasília, DF

2017

Luiz Felipe de Oliveira Campos
Solução de controle de patrimônio via Bluetooth/ Luiz Felipe de Oliveira Campos. – Brasília, DF, 2017-
107 p. : il. (algumas color.) ; 30 cm.

Orientador: Daniel Chaves Café

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
, 2017.

1. BLE. 2. Beacon. I. Daniel Chaves Café. II. Universidade de Brasília. III. Faculdade de Tecnologia. IV. Solução de controle de patrimônio via Bluetooth

CDU 02:141:005.6

Dedico este trabalho a Deus, por ser a base e o guia de toda e qualquer decisão e caminhada que tenha me feito chegar até aqui, ao meu pai Sérgio e à minha mãe Carmem.

Agradecimentos

Agradeço primeiramente a Deus, pelo dom da vida e por prover todo o necessário para que este trabalho e graduação se concretizassem.

Agradeço aos meus pais, Sérgio e Carmem, pelo apoio total e incondicional durante toda minha vida e por todo o incentivo e exemplo à vida escolar e acadêmica.

Agradeço ao Professor Doutor Daniel Café pela orientação neste trabalho e pela sugestão da integração Empresa - Universidade como trabalho de conclusão de curso.

Agradeço ao Hermano Albuquerque, presidente da Innovix, ao Felipe Machado e Marcelo Araújo, colegas de trabalho, pela oportunidade de aprender e produzir na Innovix e pelos auxílios no desenvolvimento do trabalho.

Agradeço à minha irmã, Ana Campos, e aos amigos André Costa, Arthur Carvalho, Cristina Cavaletti, Felipe Sanches, Fernanda Kucharski, Gabriel Bayomi, Gabriel Castellano, Gustavo Cid, Henrique Orefice, Igor Barroso, Isabelle Novelli, João Rondina, José Kffuri, José Paiva, Júlia Horta, Leonardo Albuquerque, Leonardo Machado, Letícia Brito, Letícia Lima, Lorrainy Braz, Luisa Marini, Luiz Bruder, Pedro Campos, Rafael Ottoni, Raíssa Ruperto, Renata Lopes, Stefano Dantas, Vinícius Cardoso e Vinícius Pinheiro, pelo apoio e companheirismo principalmente nos anos dedicados a esta graduação.

Resumo

O objetivo desse trabalho é o desenvolvimento de uma solução de segurança e controle de patrimônio, como produto da empresa de tecnologia Innovix, de nome "IDView". Houve duas principais frentes de desenvolvimento: o desenvolvimento dos sensores que serão anexados aos patrimônios e o desenvolvimento de um gateway (concentrador) que irá receber informações desses sensores e envia-las para um servidor. Para os sensores, foram utilizados beacons, pequenos dispositivos que transmitem sinais *broadcast* de Bluetooth *Low Energy* (BLE). Foram analisados alguns beacons homologados como prova de conceito e, por fim, optou-se por utilizar o nRF51822 *Smart Beacon* programável da Nordic Semiconductor. O firmware do beacon foi desenvolvido todo na linguagem C de programação. Para o gateway, optou-se por utilizar sistemas com suporte a Linux embarcado. Foi desenvolvido um código em C e um em Python, que se comunicam via *socket* TCP, para fazer a leitura de sinais BLE, processamento das informações e comunicação com o servidor via protocolo HTTP. Foram escolhidas duas opções de plataforma: a Raspberry Pi 3 e a placa Colibri VF50 da empresa Toradex. Ambas foram programadas com o mesmo código, mas possuem diferenças em relação a capacidade de processamento, configurações de kernel do sistema operacional, dentre outros. Os objetivos e funcionalidades da solução foram alcançados e o IDView encontra-se em fase de análise e testes.

Palavras-chaves: monitoramento. patrimônio. ativos. bluetooth. BLE. beacon. nordic. gateway. embedded. linux. raspberry. toradex.

Abstract

The objective of this work is the development of an asset management solution, as a product of Innovix Company of technology, named "IDView". There were two main development fronts: the development of the sensors which will be attached to the assets and the development of a gateway that will receive information from the sensors and send them to a server. For the sensors, the choice was to utilize beacons, small devices that transmit broadcast signals of Bluetooth Low Energy (BLE). Some market beacons were analyzed as proof of concept and, finally, the programmable Smart Beacon from Nordic Semiconductor was chosen as the development plataform. The firmware of the beacon was developed entirely in C programming language. For the gateway, it was chosen to use systems with embedded Linux support. Codes in C and Python were developed, communicating with each other via socket to read BLE signals, process information and communicate with the server via HTTP protocol. Two platform options were chosen: the Raspberry Pi 3 and the Colibri VF50 board from Toradex. Both were programmed with the same code, yet they have differences in processing capacity, operating system kernel configurations, among others. The objectives and functionality of the solution have been achieved and IDView is undergoing analysis and testing.

Key-words: monitoring. patrimony. management. asset. bluetooth. BLE. beacon. nordic. gateway. embedded. linux. raspberry. toradex.

Sumário

Lista de abreviaturas e siglas

AD	Analógico-Digital
API	*Application programming interface*
BLE	Bluetooth *Low Eergy*
BR/EDR	*Basic Rate/Enhanced Data Rate*
DTC	*Device Tree Compile*
EXT	Externo
FAT	*File Allocation Table*
FFD	*Full Function Device*
GND	*Ground*
GPIO	*General Purpose Input/Output*
GPS	*Global Positioning System*
HCI	*Host Controller Interface*
HTTP	*HyperText Transfer Protocol*
I2C	*Inter-Integrated Circuit*
IEEE	*Institute of Electrical and Electronics Engineers*
IO	*Input/Output*
IOT	*Internet of Things*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
LED	*Light-emitting diode*
MAC	*Media Access Control*
MQTT	*Message Queuing Telemetry Transport*
PC	*Personal Computer*

PCB	*Printed Circuit Board*
RFD	*Reduced Function Device*
RFID	*Radio Frequency Identification*
RSSI	*Received Signal Strength Indicator*
RTC	*Real Time Clock*
RTOS	*Real-Time Operating System*
SD	*Secure Digital*
SDK	*Software Development Kit*
SIG	*Special Interest Group*
SO	Sistema Operacional
SPI	*Serial Peripheral Interface Bus*
SWDCLK	*Serial Wire Debug Clock*
SWDIO	*Serial Wire Debug Input/Output*
SWO	*Single Wire Output*
TCP	*Transport Control Protocol*
TI	Tecnologia e Informação
TLM	*Telemetry*
UART	*Universal Asynchronous Receiver-Transmitter*
UID	*Unique Identifier*
URL	*Uniform Resource Locator*
USB	*Universal Serial Bus*
UUID	*Universally Unique Identifier*
VCC	*Integrated Circuit Power-Supply*

Lista de ilustrações

Lista de tabelas

1 Introdução e Estudo de Viabilidade

Muitas empresas, faculdades, lojas, hotéis e outros estabelecimentos, investem bastante em equipamentos e dispositivos de alto custo com o objetivo de melhorar sua infraestrutura, produtividade e desempenho na área em que atuam. Infelizmente, o que sempre foi uma grande preocupação de quem faz esse tipo de investimento é o furto desses equipamentos, que pode acarretar em grande prejuízo.

Há, então, uma crescente procura no mercado por soluções de segurança que inibam o furto ou de alguma forma alertem o estabelecimento de que um furto está acontecendo, de forma a serem tomadas medidas adequadas que minimizem ou até impeçam o prejuízo patrimonial. Uma solução para tal problema pode ser dividida em duas partes: uma responsável por detectar o furto e transmitir essa informação; e outra parte responsável por receber essa informação e disponibilizá-la para um usuário por meio de uma interface gráfica de monitoramento juntamente com um alarme local.

O objetivo principal desse trabalho é desenvolver uma solução de baixo custo que seja eficiente e que atenda às demandas do mercado por produtos na área de controle de patrimônio.

O tema proposto é uma iniciativa da Innovix (Logomarca na Figura 1), uma empresa especializada em utilizar tecnologias presentes no mercado e também em desenvolver suas próprias soluções para as áreas de "Controle de Acesso e Segurança" e "Gestão e Controle de Patrimônio". O projeto tem o nome de "IDView", e foi desenvolvido aliando um estágio de Engenharia Elétrica iniciado em janeiro de 2017 na empresa Innovix com o Trabalho de Conclusão de Curso de Engenharia Elétrica na Universidade de Brasília. Todo o equipamento e ferramentas usadas no desenvolvimento são propriedades da Innovix.

Figura 1 – Logomarca da Empresa. Imagem cedida pela Innovix.

A Innovix é uma empresa com o objetivo de desenvolver soluções de tecnologia nas áreas de "Gestão e Controle de Patrimônio" e "Controle de Acesso e Segurança".

Na área de "Gestão e Controle de Patrimônio" a Innovix desenvolve e oferece soluções completas com o uso de tecnologia RFID (etiquetas eletrônicas com identificação por rádio-frequência), já tendo realizado projetos para clientes como Universidade Católica

de Brasília (DF), Unileste – Universidade do Leste de Minas (MG), Total Express (SP), Walmart (SP), Hospital Albert Einstein (SP), entre outros.

Na área de "Controle de Acesso e Segurança" a Innovix oferece soluções para movimentação e controle de pessoas e veículos utilizando autenticação por RFID, cartão de proximidade e biometria. Além disso, as soluções podem ser integradas a sistemas avançados de processamento de vídeo, incluindo vídeo-analítico para movimentação de objetos, reconhecimento de padrões e reconhecimento de eventos. Nesta área, os principais clientes da empresa incluem o Grupo Telefônica (SP), CNI – Confederação Nacional da Indústria (DF), Sicoob / Bancoob (DF), Brasília Shopping (DF), Hospital de Barretos (SP), etc.

Além de utilizar tecnologias já presentes no mercado, a Innovix também desenvolve soluções próprias que integram *hardware* e *software*, com foco na evolução da chamada "IOT" ou "Internet das Coisas". A Innovix tem experiência com o desenvolvimento de circuitos eletrônicos, já tendo desenvolvido produtos que são utilizados por clientes em todo o Brasil. Alguns produtos desenvolvidos pela Innovix incluem:

- Gateway (concentrador) inteligente integrado a sensores eletrônicos;
- Sensor de consumo de energia de baixo custo;
- Sensor de controle de patrimônio;
- Sensor de temperatura e umidade;
- Sensor de corrente alternada.

Todo o desenvolvimento dos circuitos eletrônicos é realizado pela equipe de engenharia na sede da empresa, em Brasília. Para o desenvolvimento de tais circuitos, a Innovix conta com estrutura específica de desenvolvimento e equipamentos de laboratório, incluindo osciloscópio, analisador lógico, e outros equipamentos de bancada.

Adicionalmente, a empresa também desenvolve *software* próprio integrado às suas soluções de *hardware*. As soluções de *software* envolvem aplicativos específicos de comunicação e tratamento de dados primários ("middleware"), aplicativos de tratamento das regras de negócio, interface de usuário ("front-end") e gestão de banco de dados. A empresa oferece soluções de *software* que operam localmente ou remotamente, por meio do uso de infra-estrutura de TI na Amazon e Google.

A Innovix oferece soluções preferencialmente no modelo de "serviços". Neste modelo, a empresa instala suas soluções de *hardware* e mantém um contrato mensal com o cliente para utilização de seu *software* aplicativo

As possibilidades de tecnologias que podem ser utilizadas como solução para o problema de controle e monitoramento são vastas. De soluções já existentes à soluções que poderiam ser implementadas, existe um imenso ramo de opções a serem analisadas. O crítico para uma tomada de decisão sobre a tecnologia a ser utilizada é trazer o melhor custo benefício para a empresa que fornece e para a que adquire a solução.

Câmeras de segurança não estarão presentes na discussão, pois permitem apenas um monitoramento remoto visual no qual há a necessidade constante de acompanhamento humano para um eventual alarme no momento exato do furto, que não é o objetivo da solução a ser proposta. *Tags* magnéticas, geralmente encontradas em roupas de lojas de departamentos com detectores em suas entradas e saídas, também não estão entre as soluções mais procuradas, pois essas *tags* podem ser removidas sem muito esforço e sem acionar nenhum tipo de alarme.

Procura-se por uma solução o mais automatizada possível, na qual possa haver um acompanhamento humano mas que não dependa disso. Faz-se necessária a subdivisão da definição da solução em duas principais escolhas: Monitoramento de Estado e Transferência de Informação.

1.1 Monitoramento de Estado

Monitoramento de estado engloba a tecnologia que será utilizada para detectar algum tipo de movimento ou deslocamento dos objetos a serem monitorados como, por exemplo, se um objeto foi retirado do lugar. As principais opções são os sensores de proximidade, presença e movimento. Nesse conjunto de sensores, estão os sensores infravermelho, sensores micro-ondas, sensores ultrassom, sensores capacitivos e indutivos, sensores magnéticos, dentre outros.

Os sensores infravermelho captam variações térmicas e podem ser ajustados de acordo com a temperatura humana (35º a 38º graus Celsius). Dessa forma, no momento que uma pessoa entra em um ambiente que possui esse tipo de sensor, há uma mudança na radiação infravermelha da região e o sistema entra em alarme. Esse tipo de sensor é frequentemente utilizado em soluções de iluminação automatizada de ambientes e também para alarmes de segurança. Para essas aplicações, esses sensores possuem um excelente desempenho e, em geral, não demandam um custo tão alto por já ser uma tecnologia amplamente difundida. Porém, para o monitoramento de um objeto que esteja localizado num ambiente movimentado, como uma sala de aula ou uma exposição, poderia ocorrer muitos alarmes falsos devido ao movimento constante de pessoas nas proximidades do objeto. (CHILTON, 2017)

Sensores de ultrassom estão geralmente voltados para aplicações de proximidade. Eles são equipados com um emissor e um receptor de ondas. Emitem pulsos ultrassônicos inaudíveis ao ouvido humano e detectam o eco proveniente da colisão dessas ondas com alguma superfície. Com a informação do tempo que o eco levou para ser detectado e a velocidade do som no ar, é possível calcular a distância que o sensor está da superfície com a qual as ondas colidiram, que para a aplicação em análise seria o objeto de monitoramento. (KINNEY, 2001)

Existem também os sensores capacitivos e indutivos. Os sensores capacitivos utilizam o objeto o qual se quer aferir a distância como segundo polo de um capacitor e o ar entre o sensor e o objeto como dielétrico. Quando a distância entre o objeto e o sensor varia, a capacitância também varia, e dessa forma é detectado movimento do objeto. Os sensores indutivos, por sua vez, utilizam campos magnéticos para detectar objetos. Quando um objeto entra no alcance do campo, a corrente da bobina no circuito do sensor é alterada e dessa forma o movimento é detectado. Ambos sensores são soluções muito interessantes, e são implantados em sistemas de automação de muitas fábricas e ambientes industriais. No entanto seu preço chega a ser cinco vezes mais caro que o dos outros sensores já citados, o que influenciaria determinantemente no preço final da solução. (KINNEY, 2001)

Sensores magnéticos, conhecidos como *Hall effect sensors*, possuem um funcionamento simples, segundo o qual uma tensão de saída é induzida por um campo magnético externo. Dessa forma, um imã seria fixado ao objeto e posicionado próximo ao sensor, de forma que ao ser retirado do local a tensão deixaria de ser induzida no sensor e o movimento seria detectado. Porém, bastaria aproximar um imã ao sensor e retirar o objeto que o furto não seria percebido pelo sistema.

A solução com *RFID* (*Radio-frequency identification*) também chegou a ser considerada. É uma tecnologia bastante utilizada em outras soluções desenvolvidas pela Innovix. Seu funcionamento consiste em transmitir informação no chamado leitor *RFID* que transmite energia ao chip *RFID* quando posicionado próximo o suficiente e, quando energizado, o chip transmite informação armazenada para o leitor. O lado negativo dessa solução, assim como sensores capacitivos e indutivos, é o preço ao ser comparado com os outros sensores, sendo o *RFID* ainda mais caro que os outros dois citados. O objetivo é conciliar uma solução eficiente com um custo que consiga chegar a um preço acessível no mercado e que ainda assim traga retornos para a empresa desenvolvedora. (ARMSTRONG, 2011)

Por fim, optou-se por uma solução simples, de baixo custo, e relativamente menos elaborada que algumas das já citadas: botão de contato (*push button*). Exite um infinidade de opções de botões, todos de fácil integração com qualquer interface *GPIO*, de diferentes tamanhos e diferentes configurações (normalmente aberto, normalmente fechado, etc). A

solução contaria então com um dispositivo equipado com um botão de contato, que ficaria fixado ao objeto de modo que, quando o objeto perdesse o contato com o botão, o circuito ficaria aberto (ou fechado, dependendo da configuração) e isso seria interpretado pelo sistema como um estado de alarme.

1.2 Transmissão de Informação

Uma vez tendo em mãos a informação do estado de um objeto a ser monitorado a partir de um sensor, deve-se escolher uma opção de transmitir essa informação. Como, na maioria das vezes, haverá mais de um objeto a ser monitorado por ambiente, esses sensores terão que transmitir essa informação para um concentrador (*gateway*) que ficará encarregado de disponibilizar o *status* dos sensores desse ambiente para um servidor e a partir daí o usuário terá acesso a essa informação por meio de uma interface gráfica.

As opções que se têm para esse envio de informação do sensor para o gateway podem ser divididas em dois grandes grupos: Soluções com fio (*Wired*) e soluções sem fio (*Wireless*).

Soluções com fio costumam ter menores taxas de erro devido à quantidade significativamente menor de interferência externa comparada às soluções *Wireless*. Os protocolos mais simples de comunicação via cabo são os protocolos de comunicação serial. Na comunicação serial, a porta serial faz o envio e recepção de bits individuais, um após o outro, que são montados em bytes de informação. Dessa forma, a comunicação serial acaba sendo um pouco mais lenta que outros tipos de comunicação, como a comunicação paralela, que permite o envio de bytes inteiros por vez (INSTRUMENTS, 2015). Mesmo sendo mais lenta, a comunicação serial é mais simples e de certa forma mais versátil que outros tipos de comunicação. Dos protocolos de comunicação serial, pode-se citar três como os mais comuns: SPI, UART e I2C.

A Innovix já possui um produto de controle de patrimônio que utiliza sensores que se comunicam com o concentrador por meio de comunicação serial do tipo UART, chamado Segurix. Esse produto já é comercializado e tem uma eficiência muito boa. Porém, existem alguns inconvenientes em relação à solução cabeada. Em alguns casos, existem dificuldades de instalação devido a necessidade de passar fios pelo ambiente e, na maioria das vezes, essa solução acaba sendo muito invasiva à estética do estabelecimento. Outra desvantagem da solução cabeada é o custo. A quantidade de fio a ser utilizada para fazer o controle de patrimônio de diversos itens em algumas salas de um dado estabelecimento faz com que o custo dessa solução supere algumas das soluções *Wireless* consideradas.

As tecnologias *Wireless* vêm ganhando cada vez mais espaço no mercado. Elas

tendem a ser mais lentas que as comunicações cabeadas e os dados recebidos devem ser cuidadosamente validados na procura de se proteger contra possíveis perdas de dados. As tecnologias de comunicações *Wireless* que foram estudadas para possível implementação nesse produto são: WiFi (*IEEE* 802.11), Zigbee (*IEEE* 802.15.4) e Bluetooth (*IEEE* 802.15.1).

O WiFi é uma tecnologia para a interconexão de dispositivos com redes locais. Opera mais usualmente na faixa de frequência de 2,4GHz e pode também operar na faixa de 5GHz, dependendo da aplicação. Na configuração mais comum (802.11b) realiza a transferência de dados em até 11Mbps mas existem configurações, como a 802.11n, que podem chegar à uma taxa de transmissão de 600Mbps, com um alcance médio de 30 metros sem obstrução. Hoje em dia é muito conhecida por dar suporte a conexões com a Internet. Uma desvantagem dos módulos WiFi é que eles consomem muita energia, de forma que haveria necessidade de conectar os sensores com cabos de alimentação à rede de energia. Assim, uma solução para vários objetos em um ambiente seria invasiva mesmo com uma tecnologia *Wireless* para transferência de dados. Procura-se uma solução de baixo consumo que permita a utilização de bateria e que essas possam ter uma vida útil considerável. (POOLE,)

Dentre as tecnologias de baixo consumo, temos o Zigbee. Ele é utilizado para criar predominantemente redes locais pessoais para dispositivos de baixa potência, em geral voltados para a automação em setores industriais. Suas frequências de operação variam de 868MHz na Europa e Japão, 915MHz nos Estados Unidos e 2,4GHz no resto do mundo. Seu protocolo define dois tipos de dispositivos nós em uma rede Zigbee: FFD (*Full Funcion Device*) e RFD (*Reduced Function Device*). RFDs, mais baratos e com menos memória, se comunicam apenas com FFDs; e FFDs se comunicam com todos os nós da rede. É uma tecnologia ainda não muito popular e, por ainda estar ganhando o mercado, possui um preço ainda acima das outras opções *Wireless*. (NENOKI, 2013)

Outra opção de baixo consumo é o Bluetooth *Low Energy* (BLE). O Bluetooth deriva de uma tecnologia de comunicação de curta distância entre telefones, desenvolvida pela antiga empresa *Ericson* e que mais tarde avançou para um padrão a ser utilizado por uma variedade de dispositivos. Também opera na faixa de 2,4GHz. Sua topologia utiliza o esquema mestre-escravo no qual um mestre pode estar conectado (pareado) com até sete dispositivos no modo *full-duplex* (recepção e envio de dados no mesmo canal e ao mesmo instante), formando uma rede chamada de *Piconet*. Para haver a comunicação entre dispositivos Bluetooth, inicialmente existe o envio público de pacotes de anúncio, que são pacotes utilizados para fazer a requisição de pareamento. Um dispositivo Bluetooth que consegue receber e interpretar esses pacotes de *advertise*, envia uma resposta para o dispositivo de origem e então ocorre o pareamento. Após isso, é criado um *link* sincronizado entre os dois dispositivos (mestre e escravo) e é feita uma reserva de *slots* para cada um

deles (SIG, a). O Bluetooth *Low Energy* foi desenvolvido para prover uma comunicação similar ao Bluetooth convencional porém com um consumo de energia significativamente reduzido. Seu funcionamento e protocolo serão melhor detalhados no próximo capítulo.

Por ser uma opção de custo bastante acessível, pelo baixo consumo de energia que permite o uso conveniente de baterias, e por questões de padronização (tecnologia popular) e possível futura integração com outros produtos a serem desenvolvidos pela empresa, optou-se por utilizar a tecnologia Bluetooth para o envio de informação dos sensores.

Assim, a solução a ser desenvolvida para o dispositivo não invasivo de controle de patrimônio compreende um aparelho alimentado por bateria e equipado de um botão de contato cujo estado será enviado à um gateway via Bluetooth. O gateway, por sua vez, se comunicará com uma API (*Application programming interface*) que fará parte do processamento dos dados e dos alarmes e irá disponibilizar essas informações aos usuários do produto atraves de um *web service* com interface gráfica. A Figura 2 mostra um diagrama explicativo da solução completa do IDView, sendo de competência desse trabalho o desenvolvimento de uma solução para os sensores e para o gateway.

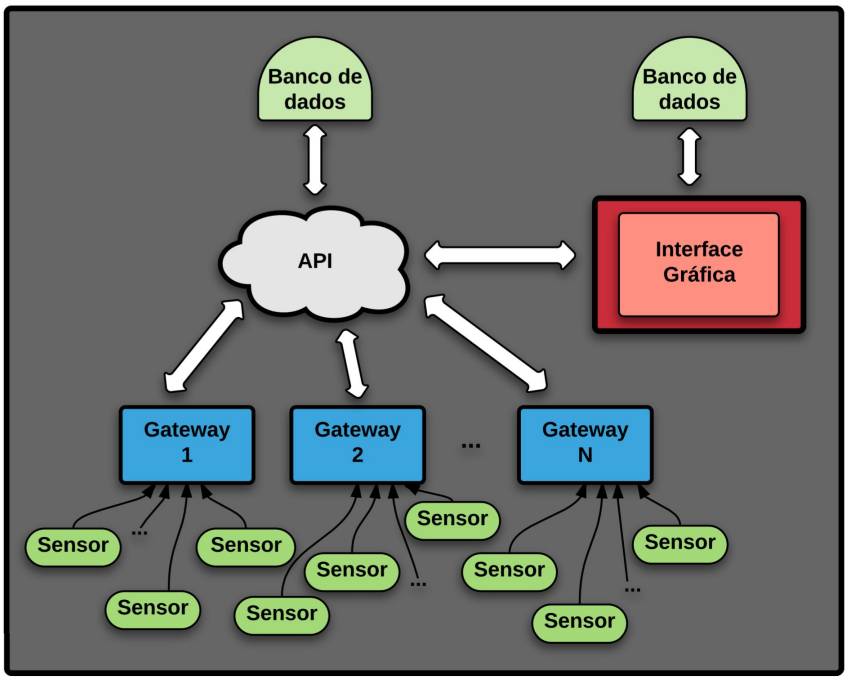

Figura 2 – Diagrama ilustrativo da solução proposta com o IDView. Fonte: Autor

1.3 Objetivos Específicos

Tendo em vista o objetivo geral proposto, os objetivos específicos aqui abordados são:

- Estudo da tecnologia BLE por meio de análise de provas de conceito;
- Utilizar uma plataforma de beacon programável para desenvolver um *firmware* com as funcionalidades esperadas do produto.
- Realizar testes e homologação da solução do beacon desenvolvido utilizando solução de receptores BLE de mercado;
- Estudar e definir, dentre as opções de desenvolvimento de um gateway, qual é a mais interessante para o produto, levando em conta tempo e custo de desenvolvimento;
- Desenvolver o firmware do gateway na plataforma escolhida, realizando a integração com uma API em um servidor local;
- Utilizar soluções de beacons de mercado para testar e homologar o gateway desenvolvido, e pra auxílio em seu desenvolvimento.

2 Tecnologias e Metodologia

2.1 Bluetooth

Figura 3 – Logomarca oficial do Bluetooth. Fonte: (SIG, 2011)

A tecnologia Bluetooh (Logomarca oficial na Figura 3) é amplamente conhecida e utilizada nos tempos atuais. As redes Bluetooth transmitem seus dados por ondas de rádio de baixa potência com alcance de até 30 metros a livre linha de visada, resiliente a espúrios de ruído no espectro por saltos curtos em torno da frequência de transmissão e na faixa entre 2.402GHz e 2.480GHz, que faz parte da banda reservada internacionalmente para ondas de radiofrequência eletromagnéticas não licenciadas (Banda *ISM*, do inglês *Industrial, Scientific and Medical*) (BOURQUE, 2014). As outras Tecnologias wireless abordadas no fim do Capitulo 1 (WiFi e Zigbee) também operam na banda *ISM*. Na verdade, essa faixa de frequência é utilizada por diversas tecnologias, então há sempre a necessidade de se evitar possíveis interferências. O Bluetooth tenta evitar essas interferências trabalhando com um baixa potência de transmissão de sinal. Enquanto celulares potentes chegam a transmitir seu sinal a cerca de 3 watts, o sinal de um dispositivo Bluetooth é de cerca de 1 miliwatt (FRANKLIN; LAYTON,).

