UNIVERSIDADE DE CAXIAS DO SUL

ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

LEONARDO GOBI

PROJETO E IMPLEMENTAÇÃO DE SOLUÇÃO IOT MODULAR COM CONTINGÊNCIA LOCAL

CAXIAS DO SUL

2019

LEONARDO GOBI

PROJETO E IMPLEMENTAÇÃO DE SOLUÇÃO IOT MODULAR COM CONTINGÊNCIA LOCAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador Prof. Me. Alexandre Erasmo Krohn Nascimento

CAXIAS DO SUL

2019

LEONARDO GOBI

PROJETO E IMPLEMENTAÇÃO DE SOLUÇÃO IOT MODULAR COM CONTINGÊNCIA LOCAL

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Trabalho aprovado em 26/06/2019

Banca Examinadora

_____

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

_____

Prof. Dr. André Luiz Martinotto
Universidade de Caxias do Sul - UCS

_____

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

RESUMO

Com a constante evolução da *Internet* das Coisas, inúmeras soluções e produtos surgem no mercado. De modo geral, boa parte das soluções presentes no mercado possuem estrutura limitada a atender uma determinada demanda conhecida, porém se for necessário expandir a arquitetura para suportar novos tipos de dispositivos ou aumentar o número existente, podem haver problemas estruturais que comprometam esta alteração. Outro problema comum em soluções IoT é a indisponibilidade em casos de falhas na conexão com a *internet*, o que pode até trazer problemas ao usuário em alguns tipos de projetos. Neste trabalho foi desenvolvida uma arquitetura para soluções IoT utilizando um *gateway* local conectado a um servidor. Os dispositivos trocam mensagens com o *gateway* que, por sua vez, comunica-se com o servidor. Caso a comunicação falhe o *gateway* pode tomar ações emergenciais pré-configuradas em um painel de controle. As mensagens recebidas durante a ausência de conexão com o servidor podem ser persistidas localmente no *gateway* para posterior envio. O trabalho envolveu ainda o desenvolvimento de uma solução para prevenção de incêndios e acidentes causados por distrações ou vazamentos de gás utilizando fogões. A solução desenvolvida fez uso da arquitetura proposta a fim de validá-la e obteve resultado positivo após os testes executados, demonstrando a eficiência da mesma.

Palavras-chaves: IoT, MQTT, AMQP, *broker*, *Raspberry Pi*, *gateway*, gás, distração.

LISTA DE ILUSTRACÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	*Atomicity, Consistency, Isolation, Durability*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
AWS	*Amazon Web Services*
BLE	*Bluetooth Low Energy*
CAP	*Consistency, Availability, Partition tolerance*
CISC	*Complex Instruction Set Computer*
CPU	*Central Processing Unit*
CSS	*Cascading Style Sheets*
DPMO	*Defeitos por Milhão de Oportunidades*
DOM	*Document Object Model*
GB	*Gigabyte*
GLP	*Gás liquefeito de petróleo*
GPIO	*General Purpose Input/Output*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
IBSG	*Internet Business Solutions Group*
IEC	*International Electrotechnical Commission*
IEEE	*Institute of Electrical and Electronics Engineers*
IOT	*Internet of Things*
IP	*Internet Protocol*
ISO	*International Organization for Standardization*

JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
LAN	*Local Area Network*
MB	*Megabyte*
MIT	*Massachusetts Institute of Technology*
MQTT	*Message Queuing Telemetry Transport*
OSI	*Open System Interconnection*
RAM	*Random-access memory*
REST	*Representational State Transfer*
RFC	*Request for Comments*
RISC	*Reduced Instruction Set Computer*
SSL	*Secure Sockets Layer*
TCC	*Trabalho de Conclusão de Curso*
TLS	*Transport Layer Security*
ULA	*Unidade Lógica e Aritmética*
URL	*Uniform Resource Locator*
USB	*Universal Serial Bus*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Vivemos em uma era de transformação tecnológica, em que a cada dia temos mais dispositivos conectados à *internet*. Junto a esta transformação, podemos observar um constante crescimento no mercado de segurança de perímetro, o que inclui alarmes, câmeras, iluminação, dentre outros dispositivos.

Emergente a alguns anos, o termo *Internet* das coisas, do inglês IoT (*Internet of Things*), vem se tornando mais popular. *IoT* pode ser definido como uma forma de conectar dispositivos de forma a realizarem transmissão e recepção de dados sem nenhuma intervenção humana (COELHO, 2017). Tais dados podem ser gerados por sensores capazes de interagir com determinados ambientes e situações de modo a extrair informações.

Presente em eletrodomésticos, tomadas, lâmpadas, carros, na indústria, sensores fazem parte do dia a dia e podem ser encontrados em qualquer lugar. O Cisco IBSG estima que deverá existir 50 bilhões de dispositivos conectados à *internet* até 2020 (EVANS, 2011).

"A imaginação é hoje o limite para aquilo que se possa conectar à Internet, havendo aplicações e dispositivos de baixo custo para tudo o que nos lembremos de fazer [...]"(COELHO, 2017).

O mercado brasileiro de *IoT* ainda está em fase inicial, na qual grande parte dos produtos do segmento são estrangeiros. Grande parte dos produtos *IoT* encontrados no mercado são destinados a um uso específico, não havendo interação entre os mesmos, criando assim uma descentralização de controles e informações, onde cada produto é controlado de maneira independente. *IoT* pode gerar uma quantidade muito alta de dados, portanto obter uma análise significativa com esta quantidade de dados variados de forma eficiente é um desafio (J.; K, 2018).

Trabalhos anteriores a este que abordavam o tema de automação residencial usavam *hardwares* de custo elevado e não tinham suporte a grande carga de dados, bem como não possuíam uma estrutura de tratamento e persistência de mensagens.

O objetivo deste trabalho foi desenvolver uma arquitetura capaz de tratar a demanda de mensagens geradas por dispositivos *IoT*, visto que existe uma carência de *softwares* que desempenhem tal tarefa. A solução é capaz de realizar a recepção de mensagens oriundas de dispositivos presentes em um determinado ambiente, bem como enviar mensagens a eles, implementando arquitetura contingente para garantir a tomada de decisões críticas mesmo quando não houver comunicação com o servidor.

1.1 PROBLEMA DE PESQUISA

Hoje no mercado pode-se encontrar inúmeros dispositivos IoT para as mais diversas funcionalidades, vindos de *startups*, pessoas que tornam seu *hobbie* um produto, e até mesmo empresas que já atuam no setor de tecnologia. Porém, grande parcela destas soluções carece de uma estrutura que consiga gerir todos os dados gerados e, se a comercialização destas soluções aumentar em larga escala e as mesmas não possuírem uma estrutura adequada, aumentam-se as chances destas soluções falharem.

Um outro ponto levado em consideração é o fato de que a minoria dos dispositivos presentes no mercado atualmente trata a questão da disponibilidade. Podemos ter sensores que nos notificam sobre as mais diversas situações em tempo real, porém, se o sensor ficar sem conexão com a *internet*, o que ocorre com estas notificações? Outro exemplo são os dispositivos que realizam automação de lâmpadas, portões e até mesmo fechaduras. A falta de acesso a *internet* não deveria impedir o funcionamento dos mesmos.

Junta-se a esses desafios a meta de produzir uma solução IoT de baixo custo que permita modularidade, possibilitando que novos dispositivos possam ser anexados à solução com o menor esforço possível. Isto consolida o objetivo deste trabalho.

1.2 QUESTÃO DE PESQUISA

É possível construir uma solução capaz de realizar monitoramento e interação de ambientes através de dispositivos de baixo custo e de forma remota e segura, bem como prover contingência e tomada de decisões predefinidas em situações críticas?

1.3 OBJETIVOS

Desenvolver uma solução *IoT* modular de baixo custo capaz de realizar interação e monitoramento simultâneo de ambientes de forma remota e segura, provendo interação com o usuário através de aplicativo para *smartphone* e possibilitando parametrizações da solução através de painel de controle *web*. A solução deverá contar também com mecanismos de contingência e tomada de ações emergenciais.

Com base no objetivo geral, foram elaborados os seguintes objetivos específicos:

a) Definir um modelo próprio de mensagem para a comunicação entre dispositivo - servidor e selecionar um método de criptografia das mesmas.
b) Implementar estrutura de filas e tópicos de mensagens através do uso dos protocolos de mensagem MQTT e AMQP, ou outros que se adequem à implementação, bem como desenvolver agentes capazes de monitorar, consumir a fila e persistir os dados.
c) Implementar uma API REST (FIELDING, 2010) para que os aplicativos clientes consigam acessar e interagir com a solução, bem como persistir dados

pertinentes.

d) Criar um mecanismo de contingência local através de uma CPU de baixo custo, como uma *Raspberry Pi* (HEATH, 2017), com o objetivo de armazenar temporariamente os dados enviados pelos sensores e que necessitam ser persistidos, realizando a retransmissão ao servidor quando a comunicação volte a ser estabelecida.
e) Planejamento, aquisição e montagem dos módulos e sensores, responsáveis por detectar movimento, luz, relés de acionamento, dentre outras ações a serem definidas no decorrer do trabalho, a fim de realizar a coleta e transmissão dos dados, bem como a recepção e execução dos comandos.
f) Realizar o desenvolvimento dos *softwares* e mecanismos de comunicação que serão usados nos dispositivos acima mencionados.
g) Implementar aplicativo para *smartphone* a fim de possibilitar interações com a solução, como realizar consultas a dados, receber notificações e enviar comandos aos dispositivos.
h) Implementar *software web* para realizar as configurações da solução.

1.4 ESTRUTURA DO TRABALHO

A estrutura deste trabalho é composta por 4 capítulos, que se subdividem da seguinte maneira:

a) No Capítulo 1 pode-se encontrar uma visão geral do tema e trabalho proposto, bem como o problema de pesquisa e os objetivos a serem alcançados.
b) No Capítulo 2 pode-se encontrar uma visão do termo *Internet of Things* e dos desafios trazidos por ela. Também estão contidos assuntos relacionados aos tipos de *hardware* que pode ser encontrado em projetos IoT e alguns dos principais protocolos de comunicação utilizados neste contexto. Por fim, o Capítulo 2 trata de algumas tecnologias de desenvolvimento para a construção dos *softwares web* e móvel propostos pelo trabalho.
c) No Capítulo 3 ocorre a modelagem de uma arquitetura genérica para atender a projetos IoT através da troca de mensagens de forma contingente, permitindo ao *gateway* a tomada de decisões sobre mensagens críticas havendo indisponibilidade do servidor.
d) No Capítulo 4 um problema é proposto, sendo este implementado através do uso da arquitetura apresentada no Capítulo 3.
e) No Capítulo 5 são apresentadas as conclusões finais do trabalho e os trabalhos futuros que podem ser desenvolvidos visando a expansão deste.

2 **REFERENCIAL TEÓRICO**

2.1 *INTERNET* DAS COISAS

O termo *Internet* das Coisas (IoT), do inglês *Internet of Things*, foi proposto em 1999 por Kevin Ashton da MIT Auto Centre (ASHTON, 2009). Descrever o termo, quase 20 anos após sua criação, ainda é uma questão que gera debate. Como o termo é amplo e genérico, vários autores criaram suas próprias definições para o mesmo. Segundo Clark (2016), IoT é uma rede gigante, onde coisas e pessoas estão conectadas, dados são coletados e compartilhados, de forma a identificar como os aparelhos são usados e o estado do ambiente em que estão. No decorrer destes quase 20 anos, o mercado passou por uma grande evolução e inúmeras tecnologias surgiram, difundindo ainda mais o termo. Nos dias atuais, pode-se dizer que é praticamente impossível entrar em uma loja que comercializa itens de tecnologia e não se deparar com algum produto envolvendo IoT.

Para Evans (2011), IoT é essencial para o progresso humano, uma vez que a coleta, transmissão e análise de dados em grande escala proporciona mais conforto e comodidade à população, bem como gera uma vida mais saudável. Dentre as inúmeras áreas de aplicação, algumas das áreas que a IoT se aplica são citadas neste capítulo.

2.1.1 **Áreas de aplicação**

Atualmente, podemos citar uso de tecnologia em quase todos os setores da economia. Direta ou indiretamente, temos contato diário com IoT. Pode-se listar alguns exemplos de uso nas mais diversas áreas, que são indústria, transporte, residências, saúde e agricultura.

2.1.1.1 Indústria

Sensores podem ser instalados em máquinas para realizar o monitoramento de seus componentes em tempo real, bem como notificar necessidade de manutenções. Podem também ser aplicados na segurança, onde máquinas possam ser configuradas para parar de operar assim que oferecem risco ao operador. Magrani (2018) cita que alguns fabricantes adicionam IoT em produtos com o intuito de prover o monitoramento do mesmo, bem como efetuar eventuais manutenções, fazendo disso uma ferramenta de customização.