Como foi dito no Capitulo 1, o Bluetooth funciona a partir de uma topologia mestre-escravo na qual 8 dispositivos podem estar conectados ao mesmo tempo, formando uma rede chamada de *Piconet*. Na prática, isso significa que um PC equipado com a tecnologia Bluetooth poderia se comunicar até 7 periféricos Bluetooth (mouse, teclado, impressora, fone de ouvido...) Além das redes *Piconet*, existem também as redes *Scatternet*, para aplicações mais específicas onde há a necessidade de envio de dados entre mais de 8 dispositivos. As redes *Scatternet* são nada mais que interconexões entre redes *Piconet*, na qual um membro da rede *Piconet* seria escolhido para participar como escravo em uma *Piconet* secundária, de maneira multiplexada temporalmente (SIG, a). Dessa forma, mais de 8 dispositivos estariam conectados na mesma rede Bluetooth. A Figura 4 exemplifica a topologia usual de uma rede Bluetooth.

Para haver a conexão entre dispositivos Bluetooth, os “mestres” devem ter a configuração de Bluetooth central e os “escravos” a configuração de Bluetooth periférico. A conexão entre os dispositivos ocorre inicialmente por meio de envio de pacotes *broadcast*

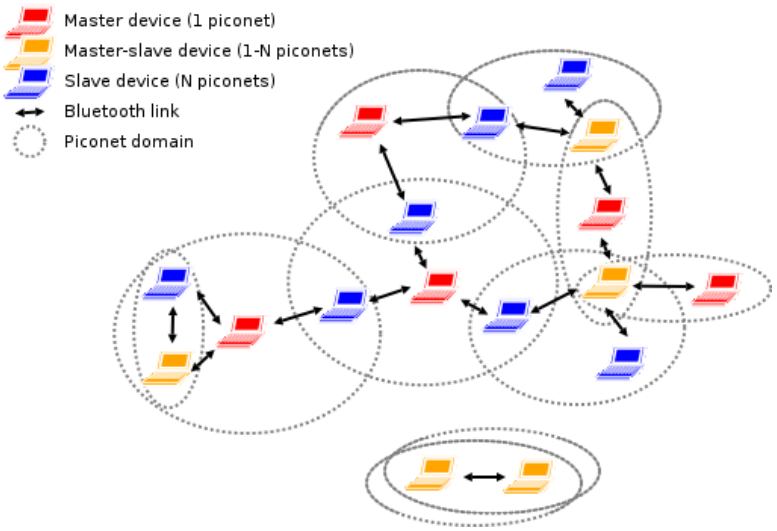

Figura 4 – Topologia de Pinconets e Scaternets Bluetooth. Fonte: (BLANCO, 2007)

de anúncios, conhecidos pelo termo consagrado de *advertise*, e as definições de central e periférico definem as habilidades do dispositivos de enviarem (periférico) e de receberem esse pacote (central).

Para iniciar um pareamento, os dispositivos periféricos fazem o envio desses pacotes de *advertise*, que são sinais com mínimas informações para alertar dispositivos vizinhos sobre sua presença e suas capacidades. Os dispositivos centrais realizam a tarefa de leitura de sinais. É um modo de recepção no qual sinais de *advertise* no ambiente são escutados e analisados. No caso, por exemplo, do pareamento entre celulares, é possível ver a lista de dispositivos Bluetooth periféricos no alcance do celular central e selecionar algum para tomar uma certa ação. No pareamento automatizado, onde não existe a necessidade da seleção por um usuário, o dispositivo central realiza a análise do pacote de *advertise* recebido, muitas vezes comparando com seções padrões de pacotes de dispositivos a serem pareados. Se o pacote recebido condizer com os parâmetros de comparação, o pareamento ocorre automaticamente.

2.1.1 BR/EDR, BLE e *Smart Ready*

O que foi apresentado até agora são conceitos gerais para as especificações do Bluetooth. Dentro dessa tecnologia, porém, existem duas grandes vertentes: O BR/EDR (*Bluetooth Basic Rate/Enhanced Data Rate*) e o BLE (*Bluetooth Low Energy*).

- BR/EDR (*Bluetooth Basic Rate/Enhanced Data Rate*)

  O BR/EDR é a forma mais tradicional e popular do Bluetooth. Ela possibilita a comunicação sem fio contínua ponto a ponto. Ao se pensar em Bluetooth, muitas vezes vêm a mente aparelhos de som automotivos, caixas amplificadoras e etc. Isso

se deve ao fato de o Bluetooth BR/EDR ter como aplicação otimizada o *streaming* de áudio (SIG, b) . Utiliza 79 canais na banda Bluetooth, indexados de 0 a 78, e transfere dados a uma taxa de mais de 2.0 Mbps. O BR/EDR em geral lida com uma grande quantidade de dados e por isso possui um consumo considerável de energia. É uma tecnologia madura, utilizada muito mais em integrações de produtos do que no desenvolvimento em si (REN, 2015).

- Bluetooth *Low Energy*.

  O Bluetooth *Low Energy*, BLE ou Bluetooth *Smart*, existe a partir da versão 4.0 do Bluetooth, introduzida em 2010. De acordo com Szymo Janiak (JANIAK, 2015), o Bluetooth *SIG* (*Special Interest Group*) prevê que em 2018 mais de 90% dos dispositivos habilitados para Bluetooth terão suporte para o BLE. O SIG é a entidade que supervisiona o desenvolvimento e licenciamento de padrões Bluetooth para fabricantes. Hoje em dia já é suportado nativamente por celulares com os sistemas iOS, Android, Windows Phone e BlackBerry, e computadores com os sistemas macOS, Linux, Windows 8 e Windows 10. O BLE é voltado para aplicações de menor taxa e quantidade de dados. Em geral, é utilizado em sensores de ambiente (temperatura, umidade, etc), de automação e de proximidade. Essa tecnologia utiliza 40 canais (quase metade da quantidade de canais do BR/EDR) indexados de 0 a 39, sendo que os canais 37, 38 e 39 são reservados apenas para o serviço de *advertising* já discutido. O envio de dados ocorre a uma taxa de cerca de 1 Mbps (SIG, b).

  O principal diferencial do BLE, como é explicitado pelo seu nome, é o consumo de energia. A tecnologia do Bluetooth original foi reestruturada em *hardware* e *software* para obter o menor consumo possível. Isso, aliado a aplicações que exigem menos fluxo de dados, faz com que dispositivos que utilizam a tecnologia BLE possam ser alimentados por baterias moedas e ter pleno funcionamento por vários meses e até anos. Uma aplicação com BLE não chega a consumir mais de 20mA de corrente de pico (SEMICONDUCTOR, 2010).

  Diferente do BR/EDR, para muitas de suas aplicações, a tecnologia BLE não tem a dependência do pareamento. Com ela, é possível o dispositivo estar configurado como central e periférico ao mesmo tempo. Se a aplicação for simples o suficiente nesse quesito, é possível tirar proveito dessa dupla configuração para usar apenas os pacotes de *advertise* para transmitir e receber dados de dispositivos vizinhos.

Além dessas duas vertentes principais, existe ainda a Tecnologia Bluetooth *Smart Ready*. Dispositivos equipados com essa tecnologia têm suporte tanto ao BLE quanto ao BR/EDR, ou seja, funcionam com os dois modos e conseguem trocar informação com dispositivos de ambos os modos (TORVMARK, 2014). Um diagrama explicativo encontra-se na Figura 5. São chamados de dispositivos "*dual-mode*", enquanto que os dispositivos

que funcionam somente com o BLE ou com o BR/EDR são chamados de "*single-mode*". Por abrangerem ambas as tecnologias, os chips *dual-mode* são consequentemente mais caros que os *single-mode*.

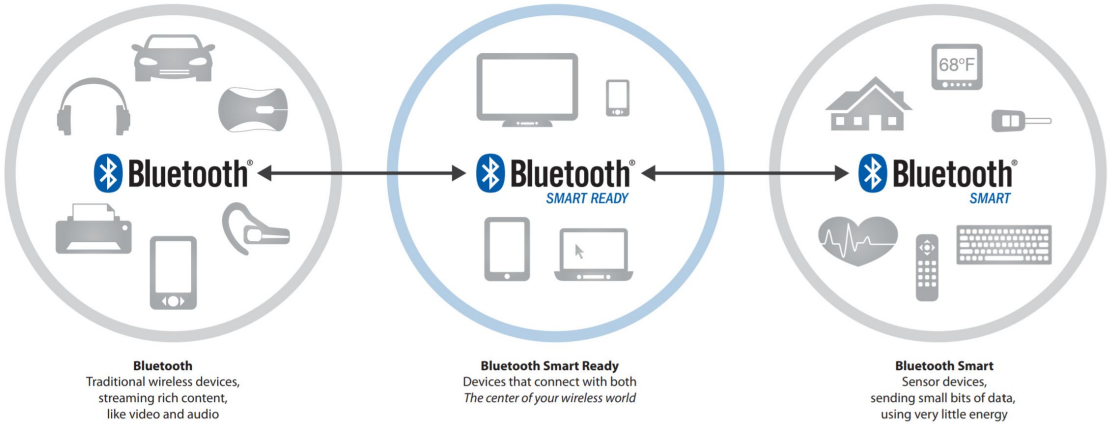

Figura 5 – Diagrama Bluetooth *Smart Ready*. Fonte: (TORVMARK, 2014)

Por questões de consumo energético foi escolhida a tecnologia BLE para se trabalhar com o desenvolvimento do sensor de monitoramento de ativos proposto.

2.2 Beacons

A principal aplicação hoje do BLE é no conceito de beacons. Os beacons são pequenos dispositivos transmissores de pacotes de Bluetooth *Low Energy* à uma certa periodicidade e com uma infinidade de aplicações. A palavra beacon vem do inglês e significa farol, sinal luminoso. Ao invés de utilizar sinal de luz visível, os beacons de Bluetooth enviam sua informação a partir dessa tecnologia *Wireless* e pode ser "enxergado" por outros dispositivos habilitados com a mesma tecnologia (KONTAKTIO, b). Em questão de consumo energético, os beacons chegam a durar até quatro anos com a mesma bateria dependendo de certas definições de *hardware* e *software*. A Figura 6 mostra um beacon monitorado por um *smartphone*.

A utilização de beacons é bastante variada, mas em sua maioria é voltada para tecnologia de proximidade. São usados para localização de objetos e até de animais em uma determinada área; para o conceito de navegação *indoor*, dando a possibilidade de um ambiente similar a um GPS *indoor*; para a área de interação, como por exemplo, aparelhos eletrônicos ligarem com a chegada de um portador de beacon no ambiente, ou registro de entrada e saída de estabelecimentos; para o intuito de segurança, fazendo *broadcast* de notificações de problemas com segurança de diversos tipos; para análise de informação, enviando o pacote de mensagem alterada pelo meio externo para ser processada e interpretada.

Figura 6 – Exemplo de um Beacon sendo monitorado por um App dedicado. Fonte (NALDER, 2014)

2.2.1 Implementações *Pseudo-Standards* para beacons

A informação enviada pelos beacons estão em campos pré-definidos pelo protocolo BLE utilizado. O pacote de informação em si é um código alfanumérico. Um aplicativo dedicado faz a interpretação desse pacote de dados em seus respectivos campos pré-definidos pelos protocolos. Pelo fato de não existir uma implementação de protocolo *standard* oficial, todas as implementações para beacons são chamadas de *pseudo-standard*. São implementadas por algumas empresas e adotados por outros grupos de empresas. As principais são o iBeacon, o Eddystone e o AltBeacon.

- **iBeacon:**

  O termo "iBeacon" é marca registrada da Apple, uma das primeiras grandes empresas a adotar essa tecnologia. É necessária uma licença da Apple pra utilizar e comercializar produtos com a logo iBeacon. Nesse *pseudo-standard*, há o Broadcast de pacotes de até 30 bytes em intervalos de até 100ms. As três principais informações em um pacote de iBeacon são: UUID, que funciona como identificação do beacon; Major, que identifica um subgrupo de beacons dentro de um grupo maior; Minor, que identifica o beacon dentro de seu subgrupo. A Figura 7 mostra a estrutura de um pacote de *advertise* em iBeacon (WARSKI, 2014).

Figura 7 – Protocolo de um pacote de dados no *pseudo-standard* iBeacon. Fonte: Autor

- **Eddystone:**

  O Eddystone é um formato de beacon de código aberto da Google para dispositivos Android e IOS. Em seu protocolo, são definidos diferentes tipos de frames exclusivos que podem ser usados individualmente ou em combinação, como Eddystone-TLM (*broadcast* de telemetria), Eddystone-UID (identificação única) e Eddystone-URL (*broadcast* de URLs). O Eddystone-URL permite o que a Google chama de "*Physical Web*", que é o conceito de uma rede física criada por dispositivos espalhados pelo ambiente que transmitem URLs com conteudo web para outros dispositivos. O Eddystone já é suportado pelo Google Chrome para IOS e para Android. Materiais de protocolo, ferramentas e códigos *open source* no Github e pela Google. A Figura 8 mostra a estrutura de um pacote de *advertise* em Eddystone. (KONTAKTIO, a)

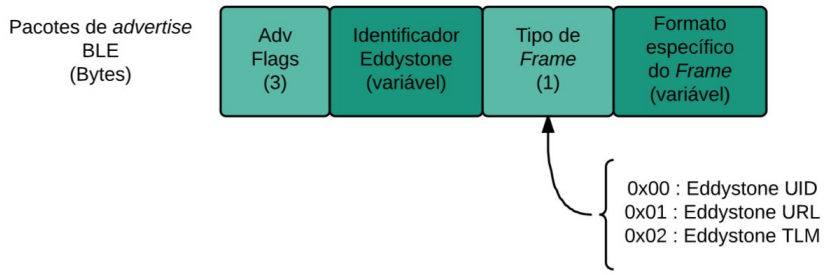

Figura 8 – Protocolo de um pacote de dados no *pseudo-standard* Eddystone. Fonte: Autor

- Altbeacon

  O Altbeacon é um formato definido pela Radius Network bastante similar ao iBeacon porém foi criado com o objetivo de ser uma implementação OS-agnóstica e *open source* que não favoreceria nenhuma companhia em particular. As especificações estão disponíveis no site da AltBeacon e são 100% livres de licença e royalties. Enquanto, por exemplo, os iBeacons tem 20 bytes disponíveis para dados do usuário (UUID+Major+Minor) os AltBeacons tem 25 bytes disponíveis. Ou seja, é possível entregar mais dados por pacote. As especificações atuais não definem um período especifico de envio dos pacotes. A Figura 9 mostra a estrutura de um pacote de *advertise* em Altbeacon. (BLACKSTONE, 2014)

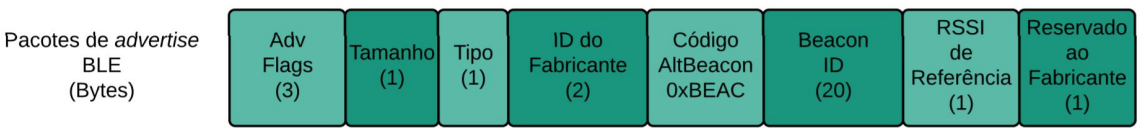

Figura 9 – Protocolo de um pacote de dados no *pseudo-standard* Altbeacon. Fonte: Autor

Foram analisadas algumas soluções já existentes antes de se fazer a escolha do Hardware a ser utilizado para desenvolvimento. A maioria dos chips e módulos Bluetooth utilizados em beacons possuem suporte a todos esses *Pseudo-Standards* e o mais utilizado para aplicações dessa natureza é o iBeacon. Então, durante todo o desenvolvimento e análises de produtos, trabalhou-se com o *Pseudo-Standards* iBeacon.

2.3 Gateway

Além dos sensores de monitoramento, é necessário o desenvolvimento de um concentrador (gateway). A sua função é coletar os dados enviados pelos sensores, processar esses dados e comunicar-se com um servidor através de uma API (*Application Program Interface*) proprietária. Esse servidor armazena as informações num banco de dados e os exibe através de uma interface web. É através dessa interface que o usuário monitora e interage com o sistema. Diferente dos beacons, não há a necessidade de um consumo mínimo de energia pois o conceito da solução é utilizar um gateway com alimentação externa por cômodo a ser monitorado.

Dessa forma, os principais requerimentos do gateway são:

- Ser equipado com a tecnologia BLE para receber os pacotes enviados pelos sensores;
- Ter acesso a uma rede, para possibilitar a comunicação com a API;
- Ser capaz de gerenciar as tarefas de leitura de sinais Bluetooth no ambiente e transmissão de dados na rede;
- Ter uma interface IO para possível acionamento de relés de modo que, no caso de um alarme de segurança, seja possível acionar uma sirene ou semelhante.

Com isso em mente, após pesquisas e análises de gateways Bluetooth no mercado, chegou-se a duas possibilidades de projeto de gateway. A primeira possibilidade é fazer um projeto de PCB que constaria de um chip microcontrolador, um chip ou módulo BLE, um chip ou módulo WiFi e/ou porta *Ethernet*, circuito de alimentação, circuito de relés e, no caso de serem utilizados chips BLE e WiFi ao invés de módulos, circuito de antenas. A segunda possibilidade em que se chegou é utilizar uma placa pronta e equipada com as tecnologias necessárias e na qual fosse possível utilizar um Sistema Operacional embarcado. Mais especificamente, um Linux de distribuição livre.

A opção com um microcontrolador é mais barata que a opção com o Linux embarcado. Porém, é de desenvolvimento muito mais trabalhoso, em questão da placa em si e das funcionalidades a serem implementadas, as quais o Linux já possui.

Como uma possível solução para isso, foi pensada ainda na possibilidade de se utilizar um RTOS (*Real-Time Operating System*) na opção com o microcontrolador. O maior diferencial que um sistema operacional tem em relação a um sistema microcontrolado é a função de multitarefa, ou seja, executar vários programas ao mesmo tempo. Isso é feito a partir do agendador do sistema, que é responsável por decidir quando cada programa irá executar e realizar a comutação rápida entre eles, o que gera a ilusão de execução simultânea. (INSTRUMENTS, 2011)

O agendador de um RTOS é voltado para realizar um padrão previsível de execuções em grande parte dos casos com um número reduzido de processos. Isso torna a resposta de um RTOS mais precisa que a de um S.O. No entanto, as aplicações de Linux embarcado são geralmente motivadas pela disponibilidade de dispositivos que o suportam, de conectividade com redes em geral, de suporte a interface de usuário, pela facilidade de acesso a banco de dados e linguagem SQL (*Structured Query Language*, linguagem de comunicação com banco de dados), de acesso remoto ao sistema (que para fins de manutenção do produto, atualização de software e afins, é de grande interesse para a empresa), dentre outros. (CANNON, 2017)

Todas essas funcionalidades são possíveis em um microcontrolador com RTOS. Porém, são de implementação muito complexa que demandariam muito mais tempo. O preço de um microcontrolador que tem suporte ao RTOS também é bem mais alto que o de um microcontrolador comum, o que faz com que essa opção seja mais trabalhosa por um preço não tão distante do preço de uma placa com Linux embarcado. Por esses motivos, optou-se por utilizar uma placa com suporte a Linux embarcado como o gateway da solução.

2.3.1 Linux Embarcado

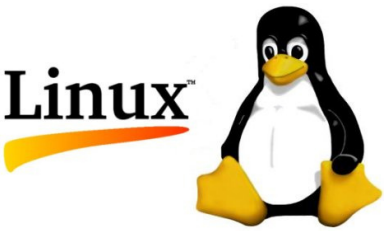

Figura 10 – Logomarca e mascote oficial da marca Linux. Fonte:(ACCARRINO, 2011)

A Figura 10 mostra a logo oficial da marca Linux. O Linux foi criado em 1991 pelo engenheiro de software Linus Trovalds. É importante ressaltar que o nome Linux não se refere estritamente a um sistema operacional e sim ao termo Linux *kernel*, que será explicado mais adiante. Na verdade, existem vários Sistemas Operacionais conhecidos genericamente como Linux que são de fato sistemas que unem o Linux *kernel* à diversos *softwares*, bibliotecas e ferramentas (CANNON, 2017). Esses sistemas operacionais são

chamados de distribuições Linux (ou, de maneira abreviada, apenas *distros*) e as mais conhecidas são: Ubuntu, Fedora, Arch Linux, CentOS, openSUSE, Debian, Linux Mint (WILLIAMS, 2014). O intuito do Linux é ser livre e de código aberto. Dessa forma, a quantidade de distribuições é bastante vasta e sua grande maioria é também livre.

Os principais pontos para se entender o funcionamento básico de um Linux embarcado são:

- *Kernel*

  O *Kernel* é o núcleo do Sistema Operacional. Ele controla todas as tarefas, as requisições de entradas e saídas, gerencia a memória do sistema e os periféricos controlados por ele (CANNON, 2017). A Figura 11 ilustra essas relações. É a parte do sistema que coordena os processos para que não haja conflito através do agendador já mencionado. O agendador de tarefas é a parte mais importante do *Kernel*. Ele gerencia uma tabela de metadados (descrição sucinta dos próprios dados) que tem por finalidade alocar tempo de execução para as tarefas do sistema. As tarefas de maior importância são executadas com maior prioridade.

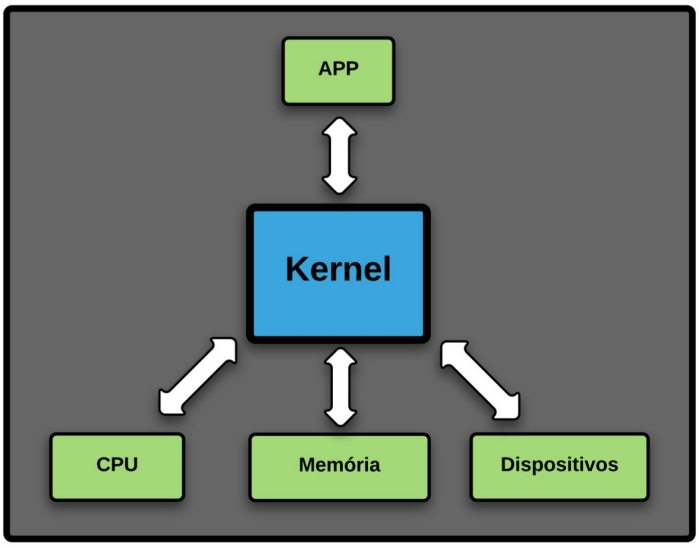

Figura 11 – Diagrama de relações com Kernel. Fonte: Autor

- *Bootloader*

  É o primeiro programa carregado ao se ligar ou reiniciar o sistema e é responsável por carregar o *kernel* na memória principal e executar o sistema operacional. Ele recebe comandos de configuração por arquivo ou linha de comando e pode passar parâmetros de configurações para o *kernel* (SUEIRO, 2013).

- *Toolchain*

  A *Toolchain* é um conjunto de programas e ferramentas voltados para desenvolvimento. É o elemento central do Linux embarcado, pois gera todos os outros elementos a serem desenvolvidos. A *Toolchain* pode ser configurada de modo a realizar a chamada compilação cruzada (*cross compile*), na qual um sistema *host* rodando em uma arquitetura "A"pode produzir códigos binários que serão executados em uma arquitetura "B"(SUEIRO, 2013).

2.3.2 TCP e HTTP

Como já foi dito, a ideia do gateway é coletar os dados Bluetooth e transmitilos a um servidor pela rede. Foram analisados e testados ambos os protocolos TCP e HTTP para realizar a comunicação do gateway com o servidor. As divisões de camadas aqui citadas (Figura 12) estão de acordo com o modelo OSI de redes de computadores (WILKINS, 2011).

O termo TCP vem do inglês *Transmission Control Protocol* e designa um protocolo de comunicação a nível de camada de transporte. É um dos principais protocolos do *Internet Protocol Suite* (conjunto de protocolos de comunicação). Ele é responsável por realizar a transferência de dados entre computadores em redes IP de maneira confiável e ordenada. Possui mecanismos de aferição de erros para se certificar de que os pacotes enviados pelo cliente TCP chegaram ao servidor TCP. Números sequenciais utilizados no protocolo permitem que seja feito o descarte de pacotes duplicados e também reordenar os pacotes recebidos na sequência correta. A utilização de *acknowledgments* (mensagem de confirmação respondida ao se receber um pacote pelo protocolo) permite que se possa determinar quando retransmitir um pacote perdido (EGGERT, 2014).

O protocolo HTTP (*HyperText Transfer Protocol*) é um protocolo a nível de camada de aplicação, ou seja, está um nível acima do TCP (vide Figura 12). O HTTP utiliza a conexão entre máquinas fornecida pelo TCP para realizar várias sessões de comunicação, controladas por pedidos de requisição e resposta entre seus clientes e servidores (EGGERT, 2014). O protocolo TCP pode ser usado diretamente em aplicações mais simples, enquanto aplicações que exigem mais suporte e mais capacidade geralmente necessitam da camada de aplicação com protocolos como HTTP.

2.4 Procedimento de Desenvolvimento

Recapitulando, o objetivo é desenvolver sensores Bluetooth e um gateway que fará a leitura e processamento das informações desses sensores e repassá-las para uma API. Em ambas as vertentes de desenvolvimento (beacon e gateway) existe uma interdependência quando se trata de testes e análises da solução. Para o gateway, é necessário receber dados

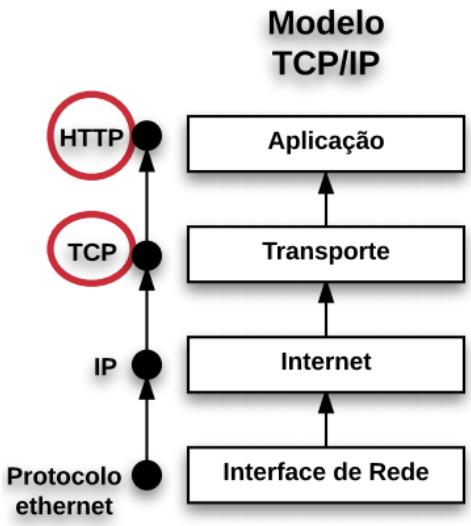

Figura 12 – Camadas do modelo TCP/IP. Fonte: Autor

para testar seu correto funcionamento. Já para os beacons, é necessário um gateway para verificar se seu envio de dados está sendo realizado da maneira esperada.

Um desenvolvimento simultâneo e interdependente de ambas as vertentes ocasionaria uma maior complexidade de se encontrar fontes de erros durante testes de funcionamento. No intuito de minimizar erros e isolar variáveis na análise durante a construção da solução, no desenvolvimento do beacon serão utilizados gateways prontos para homologação e durante o desenvolvimento do gateway serão utilizados beacons prontos com a mesma finalidade.

Assim, no começo do desenvolvimento foram testados diferentes chips e módulos BLE, assim como beacons e gateways prontos, para adquirir um bom conhecimento da tecnologia e para servir como prova de conceito. Além disso, é de bastante importância ter uma base de dispositivos para servir de auxilio em testes e homologações da solução proposta pela Innovix.

3 Desenvolvimento do Beacon

3.1 Provas de conceito

3.1.1 Estudos com Soluções Prontas da Ingics Technology

Após alguma pesquisa a procura de soluções semelhantes a proposta, chegou-se à empresa Ingics Technology, fabricantes de beacons e gateways Bluetooth. A Ingics Technology é uma empresa Tailandesa que fabrica e exporta produtos voltados para aplicação em IOT. Eles possuem quatro beacons diferentes, todos munidos de um botão de contato e três com sensores adicionais: sensor de umidade, sensor magnético e acelerômetro. Além disso, também possuem um Gateway BLE to WiFi que possui módulo WiFi para comunicação TCP, HTTP e MQTT. Foram comprados alguns beacons (iBS01) e Gateways (iGS01) para testes e estudos sobre a solução, vide Figura 13.

Figura 13 – Produtos da Ingics Technology: à esquerda, um Gateway BLE to WiFi; à direita, um beacon. Fonte: Imagens cedidas pela Innovix.

As documentações dos dispositivos estão disponíveis em (TECHNOLOGY, 2017). Os beacons da Ingics são equipados com 2 suportes de baterias CR2032 e com a alternativa de alimentação micro USB (5V). É desenvolvido a partir do chip nRF51822 da Nordic Semiconductor, que será abordado mais adiante neste trabalho. O período de transmissão de seus pacotes de *advertise* é configurável de 100ms à 10s via aplicativo para *smartphone*. Pelos pacotes, é enviada a informação do endereço MAC do beacon, estado do botão (e dos outros sensores, no caso dos modelos com sensor magnéticos, de umidade e com acelerômetro) e estado da bateria. Quando alguma ação é detectada (botão apertado, sensor magnético acionado, etc) o dispositivo entra em modo alarme e passa a enviar mensagens de 100ms com o estado dessa tal ação.

Os Gateways BLE to WiFi possuem alimentação exclusivamente por micro USB

(5V) e geram suas próprias redes WiFi com nome BLEWIFI_XX_XX (os termos X são únicos para cada Gateway) e de senha padrão "12345678". Eles possuem antena com sensibilidade de -96dBm e bidirecional impressas no circuito, ou seja, além de ler sinais de beacons pode também enviar pacotes BLE para eles, com alcance de até 30m. A antena para o WiFi é uma antena dipolo de 2dBi com taxa máxima de 72.2Mbps e com alcance de até 100m.

Na análise do envio de dados dos beacons para os gateways, foi utilizado o *software* Hercules SETUP Utility, do HWgroup, como cliente TCP para visualizar o conteúdo emitido por um gateway via WiFi. Para tal, é necessário estar conectado à rede WiFi de um gateway e informar ao programa o IP e a porta habilitada para essa comunicação. O protocolo do pacote de mensagem enviado pelo gateway da Ingics consiste em 5 partes separadas por vírgula na seguinte ordem: *Report type*, reservado para uso futuro da empresa, atualmente definido apenas por GPRP (*General Purpose Report*); Endereço MAC ou ID do beacon vinculado à essa mensagem; endereço MAC do gateway que está enviando essa mensagem; RSSI (*Received Signal Strength Indicator*) do beacon vinculado à mensagem, muito utilizado para aplicações de rastreamento de proximidade; e finalmente o pacote de dados enviado pelo beacon vinculado à mensagem. A Figura 14 mostra parte da interface do Hercules recebendo dados de um gateway. Nela, é possível ver dados de dois beacons diferentes (MACs: E0-0E-F5-1A-42-96 e D1-85-59-9F-0E-AF) sendo enviados por um gateway (MAC: F7-A1-AF-37-01-CE), com RSSIs entre -44dBm e -58dBm.

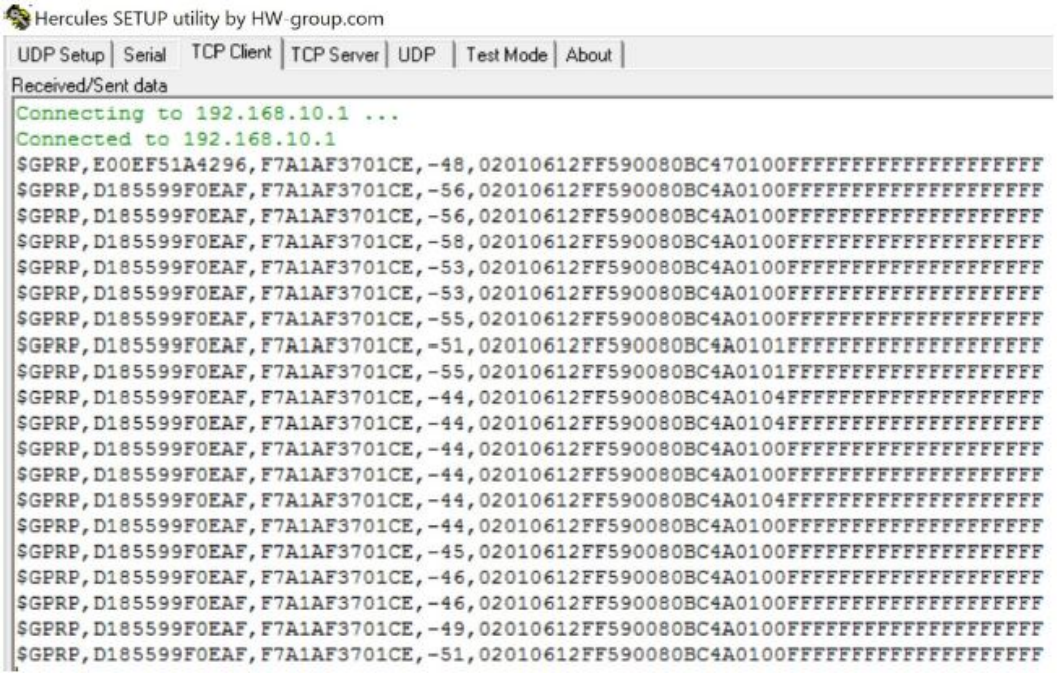

Figura 14 – Parte da interface do *software* Hercules recebendo dados de um Gateway da Ingics. Fonte: Imagem cedida pela Innovix

Foram feitos estudos de alcance do beacon iBS01 pelo gateway iGS01, em linha

reta com linha de visada não obstruída e também com beacons espalhados em diferentes localidades da empresa. Além dos testes de alcance, também foi realizado um teste em relação ao funcionamento dos beacons Ingics de acordo com o decaimento de bateria. Esse teste foi realizado no Laboratório de Circuitos Elétricos do SG11, na UnB. A descrição e resultado desses testes encontram-se no Capítulo 5.

A solução proposta pela Ingics é satisfatória e se assemelha muito ao objetivo de solução da Innovix. Foi uma das únicas opções de beacons encontradas no mercado que possuem também um botão de alarme cujo estado é enviado no pacote de transmissão, assim como se pretende desenvolver na Innovix. No entanto, um dos inconvenientes para a aplicação desse beacon no âmbito de controle de patrimônio é o tamanho do beacon (42mm x 58mm x 10mm). Para suportar 2 baterias CR2032, a PCB dos beacons desenvolvidos pela Ingics é grande se comparada com algumas outras soluções no mercado. Se o patrimônio a ser monitorado pelo beacon for pequeno demais, a solução se torna inviável. Os beacons Ingics foram escolhidos como opção para testar e homologar o gateway a ser desenvolvido, mas ainda há a necessidade do desenvolvimento de um beacon mais conveniente para a solução final.

3.2 Desenvolvimento do beacon

3.2.1 Hardware da Nordic Semiconductor

A Nordic Semiconductor é uma empresa líder mundial em soluções e dispositivos Wireless voltados para aplicação *Ultra Low Power*. A maioria dos beacons encontrados hoje são desenvolvidos a partir de módulos e chips da Nordic, incluindo a solução da Ingics Technology discutida anteriormente. O chip da Nordic mais comumente utilizado por esses beacons é o nRF51822.

O chip nRF51822 é um SOC (*System on Chip*) multiprotocolo projetado para aplicações BLE. Possui arquitetura de CPU ARM Cortex M0 32-bit com versões de 256kB ou 128kB de memória flash e 32kB ou 16kB de memória RAM. É equipado com um *transceiver* de 2.4GHz com sensibilidade de -93dBm que suporta tanto o BLE quanto um protocolo próprio da Nordic chamado Nordic Gazell 2.4GHz. Possui 31 portas GPIO e também conversor AD com múltiplos canais, o que possibilita a interação do chip com diversos periféricos.

Em 2014, a Nordic passou a comercializar um beacon programável baseado no seu próprio chip nRF51822, chamado *"nRF51822 Bluetooth Smart Beacon Kit"*, Figura 15. Diferentemente do beacon da Ingics que é comercializado como produto final, o conceito do beacon da Nordic é justamente o de um beacon de desenvolvimento, versátil, no qual o usuário pode desenvolver seu próprio firmware do zero e atualizá-lo quantas vezes quiser.

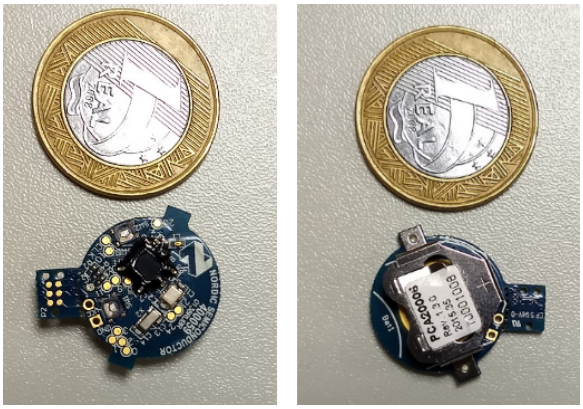

Figura 15 – *"nRF51822 Bluetooth Smart Beacon Kit"* da Nordic. Fonte: Imagens cedidas pela Innovix.

Esse beacon tem como fonte de alimentação apenas uma bateria CR1632 e portanto consegue ter um tamanho de PCB bastante reduzido (formato circular com 20mm de diâmetro). Possui 2 botões de contato além de 10 pinos GPIO distribuídos pela placa, 1 *LED* RGB, conectores para programação/*debug* externo (com a necessidade de um cabo específico) e todas as outras propriedades e funcionalidades do chip nRF51822 descritas anteriormente. Esse beacon se destaca por já ser um dispositivo com *hardware* pronto e bem equipado. Além disso, o chip tem uma documentação bem escrita e completa, tem um SDK (*Software Development Kit*) rico com mais de 50 exemplos de aplicativos prontos para uso e é o chip por trás da grande maioria de beacons encontrados no mercado hoje em dia. Decidiu-se então por adquirir alguns desses beacons para dar início ao desenvolvimento da solução proposta a partir dessa plataforma.

Além de alguns beacons, adquiriu-se também a placa de desenvolvimento para o chip da Nordic (Figura 16) . A placa traz a conveniência de poder ser alimentada tanto por 1 bateria CR2032 quanto via cabo micro USB e o mesmo também disponibiliza a funcionalidade programação/*debug*. Além disso, a placa vem com 4 botões de contato programáveis, 1 botão de *reset*, 5 LEDs (quatro programáveis e um para indicar a comunicação USB), 29 pinos GPIO digital e 6 analógicos e pinos para medição de consumo de energia. Existem também pinos de programação/*debug* externos, que são utilizados para programar dispositivos da Nordic, como o próprio chip nRF51822 presente no beacon, através da placa de desenvolvimento. Algo interessante é que a placa de desenvolvimento é compatível com *shields* para o conhecido Arduino Uno.

Então, o firmware será desenvolvido na placa de desenvolvimento e, depois de testado nela, será migrado para o beacon, sendo o principal facilitador da placa a programação via USB. Um detalhe a ser mencionado é que o chip da placa é o nRF52832, e o chip do beacon é o nRF51822, já abordado anteriormente. A única diferença prática encontrada entre os chips foram algumas bibliotecas do SDK do chip nRF52832 (placa)

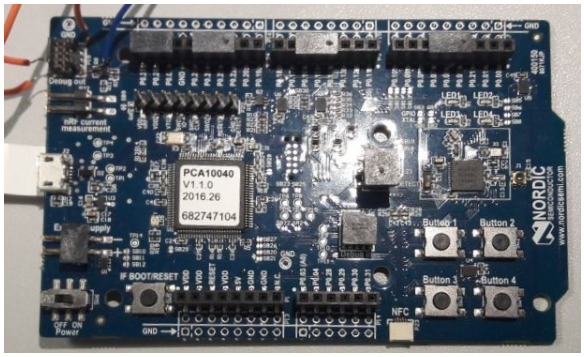

Figura 16 – Placa de Desenvolvimento da Nordic para o chip nRF52832. Fonte: Imagem cedida pela Innovix.

que não são compatíveis com o nRF51822 (beacon) então, ao final do desenvolvimento do firmware na placa, foram necessárias algumas adequações de certas funções para bibliotecas compatíveis com o chip do beacon. As bibliotecas que precisaram ser adequadas foram a de gerenciamento dos botões e a de gerenciamento do conversor AD, que foi utilizado para medição do nível de bateria do beacon.

3.2.2 Programação do Firmware

A solução proposta é a de um beacon que será anexado a um objeto, de modo que se o objeto for retirado do campo de alcance do gateway que monitora esse beacon, o sistema de controle entra em modo de alerta. Também, se o beacon for retirado do objeto para que permaneça na área de alcance do gateway enquanto o objeto é furtado, um botão deixaria de fazer contato com o objeto e assim o sistema também entraria em modo de alarme. Dessa forma, no quesito firmware do beacon, existem alguns pontos chaves a serem atendidos:

- **Transmissão de *advertises* BLE mesmo no modo não-alerta**

  O beacon fará o *broadcast* de pacotes de *advertise* constante e incessantemente, pois mesmo que não haja nenhum alerta é necessário verificar se o monitoramento está acontecendo ou se há algum erro em alguma parte do sistema. Assim, o processamento desse sinal, seja no gateway seja na API dedicada a isso, poderá interpretar que se o beacon não enviou nenhuma mensagem em um dado intervalo, deve-se então entrar um estado de alerta.

- **Monitoramento do estado do botão e envio imediato no caso de mudança de estado**

  A função de monitoramento do botão e de incluir seu estado na mensagem transmitida é essencial para a solução proposta. Além disso, é necessário que quando

esse botão for liberado, ou seja, quando o beacon for retirado do objeto monitorado, a mensagem com essa informação seja enviada imediatamente e não com o mesmo intervalo da mensagem periódica abordada no item anterior. Por exemplo, se o beacon for programado para enviar seus *advertises* a cada 1 minuto, uma pessoa poderia retirar o beacon do objeto e sair do estabelecimento com um intervalo de 1 minuto até que a próximo pacote com a informação de que o objeto foi furtado fosse transmitido. Dependendo da situação, 1 minuto é um intervalo muito grande entre o furto acontecer e o o sistema reconhecer um alarme. Simplesmente diminuir o intervalo de transmissão do beacon não é uma solução eficiente, pois quanto mais radiotransmissões o chip realizar, mais ele gasta energia. Portanto, o objetivo é fazer com que o beacon envie mensagens periódicas e, além disso, envie mensagem imediatas a cada evento de mudança de estado do botão através de interrupções.

- **Medição do nível de bateria do beacon e incluir esse nível na mensagem**

  Como foi explicado no primeiro item, há a necessidade de envios periódicos de pacotes para checar se o sistema está funcionando. Caso a bateria do beacon acabe, esse envio não terá continuidade e o gateway ou API irá interpretar como um estado de alerta. Para evitar esse acontecimento, que pode também dar oportunidade para algum furto, é interessante que o usuário tenha a informação dos níveis de bateria e se programe para fazer as devidas trocas de baterias. Assim, é necessário incluir o nível de bateria no pacote de informação.

- **Desenvolver um log de eventos ocorridos**

  É possível que eventualmente ocorra a queda da rede de comunicação do gateway com o servidor e com a API, ou até mesmo que o gateway seja desligado intencionalmente ou por uma queda de energia. Nesse intervalo, qualquer evento que ocorra com o beacon não será repassado para a API e consequentemente para o usuário. Assim, é necessário que o pacote transmitido tenha um campo que guarde alguma informação sobre a ocorrência de eventos anteriores. Dessa forma, quando a comunicação for reestabelecida, o pacote enviado pelo beacon pode fornecer a informação de que algum evento aconteceu durante o período de queda da comunicação.

A plataforma utilizada para o desenvolvimento do *firmware* foi a uVision da Keil. No Anexo A deste documento, encontram-se detalhes sobre a o uVision e sobre todas as outras ferramentas utilizadas para desenvolver o *firmware* e transferi-lo para a placa de desenvolvimento. No Anexo B, encontram-se estudos e análises de projetos exemplos de *firmware* da Nordic com suporte no uVision.

O firmware para a solução de controle de patrimônio foi desenvolvida baseando-se no projeto exemplo *ble_app_uart*, abordado no Anexo B, por ser um exemplo bastante

completo. Foram retiradas, então, algumas funcionalidades presentes nesse projeto exemplo que não seriam utilizadas, como a comunicação serial UART (através das bibliotecas *app_uart*). Também foram removidos todos os arquivos e bibliotecas relacionadas à conexão/pareamento Bluetooth, uma vez que a solução proposta utilizará apenas a funcionalidade de *broadcast* dos pacotes de *advertise* do serviço Bluetooth, não havendo o pareamento propriamente dito ente dispositivos. No fim, o código do exemplo foi reduzido até somente as funções que configuram o *softdevice*, a função responsável por montar e enviar o pacote de *advertise* e a função *main* para chamar essas demais.

3.2.2.1 Transmissão de *Advertises* BLE

Para os serviços Bluetooth é necessária que haja a configuração do softdevice na inicialização do código. Existem funções prontas para fazer essa configuração e essas funções foram utilizadas. Elas constam basicamente de habilitar algumas funcionalidades do chip, ativar alguns parâmetros, reservar memória RAM para os serviços Bluetooth, dentre outros. Uma vez configurado o softdevice, o próximo passo é configurar o pacote de *advertise*.

A biblioteca do SDK que foi usada para configurar o pacote de transmissão é a *ble_advdata*. Além disso, foi criada a função *advertising_init*, que chamaria várias das funções dessa biblioteca para gerar o pacote.

Na *advertising_init*, é criado um vetor de 5 posições. A primeira posição é reservada para passar a informação do status do botão, sendo 0 se o botão está solto ou 1 se o botão está pressionado. O conjunto formado pelas posições de 2 a 4 desse vetor será utilizado para armazenar a informação do nível de bateria, que idealmente será passado em formato de porcentagem e, portanto, temos uma posição para cada uma das três casas decimais utilizadas de 0 a 100. Já a última posição será utilizada para a funcionalidade de log de eventos.

O iBeacon possui uma parte de seu pacote de dados chamado *AD Flags* (como na Figura 7). Esse identificador indica o tipo de transmissão que está sendo feito. No caso do serviço de *advertise*, o dado "020106", conhecido com *Avdertize Flag*, é padrão em todo beacon e traz a informação de que o beacon está no modo de *advertise*, e não de pareamento, por exemplo. Após o *AD Flags*, temos o campo *Length*, que informa o tamanho da mensagem, em bytes, desse campo em diante. Para o caso da mensagem construída para a solução proposta, esse dado é "08", indicando 8 bytes. Depois, existe um campo chamado de *Manufacturer Specific Data*. Esse campo é utilizado para transmitir qualquer tipo de informação que não são *Proximity UUID*, *Major* ou *Minor* (Capítulo 2). Ou seja, é o espaço no pacote de *advertise* no qual o desenvolvedor pode inserir o que quiser. O campo de *Manufacturer Specific Data* é identificado pelo padrão "0xFF", seguidos por um combinação de 2 bytes que variam de acordo com o tamanho da mensagem de informação

a ser inserida, o intervalo de transmissão e outros parâmetros. Após essa identificação estarão os dados de informação inseridos pelo desenvolvedor (no caso, o vetor de 5 posições). Todos esses campos citados foram configurados dentro da função *advertising_init* criada para gerenciar o pacote de *advertise*. Assim, o pacote enviado é o seguinte:

```
1 02010608FF0800BBEEEEEELL
```

Sendo "BBEEEEEELL" o vetor com 1 byte de estado do **B**otão, 3 bytes de nível de **E**nergia e 1 byte final de **L**og de eventos. Vale lembrar que esse é o pacote de *advertise*, mas todo dispositivo emissor de Bluetooth transmite em sua mensagem o seu ID (ou endereço MAC) como foi abordado na análise dos produtos da Ingics.

Para a configuração do intervalo de transmissão, o softdevice possibilita um intervalo de 100ms até 10,24s. Esse número foi discutido durante o desenvolvimento e foi decidido que um intervalo mais longo causaria um menor consumo de energia, por menos ativações do radiotransmissor, e consequentemente uma maior vida útil para a bateria. Além disso, não seriam feitas aferições de intervalos tão pequenos para checar se o sistema está funcionando. Então, como será desenvolvido um envio automático no caso de evento de botão, não há necessidade de envios de períodos tão curtos para um estado de não-alerta.

Para contornar a limitação de 100ms a 10,24s, foi adicionada a biblioteca *app_timer* para criar um temporizador. O temporizador é inicializado com a função *timer_create*, dentro da função *main*, com modo *APP_TIMER_MODE_REPEATED*. Esse modo que faz com que o temporizador reinicie automaticamente a sua contagem após seu término. Ainda na *timer_create*, é atrelado a esse temporizador uma função de gerenciamento que foi chamada de *timer_handler*. Após a iniciação do temporizador, ainda na *main* é chamada a função *timer_init* para dar início à contagem do temporizador. Quando o temporizador chega ao final de sua contagem, a função de gerenciamento *timer_handler* é chamada por interrupção e nela ocorre a chamada da função *advertising_init* para configurar o pacote de *advertise*. Seguindo a *advertising_init*, é chamada a função *sd_ble_gap_adv_start*, configurada para realizar o envio de apenas 1 pacote de *advertise*. Dessa forma, a cada contagem do temporizador, existe a configuração do pacote de *advertise* e o envio de um pacote, de forma que o intervalo efetivo de transmissão seja, na verdade, o tempo a ser configurado para o contador desse temporizador. Foi definido que um tempo de 5 minutos seria adequando. Esse parâmetro é facilmente alterável no código caso futuramente decida-se que 5 minutos é um intervalo longo demais.

3.2.2.2 Monitoramento do estado do botão

Para o monitoramento o estado do botão, foi utilizada inicialmente a função *bsp_button_is_pressed* da biblioteca *bsp*, abordada no relato sobre o teste com o exemplo ble_app_uart. Da mesma forma como foi feito no exemplo, essa função foi adicionada no loop principal da função *main* mas, dessa vez, monitorando apenas um dos botões da placa de desenvolvimento. Assim que um estado de botão apertado fosse detectado, seriam imediatamente chamadas as funções *advertising_init* e *sd_ble_gap_adv_start* para configurar e fazer o envio de um pacote de *advertise* que carregue essa informação.

Após uma série de testes, essa função se mostrou pouco eficiente e também percebeu-se um grande inconveniente em seu uso. Utilizando essa função no loop principal, o código estaria constantemente executando seus comandos, aferindo o estado do botão a todo momento, o que resultaria num processamento excessivo e desnecessário, visto que informação interessante é apenas a mudança de estado.

Após pesquisas na plataforma *Developer Zone* e no *Infocenter* da Nordic, chegou-se a uma biblioteca com funções mais eficientes para o que se foi proposto fazer. A biblioteca *bsp_btn_ble* foi adicionada ao projeto. Com ela, foi utilizada a função *app_button_init* para configurar uma função de gerenciamento de eventos de botão a ser chamada via interrupção a cada troca de estado do botão. A função criada para realizar esse gerenciamento das interrupções do botão foi chamada de *button_event_handler*. Essa função recebe duas variáveis: uma que armazena a ID do botão responsável pela interrupção detectada e outra que armazena a ação que ocasionou a interrupção no dado botão (ação de pressionar ou de soltar o botão). Dentro dessa função fez-se o uso de uma simples estrutura de *switch case* para tomar diferentes ações caso o botão tivesse sido pressionado ou liberado. Foi criada uma *flag* a qual seria atribuída o valor 0 ou 1 de acordo com o estado liberado ou pressionado do botão, respectivamente. Essa *flag* é a variável utilizada na função *advertising_init* para definir se o pacote de *advertise* carregará o valor de 1 ou 0 no espaço reservado para o estado do botão.

Após o *switch case*, a função de gerenciamento de interrupções do botão finaliza a contagem atual, chama as funções *advertising_init* e *sd_ble_gap_adv_start* para fazer a configuração e o envio de um pacote de *advertise*, e depois reinicia o contador para manter o envio periódico dos pacotes. A estrutura dessa função de gerenciamento ficou da seguinte forma:

```
static void button_event_handler(uint8_t pin, uint8_t action){

if (pin == BUTTON_1)
{
   switch (action)
    {
      case APP_BUTTON_PUSH: // BOTAO PRESSIONADO
         FLAG = 1;     // Levanta a Flag indicando modo "ok" (Botao Pressionado)
         break;

      case APP_BUTTON_RELEASE:// BOTAO SOLTO
         FLAG = 0;     // Abaixa a Flag indicando modo "Alarme" (Botao Solto)
   }

   advertising_init();           // Configura o pacote de adv
   sd_ble_gap_adv_start(&m_adv_params);  //Faz o envio de um pacote de adv

   app_timer_start();            // Reinicia as transmissoes periodicas.
}
```

Dessa forma, a detecção da mudança de estado dos botões é muito mais eficiente pois não existe processamento constante do chip, economizando energia e aumentando a vida útil da bateria.

3.2.2.3 Medição dos níveis de bateria

Para medir os níveis de bateria, foi necessário o uso do conversor AD presente no chip. Para acessar as funcionalidades do conversor AD, adicionou-se a biblioteca *nrf_drv_saadc* ao projeto. Foi criada a função *adc_configure* para configurar o conversor AD. Foi necessário definir um valor de tensão de referência e um valor de *prescaler*, que é um multiplicador usado para adequar o valor medido à referência definida e a resolução digital a ser utilizada pelo conversor. O canal padrão para realização de amostras pelo conversor AD já faz a leitura da tensão de alimentação do chip, então não houve necessidade de configurar o canal.

Com a correta configuração do conversor AD, a função *nrf_drv_saadc_sample* da biblioteca *nrf_drv_saadc* é utilizada para realizar uma amostra do valor de tensão do canal definido nas configurações. Logo no início da função *main*, é chamada a função *nrf_drv_saadc_sample* para que o primeiro pacote enviado já contenha a informação dos níveis de bateria.

Quando chamada, a função *nrf_drv_saadc_sample* inicia a amostragem do canal definido, o que gera uma interrupção que é gerenciada pela função *saadc_event_handler*. Essa função dá início a conversão do valor amostrado para milivolt, de acordo com os parâmetros definido anteriormente. Depois disso, ainda na função *saadc_event_handler*, o valor em milivolt é convertido para porcentagem com a utilização da função *battery_level_in_percent*. Essa função procura linearizar a curva de descarga da bateria em 4 seções lineares de inclinação diferentes. Valores iguais ou superiores a 3,0V retornam 100% e o valor de 2,1V retorna 0%, visto que valores menores que esse são considerados inadequados para correta alimentação do chip. As quatro seções de linearização estão indicadas abaixo. A Figura 17 mostra a representação gráfica dessas seções de linearização.