2.1.1.2 Transporte

Muito tem-se falado sobre carros autônomos nos tempos atuais, porém já conseguimos observar uso de IoT em carros guiados, onde dados são coletados e exibidos para o motorista em tempo real. Segundo (CLARK, 2016), um exemplo de aplicação IoT no

transporte são carros que coletam dados de um determinado veículo e os enviam para seu fabricante. O fabricante por sua vez, consegue diagnosticar possíveis problemas no veículo e providenciar as peças necessárias para reparo, agilizando e sendo mais assertivo no processo de reparo do veículo. Carros elétricos e autônomos são tendências tecnológicas e atualmente vêm ganhando atenção do público e, de acordo com Lopez (2018), os carros elétricos possuem diversas vantagens quando comparados com carros a combustão, como menor manutenção, maior conforto e menor poluição. Já carros autônomos ainda estão sendo testados e seu valor é elevado, cerca de US$ 75.000,00 por um *Tesla Model S*, carro autônomo da empresa *Tesla Motors* (TESLA, 2019).

2.1.1.3 Residências

Foco deste trabalho, a automação residencial está em crescimento constante. Como exemplo, podemos citar lâmpadas automatizadas, cortinas com acionamento automático, eletrodomésticos que interagem com o usuário de forma remota, dentre outras inúmeras aplicações existentes. Com todas estas tecnologias à mão, podemos ter maior conforto e qualidade de vida, bem como economizamos tempo com tarefas rotineiras que agora são realizadas por dispositivos IoT. Clark (2016) exemplifica o uso de IoT através de um dispositivo trivial, o despertador. Um simples despertador tem a tarefa de emitir alarmes em determinados horários, porém se utilizado com IoT pode cruzar estatísticas sobre meteorologia e trânsito, de modo a ajustar o horário do alarme levando em consideração tais dados, evitando assim possíveis atrasos à compromissos.

2.1.1.4 Saúde

A IoT é uma grande aliada da saúde, sendo que um dos seus maiores objetivos é disponibilizar informações de forma eficiente. Como menciona Magrani (2018), dispositivos voltados a saúde podem aproximar médicos e pacientes além de maior eficiência na coleta de dados voltados à área.

Gadgets como pulseiras e relógios já conseguem informar aos seus donos dados sobre frequência cardíaca, número de passos diários, alertas de ociosidade, dentre outras métricas existentes.

2.1.1.5 Agricultura

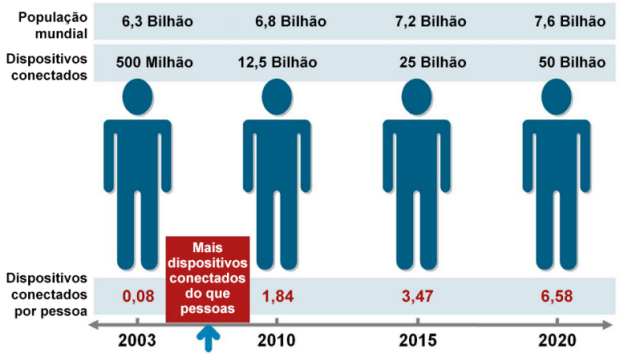
De acordo com Albertin e Albertin (2017), estudos de organizações internacionais apontam que a população mundial precisará crescer perto de 60%, enquanto o crescimento de terra produtiva deverá crescer cerca de 5%. Dado estes valores, a IoT é fator fundamental para que a agricultura consiga suprir toda esta demanda. Através da tecnologia, uma série de fatores podem ser monitorados e calculados, gerando maior precisão no uso de fertilizantes,

maior economia e alta produtividade.

2.1.2 **Desafios**

Conforme Albertin e Albertin (2017), projeções indicam aproximadamente 50 bilhões de dispositivos conectados até 2020. Ainda, segundo o Ministério da Ciência, Tecnologia e Inovação MCTIC (2019) e a Agência Nacional de Telecomunicações ANATEL (2019), o Brasil poderá ter 43 milhões de dispositivos conectados em 2020 e, em 2025, ultrapassará os 100 milhões e podendo chegar a 200 milhões. Através de um estudo feito pela Evans (2011), ilustrado na Figura 1, pode-se observar o número de dispositivos por habitante no mundo em função do tempo.

Figura 1 – População Mundial X Dispositivos conectados



Fonte: EVANS (2011)

Este crescimento de dispositivos conectados tende a crescer em ritmo cada vez mais acelerado, gerando uma quantidade de dados cada vez maior. Um dos desafios criados pela IoT é gerir todas estas informações. Serviços podem ter milhares de sensores enviando e recebendo dados simultaneamente, e sem uma estrutura robusta não é possível tratar toda esta massa de dados.

Outro fator importante a ser levado em consideração no momento do projeto de uma arquitetura para IoT é a escalabilidade. Em alguns cenários, podemos ter picos na geração de dados, sendo necessário aumentar os recursos computacionais em momentos específicos do dia, para tratar tais demandas.

Para Evans (2011), a falta de normas técnicas ainda é um desafio, principalmente nas áreas de comunicação, arquitetura, privacidade e segurança, áreas que são a base e de fundamental importância para IoT.

A segurança da informação deve ser motivo de atenção em qualquer comunicação com a *internet*, uma vez que o número de ataques está em constante crescimento. Com a IoT a atenção deve ser redobrada, uma vez que sensores podem impactar diretamente na

vida de seus usuários, e qualquer comando executado não proveniente do usuário pode trazer riscos a ele.

Conforme aponta Albors (2018), alguns estudos estimam que gastos com segurança em IoT projetam crescimento a ponto de triplicar entre 2018 e 2023. Problemas envolvendo privacidade são os mais recorrentes nos dispositivos atuais, onde lacunas de segurança no desenvolvimento causam brechas para que pessoas mal-intencionadas as explorem e coletem dados sensíveis de forma transparente. Albors (2018) ressalta também que um especialista de segurança da ESET[1], através de uma pesquisa, analisou doze produtos IoT para casa inteligente, sendo que todos eles apresentaram algum problema que poderiam colocar a solução em situação vulnerável, a ponto de comprometer a privacidade do produto.

Para algumas aplicações e sensores específicos, a autonomia da bateria é um fator importante e deve ser levado em consideração. Projetar baterias com autonomia cada vez maior e tamanho menor ainda faz parte dos desafios da IoT.

Não menos importante, o custo dos dispositivos pode tornar-se decisivo na escolha de uma solução IoT. Na Seção 2.2 e Seção 2.3 são citados alguns exemplos de dispositivos que podem ser utilizados em projetos IoT, permitindo que este conceito se torne cada vez mais acessível e que novas ideias inovadoras estejam ao alcance do maior público possível.

2.1.3 **Modelos de comunicação IoT**

De acordo com a RFC 7452 descrita por Tschofenig et al. (2015), definem-se 4 modelos de comunicação IoT, descritas nas subseções que se seguem.

2.1.3.1 *Device-to-Device*

Neste modelo, a comunicação ocorre diretamente entre dois dispositivos, os quais podem ser de fabricantes distintos, uma vez que sigam os mesmos protocolos, de forma a padronizar a comunicação. Como exemplo, pode-se citar um *smartphone* que se comunica com uma chaleira inteligente através de conexão BLE (*Bluetooth Low Energy*), conforme Figura 2, enviando-lhe comandos para, por exemplo, aquecer a água em determinada temperatura no horário definido.

[1] ESET: Empresa desenvolvedora de soluções e segurança para a *internet*.

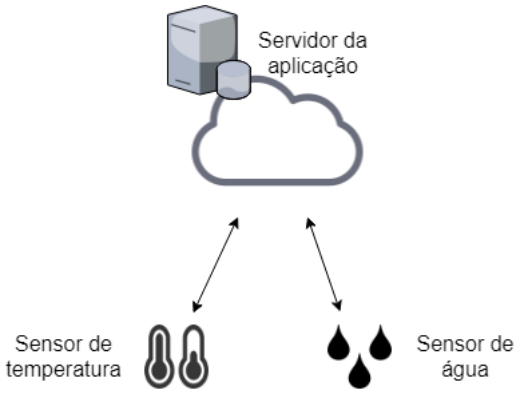
Figura 2 – Exemplo modelo de comunicação Device-to-Device

Fonte: Elaborado pelo autor (2018)

2.1.3.2 *Device-to-Cloud*

O modelo *Device-to-Cloud* é designado a dispositivos que se comunicam diretamente a servidores remotos, realizando a comunicação através da *internet*. Através deste formato de comunicação é possível eliminar equipamentos intermediários que realizariam o intermédio das mensagens entre dispositivos e servidor. Por exemplo, pode-se ter sensores de temperatura e água enviando dados diretamente ao servidor da solução, como pode ser visto na Figura 3.

Figura 3 – Exemplo modelo de comunicação Device-to-Cloud



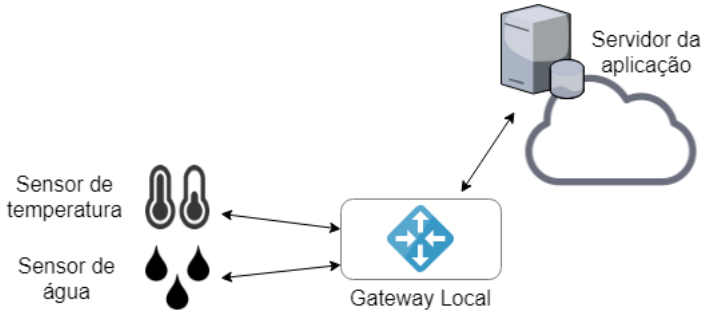
Fonte: Elaborado pelo autor (2018)

2.1.3.3 *Device-to-Gateway*

No modelo *Device-to-Gateway* o dispositivo também pode enviar dados para um servidor remoto, porém a comunicação passa por um *gateway*, responsável pela comunicação com a *internet*, intermediando a troca de mensagens.

Dentre as aplicações deste modelo de comunicação, pode-se ressaltar o uso em tecnologias vestíveis (*wearables*) e dispositivos sem necessidade de conexão constante com a *internet*, uma vez que geralmente usam o *smartphone* como *gateway*, por exemplo pulseiras e *smartwatches* transmitindo dados coletados para posteriores cálculos estatísticos. Outro exemplo de aplicação é para centralizar o envio de dados dos sensores, conforme mostrado na Figura 4.

Figura 4 – Exemplo modelo de comunicação Device-to-Gateway

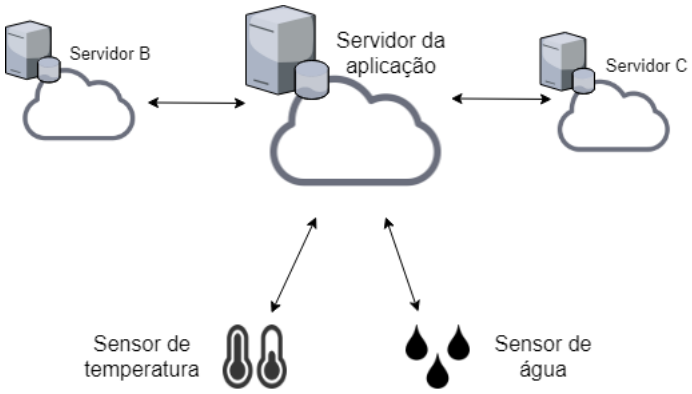


Fonte: Elaborado pelo autor (2018)

2.1.3.4 *Back-End Data Sharing*

Após o recebimento das informações enviadas pelos sensores, em algumas aplicações torna-se necessário o processamento e análise destas mensagens junto a outras fontes de dados, o que caracteriza este modelo de comunicação. Os dados recebidos dos sensores podem, por exemplo, serem acessados de outros serviços através de uma API *Restful*, para tornar possível a análise e cruzamento de dados, ou ainda para comparação com outros parâmetros. A Figura 5 ilustra um exemplo de aplicação.

Figura 5 – Exemplo modelo de comunicação Back-End Data Sharing



Fonte: Elaborado pelo autor (2018)

2.2 MICROCONTROLADORES

Atualmente, o mercado de microcontroladores e embarcados está em expansão. Novidades surgem o tempo todo e cada vez mais desenvolvem-se dispositivos com menor custo e maior capacidade. Segundo Reis (2015), podemos definir um microcontrolador como uma CPU de baixo custo e poder computacional, contendo sub-circuitos capazes de realizar o processamento das instruções, o gerenciamento da entrada e saída de dados e sinais, temporizar o circuito, converter sinais, dentre outras atividades. Normalmente estão atrelados a algum outro circuito, onde sua tarefa pode fazer parte de um objetivo maior,

sendo essencial para o alcance do mesmo.

No segmento de IoT, microcontroladores estão presentes em praticamente todos os dispositivos, atuando na leitura e envio de informações, bem como no recebimento e execução de comandos. Tanenbaum (2007) escreveu em 2007 que carros de alto padrão poderiam conter facilmente 50 microcontroladores, responsáveis por executar e gerenciar vários circuitos elétricos do veículo. Hoje, mais de 10 anos depois, pode-se presumir que este número deve ser maior. “Uma família poderia possuir facilmente centenas de computadores sem saber. Dentro de alguns anos, praticamente tudo o que funciona por energia elétrica ou baterias conterá um microcontrolador.” (TANENBAUM, 2007).

Dentre os microcontroladores existentes, neste trabalho optou-se, devido ao seu custo/benefício, pelo microcontrolador ESP8266, desenvolvido e produzido pela empresa Chinesa *Espressif Systems*. Dentre suas principais características, destaca-se a comunicação sem fio por tecnologia *WiFi*, operando na faixa de 2.4 GHz, implementando os recursos do protocolo TCP/IP. Ele utiliza uma unidade de processamento RISC *Tensilica* L106 32-bit, operando em uma velocidade máxima de 160 MHz.