- Seção 1: 3.0V - 2.9V = 100% - 42% (58% de queda em 100 mV)
- Seção 2: 2.9V - 2.74V = 42% - 18% (24% de queda em 160 mV)
- Seção 3: 2.74V - 2.44V = 18% - 6% (12% de queda em 300 mV)
- Seção 4: 2.44V - 2.1V = 6% - 0% (6% de queda em 340 mV)

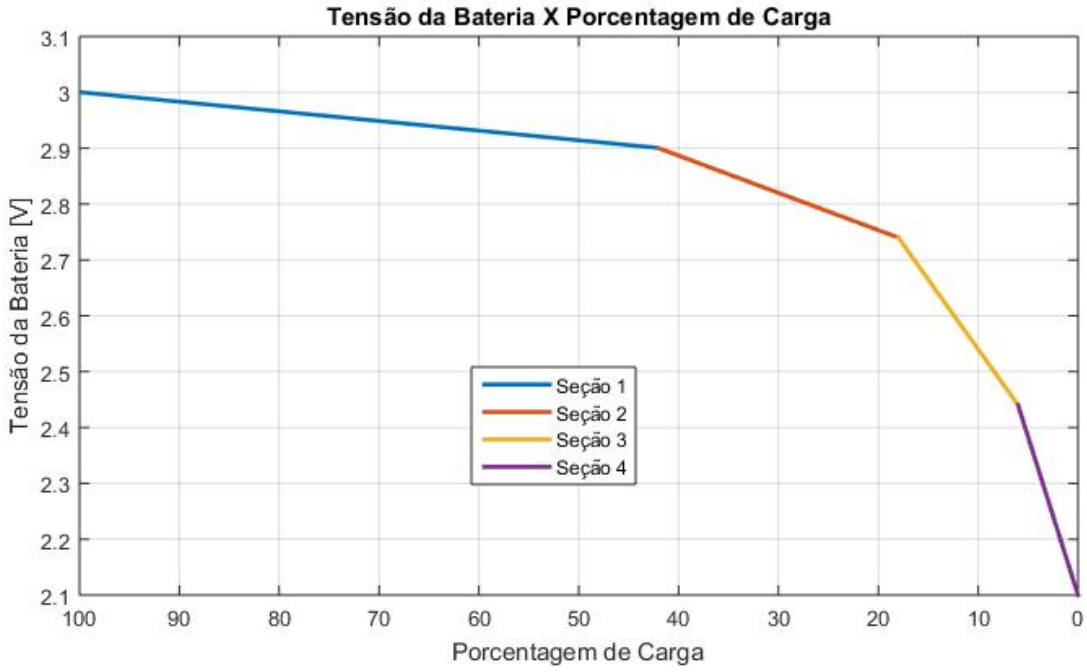

Figura 17 – Representação das seções lineares de decaimento da carga da bateria.

Assim, a função *battery_level_in_percent* é definida da seguinte forma:

```
uint8_t battery_level_in_percent(const uint16_t mvolts)
{
    uint8_t battery_level;

    if (mvolts >= 3000)      battery_level = 100;

    else if (mvolts > 2900)  battery_level = 100 - ((3000 - mvolts) * 58) / 100;

    else if (mvolts > 2740)  battery_level = 42 - ((2900 - mvolts) * 24) / 160;

    else if (mvolts > 2440)  battery_level = 18 - ((2740 - mvolts) * 12) / 300;

    else if (mvolts > 2100)  battery_level = 6 - ((2440 - mvolts) * 6) / 340;

    else                     battery_level = 0;

    return battery_level;
}
```

Os valores de bateria a serem passados para o vetor que irá compor o pacote de *advertise* devem ser valores hexadecimais. Poderia ser usado 1 byte para transmitir essa informação, visando uma mensagem mais curta e uma consequente economia de energia. Em 8-bits, pode-se codificar valores de 0 a 255, o que bastaria para enviar os dados de porcentagem. No entanto, assim como o estado do botão e o log de eventos que será abordado, é interessante para desenvolvimento e para reconhecimento mais rápido de erros que essas informações sejam de fácil visualização na mensagem. Além disso, o pacote de *advertise* aqui proposto (12 bytes) já possui tamanho inferior ao da maioria de beacons encontrados no mercado (beacons Ingics possuem 22 bytes em seus pacotes, por exemplo) utilizando o mesmo chip nRF51 e que possui expectativa de mais de 2 anos de bateria.

Assim, conclui-se que a economia de 2 bytes não traria grande efeito nos gastos energéticos do beacon. O valor da porcentagem de bateria foi transmitido na mensagem utilizando um byte para cada casa decimal do possível valor entre 0 e 100. Ou seja, a transmissão de um valor de 63% de bateria será feita com o envio de um byte 0, um byte 6 e um byte 3, em sequência, no campo do pacote de *advertise* reservado para isso.

Para converter o valor de porcentagem que está em uma variável do tipo *uint8_t* em um vetor, foi criada a variável global "a" como um vetor *uint8_t* de três posições. Para a conversão, foi criada a função *int_to_vec* a seguir:

```
int_to_vec(uint8_t num){
   if (num>100) num = 100;

   a[0] = num/100;
   a[1] = (num - a[0]*100)/10;
   a[2] = num - a[0]*100 - a[1]*10;}
```

O conversor AD aumenta consideravelmente o consumo de energia do chip durante a amostragem e conversão do valor analógico. A corrente do sistema, utilizando o modo *low power mode*, é em média 2.6uA. Com o funcionamento do conversor AD, a corrente chega a pouco mais de 1mA. Optou-se então por realizar a aferição do nível de bateria apenas duas vezes ao dia (a cada 12h). Uma vez que é esperado que em 12 horas ocorram 144 interrupções do temporizador da transmissão periódica de *advertise* (uma interrupção/transmissão a cada 5 minutos) foi definido um contador a ser incrementado dentro da função de gerenciamento de interrupção do temporizador que, ao chegar em 144, retorna ao valor 0 e faz uma aferição de nível de bateria. A variável global "a" que guarda o vetor com os algarismos de porcentagem de bateria é a variável utilizada pela função *advertising_init* para o campo do pacote destinado à essa informação. Assim, todo pacote de *advertise* carrega a informação de nível de bateria, e essa informação é atualizada a cada 12h.

3.2.2.4 Log de eventos ocorridos

Muito se pensou sobre qual seria a melhor estratégia para carregar a informação sobre eventos anteriores na mensagem transmitida pelo beacon, para que, em caso de perda de comunicação com o gateway, servidor ou API, o usuário tenha a informação de que algum evento ocorreu quando a comunicação for reestabelecida.

Para tal, chegou-se a considerar a utilização do *Real time Counter* (RTC) do chip para implementar um sistema que fornecesse a informação de dia, hora, minuto e segundo do último alarme (botão solto) ocorrido. O único exemplo no SDK fornecido pela Nordic que possui uma implementação como essa, assim como aconteceu com os exemplos de níveis de bateria, é implementado em cima do modo de conexão/pareamento e não no modo de *advertise*. Nessa implementação, o sistema recebe do dispositivo ao qual está pareado a informação de data e horário atual e utiliza o RTC para dar continuidade a essa contagem a partir desse valor inicial.

A implementação de um sistema como esse a partir do zero levaria um tempo considerável. Além disso, o objetivo é utilizar apenas o modo *advertise* do Bluetooth para potencializar o conceito *low energy* do BLE. Ou seja, mesmo que se implementasse um

sistema de contagem de dias, horas, minutos e segundo com o RTC, haveria a dificuldade da aquisição da informação de data e horário inicial sem o pareamento com um dispositivo externo. Essa opção foi então descartada.

O problema da perda de conexão é que junto com ela é perdida toda a informação do que ocorre até seu reestabelecimento. A informação do horário exato de um evento ocorrido nesse intervalo não é necessariamente de grande importância. Se, em uma queda de conexão que demorou uma hora, um objeto foi desassociado de um beacon (botão do beacon foi solto), o mais importante para o usuário é ser notificado de que houve um furto naquele intervalo, não necessariamente que o furto ocorreu num dado minuto específico daquela hora, dado que o furto já aconteceu. Lembrando que, uma vez que o principal objetivo da solução é alertar um furto no momento imediato de seu acontecimento, o problema tratado agora é de contornar uma situação extrema que possa vir a ocorrer, na qual não há conectividade do beacon com a interface do usuário.

Por fim chegou-se numa solução de implementação bastante trivial que soluciona o problema de maneira satisfatória. O firmware do beacon terá uma variável global que funcionará como contador que será incrementado na função de gerenciamento de eventos de botão, a cada detecção de uma ação de botão solto. Dessa forma, esse contador guardará consigo a informação de quantas vezes o botão foi solto, ou seja, de quantos alarmes ou eventos relevantes aconteceram. Essa variável contador será a informação repassada no pacote de *advertise*. A eficiência dessa solução será dependente também da interpretação dessa informação pelo gateway ou pela API. Assim, se a um certo beacon estava associado um número 0 de eventos e, após uma queda e reestabelecimento de conexão, esse mesmo beacon traz a informação de um número 1 de eventos, significa que durante esse intervalo sem comunicação houve um evento de botão solto e a API deve entrar em modo de alarme. Da mesma forma, se um beacon ao qual já está associado o número de 11 eventos passar apresentar 13 eventos após uma queda de conexão, significa que ocorreram 2 eventos de botão solto e a API deve entrar em modo de alarme para que a situação seja investigada. Esse contador retorna para 0 após 100 eventos.

Esse Log de eventos acabou por ser um recurso redundante, uma vez que durante o desenvolvimento do firmware do gateway foi criado um módulo *offline* para lidar com as situações de desconexões de maneira muito mais eficiente que esse método implementado apenas no beacon.

3.2.2.5 Migração do código do chip nRF52832 para o nRF51822

Estando o firmware pronto e testado na placa de desenvolvimento, o próximo passo foi migrar o código do chip nRF52832 utilizado pela placa para o chip nRF51822 utilizado pelo *Smart Beacon*. O aplicativo exemplo *ble_app_uart*, utilizado como base para a

construção do firmware no chip nRF52, não é disponível para a versão de placa utilizada no beacon com o chip nRF51, uma vez que para a conexão UART era utilizado o cabo USB e o beacon não possui tal entrada. Foi necessário então utilizar outro exemplo como base para o chip nRF51 e adicionar/remover alguns arquivos necessários/desnecessários no projeto.

Fez-se o uso do aplicativo padrão do beacon chamado *ble_app_beacon* como projeto base para o nRF51. Inicialmente foi feita a cópia de todo o conteúdo do arquivo main.c desenvolvido no projeto para a placa de desenvolvimento para o main.c do projeto do *ble_app_beacon* e, a partir das notificações de erro em compilações, foram sendo feitas as alterações.

A indexação dos botões na inicialização dos pinos de GPIO, no começo da função *main*, teve que ser refeita visto que o beacon possui 2 botões e a placa de desenvolvimento possui 4. A maior diferença foi em relação a aferição de bateria. O conversor AD dos dois chips utiliza configurações diferentes. Para cada um, existe sua determinada biblioteca, com as determinadas funções. Ao invés da biblioteca *nrf_drv_saadc* para o chip nRF52, foi preciso adicionar a biblioteca *nrf_adc*. As funções referentes ao funcionamento do conversor AD foram todas substituídas por funções correspondentes nesse nova biblioteca. Por exemplo, para realizar as amostras de sinal analógico no canal definido, a função *nrf_drv_saadc_sample* foi substituída pela função *nrf_adc_start* de mesmo funcionamento. Em geral, a migração entre os chips não foi complicada.

A placa do beacon possui apenas *pads* de contato para as trilhas utilizadas para programação (vide Figura 18), e essas trilhas devem ser conectadas à interface de programação/*debug* externo da placa de desenvolvimento (vide Figura 18). De acordo com a documentação dos dispositivo ((SEMICONDUCTOR, 2014) e (SEMICONDUCTOR, 2016) de acesso restrito à membros registrados na plataforma da Nordic Semiconductor), é possível chegar à relação de pinos vista nas Tabelas 1 e 2.

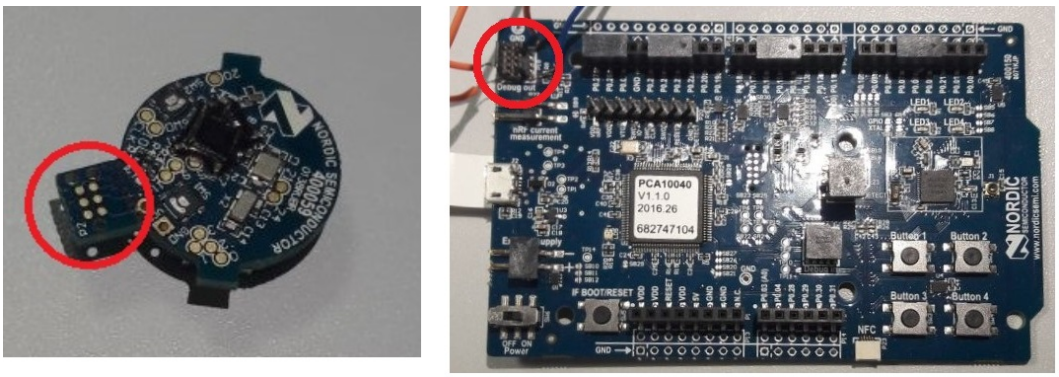

Figura 18 – À esquerda, indicação dos 6 pinos de programação do *Smart* Beacon da Nordic. À direita, indicação dos 10 pinos *debug-out* da placa de desenvolvimento. Fonte: Imagens cedidas pela Innovix

O cabo utilizado para fazer a conexão do beacon com a interface de programação/*debug* da placa de desenvolvimento é o *TC2030-CTX-NL 6-Pin "No Legs"Cable with 10-pin micro-connector*, da fabricante Tag Connector. Por ter um preço muito elevado e o projeto ainda estar em fase de teste, foi preferível soldar alguns *jumpers* e utilizar uma pequena *protoboard* para fazer a conexão dos terminais do beacon com os da placa de desenvolvimento. Assim, o beacon foi inicialmente programado da forma mostrada na Figura 19, a partir de *jumpers* soldados a esses terminais, realizando a correta relação dos pinos do *Smart* Beacon de acordo com a pinagem das Tabelas 1 e 2. Atualmente, foi adquirido o cabo da Tag Connector uma vez que, caso seja decidido que essa passe a ser a solução final e comercializada da empresa, seria inviável fazer a soldagem manual de 4 pinos para remessas maiores de beacons, pois os clientes dessas soluções geralmente fazem pedidos na ordem de centenas de unidades.

Tabela 1 – Pinos de programação do *Smart* Beacon da Nordic e suas funções.

Pinos da Placa do *Smart* Beacon	Função
1	VCC
2	SWDIO
3	-
4	SWDCLK
5	GND
6	-

Tabela 2 – Pinos de programação da Placa de Desenvolvimento e suas funções.

Pinos da Placa de Desenvolvimento	Função
1	EXT VTG
2	EXT SWDIO
3	GND
4	EXT SWDCLK
5	GND
6	EXT SWO
7	-
8	-
9	EXT GND DETECT
10	EXT RESET

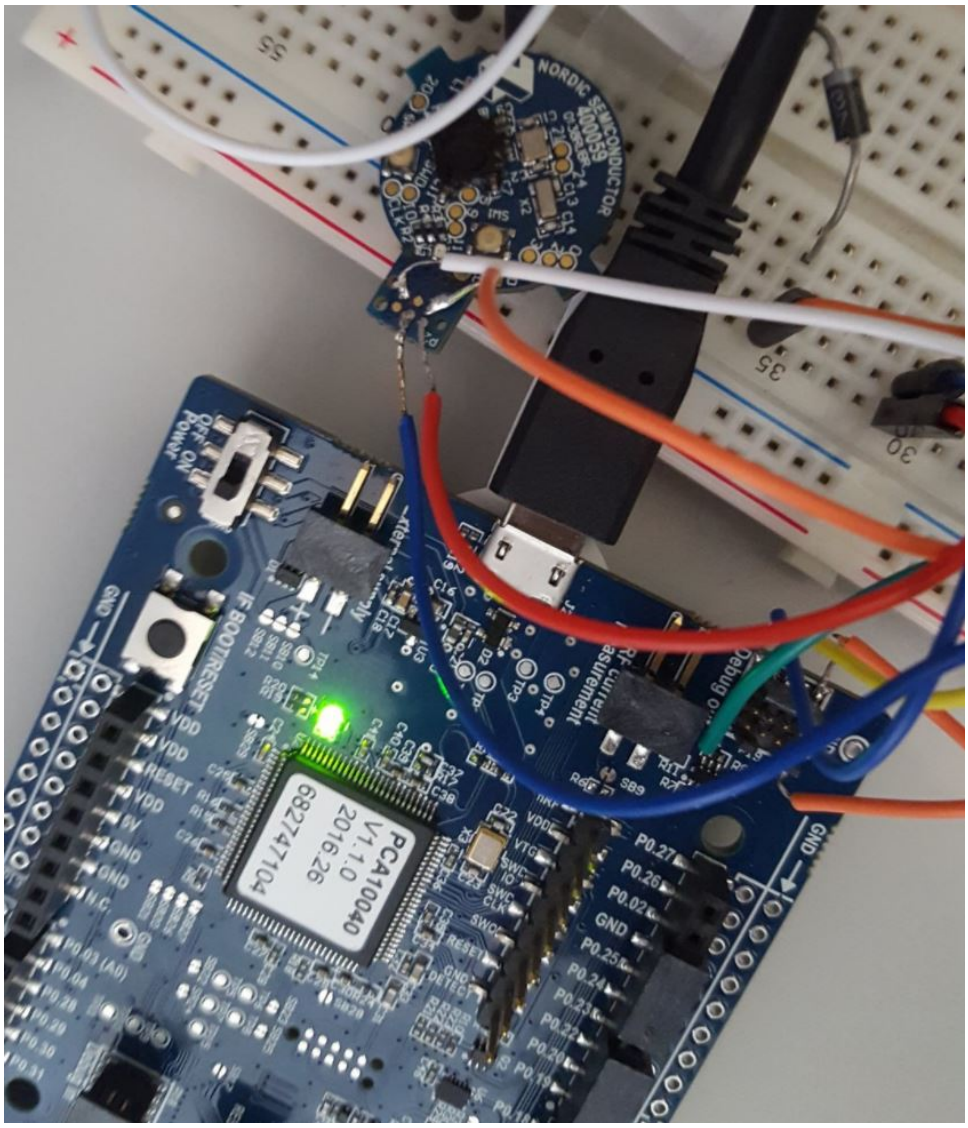

Figura 19 – Jumpers soldados aos terminais de programação do beacon e da placa de desenvolvimento. Fonte: Imagem cedida pela Innovix.

4 Desenvolvimento do Gateway

4.1 Raspberry Pi 3

Após chegar-se a um beacon funcional, iniciou-se o desenvolvimento do gateway. A primeira opção de sistema embarcado com Linux que chamou muita atenção foi a Raspberry Pi 3 (vide Figura 20).

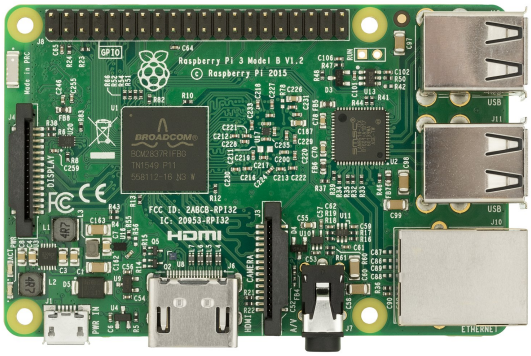

Figura 20 – PCB da Raspberry Pi 3. Fonte: (EVAN-AMOS, 2016)

A empresa Raspberry Pi se propõe a fabricar pequenos computadores de baixo custo, inicialmente voltados para o aprendizado didático de programação, que vêm ganhando visibilidade nas mais variadas aplicações. O último modelo até então é a Raspberry Pi 3, lançada em fevereiro de 2009. Suas especificações incluem o processador Broadcom BCM2837 (um ARM Cortex-A53 Quad Core 1,2GHz de 64bits), 4 portas USB, 40 pinos de GPIO que incluem UART, I2C e SPI (FONDATION,). Aceita diversos sistemas operacionais, de Windows gratuito à diferentes distribuições de Linux e possui módulos WiFi e Bluetooth *Low Energy* embarcados. Essa última característica foi determinante para que a Raspberry Pi fosse um hardware elegível para se desenvolver.

Apesar de sua inquestionável capacidade de processamento e da grande popularidade que vem ganhando, a Raspberry Pi ainda é muito vista como um produto usado para *hobby* e para aprendizado, que foi de fato o intuito de sua concepção. Assim, não é muito utilizado em produtos profissionais. Por esse motivo, a Raspberry Pi foi utilizada como um ambiente de prova de conceito, de teste e de desenvolvimento, uma vez que o código produzido nela pode ser migrado para outra plataforma com Linux embarcado.

O primeiro passo na utilização de uma Raspberry Pi é a instalação de um sistema operacional. Foi escolhida a distribuição Linux Raspbian como sistema operacional, por ser tomada como o sistema operacional oficial suportado pela Raspberry Pi, com base

na distribuição Debian. O Raspbian já possui Python, Java, Mathematica dentre outros pacotes. A imagem da versão mais atual do Raspbian encontra-se no site da Raspberry Pi. É necessário um computador para baixar a imagem e gravá-la em um cartão SD a ser inserido na placa.

Para a gravação da imagem no cartão SD, foi utilizada a ferramenta "Etcher" para Linux, sugerida pela própria Raspberry Pi. Essa ferramenta tem interface gráfica e é bastante intuitiva. Basta selecionar o arquivo de imagem baixado e o driver no qual o cartão SD está conectado. Ao inserir o cartão na Raspberry Pi 3 e dar o primeiro *boot* do sistema, o usuário passa por algumas telas de configurações iniciais simples e rapidamente o sistema já está operacional.

4.1.1 Código teste em Python

A primeira coisa a ser implementada é a leitura dos sinais BLE, mais especificamente dos pacotes de *advertise*. Para tal, inicialmente foi utilizado a linguagem de programação Python com suas bibliotecas *bluetooth*, *bluez* e *hcitool*, para validar a leitura de Bluetooth da Raspberry Pi. Essas duas primeiras bibliotecas disponibilizam funções relacionadas à mensagem transmitida via Bluetooth e seus protocolos. A última comunica via uma porta HCI (*Host Controller Interface*) com os dispositivos Bluetooth, que nesse caso é o chip Bluetooth da Raspberry Pi. A biblioteca *socket* é utilizada para realizar uma conexão TCP de comunicação com o servidor.

Inicialmente, o código tenta acessar o driver Bluetooth embarcado da Raspberry Pi, passa os parâmetros de leitura que foram configurados e habilita o serviço de leitura através das seguintes linhas:

```
1 try:
2     ble_sock = bluez.hci_open_dev(dev_id)
3     blescan.hci_le_set_scan_parameters(ble_sock)
4     blescan.hci_enable_le_scan(ble_sock)
5
6 except:
7     print "error accessing bluetooth device..."
8     sys.exit(1)
```

Os parâmetros de IP, porta e tamanho do *buffer* são definidos e em seguida é feita a conexão com o servidor TCP:

```
TCP_IP = '192.168.XX.XX'
TCP_PORT = 10005
BUFFER_SIZE = 1024

try:
    tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = (sys.argv[1], int(sys.argv[2]))
    tcp_sock.connect(server_address)
```

Então é iniciado o processo de leitura. Para evitar sinais Bluetooth indesejáveis que possam estar no ambiente, foi implementado um filtro que compara o MAC presente em cada pacote de *advertise* recebido com uma lista de MACs dos sensores, de acordo com linhas de código abaixo. Acompanhando pela linha dois, o programa compara se está na lista de MACs *registered_beacons* a informação contida no espaço de 0 até 17 na variável *beacon* (12 hexadecimais de MAC separados por 5 ":" a cada par de hexadecimal). O MAC geralmente é utilizado com caixa alta, porém a leitura da biblioteca *bluez* retorna valor da mensagem somente com caracteres em caixa baixa, por isso há a necessidade da função *upper*, ainda na linha dois, para ser feita a comparação. Se o MAC constar na lista, a mensagem é enviada para um servidor TCP, de acordo com a linha 3.

```
    for beacon in beacon_list:
        if beacon[0:17].upper() in registered_beacons:
            tcp_socket.sendall(beacon)
```

O programa funcionava de maneira satisfatória quando eram feitos testes com dois ou três beacons. Porém, ao aumentar a quantidade de beacons para aproximadamente 20, era observada uma grande quantidade de perda de pacotes. Essas perdas se davam principalmente por dois principais problemas.

O primeiro problema que foi levado em conta foi a linguagem de programação utilizada para fazer essa leitura. Segundo Henrique Bastos (BASTOS, 2008), o Python, assim como Java e outros, é uma linguagem interpretada, o que significa que seu código fonte é convertido para uma linguagem intermediária para depois ser interpretado por ferramentas dessa própria linguagem durante a execução do programa. Esse processo de interpretação é mais lento que a execução de códigos binários compilados de linguagens como C e C++. Para diversas aplicações, essa diferença de velocidade é desprezível. Porém, qualquer lentidão na execução do código poderia afetar fatalmente a solução proposta, gerando falsos alarmes ou ignorando alarmes reais. Dessa forma, optou-se por desenvolver um código em C para fazer a leitura de Bluetooth.

O segundo problema é que as tarefas não estavam sendo feitas de maneira independente, de forma que enquanto o programa fazia o envio TCP dos dados Bluetooth para o servidor, ele momentaneamente interrompia a função de leitura e retornava a realiza-la quando o envio de um pacote fosse finalizado. Mesmo que esse período de envio TCP fosse curto, ele interferia na leitura total de pacotes por ser algo muito recorrente durante a execução. Como solução para isso, decidiu-se por fazer a utilização de *threads*, ou seja, tarefas independentes, para que a recepção de dados seja feita simultaneamente ao envio deles para o servidor e não de maneira sequencial.

4.1.2 Solução desenvolvida na Raspberry Pi

O programa principal do gateway foi mantido em Python. O código em C proposto foi desenvolvido apenas para a função de leitura, que é a parte de execução onde mais há demanda por velocidade de processamento. O programa principal possui três principais seções, ilustradas na Figura 21:

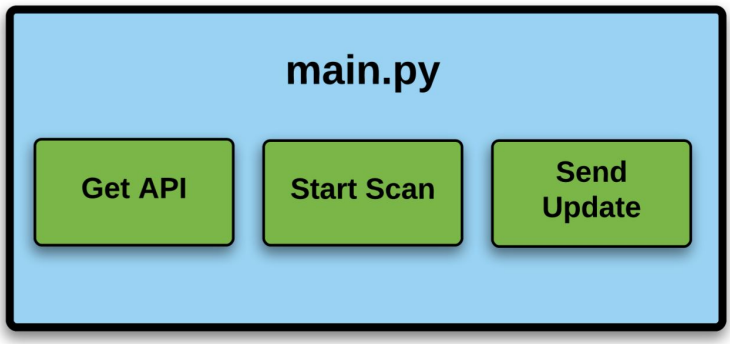

Figura 21 – Seções do código principal. Fonte: Autor

Inicialmente, ao rodar o programa *main.py*, é estabelecida uma comunicação com o servidor, em HTTP, e a seção "*Get API*" é iniciada. Ela manda uma requisição para a API que responde com 3 arquivos:

- sensors.txt : esse arquivo contém uma lista de MACs de beacons que são associados ao gateway que fez a requisição.
- config.txt : contém algumas configurações que possam ser interessantes de se repassar para o gateway, como filtro de recepção com um RSSI limite e afins. Por enquanto, não está sendo passada nenhuma configuração por esse arquivo, mas o serviço já está pronto caso futuramente seja necessário.
- token.txt : contém uma chave de autenticação do gateway, para certificar que apenas gateways autorizados possam se comunicar com a API.

Após essa requisição, o programa se divide em duas principais tarefas. Como já foi dito, chegou-se a ideia de utilizar *threads* para realizar tarefas simultâneas dentro do código. Assim, é aberta uma *thread* para a seção denominada "*Start Scan*" e uma para a seção "*Send Update*". A inicialização das *threads* é feita pela função *Thread* que tem como argumento a função ("*target*") que essa *thread* irá executar e os argumentos ("*args*") que deve passar para essa função. A função *.sleep* entre a inicialização e execução das duas *threads* garante que a seção "*Send Update*" não passe dados vazios para a API, esperando que ocorra a leitura de beacons antes de ser executada.

```
1 #Inicia thread de leitura de beacon
2 t1 = threading.Thread(target=socketBeacon, args=(sensores,args))
3 t1.start()
4
5 time.sleep(20)
6
7 #Inicia thread de envio para API
8 t2 = threading.Thread(target=sendForAPI, args=(mac,config_id, type_id,
      updateInterval))
9 t2.start()
```

Na seção "Start Scan", é chamado o código em C responsável apenas pelo leitura BLE em si. Essa chamada acontece da seguinte forma:

```
1     os.system("./scan 127.0.0.1 8888 &")
```

Isso dá um comando ao sistema operacional para que execute o programa *scan* que se comunica via socket com o programa principal em Python enviando os dados lidos para o IP e porta indicados. O caractere " &" ao final da linha de comando indica que o programa *scan* deve ser executando em *background*, ou seja, o código em Python não espera o programa *scan* retornar um valor para continuar sua execução.

A leitura em C foi de implementação muito mais complexa que o que havia previamente sido feito em Python. Em C, a maioria dos parâmetros de leitura e da comunicação via socket precisam ser configurados à mão, assim como algumas funções devem ser implementadas do zero. Em Python existe muito material já implementado, o que facilita o uso dessa linguagem. A título de comparação, apenas o código em C de leitura era cerca de 4 vezes maior, em quantidade de linhas, que o programa inicial de teste em Python. A parte principal do programa de leitura em C, de envio dos dados recebidos para a API, encontra-se a seguir:

```
            char MAC[18];
            ba2str(&(info->bdaddr), MAC);
            offset = info->data + info->length + 2;
            char *str;
            asprintf(&str,"%s,%02X%02X,%02X,%d", addr, info->data[10],
                 info->data[9],info->data[11],(uint8_t)info->data[info->length]);

            /* send the message line to the server */
               n = write(sockfd, str, strlen(str));

            if (n < 0)
                    error("ERROR writing to socket");

            free(str);
```

Esse *loop* basicamente escreve em uma *string* o MAC do beacon e as informações contidas no *advertise* do beacon. Essa escrita ocorre através da função *asprintf* que aloca espaço para uma *string*, inclui a terminação de caractere nulo e retorna um ponteiro para essa *string*. A sintaxe de escrita "%02X" é utilizada para se escrever os números hexadecimais que carregam as informações no *advertise* dos beacons.

Essa *string* é então enviada via socket TCP para o programa em Python, que continua em execução nesse meio tempo uma vez que o programa de leitura é executado em *background*. Ao final do loop, é utilizada a função *free* para liberar a memória alocada para a *string* e processo é feito a cada evento BLE detectado pelo HCI, ou seja, a cada sinal BLE que o receptor da Raspberry Pi detecta. Dessa forma, não há nenhum filtro de recepção. Todo e qualquer sinal recebido é enviado para a seção "Star Scan".

A seção "Start Scan" recebe as informações via socket do programa de leitura com a função *recv*.

```
    data = conn.recv(1024)
```

Para que as diferentes *threads* tenham acesso aos mesmos dados, foi pensado em um esquema de escrita e leitura de arquivos. Existirá um arquivo para cada beacon registrado no gateway. A cada *advertise* recebido, será aberta uma *thread* com a função de escrever os dados contidos nesse *advertise* no arquivo correspondente ao beacon. Nas linhas de código abaixo, vemos a inicialização das *threads* que irão escrever os dados recebidos em arquivos. A função *atualizarArquivos* é a responsável por essa escrita dos dados nos arquivos. A inicialização da *thread* é feita pela mesma estrutura já abordada:

```
1        t = threading.Thread(target=atualizarArquivos,
             args=(data,lista,gateway))
2        t.start()
```

A função de atualização de arquivos cria um arquivo para cada beacon presente na lista *sensors.txt* fornecida pela API e gera um arquivo de texto com o MAC do beacon, o nível de bateria em volts, o estado do botão, o RSSI e o horário no qual o *advertise* foi recebido. Portanto, essa é a fase de filtragem de sinais. O MAC de um *advertise* recebido do código de leitura é comparado com a lista *sensors.txt* e, se esse MAC estiver presente na lista, o arquivo é atualizado com os dados. Se não, os dados são simplesmente ignorados. Como exemplo, temos o seguinte arquivo do beacon E1:8B:FE:32:0B:58.

---

e18bfe320b58

3.1V

0000

182

2017-10-31 15:38:47

O estado do botão é indicado por "FFFF" se o campo de botão do *advertise* do beacon conter "01" e "0000" se o *advertise* conter "00". Caso o *advertise* recebido esteja com a mensagem de "00", os dados são escritos no arquivo e o programa continua sua rotina normal. Caso seja detectada uma mensagem de botão "01", o arquivo é escrito e em seguida é chamada uma função da próxima seção para enviar um alerta imediato para a API.

Com os arquivos atualizados, a *thread* com a seção "*Send Update*" passa a lê-los constantemente. Ela checa os dados dos arquivos e monta a mensagem que será enviada para a API. A comunicação HTTP com a API acontece por requisições e respostas no formato JSON, uma forma de estruturar e organizar os dados a serem enviados. Nessa seção, os arquivos de cada beacon são abertos e lidos. São criadas duas classes: sensores e gateways. essas classes são em seguida convertidas para o formato JSON, da seguinte forma:

```
1  class sensor(object):
2      def __init__(self, mac, _type, battery, value, rssi, _time ):
3          self.mac = mac
4          self.type = _type
```

```
        self.batteryLevel = battery
        self.value = value
        self.rssi = int(rssi)
        self.LastUpdate = str(_time)

class gateway(object):
    def __init__(self,mac,relay,config,_type,sensors):
        self.mac = mac
        self.relayState = relay
        self.gatewayConfigurationId = int(config)
        self.typeGatewayId = int(_type)
        self.sensors = sensors

    #metodo que transforma classe em json
    def toJSON(self):
        return json.dumps(self, default=lambda o: o.__dict__,
            sort_keys=True, indent=4)
```

A Figura 22 mostra um diagrama simplificado da solução:

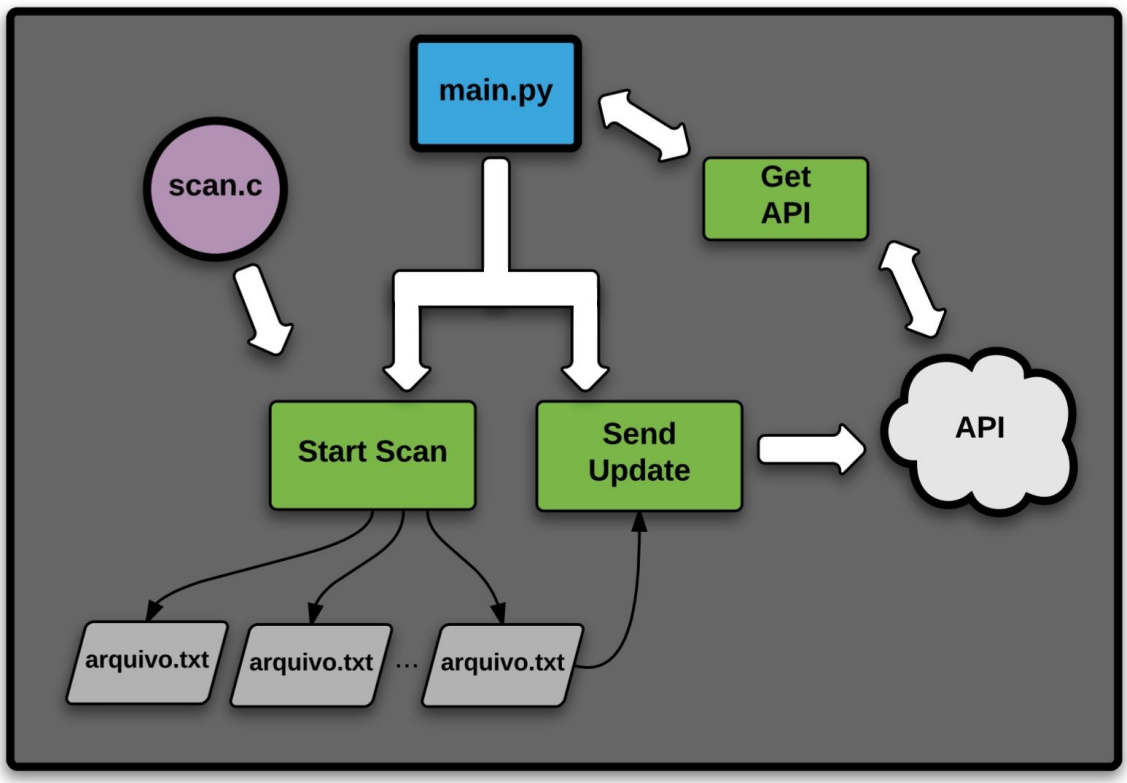

Figura 22 – Diagrama do funcionamento do gateway, relação entre os códigos, arquivos e API. Fonte: Autor

Foi criada uma variável com o nome *updateInterval* que é responsável por ditar a periodicidade das transmissões do gateway para a API. No momento, essa variável está com o valor 10, o que significa que a cada 10 segundos a API recebe a informação atualizada que consta nos arquivos. As únicas exceções para esse tempo de 10 segundos são : o alerta de botão, explicado no parágrafo anterior, e o alerta de sensor desconectado.

A desconexão de sensor é detectada na própria seção "Send Update". Foi criada uma função que faz a comparação entre o horário atual e o horário presente no arquivo de cada beacon (horário que o *advertise* foi recebido). Se essa diferença foi maior que um tempo T, ou seja, se há T segundo não houve recepção nenhuma desse mesmo beacon, significa que o o beacon provavelmente foi desligado ou encontra-se fora do alcance de recepção. Nesse caso, a mensagem JSON para a API carrega a informação de que se passam T segundo sem recepção desse beacon, para ser gerado um alarme de desconexão de sensor. A função desenvolvida para analisas a diferença entre os tempos é a seguinte, com o tempo limite para desconexão T na presente na condição da linha 7:

```
1
2  def diff_dates(date1, date2):
3      f = '%Y-%m-%d %H:%M:%S'
4      dif = (datetime.strptime(date2, f) - datetime.strptime(date1,
           f)).total_seconds()
5      try:
6          print dif
7          if(dif > T):
8              return False
9          else:
10             return True
11     except Exception,e:
12         print str(e)
13         return True
```

Modo *Offline*

Da mesma forma que o beacon deve fazer transmissões periódicas para que seja detectada uma possível desconexão, existe a necessidade da mesma função no gateway. A desconexão do gateway implica em perda de qualquer evento detectado pelos beacons. Essa desconexão pode acontecer por duas formas: falta de alimentação, no caso do gateway ser retirado da tomada, ou algum problema de comunicação com a Internet, por exemplo uma queda da rede.

Para resolver esse problema, a princípio, utilizou-se o mesmo conceito de mensagem periódica dos beacons. O gateway deve então se comunicar com a API com uma

certa periodicidade, que foi inicialmente definida como 10 segundos. Não houve necessidade de criar uma mensagem específica somente para essa função. A mensagem com as informações do beacons que era enviada a cada alteração ou detecção de alarme, passou a ser enviada também periodicamente. Caso essa mensagem não fosse recebida no intervalo determinado, a API mandaria um sinal de alarme para a interface gráfica de usuário para que o problema fosse investigado.

Ainda assim, no caso de perda de conexão com a rede e consequentemente com o servidor da API, é possível armazenar os dados e eventos recebidos pelos beacons durante esse período e atualizar as informações do sistema sobre o ocorrido quando a conexão for reestabelecida. Foi desenvolvido então o Modo *Offline*, no qual o gateway operaria na falta de conexão com a rede.

Primeiramente, foi criada um arquivo chamado *desconexão.txt* para ser utilizado como *Flag*, que levaria o conteúdo "offline" ou "online", dependendo do estado de conexão do gateway com o servidor. O gateway tenta enviar uma requisição de comunicação com o servidor e em seguida entra na condição:

```
1   if response.status_code != 200: # Resposta diferente de 200 = comunicacao
        falhou
2      arquivo4 = open('init/connection.txt', 'wr')
3        arquivo4.writelines('offline')
4        arquivo4.close()
5
6     else:
7        arquivo4 = open('init/connection.txt', 'wr')
8        arquivo4.writelines('online')
9        arquivo4.close()
```

Se a comunicação falhou, o arquivo é atualizado com a palavra "*offline*". Se a comunicação foi um sucesso, é atualizado com "*online*". Então, sempre que um alarme é detectado, o programa checa o estado de conexão por esse arquivo. Se o gateway estiver *online*, a mensagem é mandada para a API. Se estiver *offline*, chama-se a função *logOffline*:

```
1 def logOffline(value, mac):
2     with open('logs.txt', "a") as arquivo:
3         _type = []
4         _type.append(str(mac) + ';')
5         _type.append(str(value) + ';')
6         time = strftime("%Y-%m-%d %H:%M:%S", gmtime())
7         _type.append(str(time) + '\n')
8         arquivo.writelines(_type)
```

Ela gera um arquivo *logs.txt*, armazena nele o MAC do beacon, o estado do botão ou a mensagem "Sensor Desconectado" na variável *value* e o horário do alarme com a função *strftime*. Ao ser retomada a conexão com o servidor, o gateway faz requisições para a API e envia todas as linhas de *log* para serem tratadas por ela, ao mesmo tempo em que exclui essas linhas do arquivo *logs.txt*.

Com isso, o código do gateway foi finalizado utilizando uma Raspberry Pi. Ainda, foi desenvolvido um *script* que faz de maneira automática todas as configurações, baixa os códigos dos repositórios e inicializa os programas em uma Raspberry Pi nova que tenha apenas o sistema operacional. O *script* desenvolvido para tal função encontra-se no Anexo C desse documento, com comentários.

4.2 Toradex

Como já foi abordado, a Raspberry Pi, apesar de seu poder de processamento, memória, periféricos e baixo custo, em geral não é utilizada para soluções a serem comercializadas. A Innovix já havia utilizado placas de Linux embarcado em projetos anteriores ao IDView, portanto já tinha contato com a empresa Toradex. A Toradex é uma empresa Suíça com filial no Brasil líder em módulos embarcados com processadores ARM. Suas soluções em Linux embarcado compreendem módulos e placas portadoras (*carrier boards*) como as da figura abaixo.

Os modelos da Figura, módulo Colibri VF50 e portadora Iris, foram os escolhidos para serem utilizados no produto (vide Figura 23). O módulo Colibri FV50 é baseado no chip "*Freescale Vybrid embedded SoC*", possui um processador Cortex-A5 com clock de até 400MHz, 128MB de memória RAM e 128Mb de memória Flash. É possível ver que os recursos de memória desse modelo da Toradex são mais escassos que os da Raspberry Pi. A Toradex possui módulos muito mais potentes. No entanto, escolheu-se esse modelo devido ao custo que um modelo superior iria acarretar ao produto.

A placa escolhida possui chip WiFi e entrada Ethernet para fazer a comunicação com o servidor da API. Porém, não possui chip Bluetooth. Até o final desse ano, a Toradex pretende lançar uma versão da placa Colibri com ambos os chips WiFi e Bluetooth 4.0. Como solução para a falta do chip Bluetooth no módulo atual, foi utilizado um Dongle Bluetooth 4.0 na entrada USB da placa. Futuramente, serão feitas análises de diferentes Dongles para avaliar sua performance, mas em desenvolvimento foi utilizado o Dongle BLE ABT40 da fabricante Vinik (vide Figura 24).

O código para a solução já havia sido desenvolvido na Raspberry Pi e foi então migrado para a Toradex. Porém, antes disso, precisaram ser feitas algumas preparações.

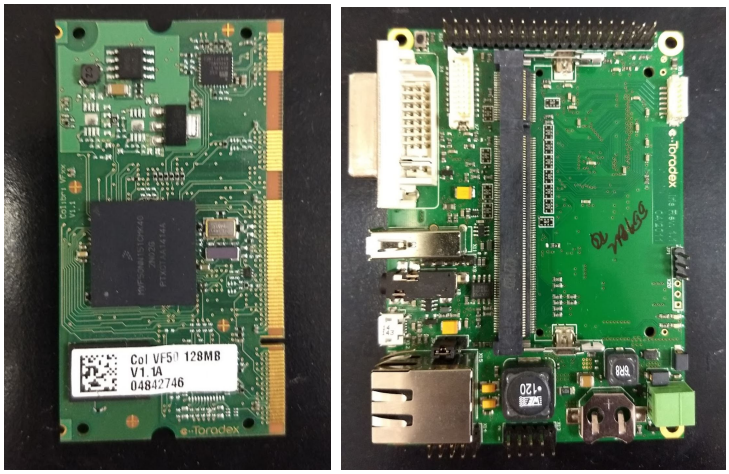

Figura 23 – À esquerda, o módulo Colibri. À direita, placa portadora Iris. Fonte: Imagens cedidas pela Innovix.

Figura 24 – Dongle da Vinik. Fonte: Imagem cedida pela Innovix.

4.2.1 Preparar o Linux para a Toradex

Com a pouca memória da placa, é necessário um sistema operacional o mais enxuto possível, que contenha apenas o essencial para o código do gateway ser executado. A Toradex fornece a distribuição *OpenEmbedded Linux*, que foi desenvolvida especialmente para seus módulos. Para que o sistema rode de maneira eficiente mesmo com a limitação de memória, muitos drivers e funcionalidades do Sistema Operacional vêm desabilitadas por padrão nas configurações do Kernel. Algumas dessas configurações precisavam ser alteradas para possibilitar o funcionamento da solução, como a habilitação da interface de rede para possibilitar a conexão com a Internet e a habilitação do HCI para a comunicação do chip com o Dongle BLE e das demais funcionalidades Bluetooth.

Além disso, para rodar o código na extensão .py era necessário compilar o Python para a arquitetura ARM da Toradex. Não é possível fazer a compilação na Toradex, pois o *OpenEmbedded Linux* não possui um compilador, mais uma vez devido à limitação de memória desse modelo. Foram feitas algumas tentativas de se compilar o Python externamente à Toradex utilizando ferramentas de compilação cruzada, em um computador com arquitetura x86. A compilação era bem sucedida, mas ao passar para a Toradex e tentar executar um código em Python surgiam erros na execução do binário.

A conclusão em que chegou-se foi que existiam dependências que o Python compilado não conseguia encontrar pois o caminho até elas tinha sido definido durante a compilação, no computador de arquitetura x86. Esse problema poderia ser solucionado com mais algumas tentativas mas, o suporte da Toradex no Brasil nos forneceu uma imagem do *OpenEmbedded Linux* já com um Python compilado. Portanto, era necessário somente instalar esse sistema operacional na Toradex e fazer as configurações de Kernel para habilitar o que fosse necessário.

Para instalar o Sistema Operacional na Toradex, foi necessário um cartão SD de mais de 1GB formatado em FAT e um cabo conversor de comunicação serial para USB para conectar a Toradex a um PC x86 "*host*" e passar os comandos de preparação. No cartão SD formatado, foi gravada a imagem do *OpenEmbedded Linux* com o Python, fornecida pelo equipe de suporte da Toradex. No PC "*host*", os seguintes comandos de configuração e instalação de ferramentas foram previamente executados, de acordo com o tutorial de instalação de OS da Toradex:

```
1 sudo dpkg --add-architecture i386
2 sudo apt-get update
3 sudo apt-get install dosfstools e2fsprogs gawk mtools parted
4 sudo apt-get install zlib1g:i386 liblzo2-2:i386 libuuid1:i386 libusb-1.0-0:i386
```

Então, a Toradex foi conectada ao PC com o cabo serial em uma conexão UART à taxa 115200bps e em seguida foi ligada. O *prompt* de comando do *bootloader* pôde ser então visualizado. Em seguida foram passados os comandos *run setupdate* e *run update* completar a instalação do S.O. na placa.

Para fazer as habilitações necessárias no Kernel, foi preciso baixar código fonte do Kernel, fazer as devidas configurações, fazer a compilação e atualizar esse novo Kernel no S.O. da Toradex. O código fonte do Kernel disponibilizado pela Toradex foi baixado em um PC com Linux com o uso da ferramenta de controle de versão Git, utilizando o seguinte comando:

```
1 git clone -b tegra git://git.toradex.com/linux-toradex.git
```

A Toradex fornece, junto com o código fonte, um arquivo de configuração padrão de seus módulos. Para aplicar essas configurações, basta utilizar o comando *make* seguido do nome de arquivo de configuração. Como o intuito é fazer uma configuração própria para a solução proposta, foi criado arquivo de configuração próprio ao invés de utilizar o fornecido pela Toradex, através do comando a seguir.

```
1 make menuconfig
```

Em seguida, a interface de configuração do kernel é inicializada (Figura 25).

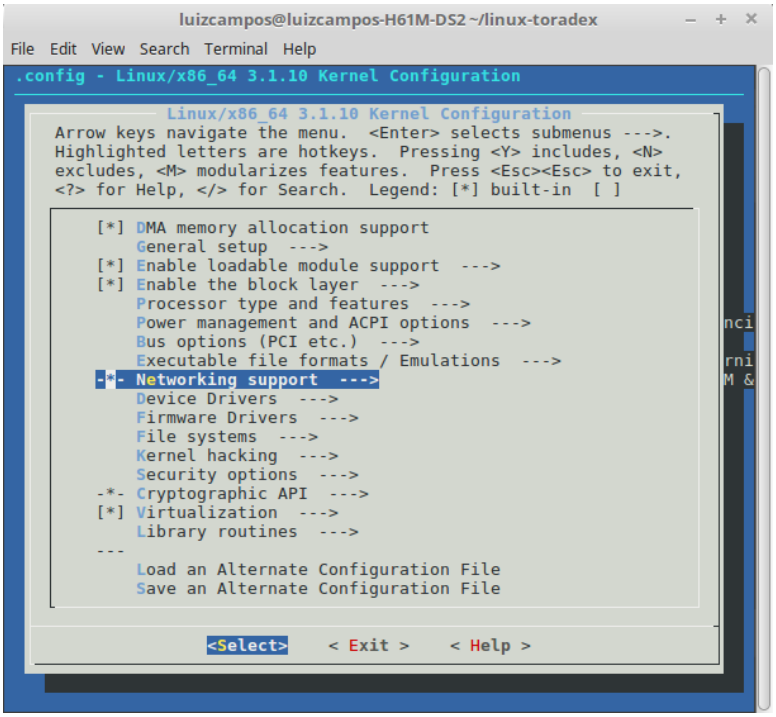

Figura 25 – Interface inicial de configuração do Kernel. Fonte: Imagem cedida pela Innovix.

Um arquivo de configuração inicializado do zero, por padrão, já tem habilitadas algumas das propriedades que estavam desabilitadas na versão enxuta da Toradex, como a interface de rede. Desse modo, as únicas funcionalidades que precisaram ser habilitadas foram as relacionadas ao Bluetooth. No menu de configuração, em "*Networking support*" deve-se habilitar o "*Bluetooth subsystem support*" apertando a tecla "y" ou a barra de espaço até que a opção fique com um asterisco, como é possível observar na Figura 26. Com o asterisco, a opção será inclusa no kernel durante a compilação.

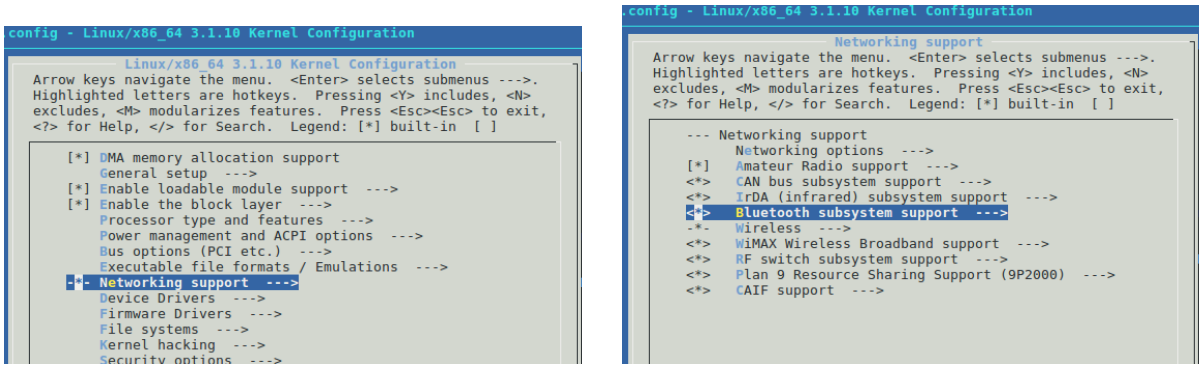

Figura 26 – À esquerda, o submenu "*Networking support*" é acessado para habilitar a aba "*Bluetooth subsystem support*", à direita. Fonte: Imagens cedidas pela Innovix.

Dentro dessa aba, é preciso entrar em "*Bluetooth devices driver*" e se certificar que a opção "*HCI USB driver*" está também inclusa, para a utilização do Dongle Bluetooth (Figura 27).

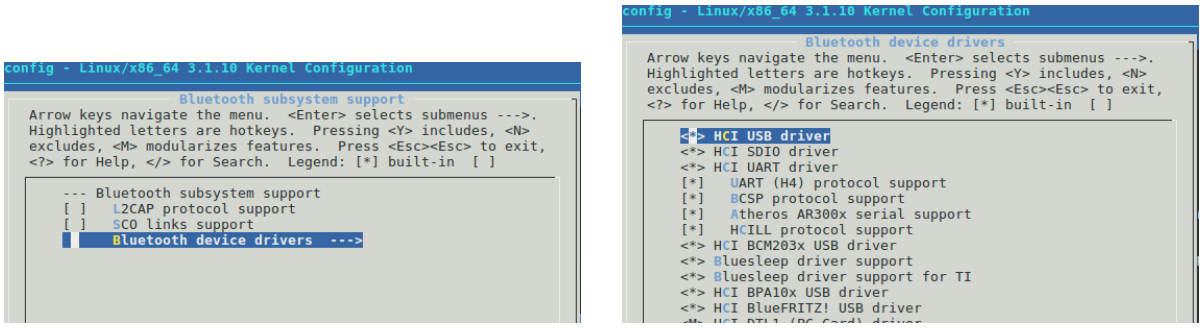

Figura 27 – Habilitação do HCI USB no diretório de *drivers*. Fonte: Imagens cedidas pela Innovix.