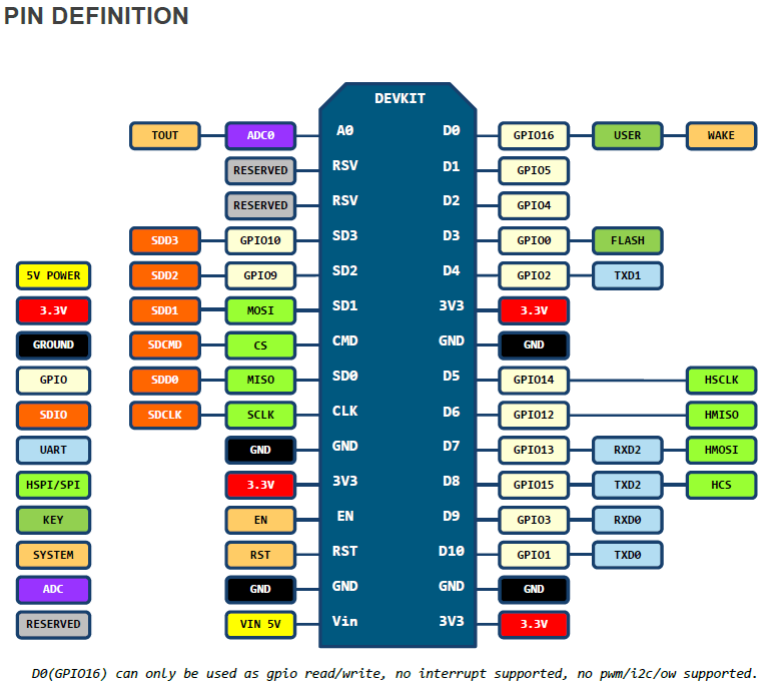
Sendo projetado para ser usado em dispositivos móveis e em aplicações que requerem autonomia de energia através do uso de baterias, o dispositivo possui tecnologia para otimização de consumo elétrico, o tornando uma boa opção para projetos com esta característica. Além disso, o mesmo opera em uma faixa de temperatura bem flexível, podendo ser implementado em ambientes inóspitos e de grande oscilação climática.

O dispositivo possui 17 pinos de sinal digital com propósito geral (GPIO), onde o uso dos mesmos é definido pelo arquiteto da solução. Possui também 3 pinos para comunicação serial (SPI), utilizado para comunicação de curta distância (SYSTEMS, 2018). Mais informações podem ser encontradas no *datasheet* disponibilizado pelo fabricante, em (SYSTEMS, 2018).

Uma vez que o ESP8266 é bastante difundido comercialmente, surgiram no mercado alguns módulos e placas de prototipagem que embarcam este microcontrolador. Dentre os existentes, um que se destaca é o NodeMCU ESP8266.

O dispositivo NodeMCU é um hardware *open source* que utiliza um microcontrolador ESP8266. Possui preço acessível e competitivo, cerca de 20 a 30 reais. Implementa todas as características do ESP8266, acrescido de conexão USB para possibilitar a gravação do software, bem como componentes para regulagem de tensão, conversor USB X Serial e outros componentes necessários para o tornar operacional. Na Figura 6 pode-se observar um diagrama identificando as portas disponíveis no dispositivo.

Figura 6 – Diagrama de pinos do NodeMCU ESP8266



Fonte: PALANISAMY (2016)

O diagrama de *hardware* do dispositivo, bem como todos os subcomponentes necessários para sua fabricação podem ser encontrados em (TEAM, 2015).

2.3 MICROPROCESSADORES

Diferente de microcontroladores, os microprocessadores são circuitos muito mais robustos e complexos, possuindo milhares de transistores, custo mais elevado e, de modo geral, são projetados para atender o mercado de forma mais ampla e genérica, possuindo aplicações em diversas plataformas e cenários.

A composição básica de uma CPU, com base em Tanenbaum (2007), parte de: uma unidade de controle, responsável por buscar instruções na memória principal; uma unidade lógica e aritmética (ULA), encarregada de efetuar operações lógicas e aritméticas e registradores, que fornecem um armazenamento temporário de resultados.

Existem atualmente duas arquiteturas de processadores, denominadas CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instructions Set Computer*). Processadores CISC são caracterizados por possuírem unidades de controle poderosas e um grande conjunto de instruções, possibilitando assim a execução de tarefas complexas e facilitando a implementações de compiladores. RISC, por sua vez, pode ser considerada um marco da história dos processadores, a qual faz uso de um conjunto de instruções pequeno e de

formato fixo, unidades de controle mais simples que CISC e um grande número de registradores. Alguns dos motivos pelos quais o desempenho da arquitetura RISC ser superior à CISC devem-se ao fato de que todas as instruções comuns são executadas pelo hardware, não sendo interpretadas por microinstruções e o *pipelining* realizar a execução de uma instrução por ciclo (STALLINGS, 2010), (TANENBAUM, 2007). Um exemplo de arquitetura RISC são os processadores ARM, utilizados pelo dispositivo *Raspberry Pi*, descrito na Subseção 2.3.1, dentre outros existentes no mercado, como:

a) *Orange Pi*: placas *open source* que utilizam processador RISC ARM, possuindo diversos modelos com variadas opções de configuração. Possuem comunicação WiFi e suporte à entrada/saída. Utilizam sistemas operacionais como Android, Debian, dentre outros (XUNLONG, 2018).
b) *Asus Tinker Board*: placa robusta desenvolvida pela ASUS, também utiliza processador RISC ARM. Suas configurações são superiores ao *Raspberry*, embarcando um processador *quad-core* e memória RAM de 2 GB. Devido a sua configuração ser superior, seu valor é bem maior, cerca de R$ 450,00 no Brasil (ASUSTEK, 2018).
c) ROCK64: placa desenvolvida pela PINE64, possui configuração mais robusta do que a *Raspberry*, utiliza processador ARM *quad-core* e alguns modelos incorporam memória RAM de até 4 GB. Esta placa não possui muitos pontos de comércio no Brasil e seu custo parte de, aproximadamente, R$ 600,00 (PINE, 2018).

Dispositivos microprocessados são normalmente utilizados em *gateways* devido ao maior desempenho e dispositivos microcontrolados são geralmente utilizados junto aos sensores e dispositivos de ação.

2.3.1 ***Raspberry Pi***

Raspberry é um computador de valor acessível que utiliza um microprocessador RISC ARM. Possui suporte a entrada e saída através de portas USB, como teclado e mouse, bem como possui saída para monitor externo. Seu tamanho é de aproximadamente um cartão de crédito.

O dispositivo é utilizado através de um sistema operacional, que deve ser instalado quando o dispositivo é adquirido, e o mesmo fica armazenado em um cartão de memória inserido no dispositivo. Atualmente existem várias distribuições de sistemas operacionais disponíveis, e algumas são listadas a seguir.

a) Raspbian: distribuição oficial do sistema operacional Linux para os dispositivos *Raspberry*, sendo compatíveis com todos os modelos fabricados.
b) Ubuntu *Core*: distribuição do sistema operacional Linux Ubuntu que pode ser utilizada na *Raspberry* 2 e 3.

c) Windows 10 IoT *Core*: distribuição desenvolvida pela Microsoft, utilizado principalmente para prover comunicação com os serviços IoT da empresa.

Dentre os modelos de *Raspberry Pi* existentes, pode-se encontrar no mercado com maior facilidade os modelos *Raspberry Pi 3* e *Raspberry Pi Zero W*, sendo este último uma versão mais compacta e de *hardware* inferior. Levando isso em consideração, foi optado pelo modelo *Raspberry Pi 3*, pois possui maior memória e capacidade de processamento.

A placa *Raspberry Pi* 3 *model* B foi lançada em fevereiro de 2016. Suas especificações de *hardware* são: Processador ARM 1.2GHz, 1GB de memória RAM, suporte para cartão MicroSD, comunicação com fio através de porta LAN e sem fio por *Bluetooth* (V4.1) e WiFi (802.11n). O suporte a entrada/saída é feito através de quatro portas USB, uma conexão HDMI, uma conexão de áudio P2, uma saída para *display* LCD e uma entrada para câmera. Além destas características, ela também possui pinos GPIO de uso geral, assim como em microcontroladores, citados na Seção 2.2. (ZIEMANN, 2018).

2.4 MÓDULOS E SENSORES

Podendo ser considerado o centro do IoT, os módulos e sensores são fundamentais para qualquer projeto. Microcontroladores e microprocessadores fazem a leitura, transmissão e recepção de dados, os quais são gerados ou interpretados por sensores ou módulos.

Sensores são dispositivos que, dado um determinado ambiente e situação, podem interagir com o mesmo e extrair informações, convertendo estas para sinais analógicos ou digitais. Tais informações podem vir das mais variadas fontes, através da detecção de temperatura, umidade, gases, movimentos, luminosidade, dentre outras diversas aplicações.

Outra abordagem adotada junto aos sensores são os módulos, permitindo que um ou mais componentes eletrônicos possam ser montados em uma placa, de modo a facilitar o desenvolvimento de projetos, fornecendo maior praticidade na montagem dos circuitos e exigindo menor conhecimento em eletrônica. Além de sensores e módulos capazes de extrair informações, existem também módulos capazes de realizar ações, como um relé capaz de interromper a corrente de um fio, ou uma válvula que é aberta e fechada eletronicamente.

Como exemplo, pode-se citar alguns sensores e suas respectivas funcionalidades:

a) Sensor de presença: desenvolvido através da tecnologia infravermelho, pode detectar movimentos com distâncias de até 7 metros, possibilitando ajustes na distância de detecção e intervalo de tempo entre capturas.
b) Sensor de temperatura e umidade: pode ser constituído por um módulo capaz de detectar temperatura e umidade capacitiva, transformando tais leituras em sinais digitais.

c) Sensor de luminosidade: dispositivo capaz de detectar a luminosidade e converter em sinal. Normalmente realizado através de um sensor LDR (*Light Dependent Resistor* - Resistor dependente de luz), onde a resistência oriunda do nível de luz que incide sobre um material semicondutor varia, sendo esta resistência transformada em tensão.
d) Sensor de corrente: sensor utilizado na aferição da corrente que está sendo conduzida por um cabo elétrico.
e) Sensor de gases: dispositivos capazes de detectar vários tipos de gases e fumaças.
f) Relé mecânico e de estado sólido: dispositivos com funcionamento similar a um interruptor, possibilitando e interrompendo a passagem de energia em um determinado meio. O relé mecânico tem funcionamento baseado em uma bobina e um eletroímã, já o relé de estado sólido, um semicondutor com mesmas funções de um relé mecânico, não possui partes móveis.
g) Válvula solenoide: a válvula solenoide é uma válvula mecânica controlada eletronicamente, a qual permite passagem e interrupção de fluxo de substâncias que passam pela válvula.

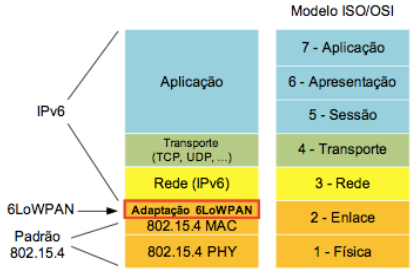
2.5 PROTOCOLOS DE COMUNICAÇÃO

Atualmente existem inúmeros protocolos para as mais variadas aplicações, tornando o estabelecimento de padrões um desafio para a IoT. Osako e Gimenez (2016) citam que hoje grande parte dos protocolos IoT são baseados no modelo TCP/IP, permitindo o uso de estruturas de comunicação existentes, consolidadas e difundidas no mercado, podendo ser enumeradas as redes WiFi, 3G, Ethernet, dentre outras. Além dos protocolos que rodam sob IP, existem também outros protocolos com implementações diferentes, dentre estes, os 2 com maior difusão no mercado são:

a) 6LoWPAN: seu nome vem de *IPv6 over Low Power Wireless Personal Area Network* e seu conceito básico é comunicar dispositivos através de pacotes IPv6 sob redes que seguem o padrão IEEE 802.15.4, desenvolvido para especificar redes sem fio pessoais com baixas taxas de transmissão. Possui ênfase no baixo consumo de energia utilizado para fazer o transporte. Mais informações podem ser encontradas em (OLSSON, 2014).
b) ZigBee: padrão criado com o intuito de tornar o uso de energia o menor possível durante comunicações com dispositivos, utilizando como base a *standard* IEEE 802.15.4, mesma utilizada pelo 6LoWPAN. Para fazer uma comunicação ZigBee, os dispositivos devem estar a uma distância máxima de funcionamento. A taxa de transmissão é pequena ao comparar com outras formas de comunicação, Saleiro e Ey (2018) pontua que é de aproximadamente 250Kbps. Mais informações estão presentes em (SALEIRO; EY, 2018).

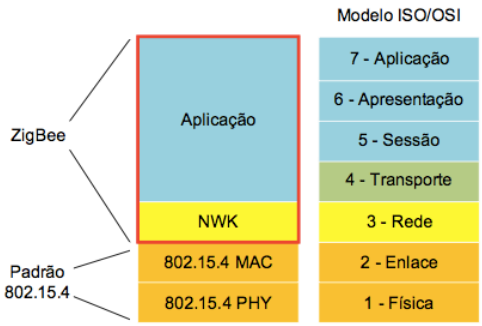
Os protocolos 6LoWPAN e ZigBee não se comunicam diretamente a redes IP e *internet*. Osako e Gimenez (2016) cita que é necessário o uso de *gateways* para realizar estas comunicações. Ressalta também que no caso do padrão 6LoWPAN, por se tratar do mesmo protocolo, a comunicação é facilitada, porém já no padrão ZigBee, a comunicação é complexa, uma vez que o protocolo não é o mesmo. Na Figura 7 pode-se observar uma comparação entre os padrões e o modelo OSI.

Figura 7 – Comparações entre 6LoWPAN e ZigBee com o modelo OSI



(a) 6LoWPAN

Fonte: IPV6.BR (2013)



(b) ZigBee

Fonte: IPV6.BR (2013)

Referente aos protocolos baseados em IP, os principais são enumerados e detalhados nas subseções que se seguem.

2.5.1 *Hyper Text Transfer Protocol* (HTTP)

HTTP, do inglês *Hyper Text Transfer Protocol*, é um protocolo de comunicação da camada de aplicação baseado na transmissão de documentos hipertexto, trafegando sob o protocolo TCP/IP. Seu funcionamento baseia-se em requisição e resposta. A comunicação ocorre entre cliente e servidor, onde o cliente faz uma requisição ao servidor e obtém uma resposta.

Conforme Gourley et al. (2002), o HTTP suporta vários comandos de requisição, chamados de métodos, que também são conhecidos como verbos HTTP. Os mais utilizados são descritos no Quadro 1.

É através destes métodos que ocorre a comunicação entre o cliente e o servidor. Além disso, todas as respostas HTTP possuem um código de status, facilitando a identificação de um sucesso ou erro. Eles podem ser divididos em 3 grandes categorias, sendo que cada uma das categorias contém vários códigos de status. O Quadro 2 enumera as categorias de status no protocolo HTTP.

Junto ao HTTP temos o padrão REST, do inglês *REpresentational State Transfer*, que, segundo Saudate (2014), é um estilo de programação de *web services*. Ainda de

Quadro 1 – Principais métodos HTTP

Método	Descrição
GET	Utilizado para obter informações. Ocorre o envio de determinado dado do servidor para o cliente.
POST	Utilizado para envio de dados através do corpo da requisição do cliente para o servidor, possibilitando ao servidor o armazenamento dos mesmos.
PUT	Cria ou altera um determinado dado no servidor. Sua principal característica é ser idempotente, onde pode-se chamar mais de uma vez o verbo PUT sem ocasionar inconsistências ou duplicidades.
DELETE	Requisita a remoção de um dado do servidor.
PATCH	Requisita alteração de somente determinados campos em um elemento no servidor, evitando o envio desnecessário de todo o elemento.
HEAD	Retorna o cabeçalho de uma resposta, sem o corpo.

Fonte: Elaborado pelo autor (2018)

Quadro 2 – Classificação dos códigos de status HTTP

Grupo global	Grupo definido	Categoria
100-199	100-101	Informacionais
200-299	200-206	Sucesso
300-399	300-305	Redirecionamento
400-499	400-415	Erro no cliente
500-599	500-505	Erro no servidor

Fonte: Traduzido de GOURLEY et al. (2002)

acordo com Saudate (2014), o REST tem sua fundamentação no HTTP através do uso adequado de verbos HTTP e URLs, uso dos códigos de *status* mencionados no Quadro 2, dentre outros itens que venham a caracterizar boas práticas.

De acordo com REST, cada entidade da aplicação é tratada como um recurso. Estes recursos, por sua vez, devem ser acessados através de URLs próprias, legíveis e padronizadas. Exemplos de URLs padronizadas são: “http://exemplo.rest/pessoas” e “http://exemplo.rest/pessoas/11708dae-84a2-4342-8095-3254ce80a09f”. Ao visualizar estas URLs, fica fácil compreender que, ao usar um verbo GET com a primeira URL, todas as pessoas serão retornadas. Logo, ao usar GET com a segunda URL, a pessoa com aquele ID passado na URL será retornada.

O protocolo HTTP é muito utilizado para requisições envolvendo cliente e servidor, porém, por ser síncrono, ele sempre aguarda por uma resposta, o tornando ineficiente em termos de escalabilidade. Para tanto, existem protocolos específicos para comunicação assíncrona e IoT que se diferenciam do HTTP por várias características além da assincronia, como por exemplo tamanho do pacote menor, padrões de mensagens diferenciados,

garantia de ordem de entrega, dentre outras. Alguns destes protocolos são descritos nas subseções seguintes.

2.5.2 ***Message Queuing Telemetry Transport*** **(MQTT)**

O protocolo MQTT, do inglês *Message Queuing Telemetry Transport*, segundo Michael (2017), é um protocolo designado a IoT, o qual foi criado pela IBM no final de 1990 com a finalidade de comunicar sensores ligados a atividades com petróleo. Assim como o protocolo HTTP citado na Subseção 2.5.1, ele pertence à *stack* do protocolo TCP/IP.

Seu funcionamento baseia-se em um sistema de *publish/subscribe* (publicação assinatura) e está consolidado sobre a ISO/IEC PRF 20922. Michael (2017) ainda salienta que, embora o nome contenha o termo “fila”, o protocolo não possui ligação com estas.

Segundo Michael (2017), pelo fato do protocolo MQTT ser assíncrono, ele pode ser escalável até mesmo em redes não confiáveis, uma vez que o emissor e o receptor não estão acoplados. Desde 2014 ele é um padrão da Organização para o Avanço de Padrões em Informação Estruturada (OASIS - *Organization for the Advancement of Structured Information Standards*).

O protocolo possui um servidor responsável pelo recebimento e encaminhamento das mensagens (*broker*), funcionando de forma similar a um roteador. Optou-se por não traduzir o termo “*broker*” ao longo do trabalho por não possuir uma palavra com total correspondência de característica na língua portuguesa.

O protocolo possui também clientes, responsáveis pela geração (*publishers*) e recebimento (*subscribers*) das mensagens. Implementa um sistema de tópicos, que funcionam como rótulos, possibilitando ou impedindo o acesso de determinados clientes a determinadas mensagens.

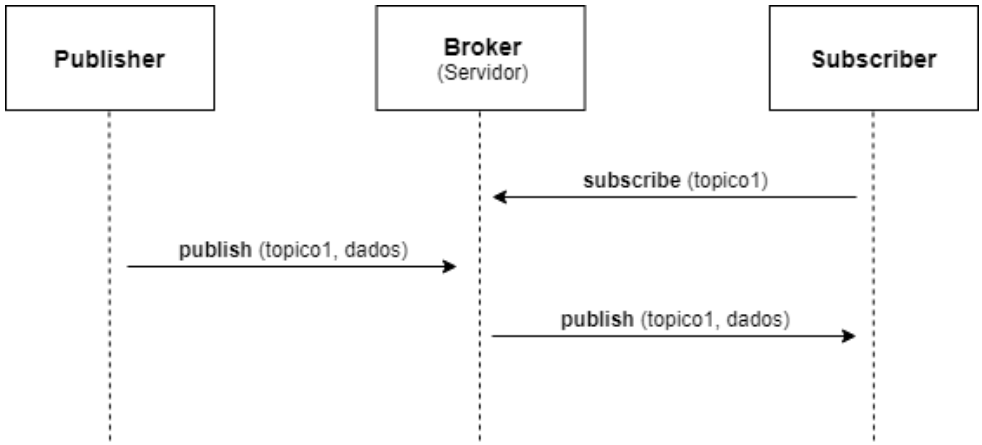
Durante seu funcionamento, clientes podem conectar-se ao *broker* e inscrever-se (*subscribe*) para o recebimento de mensagens oriundas de determinados tópicos; Quando um determinado cliente envia ao *broker* uma mensagem e um tópico (realizando *publish*), o mesmo encaminhará a mensagem para todos os clientes que inscreveram-se neste tópico. A Figura 8 ilustra este funcionamento.

A estrutura de um pacote do protocolo MQTT é basicamente constituída por 3 itens:

a) Cabeçalho fixo: possui informações sobre tipo do pacote de controle, *flags* para os tipos de pacotes de controle e informação sobre o número de bytes restantes do pacote, contando com o cabeçalho variável e *payload*.
b) Cabeçalho variável: presente em alguns tipos de pacotes, seu conteúdo varia de acordo com o tipo do mesmo. Possui informações referentes a identificação do pacote.

c) *Payload*: possui a mensagem a ser enviada. Alguns tipos de pacotes não necessitam *payload*.

Figura 8 – Diagrama de funcionamento *publish-subscribe* MQTT



Fonte: Adaptado de MCKAY (2014)

Outro ponto importante do protocolo MQTT são os tópicos. Em uma publicação de mensagem, deve-se informar o tópico em que a mensagem deve ser publicada, funcionando como uma espécie de canal. Como explica OASIS (2015), o caractere "/" é o responsável por separar os níveis da árvore de tópicos.

Além da separação, existem os caracteres curingas "+" e "#" que significam, respectivamente, curinga de nível único (*single level wild card*) e curinga multi-nível (*multi-level wild card*). *Single level wild card* corresponde a somente um nível de tópico e *multi-level wild card* corresponde a qualquer nível em um tópico. A seguir é exemplificado o uso de tópicos e de caracteres curingas.

a) casa18278/cozinha/dispositivo/scl1/iluminacao
b) casa18278/cozinha/dispositivo/scl2/iluminacao
c) casa18278/cozinha/dispositivo/sct1/temperatura
d) casa18278/quarto/dispositivo/sql1/iluminacao
e) casa18278/quarto/dispositivo/sqt1/temperatura

Os tópicos acima exemplificam uma casa que possui 2 sensores de luminosidade na cozinha e um no quarto, bem como um sensor de temperatura em cada um destes ambientes. A partir destes tópicos, pode-se observar o uso dos caracteres curingas da maneira que se segue:

a) "casa18278/cozinha/dispositivo/+/iluminacao": Utilizando o curinga "+", neste exemplo pode-se inscrever-se no recebimento de todas as mensagens de dispositivos de iluminação da cozinha.

b) “casa18278/cozinha/dispositivo/#”: Com o caractere curinga “#” pode-se obter qualquer mensagem abaixo de um determinado nível, como neste exemplo, todas as mensagens oriundas de qualquer dispositivo na cozinha. OASIS (2015) ressalta que o curinga “#” deve ser o último caractere especificado no filtro.

Segundo Piper (2013), o foco do protocolo MQTT é bem mais simples que o protocolo AMQP (*Advanced Message Queuing Protocol*), apresentado na Subseção 2.5.3, uma vez que não dispõe de sistema de filas e seu projeto foi idealizado com base em dispositivos de poder computacional limitado e providos de baixa largura de banda. Neste trabalho é apresentado, na Subseção 2.6.1, um intermediador de mensagens capaz de trabalhar tanto com o protocolo MQTT quanto o AMQP, sendo possível a utilização de ambos em conjunto.

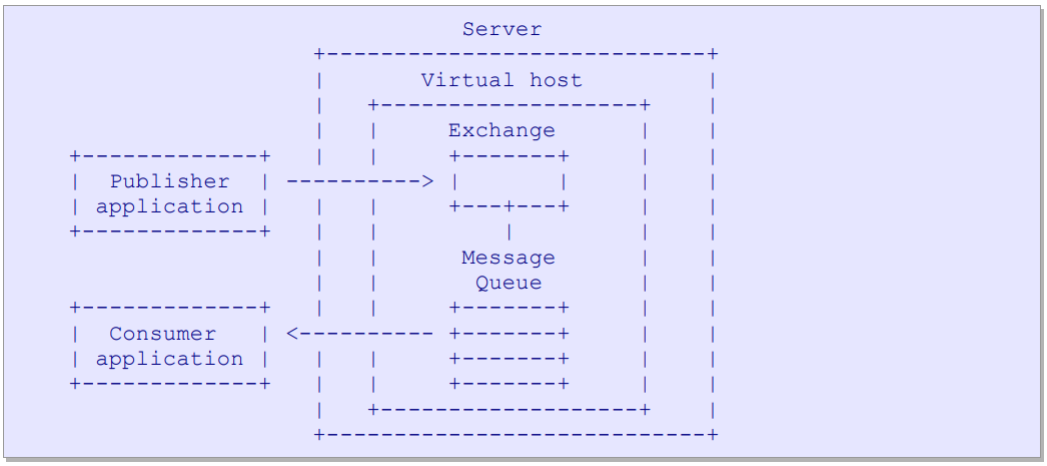
2.5.3 ***Advanced Message Queuing Protocol* (AMQP)**

O Protocolo Avançado de Filas de Mensagem (AMQP - *Advanced Message Queuing Protocol*) é, segundo Conway et al. (2008), uma especificação de protocolo de rede binário para trocas de mensagens assíncronas, incluindo uma definição dos serviços a serem executados pelo servidor. Dentre suas características, ressalta-se o fato de ser multicanal, assíncrono e eficiente. Também atua sob o protocolo TCP/IP.