Em seguida, basta utilizar a opção "*exit*" recursivamente para voltar à tela inicial e sair do menu de configuração e salvar esse arquivo de configuração.

Depois da configuração, o próximo passo é compilar o Kernel. Para a compilação, é necessário instalar um pacote de *Device Tree Compiler* (DTC), que é nada mais que uma estrutura de dados em byte que auxilia o Kernel durante sua inicialização. É aconselhado pela Toradex uma versão de DTC de 1.3 ou superior. Para fazer a instalação, foi utilizado o seguinte comando:

```
1 sudo apt-get install device-tree-compiler
```

E então foi feita a compilação do Kernel e do DTC referente a placa Colibri VF50 utilizada:

```
1 make -j3 zImage | tee build.log
2 vf500-colibri-eval-v3.dtb
```

Com um cartão SD preparado, foram copiados os arquivos compilados de Kernel e DTC. Ao inserir o cartão na Toradex, inicializa-la e esperar o *prompt* de comando do bootloader, são passados os seguintes comandos para atualizar o Kernel:

```
1 run setupdate
2 run prepare_ubi
3 run update_fdt
4 run update_kernel
```

Dessa forma, a Toradex encontrava-se devidamente preparada para receber o código desenvolvido na Raspberry Pi. A única alteração no código foi o incremento de

algumas linhas no *script* de inicialização do progama, encontrado na pasta /etc/init.d/ para ser executado após o boot do sistema, para inicializar o serviço de Bluetooth. Esse passo não foi necessário na Raspberry Pi pois é feito de forma automática.

```
#start bluetooth
systemctl enable bluetooth.service
systemctl start bluetooth.service
rfkill unblock all
hciconfig hci0 up
```

Assim, o firmware dos gateways foi concluído e iniciou-se os períodos de testes e ajustes.

As informações recebidas do gateway pela API são processadas, enviadas para dois bancos de dados e para um *web service* no qual existe uma interface para o usuário do produto. Os dois bancos de dados são um destinado à API e outro destinado ao *web service*. O desenvolvimento da API, da interface gráfica e a comunicação com os bancos de dados foram implementadas por outros membros da empresa, e portanto não foi foco desse trabalho. Nos Anexo D deste documento, encontram-se algumas imagens e breve comentários sobre a API e a interface de usuário.

5 Medidas e Testes

5.1 Testes de alcance com os dispositivos Ingics

O primeiro estudo feito com os dispositivos da Ingics foi um teste de alcance em linha reta sem obstáculos, no qual uma gateway foi mantido fixo e o beacon foi sendo deslocado em uma linha de 15 metros e a cada metro foi medido o RSSI indicado pelo gateway. Os resultados encontram-se na Tabela 3 e na Figura 28 em forma gráfica.

Tabela 3 – Tabela de resultados do teste de alcance em linha reta sem obstrução.

Distância [m]	0	1	2	3	4	5	6	7
RSSI [dBm]	-1	-65	-69	-73	-70	-75	-78	-78

Distância [m]	8	9	10	11	12	13	14	15
RSSI [dBm]	-75	-78	-81	-82	-80	-79	-82	-81

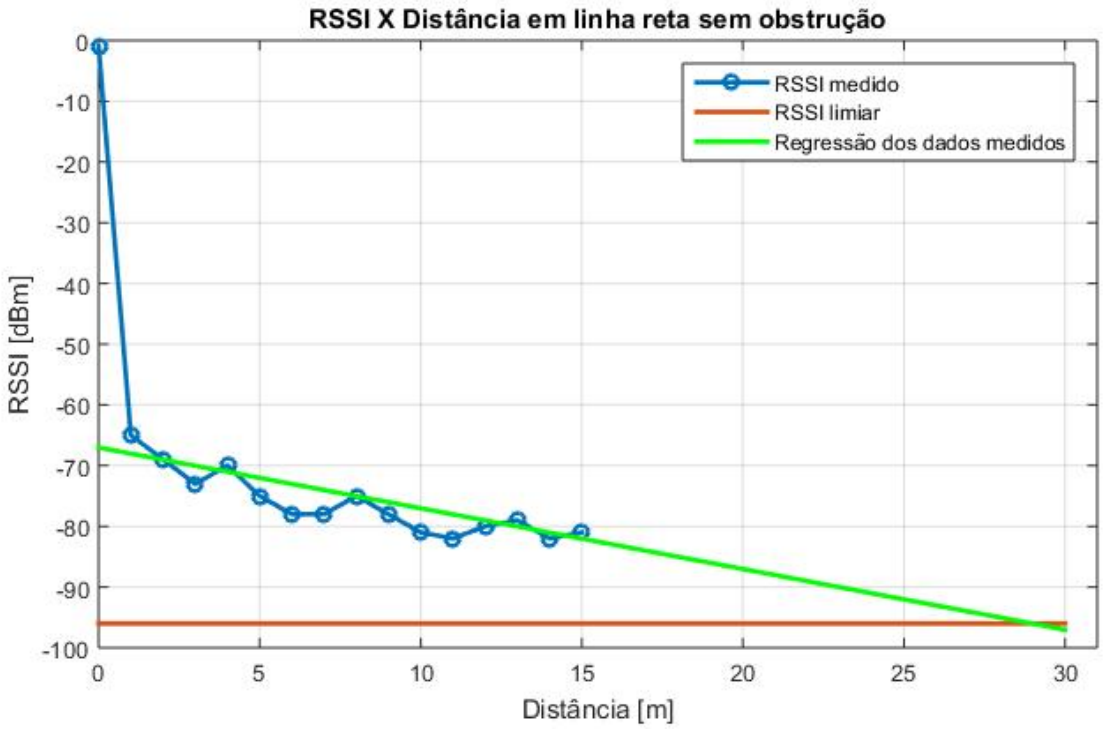

Figura 28 – Resultado gráfico do estudo de alcance em linha reta. Fonte: Autor

Por questões físicas do local onde o estudo foi realizado, não se pôde chegar à distância limite de 30 metros de alcance informado pela fabricante, de forma que o estudo foi feito apenas até 15 metros. Porém, ao plotar uma reta de regressão dos valores medidos numa tentativa de simular testes com distâncias maiores, observa-se que ela se encontra

com a reta de sensibilidade próximo ao ponto de 29 metros. Então, é razoável concluir que o alcance de 30m indicado pela Ingics é possível, para o caso sem obstáculos. A reta com o valor de sensibilidade do Gateway está presente no gráfico apenas como referência já que, por definição, não é possível medir valores abaixo dele.

Um outro estudo realizado foi do alcance do gateway de beacons espalhados nas imediações externas do local da empresa. A Figura 29 ilustra o posicionamento do gateway e dos beacons com os respectivos valores de RSSI. É possível ver que o alcance acaba sendo menor que 30m para esse caso, porém isso já era esperado visto que nesse teste houve a obstrução da linha de visada por paredes, equipamentos eletrônicos e diversos outros atenuadores e geradores de interferência.

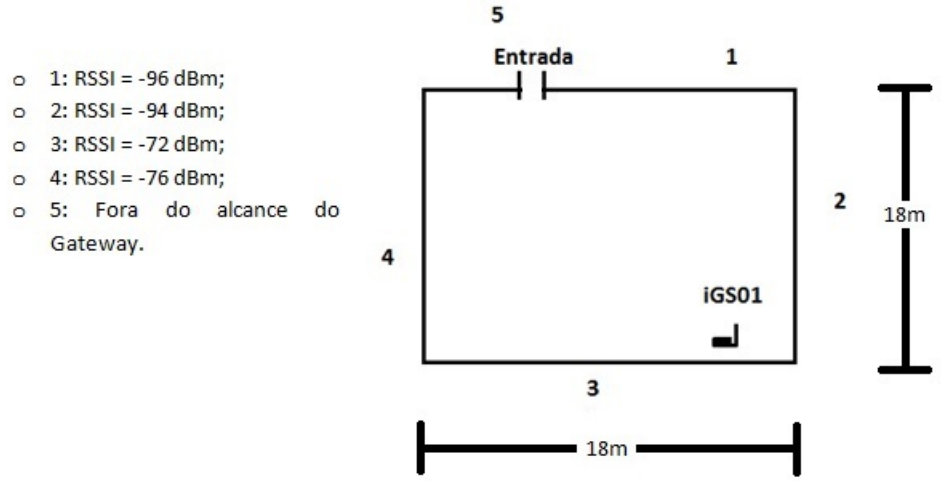

Figura 29 – Resultado de estudo de alcance na área externa à empresa. Fonte: Autor

5.2 Teste de *performance* de Beacons Ingics variando a alimentação

Esse teste foi realizado no Laboratório de Circuitos Elétricos no SG11, Universidade de Brasília. Foram utilizados uma fonte *Agilent Triple Output DC Power Supply E3631A* com saídas de 0-6V 5A e +/- 25V 1A e um multímetro *Agilent 6 1/2 Digit Multimeter 34410A* (Figura 30).

Foram analisados 5 Beacons Ingics, com o auxilio de um Gateway Ingics para a recepção dos pacotes BLE (Figura 31).

O intuito do teste foi definir qual o limiar inferior de tensão para o funcionamento dos Beacons da Ingics e analisar se esse funcionamento entre a tensão nominal e a tensão limiar corresponde às expectativas de funcionamento normal do produto.

Para isso, foi definida a seguinte metodologia:

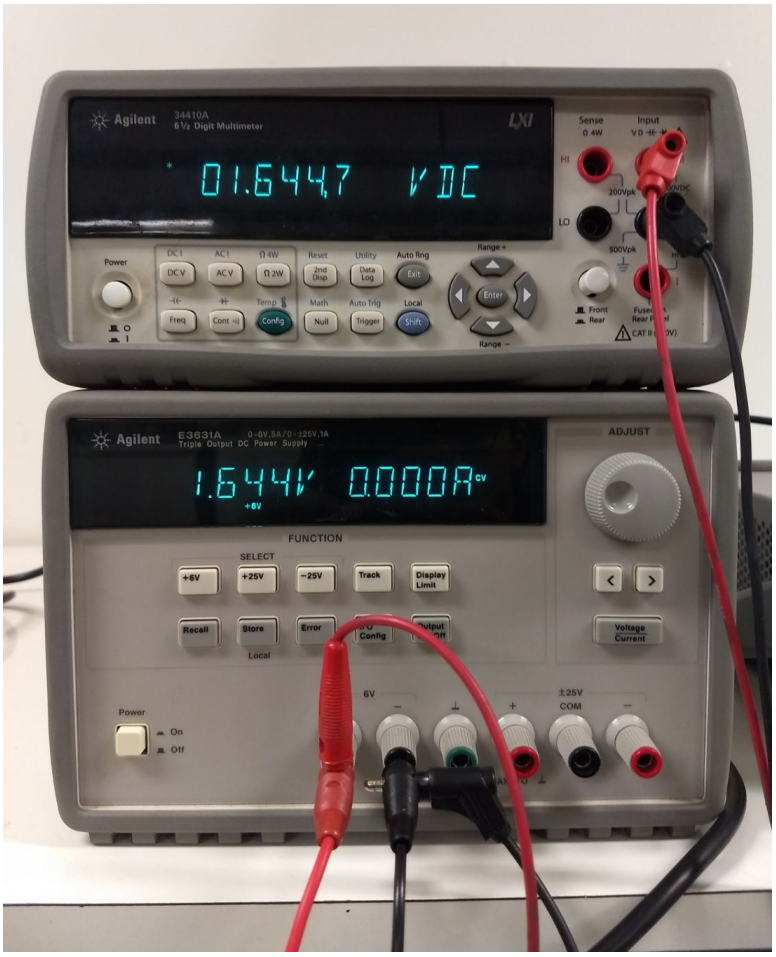

Figura 30 – Em baixo, a fonte E3631A. Acima, o multímetro 34410A. Fonte: Imagens cedidas pelo autor.

- Todos os Beacons foram alimentados ao mesmo tempo, com uma tensão inicial de 3.3V (tensão nominal dos beacons). Todos estavam configurados com a mesma potência de transmissão (4dBm) e o mesmo intervalo de transmissão (5s);

- Foram mantidos sob essa tensão por 5 minutos, e a cada minuto o botão de cada Beacon era apertado;

- Depois dos 5 minutos, a tensão foi decrescida em 100mV e repetiu-se o mesmo processo;

- O Gateway Ingics utilizado para a recepção estava configurado como servidor TCP e estava filtrando apenas Beacons Ingics.

- Foi utilizado o *software* Hercules como cliente TCP para gerar um arquivo de *log* com todas os pacotes recebidos dos 4 Beacons pelo Gateway durante o período de teste.

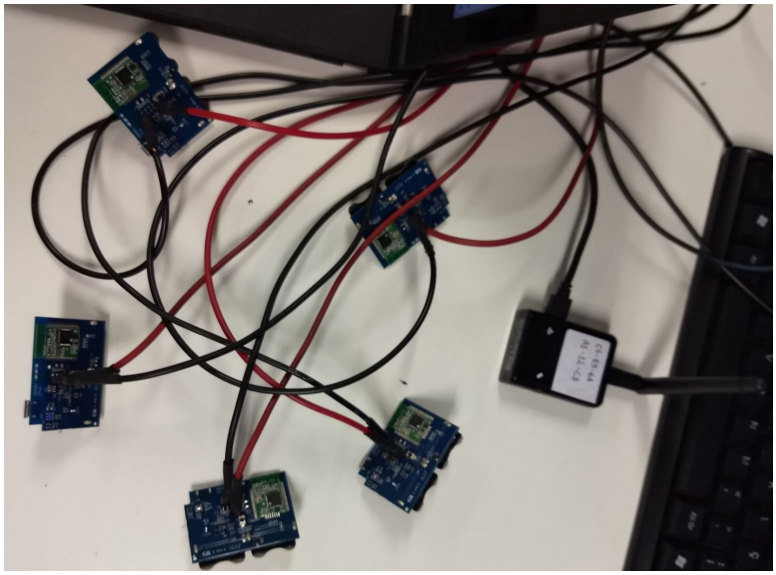

Figura 31 – Beacons em teste, conectados à fonte Agilent. Fonte: Imagem cedida pela Innovix.

Com o arquivo de *logs*, para facilitar a análise, foi utilizado o seguinte comando no terminal do Linux para separar o arquivo em 5 arquivos, um para os pacotes de cada um dos 5 Beacons:

```
1 cat logs.txt | grep -i 'MAC_BEACON' > arq_1.txt
```

O arquivo total de *log* tem aproximadamente 4000 pacotes (linhas), portanto não será incluso neste documento. A tensão foi decrescida em passos de 100mV. Ao chegar em 1,6V, não foi observada atividade dos Beacons. Então testou-se valores entre 1,7V e 1,6V para chegar-se ao valor mais aproximado do limiar. Abaixo, encontra-se a última secção do *log*, quando a tensão aplicada era de 1,654V. Percebeu-se que a recepção de pacotes se manteve dentro da normalidade nessa última fase do teste.

Assim, pôde-se concluir que a constância da transmissão de pacotes dos beacons não é influenciada pela tensão aplicada, desde que essa esteja no intervalo de 3.3V e aproximadamente 1.65V. Da mesma forma, houve funcionamento normal do alarme de botões, até essa mesma tensão limite. A única diferença em relação às funcionalidades do dispositivo foi a diminuição do brilho do *LED* de alarme.

Também foi observado que abaixo de 2,2V o Beacon alerta status de bateria fraca. Ao chegar no passo de 2,2V, o Beacon pisca o *LED* de alarme a cada 2 segundos e permanece nesse estado até parar de apresentar atividade por falta de alimentação. Essa funcionalidade não se encontra documentada pela Ingics.

5.3 Testes de bateria do Smart Beacon Nordic Semiconductor

Os testes de bateria tiveram início com a placa de desenvolvimento. A durante a programação e os testes, a placa era constantemente alimentada via USB. Para teste, foi removido o cabo USB para que a placa funcionasse pela alimentação de sua bateria CR2032. Em dois diferente testes, a bateria da placa caiu aproximadamente 30% em questão de 3 horas, o que é um resultado completamente inaceitável para um dispositivo *low power* que deveria durar até anos com uma bateria. Chegou a ser considerado, então, que talvez a placa de desenvolvimento não tivesse a consumo de energia tão minimizado quando o beacon em si, devida a quantidade substancialmente maior de funcionalidades presente nela.

Os testes de bateria no beacon, após a programação com o firmware desenvolvido, tiveram início no dia 25 de maio de 2017. A bateria utilizada nesse teste é a que estava presente no beacon desde os primeiros testes com o aplicativo *default*. O multímetro media 2.7V e o pacote de *advertise* transmitia um nível de 23% de bateria. No dia seguinte, 26 de maio de 2017, o beacon já não mostrava nenhuma atividade. Nesse mesmo dia, uma sexta feira, foi utilizada uma bateria nova e o beacon foi deixado em funcionamento com a bateria durante o final de semana, com o beacon transmitindo inicialmente um nível de 100%. Na segunda feira, dia 29 de maio, o beacon não apresentava mais atividade. Esses resultados não faziam sentido levando em conta a proposta *low power* do dispositivo, então era certo que alguma coisa estava causando um consumo excessivo de energia.

A primeira coisa a ser considerada foi que poderia haver algo no código causando algum processamento desnecessário que portanto estaria implicando em um consumo anormal de bateria pelo chip. Dias de análise do código foram inconclusivos quanto a essa suposição. Chegou-se a questionar também a qualidade das baterias utilizadas.

A solução foi encontrada em contato direto com os engenheiros da Nordic. A Nordic possui em seu site um meio de interação direta entre desenvolvedores e seus engenheiros, para perguntas mais específicas e que muitas vezes de informações que não podem ser expostas em fóruns abertos como é o caso da plataforma *Developer Zone*, muito utilizada para solucionar outras dúvidas durante o desenvolvimento. Foi aberto um caso no site a ser analisado por um engenheiro do Nordic no qual foi anexado também o código desenvolvido. A resposta recebida do engenheiro foi de um problema que não havia sido sequer considerado: pinos de entrada que estavam flutuando (desconectados).

Em resumo, o problema é que ruídos eletromagnéticos e até mesmo descargas eletrostática poderiam estar induzindo os pinos de programação/*debug* a serem ativados, de modo a deixar o beacon em constante modo de debug, pelo fato da PCB estar exposta. O mesmo aconteceu com a placa de desenvolvimento, que possui os pinos de programação/*debug* também em uma PCB que está exposta. Nos testes com o beacon, a morte

da bateria era ainda mais rápida tanto pela bateria CR1632 do beacon ser menor que a bateria CR2032 da placa quanto pelo fato de os *jumpers* soldados aos pinos de programação/*debug* do beacon estarem provavelmente potencializando o efeito de interferência, funcionando como antena.

Foi sugerido pelo engenheiro da Nordic, então, que fosse utilizado um resistor de *pull-down* de 1kohm do pino de programação SWDCLK (*Serial Wire Debug Clock*) para o *ground* (0V) pois a interferência nesse pino em específico é que estaria causando a ativação do modo *debug* no chip do beacon. Foi feita a conexão do pino SWDCLK com o ground da própria PCB do beacon utilizando o resistor de 1kohm sugerido (vide Figura 32) e deu-se início a novos testes.

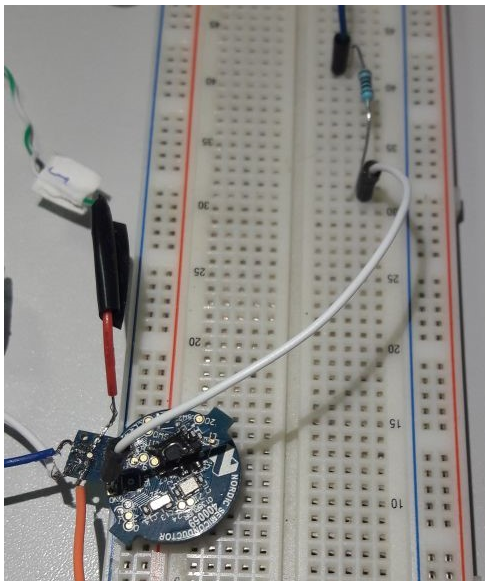

Figura 32 – Conexão do pino de programação SWDCLK ao ground utilizando um resistor de pull-down de 1kohm (parte superior direita). Fonte: Imagem cedida pela Innovix.

Com essa configuração, após 16 dias de teste, o multímetro ainda mostrava uma tensão de 2.973V (queda de apenas 0.027V) e o beacon transmitia um nível de 57%. Temos a seguinte linearização da curva de descarga da bateria, abordada no Capítulo 2 (Figura 17), utilizada pelas bibliotecas da Nordic para o cálculo das porcentagens de bateria:

- Seção 1: 3.0V - 2.9V = 100% - 42% (58% de queda em 100 mV)
- Seção 2: 2.9V - 2.74V = 42% - 18% (24% de queda em 160 mV)
- Seção 3: 2.74V - 2.44V = 18% - 6% (12% de queda em 300 mV)
- Seção 4: 2.44V - 2.1V = 6% - 0% (6% de queda em 340 mV)

Mesmo com apenas 4 seções de aproximações lineares, esse resultado é bastante satisfatório. O intuito de repassar a informação do nível de bateria do beacon no pacote

de *advertise* é de dar uma noção aproximada da vida útil da bateria, segundo a curva de descarga utilizada, e isso certamente está sendo feito.

Cinco meses depois, o nível de bateria foi medida com o multímetro mais uma vez e o resultado foi de 2,917V (queda de 0,087V). Ou seja, com o resistor de *pull-down*, o problema de interferência com os pinos flutuantes foi resolvido. Esse teste mostrou a importância do resistor de *pulldown* e para resolver o problema da interferência. No entanto, não mostra reais expectativas de decaimento de bateria para a aplicação da solução proposta, uma vez que o intervalo de transmissões utilizado nesse teste (de 5min) teve que ser drásticamente reduzido a alguns segundos para que a solução fosse eficiente, como será visto no próximo teste. Assim, novos testes em relação ao decaimento de bateria são necessários.

5.4 Testes de desconexão de beacon com os Gateways

Para realizar os testes com os gateways desenvolvidos (tanto a versão Raspberry Pi quanto Toradex) foram utilizados os beacons da Ingics abordados no Capítulo 3. Outra unidade da Raspberry Pi 3 foi adquirida pela empresa, configurada e programada da mesma forma que a inicial. Portanto os testes incluíram duas Raspberry Pi, uma placa Toradex e 25 beacons Ingics.

Uma vez desenvolvida a solução, faltava ainda definir dois parâmetros principais do projeto:

- Intervalo de transmissão do beacon: O intervalo de transmissão dos *advertises* é crucial tanto para o gerenciamento de bateria do beacon quanto para que a funcionalidade de detectar o beacon no ambiente seja eficiente. Os intervalos de 1 ou até 5 minutos que foram utilizados no teste do firmware desenvolvido para o beacon da Nordic se mostraram valores inadequados em relação a eficiência da solução, como os testes a seguir puderam mostrar.

- Intervalo para alarme de desconexão: Esse intervalo define o limite de tempo em que o gateway ficará sem receber *advertises* de um beacon antes de enviar à API um alarme de beacon desconectado. Esse tempo é crucial para a solução, pois em termos práticos é o tempo no qual uma pessoa poderia pegar o ativo, juntamente com o beacon, e se retirar do local antes de um alarme no sistema.

Os testes foram feitos em duas salas de dois ambientes cada, que serão referidas como Sala 1 e Sala 2, ilustradas respectivamente nas Figuras 33 e 34 . As figuras mostram também a disposição dos beacons Ingics, dos gateways Raspberry Pi e do gateway Toradex

durante os testes. Os beacons estavam anexados a aparelhos como impressoras, gabinetes e monitores, *laptops*, projetores, cafeteiras elétricas e similares.

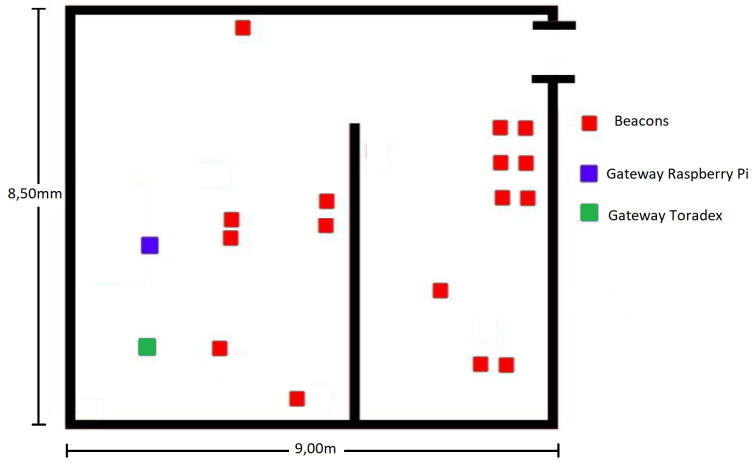

Figura 33 – Disposição de beacons e gateways na Sala 1. Fonte: Autor

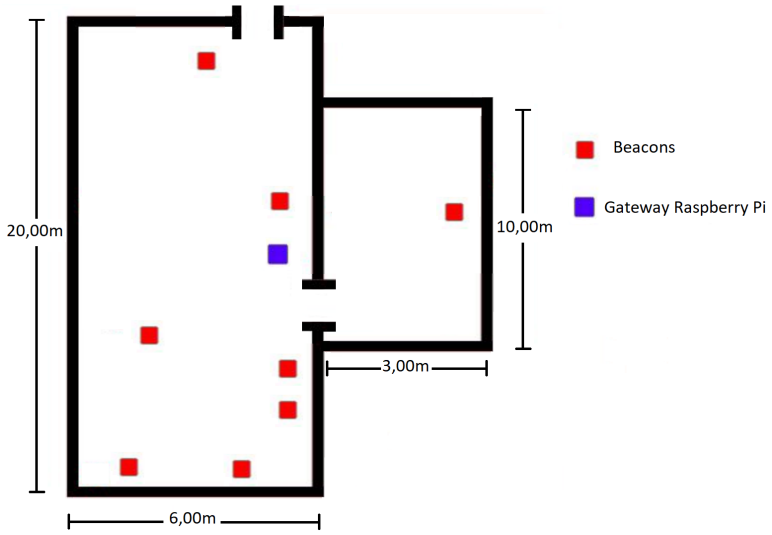

Figura 34 – Disposição de beacons e gateway na Sala 2. Fonte: Autor

O intervalo de transmissão dos beacons Ingics é programável de 1 a 10 segundos. Por padrão de fábrica, eles são configurados para um intervalo de 5 segundos, e por isso os testes foram iniciados com essa configuração.

O teste de consistência de transmissão e recepção com os dispositivos Ingics mostrou que, com uma certa proximidade entre gateway e beacon, a porcentagem de pacotes perdidos era pequena, porém considerável para a aplicação. Portanto, deve haver uma margem de segurança para evitar falsos alarmes devido algumas perdas de pacotes. Essa margem é o intervalo para alarme de desconexão, abordado anteriormente. Os testes foram iniciados com um intervalo de 30 segundos.

Tabela 4 – Tabela dos testes realizados com os de gateways desenvolvidos.