O protocolo AMQP possui mais de uma versão, sendo 1.0 (OASIS, 2012) a versão mais recente, mantida pela OASIS. Porém, esta não especifica a arquitetura do *broker*, definindo somente o protocolo. Devido a versão 0.9.1 existir a mais tempo e possuir definições de implementação de *broker*, a mesma será tratada neste trabalho, sendo descrita nesta subseção.

O funcionamento do protocolo se dá através de filas mantidas pelo servidor (*broker*), nas quais as mensagens recebidas são persistidas para posterior entrega. As próximas subseções apresentam os principais elementos que compõem o fluxo de recebimento e encaminhamento do protocolo, desde a recepção da mensagem, o roteamento das mesmas (*exchange*) com base nos *bindings* e armazenamento nas filas para consumo dos clientes, conforme pode ser observado na Figura 9.

Figura 9 – Diagrama básico de funcionamento do protocolo AMQP



Fonte: CONWAY et al. (2008)

2.5.3.1 *Virtual host*

Richardson (2012) cita o conceito de *virtual hosts*, que tem como princípio a criação de partições dentro do servidor, contendo seus próprios componentes, como filas, *exchanges*, *bindings* e demais associados, criando isolamento e segurança para as aplicações.

Um *virtual host* possui um ambiente completamente isolado de outro *virtual host*, garantindo isolamento entre os mesmos. Segundo Conway et al. (2008), o método de autenticação do servidor é compartilhado entre todos os *virtual hosts*, porém o método de autorização pode ser exclusivo a cada *virtual host*, sendo útil em casos de compartilhamento de infraestrutura.

2.5.3.2 *Bindings*

Bindings são considerados ligações entre uma fila de mensagens e uma *exchange*, criando vínculos para possibilitar o redirecionamento das mensagens recebidas. *Binding keys* são as chaves presentes nos *bindings* que serão utilizadas para comparação com a chave de roteamento da mensagem, a qual é denominada, segundo Serrano e Nunez (2018), *routing key*.

Ainda de acordo com Conway et al. (2008), *bindings* podem ser criados e destruídos pela aplicação conforme o redirecionamento necessário, e sua vida útil está relacionada com a fila de mensagens a qual está vinculado, logo quando uma fila é destruída, seus *bindings* também serão.

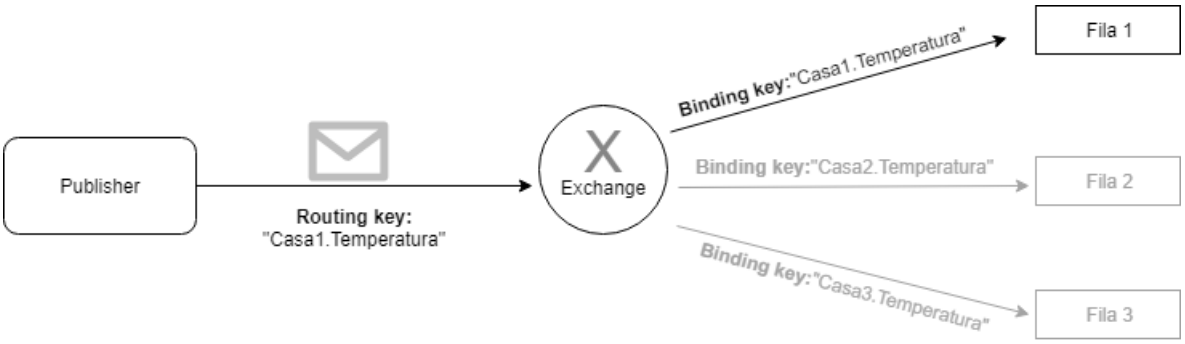
2.5.3.3 *Exchanges*

Um *exchange* é o elemento responsável por rotear as mensagens enviadas pelas aplicações clientes em um *virtual host*, encaminhando-as para uma ou mais filas de mensagens. Pode-se destacar que as aplicações clientes desconhecem completamente a existência das filas. Em suma, o trabalho das *exchanges* está diretamente relacionado aos *bindings* citados na Subseção 2.5.3.2.

Como existem mais de um tipo de *exchange*, os itens listados na sequência esclarecem a diferença entre os tipos existentes, bem como exemplificam seu uso.

(a) *Direct exchange*: método no qual uma mensagem é roteada para uma ou mais filas quando o *binding key* for igual ao *routing key* da mensagem. De acordo com Conway et al. (2008), o servidor AMQP deve implementar o *direct exchange* e deve trazer por padrão 2 *exchanges* dentro de cada *virtual host*. Uma delas deve possuir o nome de *"amqp.direct"* e a outra não deve possuir nenhum nome.
Segundo Richardson (2012), quando uma fila é declarada, ela é automaticamente vinculada a *exchange* sem nome, usando a *routing key* como chave de roteamento. A partir disso, quando uma mensagem for recebida pelo servidor com esta *routing key*, ocorrerá um redirecionamento para a fila correspondente, tornando este processo similar a um encaminhamento direto para uma determinada fila.

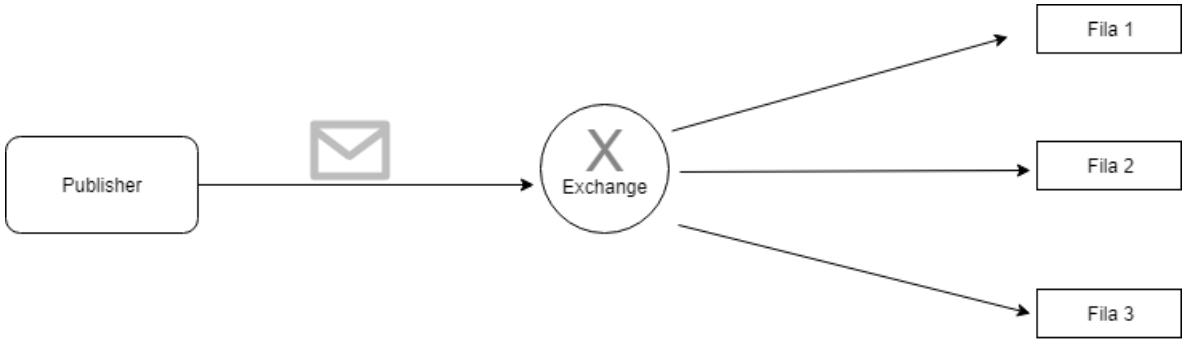
Figura 10 – *Direct exchange* no protocolo AMQP



Fonte: Elaborado pelo autor (2018)

(b) *Fanout exchange*: o método de *exchange fanout* é caracterizado por encaminhar a mensagem recebida para todas as filas que estão vinculadas a uma determinada *exchange*, não levando em consideração a *routing key*. Caso mais de uma fila esteja vinculada a *exchange*, a mensagem é copiada para todas estas filas.

Figura 11 – *Fanout exchange* no protocolo AMQP

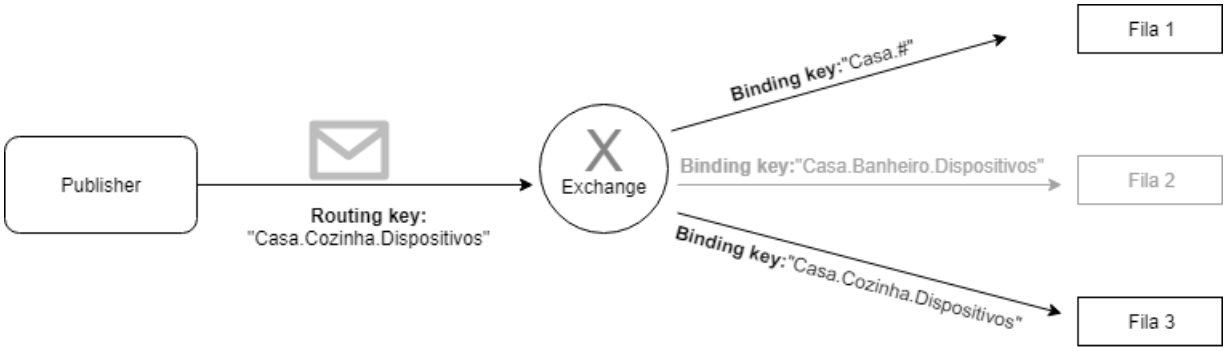


Fonte: Elaborado pelo autor (2018)

(c) *Topic exchange*: no método de *topic exchange* as mensagens são encaminhadas para as filas com base na comparação entre o *routing key* da mensagem e o *binding key* que gera o relacionamento entre *exchange* X fila. Ainda, conforme descreve Conway et al. (2008), a *binding key* usada pelo *exchange* deve possuir nenhum ou mais conjunto de caracteres compostos por caracteres [a-Z, 0-9], delimitadas por pontos (“.”).

Este tipo de *exchange* possibilita o uso de caracteres especiais, como no protocolo MQTT, citado na Subseção 2.5.2. Serrano e Nunez (2018) enumera que os caracteres disponíveis são “*”, responsável por poder substituir uma palavra no *binding-key* e “#”, característico por poder substituir uma ou mais palavras. Por exemplo, o *binding key “Casa*”*. aceitaria a *routing key “Casa.Cozinha”*, mas não aceitaria *“Casa”* nem *“Casa.Cozinha.Dispositivos”*. Já o *binding key “Casa.#”* aceitaria *“Casa.Cozinha.Dispositivos”*.

Figura 12 – *Topic exchange* no protocolo AMQP

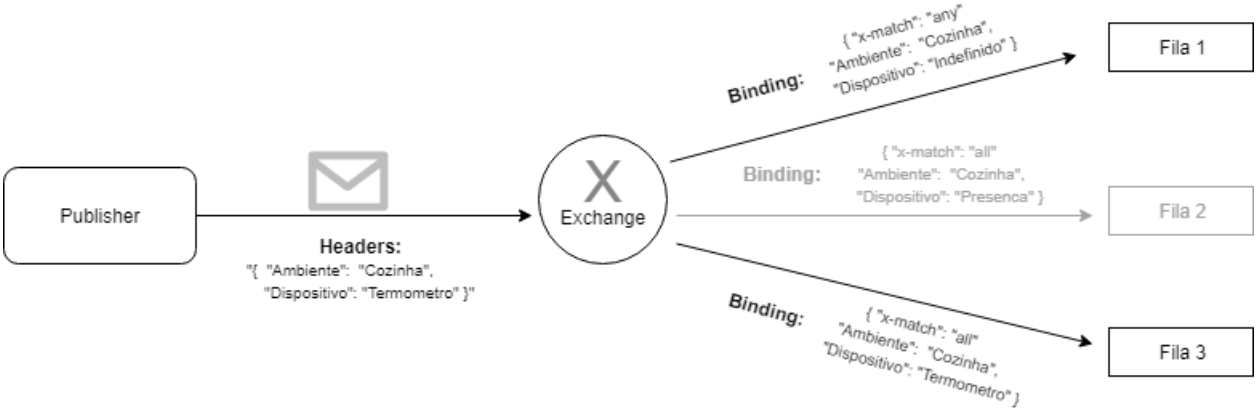


Fonte: Elaborado pelo autor (2018)

(d) *Headers exchange*: neste método, segundo Roy (2017), uma fila é ligada ao *exchange* através de uma tabela de argumentos com pares *chave/valor* contendo os cabeçalhos que devem ser aceitos por este *binding*. Quando uma mensagem for publicada, junto

a ela é também publicado um atributo chamado *headers*, na qual está contida a tabela que será usada para comparação com tabela de binding da *exchange*.
Uma vez que seja possível vincular uma fila a uma *exchange* através de mais de um cabeçalho, torna-se necessário informar mais um argumento durante a publicação, chamado de *"x-match"*, uma vez que é possível encaminhar uma mensagem a uma fila se ao menos um cabeçalho corresponder ou se todos os cabeçalhos forem correspondidos.

Figura 13 – *Headers exchange* no protocolo AMQP



Fonte: Elaborado pelo autor (2018)

Referente a persistência das filas, RabbitMQ (2018a) explica que existem 3 tipos de persistência que podem ser atribuidos a uma fila: *Durable*, onde a fila irá ser mantida após a reinicialização do broker; *Exclusive*, onde a fila é excluída após uma conexão ser fechada, sendo de uso exclusivo desta conexão; *Auto-delete*, onde a fila é excluída após o último consumidor interromper a *subscription*.

Na persistência das *exchanges*, pode-se optar entre "*transient*", que deixam de existir após o reinicio do *broker* ou "*durable*", que somente deixam de existir quando são excluídas manualmente (RABBITMQ, 2018a). *Exchanges* ainda possuem um parâmetro "*auto-delete*", que realiza a auto exclusão da *exchange* após perder o vínculo com a última fila.

2.5.4 ***Constrained Application Protocol* (CoAP)**

O protocolo CoAP, do inglês *Constrained Application Protocol*, é um protocolo para transferência de documentos, assim como o HTTP, citado na Subseção 2.5.1 Porém, como seu nome sugere, ele é projetado para necessidades de dispositivos limitados.

De acordo com Jaffey (2014), assim como no protocolo HTTP, sua estrutura de comunicação baseia-se em *requisição/resposta*, porém seus pacotes possuem tamanho

muito menor, uma vez que segundo Waher (2015), CoAP possui um cabeçalho binário compacto.

O protocolo CoAP possui o mesmo conceito de comunicação por verbos, como os citados na Subseção 2.5.1, porém Waher (2015) enumera que os verbos implementados são “GET, POST, PUT, DELETE”. Outra característica do protocolo é sua capacidade de operar junto ao protocolo HTTP e REST APIs.

Waher (2015) cita que o protocolo também suporta *multicast*, permitindo detecção de dispositivos e comunicação por meio de *firewalls*.

Ao contrário dos protocolos citados anteriormente, este protocolo trafega por UDP (*User Datagram Protocol*), e Jaffey (2014) ressalta que reenvios e reordenamentos são implementados na camada de aplicação. Referente ao QoS, o protocolo CoAP possui um controle de entrega baseado em dois parâmetros, *“confirmable”* e *“nonconfirmable”*. Mensagens que forem enviadas com o parâmetro *“confirmable”*, ao chegarem no receptor, devem ser respondidas através de um pacote *“ack”*. Outras informações sobre este protocolo podem ser encontradas no Capítulo 4 do livro (WAHER, 2015) e em (JAFFEY, 2014).

2.6 INTERMEDIADORES DE MENSAGENS

Nesta seção serão apresentados alguns dos principais intermediadores de mensagens disponíveis no mercado, responsáveis por implementar parte dos protocolos vistos na Seção 2.5 deste trabalho.

2.6.1 **Pivotal RabbitMQ**

O intermediador RabbitMQ é um software *open source* mantido pela *Pivotal Software*. Seu desenvolvimento é feito na linguagem Erlang, criada para ser distribuída, tolerante a falhas e atender aplicações que requerem disponibilidade acima de 99,9% (ROY, 2017).

O intermediador RabbitMQ suporta, por padrão, o protocolo AMQP versão 0.9.1, como citado na Subseção 2.5.3. Versões anteriores, como a 0.9 e 0.8 também são suportadas, porém a versão 1.0 do AMQP não é suportada nativamente, somente através de um *plugin*, uma vez que houve grandes alterações no protocolo. Vale destacar que não há previsão para implementação da mesma (RABBITMQ, 2018c).

Assim como existe o *plugin* para outra versão de AMQP, demais são desenvolvidos para tornar o RabbitMQ compatível com vários protocolos, como o HTTP citado na Subseção 2.5.1 e MQTT, na Subseção 2.5.2.

O intermediador possui bibliotecas clientes para as linguagens de programação mais utilizadas atualmente, como: Python, JavaScript, .NET/C#, C, C++, Java, dentre outras. Uma lista das linguagens suportadas, bem como as bibliotecas disponíveis para cada uma delas pode ser encontrado em (RABBITMQ, 2018b). Ressalta-se que estas bibliotecas, em

suma, realizam a comunicação com o intermediador através do protocolo AMQP. Uma vez que o RabbitMQ suporta mais protocolos através de *plugins*, a comunicação com estes deve ser feita através de bibliotecas clientes específicas.

2.6.2 **Apache Qpid**

Desenvolvido pela *Apache Software Foundation*, o Qpid é um projeto que tem por objetivo fomentar o crescimento de tecnologias que envolvam AMQP. Para tanto, Qpid entrega ao usuário APIs e intermediadores de mensagens, de maneira a possibilitar o uso do protocolo AMQP nas soluções (APACHE, 2018b).

Segundo Apache (2018a), Qpid fornece três APIs de mensagens, que são: Qpid Proton: biblioteca leve e de alto desempenho; Qpid JMS: cliente completo JMS (*Java Message Service*) construída com Qpid Proton; Qpid Messaging API: API orientada a conexão com suporte a inúmeras linguagens.

Um ponto a ser ressaltado é que todos os servidores de mensagens Qpid utilizam o protocolo AMQP, sendo responsáveis pelo armazenamento, roteamento e encaminhamento das mensagens. Alguns exemplos são: Broker-J, escrito em Java; C++ Broker, escrito na linguagem C++; Dispatch Router, escrito na linguagem C e construído no Qpid Proton.

Como citado por Apache (2018a), as APIs e servidores de mensagens trafegam sob a versão 1.0 do protocolo, sendo que somente alguns tem suporte a versões anteriores.

2.6.3 **Solace PubSub+**

Solace PubSub+ é um *broker* com suporte ao protocolo AMQP 1.0. Possui compatibilidade com outros protocolos e APIs, como: MQTT, REST, JMS, dentre outros (SOLACE, 2018a).

De acordo com Solace (2018b), possui uma versão gratuita limitada ao recebimento de, no máximo, 10000 mensagens por segundo e uma versão paga sem limitações e com maior escalabilidade de recursos. Possui também hardwares específicos (*appliances*) para utilização dos *brokers*, os quais são comercializados em modelos, diferenciados por seus poderes computacionais.

Mais informações podem ser encontradas em (SOLACE, 2018a) e (SOLACE, 2018b).

2.6.4 **Eclipse Mosquitto**

Eclipse Mosquitto é um *broker open source* leve que possui suporte ao protocolo MQTT versões 3.1 e 3.1.1, podendo ser utilizado tanto em servidores quanto em computadores de baixo poder computacional (ECLIPSE, 2018a).

Segundo Eclipse (2018b), a implementação atual do *broker* consome aproximada-

mente 3MB de memória RAM com 1000 clientes conectados. A segurança na troca de mensagens pode ser feita através de certificados ou por credenciais de usuário e senha.

Mais informações podem ser encontradas em (ECLIPSE, 2018a) e (ECLIPSE, 2018b).

2.7 TECNOLOGIAS E FERRAMENTAS DE DESENVOLVIMENTO

No desenvolvimento de projetos web e aplicativos móveis algumas plataformas de desenvolvimento podem ser utilizadas, de modo a estruturar e desenvolver softwares fazendo uso de boas práticas, eliminando repetições de código, criando padrões e fazendo o uso de componentes e bibliotecas oferecidas por estas plataformas.

Dentre as disponíveis no mercado, pode-se destacar a plataforma de desenvolvimento Angular, citada na Subseção 2.7.1 e o *framework* Ionic, citado na Subseção 2.7.3.

2.7.1 **Angular**

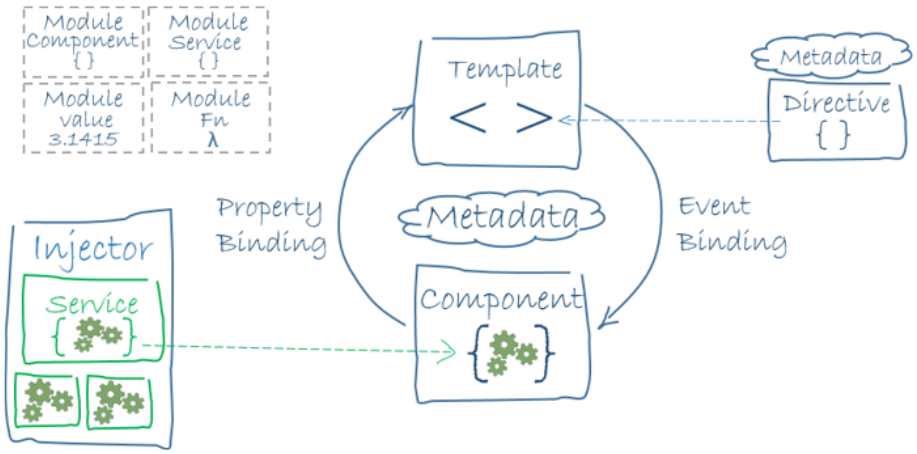
Angular pode ser descrito, segundo Wilken (2018), como uma plataforma de desenvolvimento web que tem por objetivo fornecer um conjunto de ferramentas para a construção de softwares web de grande porte e robustez.

É *open source* e subsidiado pela Google, sendo que um de seus principais objetivos é gerir a comunicação entre o código HTML responsável pela parte gráfica da aplicação e o código JavaScript/TypeScript, responsável pela parte lógica.

No decorrer do tempo, Angular sofreu uma grande alteração de versão. De fato, a versão denominada “Angular JS”, criada em 2009 foi completamente reescrita e lançada em 2014 como nome de “Angular”. A partir desta, o versionamento é dado numericamente, iniciando por “Angular 2” nesta primeira nova versão.

Wilken (2018) menciona também o equívoco em classificar Angular como um *framework*, uma vez que ele possui mais recursos do que um e, sendo assim, pode ter uma classificação mais ampla, como “plataforma de desenvolvimento”. A arquitetura de um projeto Angular pode ser vista na Figura 14.

Figura 14 – *Diagrama da arquitetura Angular*



Fonte: ANGULAR (2018)

Os elementos básicos da arquitetura, de acordo com Angular (2018) e Wilken (2018) são:

a) Módulos: uma aplicação é composta de vários módulos, os quais são utilizados para agrupar componentes em um determinado domínio da aplicação, possibilitando reutilização e maior manutenibilidade do código.
b) Componentes: são elementos que possuem suas próprias lógicas e estão associados a elementos HTML responsáveis pela parte gráfica dos mesmos. Um componente utiliza *bindings/event binding* para realizar a entrada e saída de dados e podem ser projetados para serem reutilizados em várias partes do sistema.
c) Diretivas: podem ser consideradas marcações instrutivas para o Angular. Uma vez resolvidas, fornecem instruções para realizar transformações no HTML e DOM[2] de acordo com o conteúdo das mesmas. Diretivas podem ser usadas para definir comportamentos específicos e alterar o comportamento de elementos existentes.
d) Serviços: podem ser utilizados para desacoplar uma determinada lógica utilizada em mais de um componente do software, tornando-a reutilizável e eliminando repetições de código.

2.7.2 **Apache Cordova**

De acordo com Dunka, Emmanuel e Oyerinde (2017), o desenvolvimento de aplicações nativas, as quais tem por objetivo desenvolver *softwares* para um determinado sistema

[2] DOM: Do inglês *"Document Object Model"*, caracteriza-se pela interface de representação estruturada dos documentos HTML e XML lidos pelo navegador.

operacional e *hardware*, ainda predominam o mercado, possuindo desempenho confiável e *design* diferenciado. Porém, como mencionado por Dunka, Emmanuel e Oyerinde (2017), tal abordagem possui pontos negativos, como a incompatibilidade com outras plataformas, devido a suas especificidades de desenvolvimento.

Com o constante crescimento do mercado de *smartphones*, inúmeros dispositivos surgem e saem de fabricação com muita frequência. O desenvolvimento híbrido tem como objetivo possibilitar a codificação de softwares multiplataforma, tornando possível a criação de um único aplicativo para todas. Outro ponto positivo do desenvolvimento híbrido é a curva de aprendizado, uma vez que o desenvolvedor necessita somente de conhecimento web e do *framework*, dispensando conhecimentos pontuais sobre cada plataforma do mercado.

Cordova é um *framework* para desenvolvimento de aplicativos *mobile open source*. De acordo com Cheng (2017), Cordova foi originado de um outro *framework* chamado *PhoneGap* (ADOBE, 2018), o qual foi concedido à Apache para ser utilizado como base na criação do mesmo.

Fazendo uso de tecnologias *web* como HTML, CSS e JavaScript, torna-se possível a criação de aplicativos independentes da plataforma no qual o mesmo será executado.

Como citado por Cheng (2017), aplicativos híbridos utilizam uma tecnologia denominada *webview* fornecida pela plataforma nativa, responsável por carregar e executar a aplicação no dispositivo.

Uma vez que a principal responsabilidade do Cordova é a parte estrutural do software, torna-se importante o uso de outras ferramentas para auxiliar na construção das interfaces gráficas, como o Ionic, descrito na Subseção 2.7.3 deste trabalho.

Mais informações sobre o *framework* Cordova podem ser encontradas em (CORDOVA, 2018) e no livro (CHENG, 2017).

2.7.3 **Ionic**

Ionic é um *framework open source* para desenvolvimento de aplicativos móveis híbridos, o qual pode ser utilizado com AngularJS ou Angular, possuindo um legado de milhões de aplicativos desenvolvidos através de sua tecnologia (WILKEN, 2018). Para Cheng (2017), Ionic possui diversos componentes com design elegante e multiplataforma.

Como destacado em (IONIC, 2018), Ionic é um *framework* HTML5, logo precisa de um intermediador para ser utilizado como aplicativo nativo e ter acesso aos recursos nativos do dispositivo, sendo recomendável o Cordova (citado na *Subseção* 2.7.2), mesmo utilizado pelas ferramentas já implementadas no *framework*.

Maiores informações sobre este *framework* podem ser encontradas no livro (WILKEN, 2018), no livro (CHENG, 2017) e no site oficial (IONIC, 2018).

2.8 TRABALHOS RELACIONADOS

Como citado no Capítulo 1, a *Internet* das Coisas está se difundindo e tornando-se cada vez mais presente no mercado. Com isso vários outros trabalhos foram realizados neste segmento, sendo alguns destes listados a seguir.

Pellenz (2017) e Pilotti (2014) desenvolveram projetos para automação residencial utilizando a plataforma de desenvolvimento Arduino, realizando comunicação através de meios e protocolos diferentes. Biegelmeyer (2015) utilizou um *hardware* da *Texas Instruments* com uma comunicação também diferenciada.