# TESTE	**GW.**	**INICIO**	**FIM**	**GW DESCO-NECT.**	**B. DES-CO-NECT.**
1	Rasp. 1	16h45 17/10/2017	10h00 18/10/2017	0	9
1	Rasp. 2	16h45 17/10/2017	10h00 18/10/2017	3	49
2	Rasp. 1	12h20 18/10/2017	10h00 19/10/2017	0	21
2	Rasp. 2	12h20 18/10/2017	10h00 19/10/2017	1	39
3	Rasp. 1	11h50 19/10/2017	17h15 19/10/2017	0	2
3	Rasp. 2	11h50 19/10/2017	17h15 19/10/2017	0	7
4	Rasp. 1	18h30 20/10/2017	10h20 23/10/2017	0	44
4	Rasp. 2	18h30 20/10/2017	10h20 23/10/2017	0	62
4	Toradex	18h30 20/10/2017	10h20 23/10/2017	0	31
5	Rasp. 1	13h40 24/10/2017	10h20 26/10/2017	0	0
5	Rasp. 2	13h40 24/10/2017	10h20 26/10/2017	0	0
5	Toradex	13h40 24/10/2017	10h20 26/10/2017	0	3
6	Rasp. 1	18h00 27/10/2017	10h20 01/11/2017	0	0
6	Rasp. 2	18h00 27/10/2017	16h40 01/11/2017	0	0
6	Toradex	18h00 27/10/2017	16h40 01/11/2017	0	2
7	Rasp. 1	17h30 01/11/2017	17h00 17/11/2017	0	0
7	Rasp. 2	17h30 01/11/2017	16h00 17/11/2017	0	0
7	Toradex	17h30 01/11/2017	16h00 17/11/2017	0	10

A Tabela 4 possui uma coluna que enumera os testes feitos, em ordem cronológica, uma coluna para indicar de qual dos gateways se tratam os dados, data e horário de início e fim de cada teste, o número de vezes que o gateway se desconectou da rede e o número de vezes que foi detectada uma desconexão de Beacon. A Toradex só foi incluída nos testes a partir do teste 4, porque ainda estava ocorrendo a migração do código.

No primeiro teste, a Raspberry Pi da Sala 2 perdeu a conexão com o servidor por 3 vezes e alertou 49 sensores desconectados. Ou seja, por 49 vezes ela ficou 30 segundos sem receber o *advertise* de algum beacon. O ambiente de teste da Sala 2 possui muito mais equipamentos e é menos amplo que a Sala 1, o que poderia justificar mais perdas de Radiofrequência, multicaminhos e similares e consequentemente mais falsos alarmes de desconexão que na Sala 1.

Nesse ponto do desenvolvimento, a filtragem por MAC estava sendo feita no código de leitura em C e não na seção "*Start Scan*" em Python, abordada no capítulo 4. E foi então que optou-se por retirar qualquer filtro ou processamento desnecessário no código em C, na tentativa de evitar perdas de pacote por processamento. Além disso, o intervalo foi aumentado de 30 segundos para 45 segundos.

No segundo teste, foi percebido que a Raspberry Pi da Sala 2 perdeu a conexão com o servidor mais uma vez. Foi mudada a rede na qual essa Raspberry Pi estava conectada e as desconexões com o servidor cessaram, ou seja, era um problema da rede e não do dispositivo. Como não houveram problemas com a segunda rede, análises e diagnósticos dos problemas apresentados pela primeira rede seriam feitos após os testes, que nesse momento eram prioridade. Nesse segundo teste, os números de falsos alarmes não diminuíram, e então o intervalo de desconexão foi aumentado para 60 segundos. A partir daí, observou-se uma diminuição considerável dos falsos alarmes, como mostram os resultados do teste 3. Porém ainda não era um número aceitável para a solução.

No teste 4, foi introduzido o gateway desenvolvido na Toradex e o intervalo de transmissões dos beacons foi reduzido de 5 para 3 segundos. Observou-se então um aumento no número de falsos alarmes. Foi nesse teste que detectou-se um problema com o firmware. A lógica de escrita e leitura de arquivos para que as *threads* de leitura de beacons e envio para a API compartilhassem informações estava gerando falsos alarmes. Como há um fluxo considerável de dados, o que ocorria eventualmente é que enquanto a *thread* "*Start Scan*" escrevia os dados recebidos de um beacon no arquivo referente a esse beacon, a *thread* "*Send Update*" tentava fazer a leitura do arquivo para enviar esses dados para a API. Se o arquivo fosse aberto para ser lido enquanto estava sendo escrito, ele é lido como um arquivo em branco. Daí, então era gerado um alarme de beacon desconectado, pois as informações sobre ele supostamente não estavam no arquivo.

Esse problema foi solucionado com o uso semáforos. Os semáforos são recursos utilizados para sincronização, no objetivo de controlar o acesso a dados compartilhados entre tarefas (PERES, 2013). Na biblioteca *threading* que já estava inclusa no programa, existem funções que fazem o uso dos recursos de semáforo. De maneira básica, os semáforos tem a seguinte estrutura no código:

```
semaforo.acquire()
#Codigo a ser sincronizado
semaforo.release()
```

Para o uso específico de leitura e escrita em arquivos, a estrutura

```
with open('arquivo.txt', 'r') as arquivo:
```

realiza a sincronização do *acquire* e *release* por meio do comando *with*. Essa linha foi utilizada na seção "*Send Update*", na leitura dos arquivos de cada beacon. Dessa forma, o arquivo aberto através dessa estrutura estará "travado" e nenhuma outra *thread* poderá ter acesso a ele até todos os comandos sejam finalizados. A mesma estrutura foi utilizada na seção *Start Scan* para escrita, com a mudança do argumento "r" de leitura (*read*) para "w" de escrita (*write*).

No teste seguinte, os erros diminuíram drasticamente. Como os resultados estavam mais favoráveis, decidiu-se diminuir o intervalo para alarme de desconexão de 60 segundos de volta para 45 segundos. Os resultados continuaram os mesmos. Não haviam mais falsos alarmes nas Raspberry Pi e a Toradex gerava alarmes eventualmente, aproximadamente um a cada 2 dias. No dia 17/11 foram completadas pouco mais de 3 semanas de testes ininterruptos sem nenhum falso alarme nas Raspberry Pi, e erros eventuais na Toradex.

A partir de análises dos arquivos e das transmissões entre os códigos e *threads*, foi possível concluir que os erros da Toradex eram gerados devido a um erro na comunicação entre o código em C de leitura Bluetooth com o programa principal em Python. Eventualmente, ao invés de enviar um *advertise* recebido para o programa principal, o código em C enviava uma mensagem concatenada de mais de um *advertise* recebido, que não era interpretada pelo programa principal e então gerava um falso alarme. O mais intrigante é que o código é o mesmo em ambas Raspberry Pi e Toradex, porém esse erro só acontecia na Toradex.

Chegou-se então a considerar que estaria relacionado à pouca memória da Toradex em relação A Raspberry Pi. Essa situação ainda está sendo investigada e não chegou-se a uma conclusão sobre o porque dessas mensagem serem eventualmente concatenadas (o *advertise* esperado seguida de alguns bytes de "lixo" provenientes de um outro *advertise*). Porém, ao observar o padrão de como esse erro ocorria, foi possível tratar a mensagem de modo a ignorar esse "lixo" e redimensioná-la de volta para o tamanho e formato esperado. Realizando esse tratamento, os erro originados por esse problema cessaram e o sistema encontra-se, até a finalização desse trabalho, em teste de homologação.

6 Conclusões

De início, apresentou-se o problema de controle de patrimônio, muito recorrente em diverso tipos de estabelecimentos, que motivou a empresa Innovix a levar ao mercado uma solução eficiente para esse caso. O principal objetivo da solução é monitorar objetos em um ambiente de forma que seja possível dizer se eles foram retirados desse ambiente ou ainda se os sensores vinculados a eles foram desanexados dos mesmos.

Para isso, buscou-se os métodos eficientes e pouco invasíveis que pudessem fazer a transmissão de informação para um concentrador local (gateway) e que possuíssem um tipo de sensor para detectar a remoção do objeto. Por motivos como custo, estética da solução, preferência do cliente, facilidade de instalação, facilidade de manutenção, foi decidido que o sensor que faria contato com o objeto deveria ser completamente Wireless. A implementação de um dispositivo completamente sem fio implica em alimentação a base de bateria e um desenvolvimento com foco em baixo consumo de energia.

Dentre as diversas tecnologias estudadas e analisadas, optou-se por utilizar a tecnologia Bluetooth *Low Energy* (BLE) que vem ganhando cada vez mais espaço no âmbito de automação e IOT. Para a detecção do contato do sensor com o objeto, foi escolhida a utilização de um botão de contato, visto que outras tecnologias, além de mais caras e de consumirem mais energia, poderiam ser mais facilmente burladas ou não teriam como atender às necessidades da solução.

No universo do BLE, foram analisados os Beacons e suas aplicações. Foram testados beacons e Gateways BLE de diferentes empresas e optou-se pelo desenvolvimento da solução da Innovix no *Smart Beacon* programável da empresa Nordic Semiconductor. A solução desenvolvida consta de um beacon com botão de contato cujo pacote de dados (advertise) são transmitidos a cada cinco minutos e imediatamente a cada evento de mudança de estado do botão. O pacote de advertise carrega consigo, além da ID de cada beacon, a informação do estado do botão, dos níveis de bateria e de um número identificador de eventos ocorridos. Esse número identificador de eventos foi desenvolvido para o caso em que haja uma interrupção na comunicação dos sensores com o gateway ou do gateway com o servidor ou API a ser desenvolvida. Assim, ao se reestabelecer a comunicação, seria possível saber se houve algum evento durante esse período sem conexão.

Em sequência, foi desenvolvido um gateway (plataforma concentradora) para receber os sinais dos beacons, processá-los e disponibilizar as informações para os usuários por meio de um *web service*. O gateway se comunica com uma API em um servidor local para receber informações de quais beacons ele deve passar a monitorar. Optou-se por utilizar uma plataforma que tivesse suporte a um sistema operacional Linux. Portanto,

foram utilizadas dispositivos com Linux embarcado. Inicialmente a solução foi desenvolvida na plataforma Raspberry Pi 3, por ter um custo relativamente baixo e uma grande capacidade de processamento. Foi desenvolvido um código em C para fazer a leitura de sinais BLE. Esse código se comunica com um programa principal em Python que gera uma mensagem no formato JASON e a transmite para uma API no protocolo HTTP.

Após finalizar a fase de desenvolvimento, iniciou-se a migração da solução para uma outra plataforma com suporte a Linux embarcado, da empresa Suiça Toradex. O motivo para tal é ter uma opção mais profissional, uma vez que a Raspberry Pi é vista por muitos como uma plataforma "hobbista" de desenvolvimento. Foram feitas algumas configurações no Kernel do sistema operacional para se adequar com o que era preciso para a solução proposta. Com os testes das diferentes versões de gateway desenvolvidas, chegou-se a conclusão que os gateways parecem estar estáveis e que a solução está de acordo com o proposto, com os parâmetros utilizados até então.

6.1 Trabalhos futuros

Dentre os próximos passos a serem tomados está a integração das duas frentes de desenvolvimento relatadas aqui. No estágio atual, o beacon desenvolvido poderia se comunicar e transmitir as informações dos ativos associados a eles para qualquer gateway. O mesmo pode se dizer dos gateways em relação a qualquer beacon, sendo necessárias apenas pequenas alterações no código referentes ao formato dos pacotes enviados por outros beacons. Como já foi explicado, o desenvolvimento de cada frente (beacon e gateway) teve o auxílio dos dispositivos da Ingics Technology para que fosse possível testar e homologar o que estava em desenvolvimento com o menor número de variáveis possível. Portanto, terão inícios testes com as duas frentes funcionando em conjunto.

Um grande passo a ser tomado é em relação ao botão e resistor de *pull-down* do beacon. Como o beacon deverá ser anexado à superfície de um objeto de forma que um botão será pressionado, precisa-se de um botão específico que de adeque melhor a essa função. Os botões do *Smart Beacon* da Nordic são muito pequenos e não seria possível utiliza-los dessa forma. Após pesquisas de diferentes versões de botões, chegou ao modelo ESE-11SV1, da Panasonic (Figura 35).

O pino desse modelo de botão é bastante sensível, de modo que o beacon poderia ser fixado com um fita dupla-face sobre a superfície do objeto e botão seria pressionado sem dificuldades. O produto já virá com a fita anexada, bastando o consumidor retirar o plástico e fixar o sensor a uma superfície. A fita foi escolhida como meio de fixação por trazer mais conveniência ao consumidor ou instalador, uma vez que não será necessário aplicar algum produto fixador como cola e afins. Alguns modelos de fita dupla-face da marca 3M estão sendo analisados pela empresa.

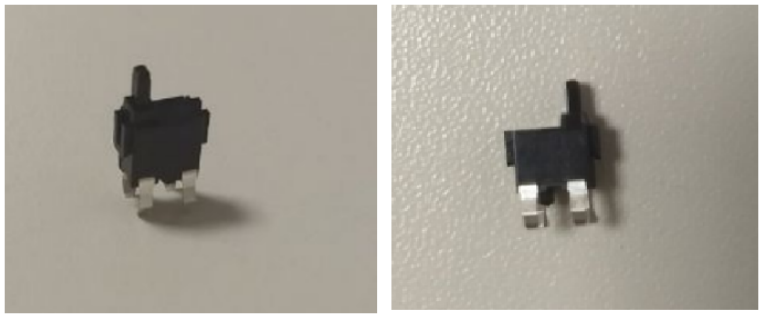

Figura 35 – Botão ESE-11SV1 da Panasonic. Fonte: Imagens cedidas pela Innovix.

Esse botão seria então soldado aos terminais GPIO da *Smart Beacon* da Nordic com o pino para fora do encapsulamento. O resistor de *pull-down*, que é necessário para evitar a interferência eletroestática nos pinos de programação do beacon, também deve ser soldado da interface de programação ao pino terra da placa.

Essa questões remetem a um outro trabalho futuro, que é o encapsulamento do produto. Uma vez que o beacon será anexado a um objeto, é conveniente que ele seja o menor possível. Portanto, deverá existir um trabalho de engenharia de produto para otimizar o espaço dentro de um encapsulamento, pesquisando modelos de caixas em empresas que fornecem caixas para dispositivos eletrônicos e também em empresas que fabricam de acordo com o modelo do produto. No caso do gateway, a ideia é conseguir um encapsulamento que sirva tanto para a versão com a Raspberry Pi quanto para a versão com a Toradex. Outras questões de produtização para o gateway como LEDs auxiliares e similares ainda estão em discussão.

Perspectivas a longo prazo sobre novas funcionalidades do IDView incluem sensores de temperatura e umidade para ambientes internos, sensores de corrente para verificar se aparelhos eletrônicos estão ligados, em *stand-by* ou desligados, dentre outros. Todas essas variações de sensores seriam totalmente integradas com o gateway, API e interface gráfica.

Por fim, conclui-se que o trabalho realizado do estudo das tecnologias abordadas e do desenvolvimento do beacon e gateway foram bastante produtivos, trouxeram muito conhecimento teórico e principalmente prático e geraram uma solução eficiente, assim como foi proposto. Mesmo que, em testes e aplicações futuras, chegue-se a conclusão de que hajam plataformas mais interessantes que os beacons programáveis da Nordic e que a Raspberry Pi 3 ou Toradex, o conhecimento do desenvolvimento em chips Bluetooth adquirido durante esse trabalho implicará em muito mais agilidade e eficiência no trabalho migração e adaptação.

Referências

ACCARRINO, J. How linux was announced to the world in 1991. 2011. [Acesso em 8 de Outubro 2017]. Disponível em: <http://www.methodshop.com/2011/05/linux-1991.shtml>. Citado 2 vezes nas páginas 15 e 34.

ARMSTRONG, S. Rfid basics: How rfid tags work. 2011. [Acesso em 15 de Maio 2017]. Disponível em: <https://blog.atlasrfidstore.com/rfid-tag-basics>. Citado na página 22.

BASTOS, H. Diferenças entre linguagem compilada e linguagem interpretada. 2008. [Acesso em 4 de Novembro 2017]. Disponível em: <http://henriquebastos.net/diferencas-entre-linguagem-compilada-e-linguagem-interpretada/>. Citado na página 59.

BLACKSTONE, A. Ble beacons: ibeacon, altbeacon, uribeacon and derivatives. 2014. [Acesso em 28 de Abril 2017]. Disponível em: <http://austinblackstoneengineering.com/ble-beacons-ibeacon-altbeacon-uribeacon-and-derivatives/>. Citado na página 32.

BLANCO, R. *Bluetooth Network Toplogy*. 2007. [Acesso em 15 de Outubro 2017]. Disponível em: <https://commons.wikimedia.org/wiki/File:Bluetooth_network_topology.png>. Citado 2 vezes nas páginas 15 e 28.

BOURQUE, B. This is how bluetooth works, and no, it's not by magic. 2014. [Acesso em 28 de Abril 2017]. Disponível em: <https://www.digitaltrends.com/mobile/how-does-bluetooth-work/>. Citado na página 27.

CANNON, J. Linux for beginners: An introduction to the linux operating system and command line. 2017. [Curso Online. Acesso em 13 de Agosto 2017]. Disponível em: <https://www.udemy.com/linuxforbeginners/>. Citado 2 vezes nas páginas 34 e 35.

CHILTON, A. The working principle and key applications of infrared sensors. 2017. [Acesso em 15 de Maio 2017]. Disponível em: <https://www.azosensors.com/article.aspx?ArticleID=339>. Citado na página 21.

EGGERT, D. Ip, tcp, and http. 2014. [Acesso em 16 de Outubro 2017]. Disponível em: <https://www.objc.io/issues/10-syncing-data/ip-tcp-http/>. Citado na página 36.

EVAN-AMOS. *Raspberry Pi 3 Flat Top*. 2016. [Acesso em 18 de Novembro 2017]. Disponível em: <https://commons.wikimedia.org/wiki/File:Raspberry-Pi-3-Flat-Top.jpg>. Citado 2 vezes nas páginas 15 e 57.

FONDATION, R. P. Raspberry pi 3 model b specifications. [Acesso em 15 de Outubro 2017]. Disponível em: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>. Citado na página 57.

FRANKLIN, C.; LAYTON, J. How bluetooth works. [Acesso em 7 de Abril 2017]. Disponível em: <https://electronics.howstuffworks.com/bluetooth2.htm>. Citado na página 27.

INSTRUMENTS, N. O que é a tecnologia de tempo real? 2011. [Acesso em 20 de Outubro 2017]. Disponível em: <http://www.ni.com/white-paper/3938/pt/>. Citado na página 34.

INSTRUMENTS, N. Conceitos gerais de comunicação serial. 2015. [Acesso em 8 de Setembro 2017]. Disponível em: <http://digital.ni.com/public.nsf/allkb/32679C566F4B9700862576A20051FE8F>. Citado na página 23.

JANIAK, S. Three ways bluetooth smart technology enables innovation for the internet of things. 2015. [Acesso em 20 de Agosto 2017]. Disponível em: <https://blog.bluetooth.com/three-ways-bluetooth-smart-technology-enables-innovation-for-the-internet-of-things>. Citado na página 29.

KINNEY, T. A. Proximity sensors compared: Inductive, capacitive, photoelectric, and ultrasonic. 2001. [Acesso em 15 de Maio 2017]. Disponível em: <http://www.machinedesign.com/sensors/proximity-sensors-compared-inductive-capacitive-photoelectric-and-ultrasonic>. Citado na página 22.

KONTAKTIO, C. Beacon profile: Eddystone. [Acesso em 28 de Abril 2017]. Disponível em: <https://support.kontakt.io/hc/en-gb/articles/206853889-Beacon-profile-Eddystone>. Citado na página 32.

KONTAKTIO, C. What is a beacon? [Acesso em 15 de Maio 2017]. Disponível em: <https://kontakt.io/beacon-basics/what-is-a-beacon/>. Citado na página 30.

NALDER, J. *Beacons by jnxyz.education*. 2014. [Acesso em 18 de Novembro 2017]. Disponível em: <https://www.flickr.com/photos/jnxyz/13570744845/>. Citado 2 vezes nas páginas 15 e 31.

NENOKI, E. Zigbee – estudo da tecnologia e aplicaÇÃo no sistema elÉtrico de potÊncia. 2013. [Acesso em 16 de Abril 2017]. Disponível em: <http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/886/1/CT_COTEL_2012_2_01.pdf>. Citado na página 24.

PERES, R. Threads, semÁforos e deadlocks. 2013. [Acesso em 4 de Novembro 2017]. Disponível em: <https://www.revista-programar.info/artigos/threads-semaforos-e-deadlocks-o-jantar-dos-filosofos/>. Citado na página 82.

POOLE, I. Ieee 802.11 wi-fi standardswireless fidelity - wifi. Citado na página 24.

REN, K. Ten important differences between bluetooth bredr and bluetooth smart. 2015. [Acesso em 8 de Agosto 2017]. Disponível em: <https://blog.bluetooth.com/ten-important-differences-between-bluetooth-bredr-and-bluetooth-smart>. Citado na página 29.

SEMICONDUCTOR, C. Comparison of rf and bluetooth. 2010. [Acesso em 16 de Agosto 2017]. Disponível em: <https://www.digikey.com/en/articles/techzone/2010/dec/comparison-of-rf-and-bluetooth>. Citado na página 29.

SEMICONDUCTOR, N. *nRF51822 Bluetooth Smart Beacon Reference Design User Guide*. 2014. [Acesso em 14 de Abril 2017]. Disponível em: <http://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF51822-Bluetooth-Smart-Beacon-Kit>. Citado na página 53.

SEMICONDUCTOR, N. *nRF52 DK Hardware Files*. 2016. [Acesso em 14 de Abril 2017]. Disponível em: <https://www.nordicsemi.com/eng/Products/Bluetooth-low-energy/nRF52-DK>. Citado na página 53.

SIG, B. Bluetooth specifications. [Acesso em 8 de Agosto 2017]. Disponível em: <https://www.bluetooth.com/specifications>. Citado 2 vezes nas páginas 25 e 27.

SIG, B. How it works: Br/edr. [Acesso em 8 de Agosto 2017]. Disponível em: <https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works/br-edr>. Citado na página 29.

SIG, B. *Bluetooth combo wordmark 2011*. 2011. [Acesso em 27 de Agosto 2017]. Disponível em: <bluetooth.com>. Citado 2 vezes nas páginas 15 e 27.

SUEIRO, D. Anatomia de um sistema linux embarcado. 2013. [Acesso em 16 de Outubro 2017]. Disponível em: <https://www.embarcados.com.br/anatomia-de-um-sistema-linux-embarcado/>. Citado 2 vezes nas páginas 35 e 36.

TECHNOLOGY, I. *iGS01 and iGS01 User Guide*. 2017. [Acesso em 14 de Abril 2017]. Disponível em: <https://www.ingics.com/>. Citado na página 39.

TORVMARK, K. Three flavors of bluetooth : Which one to choose? 2014. [Acesso em 16 de Outubro 2017]. Disponível em: <http://www.ti.com/lit/wp/swry007/swry007.pdf>. Citado 3 vezes nas páginas 15, 29 e 30.

WARSKI, A. How do ibeacons work? 2014. [Acesso em 14 de Abril 2017]. Disponível em: <http://www.warski.org/blog/2014/01/how-ibeacons-work/>. Citado na página 31.

WILKINS, S. Osi and tcp/ip model layers. 2011. [Acesso em 16 de Outubro 2017]. Disponível em: <http://www.pearsonitcertification.com/articles/article.aspx?p=1804869>. Citado na página 36.

WILLIAMS, A. The best linux distros 2017: 8 versions of linux we recommend. 2014. [Acesso em 13 de Agosto 2017]. Disponível em: <http://www.techradar.com/news/software/operating-systems/best-linux-distro-five-we-recommend-1090058>. Citado na página 35.

Anexos

ANEXO A – Ferramentas para Desenvolvimento do firmware do Smart Beacon da Nordic Semiconductor

O ambiente de desenvolvimento utilizado para produzir o código foi o uVision IDE 5, da Keil (Figura 36). É uma IDE (*Integrated development environment*) bastante completa que combina edição de código, gerenciamento de projetos e *program debugging* em uma interface simples e intuitiva.

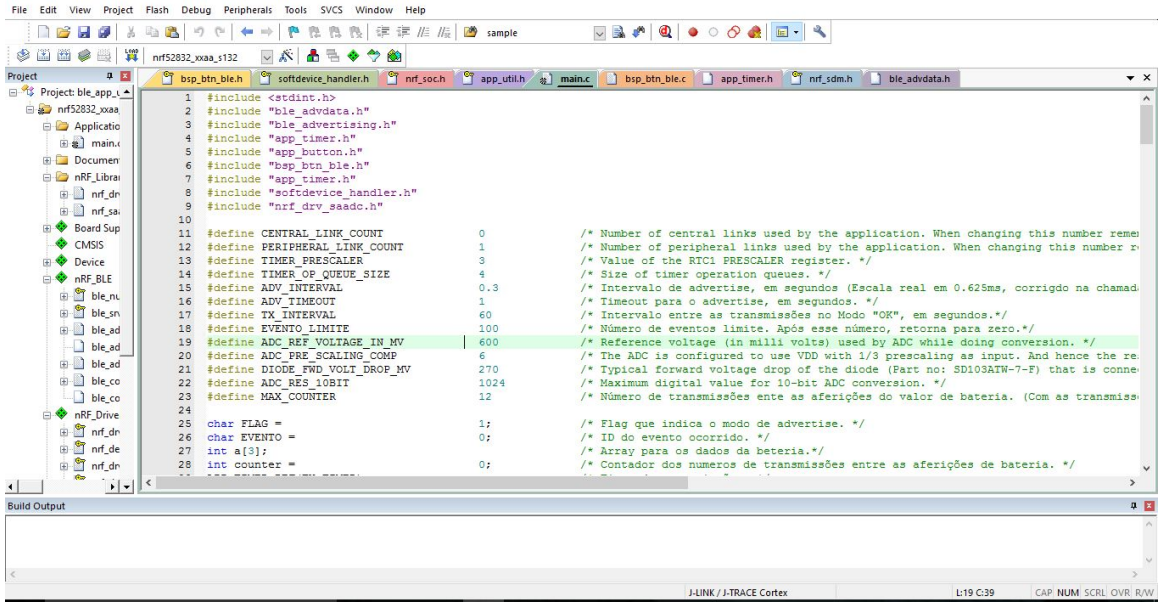

Figura 36 – Interface do uVision IDE. Fonte: Imagem cedida pela Innovix.

Na parte esquerda da tela, o usuário tem acesso a todos os arquivos que compõe o projeto. A janela inferior traz informações sobre a compilação e construção dos arquivos gerados pela IDE, assim como *log* de erros de compilação. Na parte superior da interface, existem as opções e configurações de compilação, a habilitação do modo *debug*, as opções de adicionar arquivos .h e .c ao projeto, dentre outros. Para utilizar os produtos da Nordic com o uVision, foi necessária a instalação do SDK fornecido pela própria Nordic (foi utilizada a versão 11.0.0) que contem todos os exemplos, bibliotecas e arquivos necessários para o desenvolvimento de aplicações, na linguagem C de programação. A instalação desse SDK também adiciona pacotes da Nordic ao ambiente uVision, o que permite uma total integração da plataforma com o dispositivo. Dessa forma, ao conectar a placa de desenvolvimento ao computador via USB, o uVision IDE a reconhece automaticamente

e disponibiliza a opção de *flash* para transferir a aplicação desenvolvida para o chip da placa.