Pellenz (2017) comunica os sensores ao dispositivo controlador Arduino através de fios, utiliza o protocolo MODBUS para comunicação junto ao software supervisório Elipse. Já Pilotti (2014) criou um sistema de automação voltado a portadores de necessidades especiais, utilizando comunicação *bluetooth* entre o *smartphone* e a central de automação Arduino. Biegelmeyer (2015) implementou o sistema através de uma tecnologia chamada sub-1GHz.

Além dos trabalhos envolvendo IoT como um todo, existem também trabalhos voltados a objetivos específicos, como o trabalho de Santos (2012), fazendo uso de um sensor de gás para emitir notificações sobre possíveis vazamentos de gás que foram detectados. Já o trabalho de Pereira (2016) propõe um robô capaz de responder a um alarme de incêndio, navegar pelo ambiente e fazer uso de um extintor de incêndio para combater as chamas detectadas.

Neste capítulo foram apresentados vários conceitos e tecnologias, como modelos de comunicação IoT, *hardwares*, protocolos, *frameworks*, dentre outros recursos necessários para o desenvolvimento de soluções IoT. Tais tecnologias se fazem necessárias para a criação de uma arquitetura escalável e multiplataforma.

No próximo capítulo é apresentada uma arquitetura que tem por objetivo resolver o problema proposto no Capítulo 1.

3 **ARQUITETURA PROPOSTA**

Durante o desenvolvimento deste trabalho foram analisados outros trabalhos que realizavam comunicação com dispositivos, e alguns são citados na Seção 2.8. O que se pode observar nestes trabalhos é a falta de escalabilidade, uma vez que limitações de *hardware* e de poder computacional irão impedir a adição de uma grande quantidade de sensores nas soluções, e em alguns casos a adição de sensores pode demandar alterações inviáveis na estrutura do projeto.

Com a popularização dos serviços em nuvem, como a Microsoft Azure, a AWS (*Amazon Web Services*) e *Google Cloud*, pode-se perceber uma tendência ao uso de computação em nuvem nas soluções IoT, de modo a obter maior disponibilidade dos recursos e possibilitar o escalonamento do poder computacional de acordo com a demanda do projeto. Porém, de modo geral, pode-se perceber uma despreocupação na implementação de soluções de contingência, de modo a manter os serviços essenciais em funcionamento mesmo após a perda de comunicação com a *internet* e consequentemente com o servidor.

Este trabalho propõe uma arquitetura genérica e contingente para atender a projetos IoT, permitindo a tomada de decisões críticas mesmo sem a comunicação com o servidor. A arquitetura deve ter a capacidade de prover comunicação aos diversos dispositivos presentes em um ambiente, possibilitando o armazenamento das mensagens necessárias. Além da comunicação, tornou-se necessário um aplicativo para a interação do usuário com a solução e um painel de controle para a administração da estrutura que permite a realização de parametrizações convenientes.

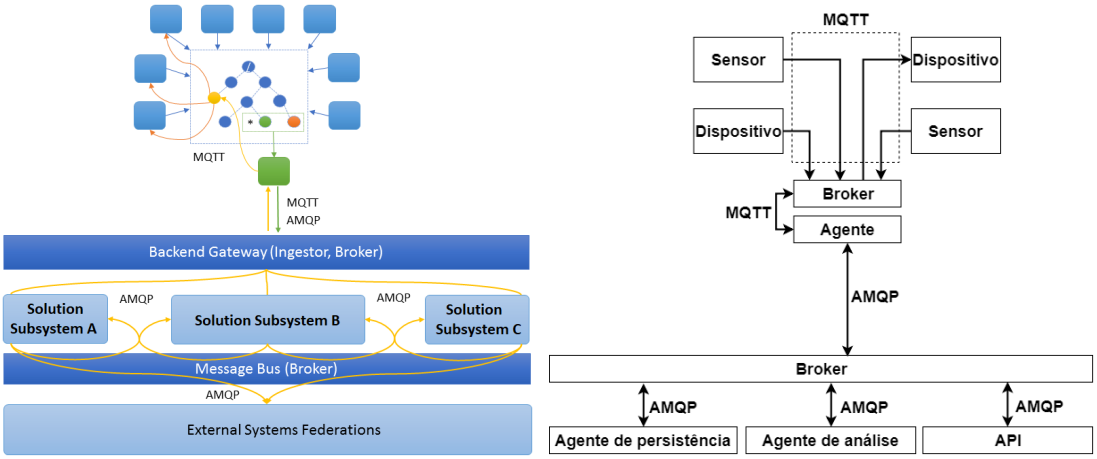
Nas seções que se seguem é apresentada uma estrutura modular junto aos elementos que a compõem. São também apresentados os itens definidos como padrão e que devem ser implementados em projetos que farão uso da estrutura. Na Seção 3.2 é definido o esquema de segurança que deve ser seguido durante a comunicação com os elementos da arquitetura.

3.1 ESTRUTURA DA SOLUÇÃO

A estrutura da solução segue o modelo de comunicação *device-to-gateway*, descrito na Subseção 2.1.3.3. Devido ao trabalho implementar um sistema de contingência local e poder suportar mensagens oriundas de diversos dispositivos, torna-se necessário a criação de uma estrutura que faz uso de mais de um protocolo de mensagem.

A combinação dos protocolos MQTT (Subseção 2.5.2) e AMQP (Subseção 2.5.3) utilizados na solução é citada por Sami (2017) e Vasters (2017), sendo considerada uma solução viável por ambos. Vasters exemplifica esta arquitetura na Figura 15a. A relação entre este trabalho e estrutura citada pelo autor é encontrada na Subseção 3.1.1 e pode ser observada na Figura 15a e Figura 15b.

Figura 15 – Estrutura híbrida de protocolos MQTT e AMQP



(a) Estrutura proposta pelo autor

Fonte: Adaptado de VASTERS (2017)

(b) Arquitetura proposta por este trabalho

Fonte: Elaborado pelo autor (2018)

A arquitetura proposta (Figura 15a) consiste de 3 ambientes distintos onde cada ambiente pode possuir dezenas de dispositivos trocando mensagens através do protocolo MQTT, os quais estão interligados a um *gateway* que possui comunicação com o servidor. Vasters (2017) menciona que a troca de mensagens entre o *gateway* e servidor pode ser feita através do protocolo MQTT ou AMQP, porém o protocolo AMQP implementa mecanismos de controle de erros com capacidade de notificações adequadas em publicações, dentre outras características, o que o torna de sua preferência. No servidor, a troca de mensagens é realizada através do protocolo AMQP.

Tal modelo arquitetural também é citado por Sami (2017), onde o mesmo menciona que em algumas aplicações pode haver necessidade de filas de mensagens confiáveis bem como dispositivos leves para trabalhar, sendo assim necessário o uso dos dois protocolos em conjunto.

Esta relação entre os protocolos pode também ser encontrada na arquitetura adotada por este trabalho, sendo descrita neste capítulo.

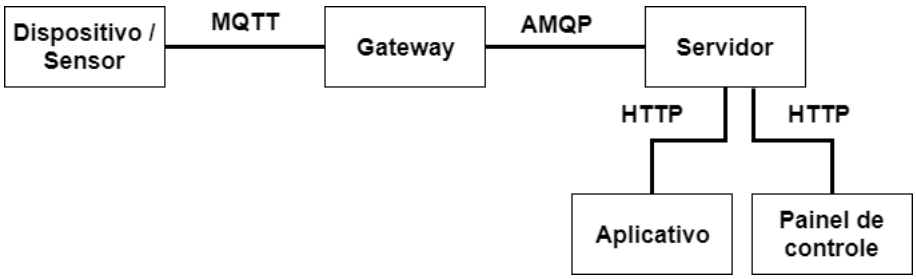
3.1.1 **Visão geral da arquitetura**

Em comparação à estrutura proposta por VASTERS na Figura 15a, a Figura 15b mostra a arquitetura implementada por este trabalho. No servidor da solução os agentes também possuem comunicação AMQP com o *broker*. O aplicativo para dispositivos móveis comunica-se com o servidor através de uma API HTTP que fornece os dados a serem exibidos no mesmo, bem como recebe os comandos a serem enviados para os dispositivos.

A comunicação do *gateway* com o servidor não é local e com isso a qualidade da mesma tende a ser inferior, podendo haver perda de pacotes e/ou menor velocidade de conexão. Com isso, o protocolo AMQP foi escolhido para realizar a conexão entre *gateway* e servidor, uma vez que possui tratamento mais apropriado na notificação de erros.

Por fim, os sensores e dispositivos locais interagem com o *gateway* trocando mensagens através do protocolo MQTT, uma vez que a comunicação é através de uma rede local e incide em uma conexão com maior qualidade e estabilidade, fazendo assim o uso de um protocolo voltado ao desempenho e baixa latência. A Figura 16 representa a topologia da arquitetura proposta.

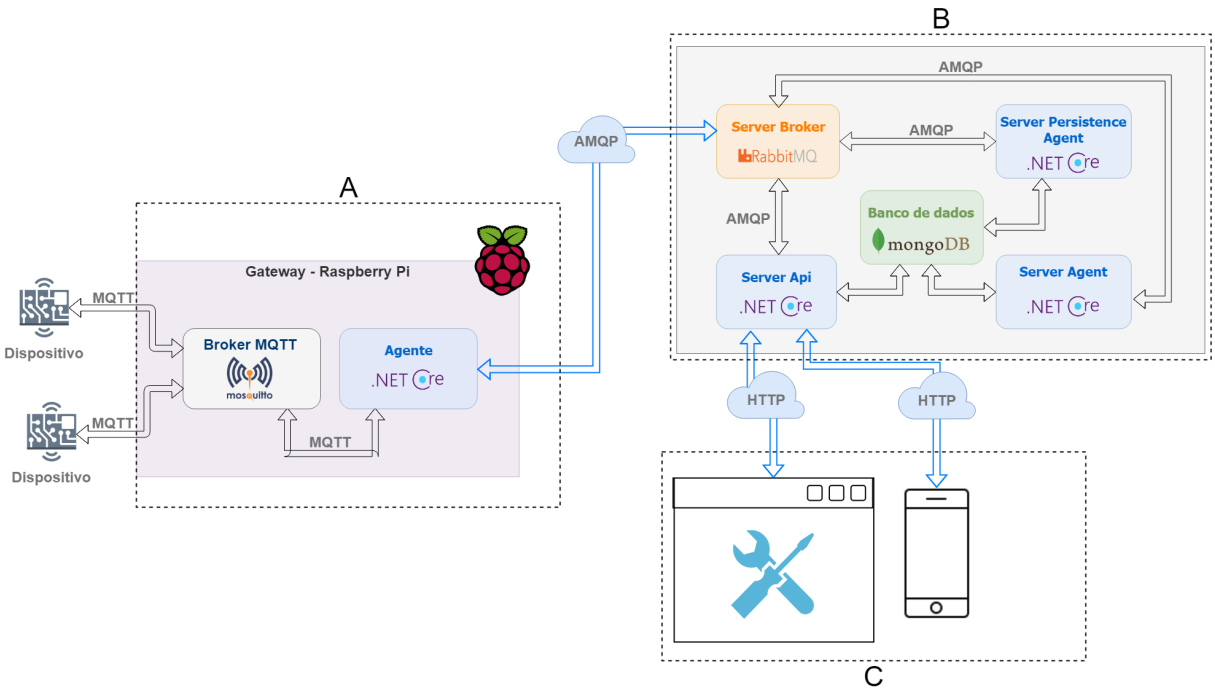
Figura 16 – Topologia da arquitetura proposta



Fonte: Elaborado pelo autor (2018)

A Figura 17 ilustra a arquitetura e os subsistemas envolvidos, onde a parte “A” representa o *gateway* da solução, a parte “B” representa o servidor da solução e a parte “C” ilustra os *softwares* clientes da solução.

Figura 17 – Diagrama com representação completa da arquitetura



Fonte: Elaborado pelo autor (2018)

O meio de comunicação dos dispositivos locais com o *gateway* é por uma rede *WiFi* e o *gateway* pode conectar-se a *internet* através de uma rede *WiFi* ou *Ethernet*. Os detalhes de cada um dos elementos que compõe a arquitetura serão apresentados neste capítulo.

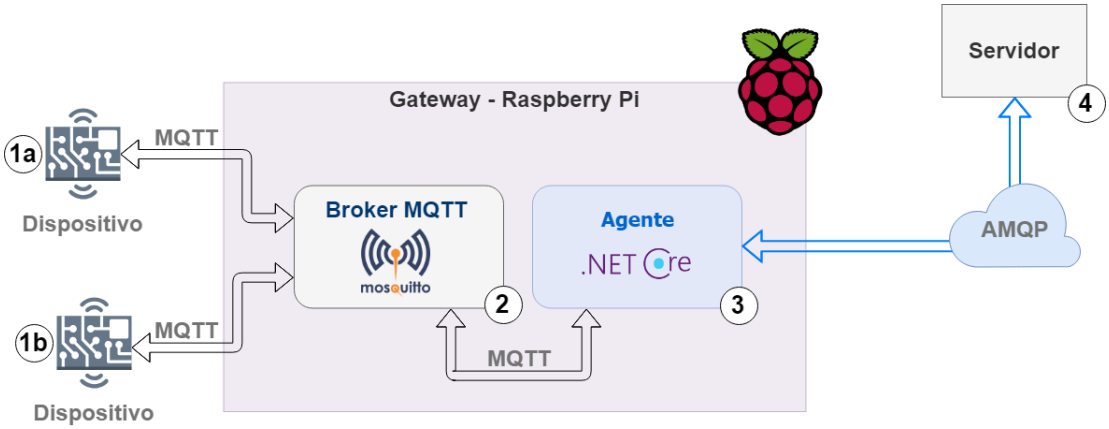
3.1.2 *Gateway* da solução

Uma vez que o modelo de comunicação escolhido é o *device-to-gateway*, o *gateway* utilizado pela solução é executado em um dispositivo *Raspberry Pi* 3B, visto que possui um excelente custo benefício e vasta documentação disponível. A mesma é utilizada com um sistema operacional Linux denominado Raspbian, possuindo distribuição livre.