A Figura 37 mostra uma janela que é acessível por um ícone na parte superior da interface do uVision. Nela, é possível selecionar os dispositivos compatíveis com a IDE e então visualizar exemplos que estão disponíveis para esses dispositivos, instalados por SDKs. Pela Figura 37, é possível visualizar a plataforma Nordic Semiconductor selecionada à esquerda. Ao clicar na opção *copy* ao lado de um dado exemplo, o usuário é redirecionado para a tela principal da IDE e o projeto do exemplo é aberto para edição e compilação. Deve-se notar que mesmo para o mesmo chip, existem diferentes versões de placa que depende do lote do produto. Diferentes versões de placas podem ter pequenas divergências entre si, como por exemplo, diferente disposição de trilhas para os pinos GPIOs. Por isso, os exemplos aparecem replicados para diferentes versões de placa. A placa de desenvolvimento utilizada é da versão PCA10040.

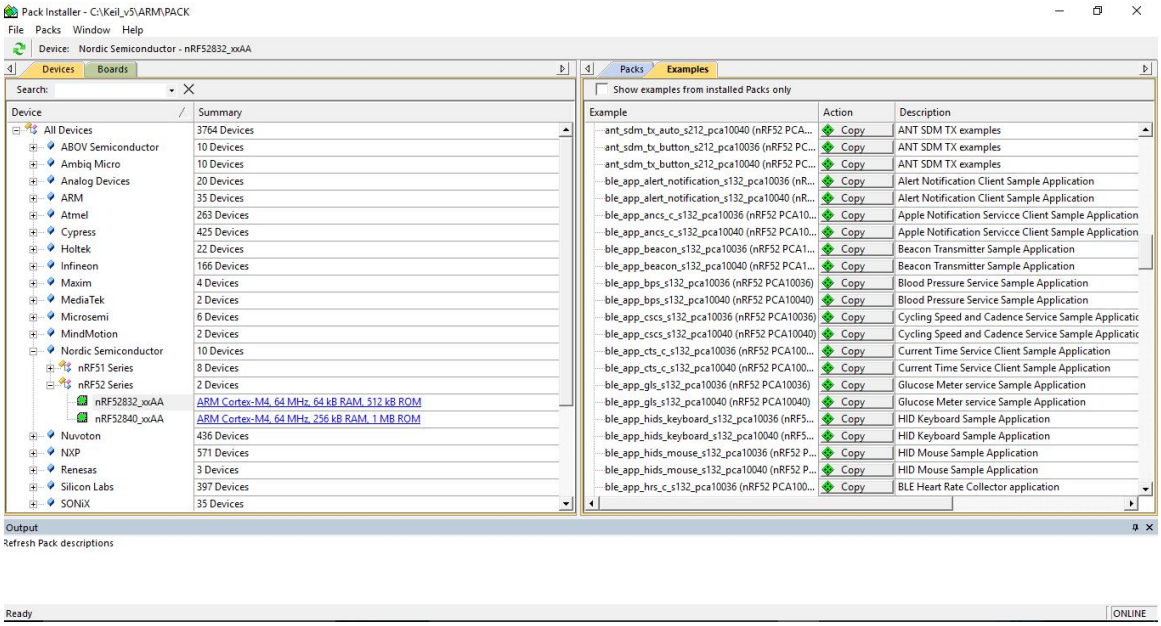

Figura 37 – Janela *Pack Installer* no uVision IDE. Fonte: Imagem cedida pela Innovix.

Além de diferentes versões de placa, existem também diferentes *softdevices* suportados por cada chip e SDK instalado. Os *softdevices* são arquivos pré programados de extensão .hex que contém todas as funções necessárias para o correto funcionamento e controle das rádio-transmissões em aplicações que utilizam BLE. O usuário não tem acesso a essas funções, de modo que a única coisa a se fazer é chamar as funções que configuram o *softdevice* utilizado durante a inicialização do aplicativo desenvolvido. O SDK 11.0.0 suporta os arquivos de *softdevices* versões s130, s132, s212 e s312. Do mesmo modo que ocorre com as diferentes versões de placa, a maioria dos exemplos também é replicada para diferentes *softdevices*.

Foram estudados diversos exemplos para que houvesse o entendimento necessário

do funcionamento de comandos, funções e bibliotecas da Nordic antes de dar início ao desenvolvimento do aplicativo próprio. Duas ferramentas foram cruciais e extensamente utilizadas durante o processo de estudo/aprendizado e também durante o desenvolvimento da solução:

- ***Infocenter* da Nordic Semiconductor:**

  O *Infocenter* da Nordic é uma plataforma online que contém todo e qualquer tipo de documentação sobre *hardware*, *software* e soluções da Nordic em geral. Disponível em http://infocenter.nordicsemi.com, seu conteúdo abrange manuais de seus produtos, referências de *layout* de seus chips e placas, comentários sobre os aplicativos de exemplos disponíveis nos SDK, documentação técnica sobre os SDKs, bibliotecas, funções, *softdevices*, dentre outros.

- ***Developer Zone* da Nordic Semiconductor:**

  O *Developer Zone* é uma plataforma de interação dedicada à comunidade que utiliza os produtos da Nordic, disponível em: https://devzone.nordicsemi.com/questions/. Nela, existem diversos tutoriais escritos por funcionários da Nordic que procuram esclarecer alguns pontos chave sobre a utilização e desenvolvimento a partir de seus produtos. Existe também o ambiente de perguntas e respostas, onde um usuário pode fazer perguntas sobre um certo tema ou problema específico que está tendo durante o desenvolvimento de uma aplicação e te-la respondida por um funcionário da Nordic ou até mesmo por outros usuários. Nessa plataforma, há uma interessante troca de experiência entre os consumidores de produtos da Nordic e funcionários da Nordic, uma vez que nela são feitas notificações de *bugs* em SDKs ou de erratas em documentações.

Além de tudo o que já foi abordado, a única outra ferramenta necessária para a programação do chip foi o *software* nRF Studio. Esse *software* é utilizado para apagar arquivos que foram programados no chip, assim como programar o *softdevice* necessário e o *firmware* desenvolvido. Foi dito que a IDE possui integração com o hardware da Nordic e, com a opção *flash*, permite programar o arquivo .hex gerado da compilação do código do aplicativo diretamente na placa. Porém, antes de programar esse arquivo, é necessário programar o *softdevice* específico a ser utilizado e isso a IDE não faz. Após a programação do *softdevice*, o aplicativo pode ser programado e reprogramado quantas vezes forem necessárias sem a necessidade de uma reprogramação do *softdevice*. Assim, ter a opção de programar o aplicativo diretamente do ambiente de edição e compilação continua sendo bastante útil. A opção de apagar os arquivos programados é utilizada no caso onde haja a mudança da versão do *softdevice* utilizado pela aplicação ou caso o arquivo programado tenha sido corrompido. A Figura 38 mostra a interface do nRF Studio

com as funcionalidades descritas. Como é possível ver ainda na Figura 38, existe também a funcionalidade de programação de um arquivo chamado *Bootloader*. Esse arquivo também de extensão .hex é utilizado para disponibilizar a a funcionalidade de atualização de *firmware* OTA (*over the air*), ou seja, sem a necessidade de cabos, feita via aplicativo da Nordic para *smartphones* "*nRF Toolbox*". Essa funcionalidade não foi muito explorada.

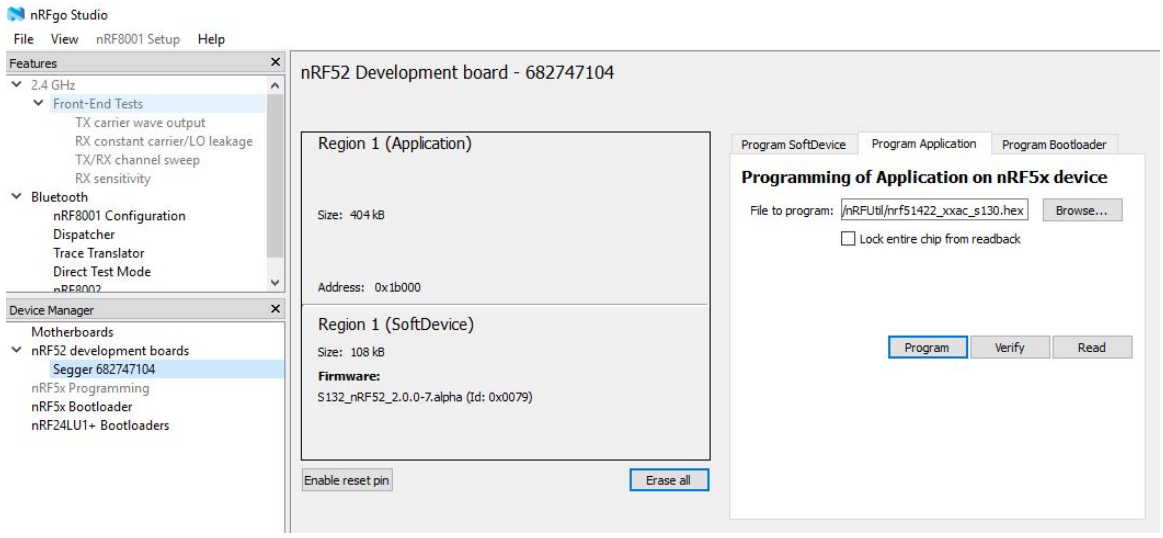

Figura 38 – Interface do *software* nRF Studio utilizado para programação dos chips. Fonte: Imagem cedida pela Innovix.

ANEXO B – Análise e Estudos de firmwares exemplo no Kit de Desenvolvimento da Nordic Semiconductor

A placa de desenvolvimento, assim como o beacon, já vem de fábrica com um aplicativo padrão chamado *ble_app_beacon* que é utilizado para demonstrações de beacons utilizando o *pseudo-standard* iBeacon em conjunto com o aplicativo da Nordic para *smartphones*.

Como já foi mencionado, o processo de desenvolvimento do firmware teve inicio com a utilização de alguns dos exemplos disponíveis para estudo e entendimento da bibliotecas e funções. Primeiramente, foi usado um exemplo mais possível, o *blinky_blanc*. Esse aplicativo demonstra a funcionalidade dos botões e portas GPIOs fazendo com que o estado dos LEDs seja controlado pelos botões da placa de desenvolvimento. Foi aberta a janela *Pack Installer* no uVision IDE e selecionou-se o exemplo *blinky_blanc*. Após compilá-lo, tentou-se passar o arquivo .hex para a placa com a opção *flash* da IDE, porém a IDE retornou um erro. Nesse momento, chegou-se à conclusão de que, quando o *softdevice* está programado no chip, não é possível a gravação de aplicativos que não utilizam a funcionalidade BLE. O *softdevice* estava programado no chip pois o aplicativo padrão do chip utiliza funcionalidades BLE. Essa impossibilidade da gravação de aplicativos não-BLE quando o *softdevice* está presente se dá por questão de disposição dos arquivos nas áreas de memória do chip. Por exemplo, o *softdevice* é gravado na memória do chip partir da posição x e os aplicativos que utilizam a funcionalidade BLE e necessitam do *softdevice* são gravados no chip a partir da posição x+[tamanho do *softdevice*]. Já os aplicativos que não fazem o uso do *softdevice* são gravados também a partir da posição x, o que gera conflito na programação dessas aplicações quando o *softdevice* já está presente no chip. Para resolver esse problemas simplesmente apagou-se todos os arquivos do chip com a ferramenta nRF Studio. Depois disso, o aplicativo *blinky_blanc* foi gravado com sucesso. A análise desse código permitiu um entendimento sobre as funções que são utilizadas para gerenciar os botões da placa e suas devidas interrupções.

Depois, foi analisado o exemplo *experimental_ble_app_eddystone*. Esse aplicativo utiliza o *pseudo-standard* Eddystone para simular o conceito de *Physical Web*. Como é um aplicativo que utiliza a funcionalidade BLE, foi necessário utilizar a ferramente nRF Studio para apagar o aplicativo anterior e programar o devido *softdevice*. O aplicativo em si foi programado pela IDE. Ele fazia o *broadcast* via BLE da url do site da Nordic (https://www.nordicsemi.com/). Foi utilizado um celular Samsung Galaxy S6 com o OS

Android 7.0 para fazer uma simulação da *Physical Web*. Para isso, foi preciso alterar as configurações do Google Chrome (Navegador de Internet da Google) do celular para habilitar o *Physical Web*. Enquanto a placa de desenvolvimento (simulando a função de um beacon) estava no alcance do Bluetooth do celular, aparecia uma notificação de *Physical Web* na aba de notificações do celular, que ao ser clicada direcionava o usuário para o site da Nordic.

Com uma certa análise desse código, foi possível alterar a url que é transmitida via BLE. Escolheu-se testar o aplicativo com o site da Innovix (http://www.innovix.com.br). Então, bastou reconfigurar a parte do código que fornecia a url para passar a transmitir a url desse novo site. A parte do código responsável pela configuração da url encontra-se abaixo.

```
1 // Eddystone URL data
2 #define APP_EDDYSTONE_URL_SCHEME    0x00
3 #define APP_EDDYSTONE_URL_URL       0x6e, 0x6f, 0x72, 0x64, \
4                                     0x69, 0x63, 0x73, 0x65, \
5                                     0x6d,0x69, 0x00
```

O termo *#define APP_EDDYSTONE_URL_SCHEME* é o byte que define o esquema identificador da url. No caso, 0x00 define "http://www.". O byte 0x01 definiria o identificador de segurança "https://www.", por exemplo. Já o termo *#define APP_EDDYSTONE_URL_URL* contém os bytes que definem o restante da url. Utilizando uma tabela de conversão hexadecimal para ASCII, é possível ver que do primeiro até o penúltimo byte é formada a sequência de caracteres "nordicsemi". O último byte desse termo #define (0x00) define o indicador do final da url, sendo 0x00 o indicador ".com/". Dessa forma, para fazer a transmissão da url da Innovix, alterou-se o termo *#define APP_EDDYSTONE_URL_URL* para a sequência "0x69, 0x6e, 0x6e, 0x6f, 0x76, 0x69, 0x78, 0x00"que forma a sequência de caracteres "innovix"e o final ".com/". Aplicações com Eddystone e *Physical Web* são de interesse para o desenvolvimento de futuras soluções da empresa.

Foi também analisado e testado um aplicativo exemplo chamado *ble_app_uart*. Esse aplicativo estabelece uma comunicação UART cabeada entre o chip e o computador e também realiza um serviço BLE. Através do serviço BLE, ele constrói um canal de comunicação entre algum dispositivo conectado à placa via Bluetooth e o computador, numa comunicação bidirecional.

De forma a se comunicar via UART com o chip, é necessário configurar um terminal no computador. Optou-se por utilizar o programa Termite versão 3.3, que é um terminal de transmissão de dados via comunicação serial de fácil configuração. Foi necessário apenas fazer a configuração da correta porta COM na qual a aplaca estava conectada.

Após programar o aplicativo na placa, fez-se a procura de dispositivos Bluetooth com o aplicativo da Nordic para *smartphones*. Quando a placa foi encontrada pelo aplicativo, foi possível fazer a conexão da placa com o celular. Depois, o usuário é direcionado para uma tela semelhante à tela de um simples *chat*.

Com a conexão estabelecida, qualquer mensagem enviada pelo celular ou pelo computador aparece na interface tanto do Termite quando do aplicativo para celular. Fez-se então uma pequena alteração no código com o objetivo de teste e de adquirir mais experiência com as bibliotecas e funções do SDK. No loop principal da função *main*, foi adicionada a função *bsp_button_is_pressed* da biblioteca bsp (da sigla *Board Support Package*). Essa função recebe dois argumentos: uma variável do tipo *uint32_t* que representa a ID de um botão a ser monitorado e um ponteiro para uma variável booleana que armazena se o dado botão está sendo pressionado (variável booleana = 1) ou solto (variável booleana = 0). Essa função foi chamada para verificar o estado dos 4 botões programáveis da placa de desenvolvimento e para tal foi utilizada uma estrutura simples de *for* para alternar entre o monitoramento dos 4 botões. Se algum dos botões fosse monitorado como pressionado, seria enviado a mensagem de "Botão x Pressionado"no terminal de comunicação. A estrutura adicionada ao código encontra-se abaixo:

```
1 for(button=0; button<4; button++)
2 {
3 app_button_is_pushed(button, &pressed)
4    if(pressed==1) printf("\nBotao %d Pressionado\n", button+1)
5 }
```

A indexação dos botões na biblioteca utilizada é de 0 a 3, por isso a necessidade da correção "button+1".

Com o estudo desses e de alguns outros exemplos foi possível desenvolver uma certa prática com as ferramentas e com as principais funções utilizadas nos programas para esse chip, então deu-se início ao desenvolvimento do firmware para a solução de controle de patrimônio. Não será disponibilizado o código final em sua integra visto que o objetivo é desenvolver uma solução que seja única da empresa.

ANEXO C – *Script* automatizado Raspberry-Pi

```
#!/bin/bash     #linha para deixar o script executavel

echo -e $"\n                    IDView - Versao Teste\n"
echo -e "--WiFi Settings--"  #Pede o nome e senha da Rede WiFi para se
    conectar a internet
echo -e "\nWiFi SSID:"
read ssid
echo -e "\nWiFi Password:"
read pass

cd ~/etc/wpa_supplicant/

echo 'network={            #Realiza a configuracao da Rede WiFi
     ssid='$ssid'
     psk='$pass'
}' >> wpa_supplicant.conf

ifconfig wlan0 up       #Habilita o WiFi.

echo -e "--Git Settings--"  #Pede usuario e senha da plataforma Git. Bitbucket
echo -e "\nWiFi Bitbucket Username:"
read bbuser
echo -e "\nWiFi Bitbucket Password:"
read -s bbpass

sudo apt-get update        #Faz atualizacoes do sistema e baixa. bibliotecas

sudo apt-get upgrade

sudo apt-get install git requests bluez bluetooth python-bluez vim

cd /home/pi

#Baixa o codigo do repositorio.

```

```
35 git clone https://$bbuser:$bbpass@bitbucket.org/idviewscp/gateway.git
36
37 sudo '/home/pi/boot.sh' > etc/rc.local #Insere um script de inicializacao no
       arquivo rc.local, de forma a inicializar o programa toda vez que a
       Raspberry for ligada (prevencao em caso de queda de energia).
38
39 sudo reboot   #Reinicia o sistema.
```

ANEXO D – API e Interface com Usuário

A API foi desenvolvida em C# e posteriormente migrada para a plataforma Swagger para testes e documentação. Encontra-se hospedada em um servidor local, assim como a interface gráfica de usuário e os bancos de dados. Na Figura 39, é possível ver a interface do Swagger e os diversos *endpoints* da API.

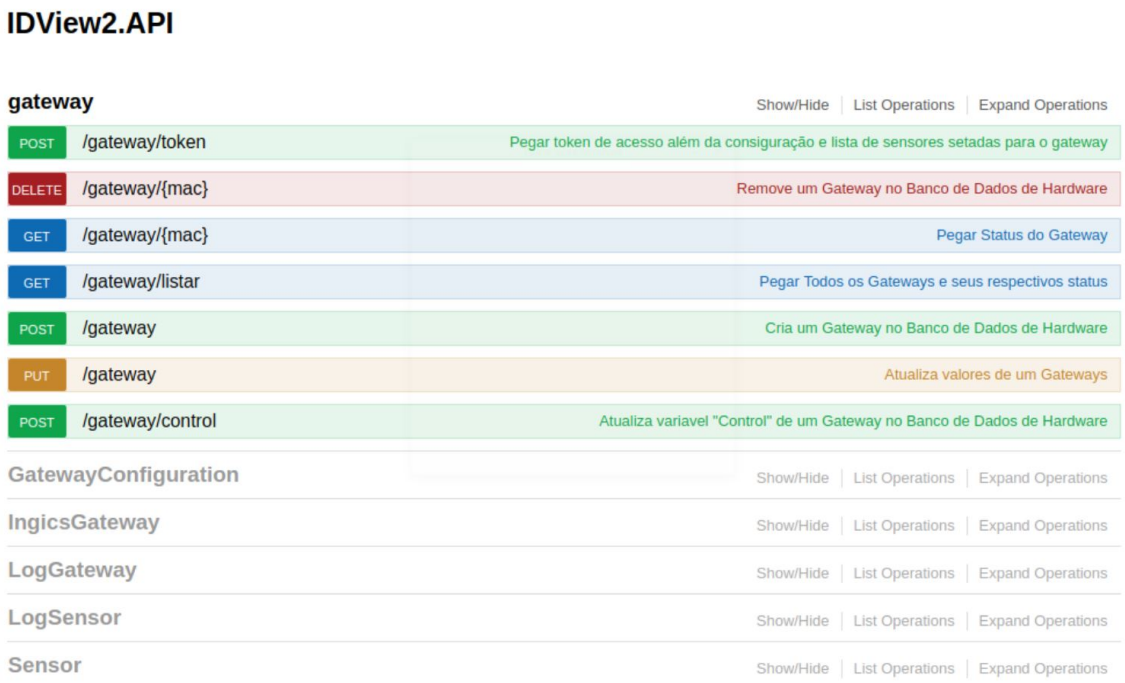

Figura 39 – Interface do Swagger com a API desenvolvida e alguns de seus *endpoints*. Fonte: Imagem cedida pela Innovix.

A interface gráfica de usuário utilizada com o IDView foi aproveitada de uma solução anterior da Innovix. Ela foi desenvolvida em PHP e para ser utilizada com o IDView foram necessários ajustes e adaptações. Planos futuros da empresa incluem o desenvolvimento de uma interface nova, em outra linguagem ainda a definir. Ao acessar o endereço fornecido, o usuário precisa digitar um login e senha para chegar a página inicial (Figura 40). A interface permite o cadastro de ativos a serem monitorados, a organização deles em blocos, salas e áreas (Figura 41) e a associação de sensores, a partir do MAC, a cada ativo. Ao clicar em um ativo(Figura 42), é possível ter acesso a informações detalhadas pelo próprio usuário. A interface utiliza um esquema de cores para diferenciar os alarmes. Os ativos aparecem em vermelho quando é detectado um estado de botão apertado aos sensores associados a eles. A sala, prédio e ambiente também ficam vermelhos nessa situação. Já no caso da detecção de uma desconexão de beacon ou de gateway, é

utilizada a cor laranja. A interface também dá ao usuário a opção de receber os alarmes por SMS e e meios similares.

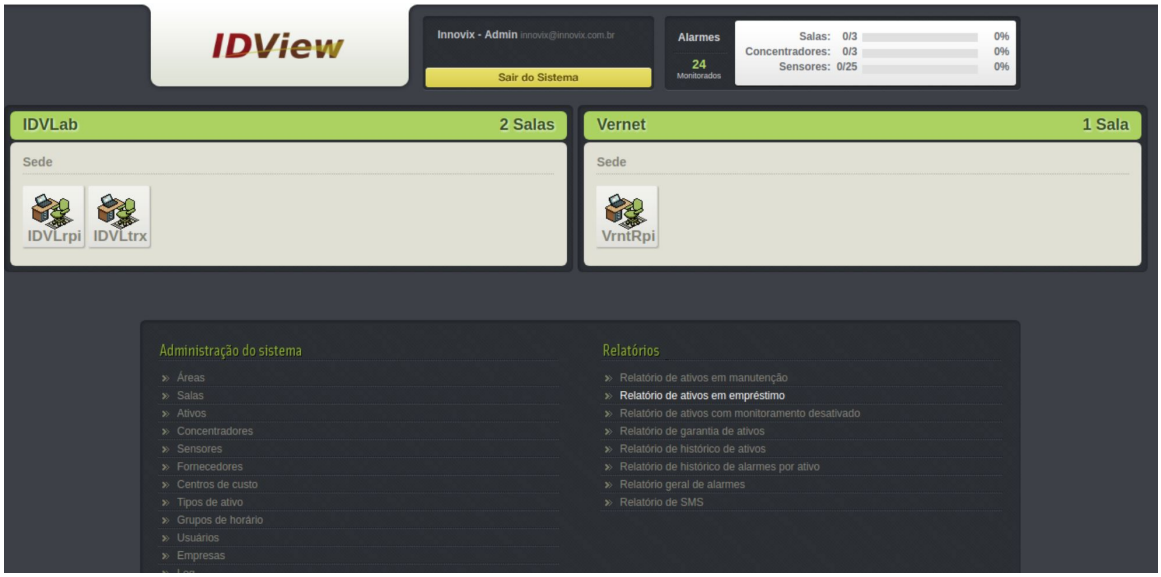

Figura 40 – Página inicial da interface de usuário. Fonte: Imagem cedida pela Innovix.

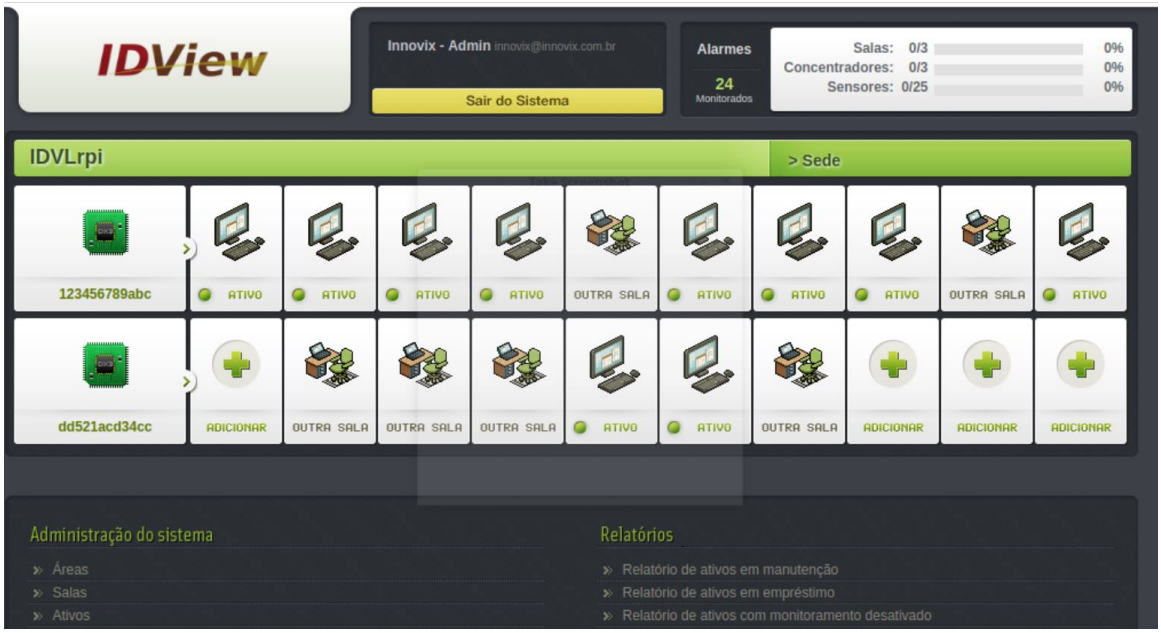

Figura 41 – Divisão entre diferentes salas. Fonte: Imagem cedida pela Innovix.

Figura 42 – Dentro de uma sala, dois gateways e os sensores associados a eles. Fonte: Imagem cedida pela Innovix.