Com base na Figura 18, como mencionado anteriormente, os sensores e dispositivos irão enviar mensagens ao *gateway* através do protocolo MQTT (1), o qual é gerido no *gateway* através do *broker* (2) Eclipse Mosquitto (ECLIPSE, 2018a). Um agente presente no *gateway* (3) consome as mensagens deste *broker*, cria as mensagens de evento e anexa a elas algumas informações de controle para posterior publicação no servidor (4).

Mais informações sobre o *broker* podem ser encontradas na Subseção 3.1.2.1 e sobre o agente na Subseção 3.1.2.2.

Figura 18 – *Broker* da solução executando na *Raspberry*



Fonte: Elaborado pelo autor (2018)

Importante observar que qualquer sensor ou dispositivo que implementa comunicação por MQTT, independente de *hardware* ou linguagem, pode se comunicar com o *broker* e consequentemente com toda a solução, bastando seguir o modelo de mensagem estabelecido na Subseção 3.1.2.3.

3.1.2.1 *Broker* do dispositivo *gateway* (2)

O *broker* que é utilizado no *gateway* é o Eclipse Mosquitto, um broker *open source* mantido pela *Eclipse Foundation* implementando o protocolo MQTT 3.1 e 3.1.1, sendo escolhido por seu baixo consumo de recursos computacionais ao ser comparado a outros *brokers* presentes no mercado que implementam mais protocolos de comunicação, porém irrelevantes neste caso.

Sua função é a recepção de mensagens vindas dos sensores presentes no ambiente (1a e 1b da Figura 18) e do agente presente no *gateway* citado na Subseção 3.1.2.2. Além da recepção, ele permite o consumo das mesmas pelos dispositivos e pelo agente. A publicação e consumo das mensagens neste *broker* é feita exclusivamente através do protocolo MQTT.

3.1.2.2 Agente do *gateway* (3)

O agente do *gateway* é constituído por um *software* desenvolvido na linguagem C#, utilizando o *framework* .NET Core.

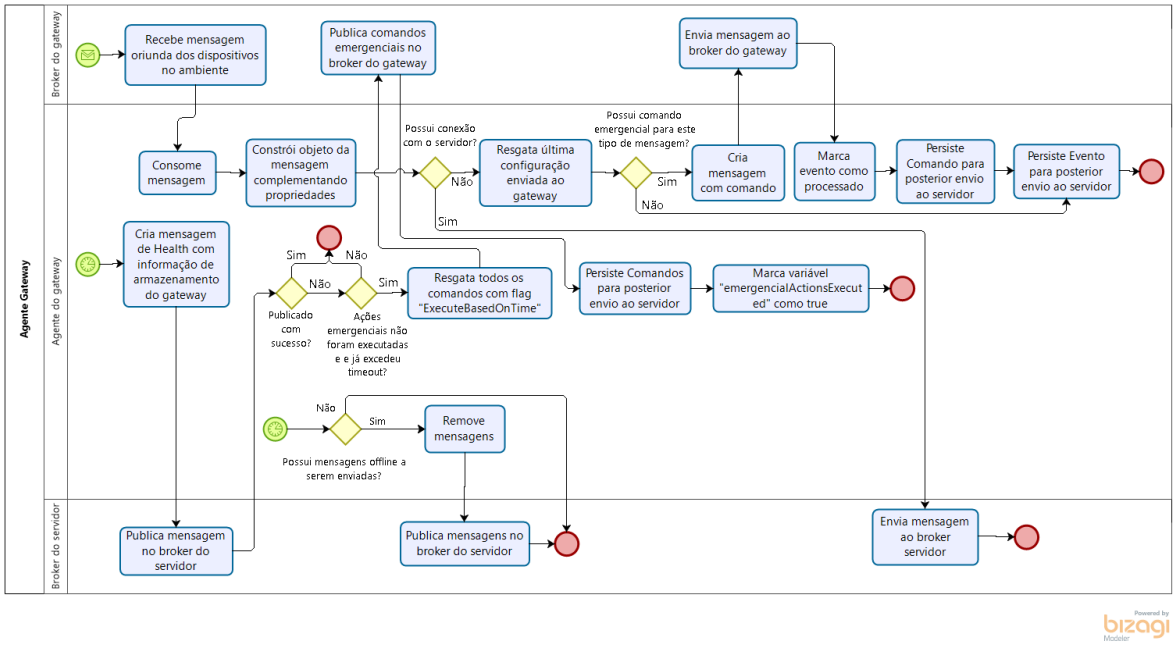
A linguagem C# é desenvolvida pela Microsoft e está padronizada pelo ECMA-334 e ISO/IEC 23270. Possui orientação a objetos e a componentes. O processo de compilação do código C# é realizado em etapas, onde inicialmente *assemblies* com código de linguagem intermediária e outras informações são geradas e possuem extensão “.dll”. Antes da execução, o compilador JIT (*Just-In-Time*) do .NET *Common Language Runtime*

converte o código intermediário para código de *hardware* específico (MICROSOFT, 2018b).

O *framework* .NET Core é um *software* de código aberto mantido pela Microsoft e comunidade. Ao contrário dos *frameworks* anteriores produzidos pela Microsoft, .NET Core é multiplataforma, podendo ser executado em sistemas operacionais Windows, Linux e MacOS, bem como pode ser utilizado em contêineres do Docker[1]. Permite o uso da linguagem C#, F# e Visual Basic para o desenvolvimento de aplicativos e bibliotecas (MICROSOFT, 2018a). Mais informações podem ser encontradas em (METZGAR, 2018).

Os diagramas com os fluxos básicos de funcionamento do agente podem ser visualizados na Figura 19 e Figura 20.

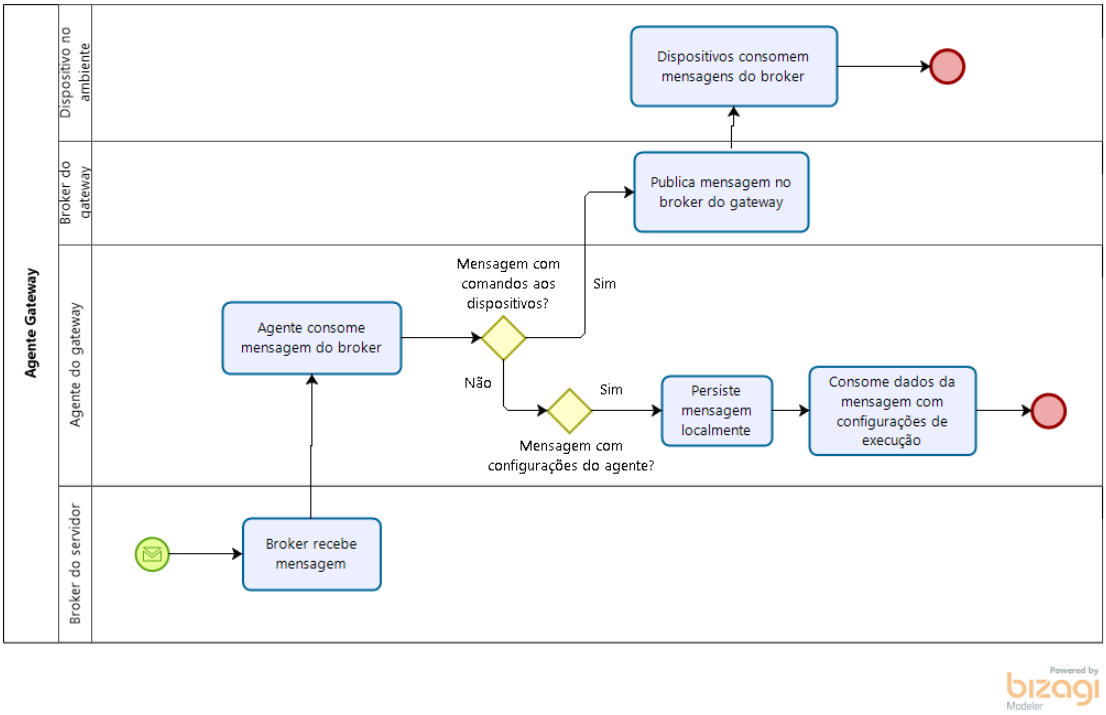
Figura 19 – Diagrama de funcionamento do agente presente no *gateway* - Fluxo de envio



Fonte: Elaborado pelo autor (2018)

[1] Docker: Contêineres que realizam o empacotamento de um *software* e todas as dependências envolvidas, de modo a realizar a execução do mesmo em qualquer outra máquina, independendo de configurações e parametrizações extras.

Figura 20 – Diagrama de funcionamento do agente presente no *gateway* - Fluxo de recebimento



Fonte: Elaborado pelo autor (2018)

O diagrama da Figura 19 representa o fluxo de envio de mensagens presente no agente, tendo o objetivo de realizar o consumo das mesmas no *broker* do dispositivo *gateway* através do protocolo MQTT, criar mensagens de evento e adicionar a elas alguns dados de controle para posterior publicação no *broker* do servidor através do protocolo AMQP, sendo que além do envio ao servidor, o agente pode persistir algumas mensagens localmente para posterior envio em caso de falha na conexão com o servidor.

Em casos de falha na comunicação com o servidor, ao receber uma mensagem, o agente verifica a criticidade da mesma através das relações *eventos X comandos* cadastrados no painel de controle e enviadas ao gateway e com isso pode, emergencialmente, enviar comandos a dispositivos no ambiente através de mensagens publicadas no *broker* do dispositivo *gateway*, como por exemplo solicitar o fechamento de uma válvula preventivamente.

Além disso, caso a comunicação esteja *offline*, o agente pode ser programado para executar comandos preventivos, como por exemplo, interromper o fluxo de uma válvula ou desligar um dispositivo após determinado tempo sem conexão com o servidor.

As mensagens não enviadas ao servidor podem ficar temporariamente armazenadas no *gateway* para posterior envio e serão reenviadas automaticamente após um determinado

tempo decorrer entre as tentativas de reenvio.

No diagrama presente na Figura 20 é descrito o fluxo de recebimento das mensagens, onde o agente consome as mesmas de filas no *broker* do servidor através do protocolo AMQP e, caso a mensagem seja consumida de uma fila de comandos, as publica no tópico de comandos do *broker* do dispositivo *gateway* com protocolo MQTT. Caso a mensagem consumida seja referente a configurações do *gateway*, o agente faz a leitura e aplica as configurações alteradas. Como a estrutura proposta é modular, podem existir mais filas e tipos de mensagens a serem consumidos pelo agente, sendo estas específicas de cada implementação.

3.1.2.3 Modelo de mensagem dos dispositivos

As mensagens enviadas e recebidas pelos dispositivos devem seguir um formato com campos e representação padrão. A mensagem será representada através do formato atributo-valor JSON (ECMA, 2017), considerando sua compacta estrutura e independência de linguagem oferecida.

O conteúdo da mensagem possui campos para identificação de dispositivo, armazenamento de valor lido, campos para controle, bem como campos para definição de tipo e ação da mesma. A mensagem representada em formato JSON tem a estrutura da Figura 21.

Figura 21 – Exemplo de mensagem de dispositivo no formato JSON


```
{
  "GatewayId":"85265f5b-c90e-4f21-8b87-d5a2f1662c4f",
  "DeviceId":"4e391d55-9c8a-42a8-b660-9872fc262a59",
  "MessageTypeCode":109,
  "Action":0,
  "Value":1.0,
  "IsDisposable":false,
  "Date":"0001-01-01T00:00:00",
  "Processed":false
}
```


Fonte: Elaborado pelo autor (2018)

No Quadro 3 são descritos os campos da mensagem, onde as colunas representam, respectivamente, o nome da propriedade e seu uso. Alguns campos utilizam um identificador único UUID [2] como formato.

[2] *UUID (Universally Unique IDentifier)*: Identificador único universal que possui formato de caracteres 8-4-4-4-12.

Quadro 3 – Conteúdo da mensagem da solução proposta

Propriedade	Descrição
GatewayId	Destinado a identificar o *gateway* ao qual o dispositivo está vinculado. Representado por um UUID.
DeviceId	Destinado a identificar o dispositivo quando for um sensor ou dispositivo destino quando for uma ação. É único e representado por um UUID.
MessageTypeCode	Destinado a identificar o tipo da mensagem. São previamente cadastradas no painel de controle da solução.
Action	Representa se a mensagem é um evento ou um comando.
Value	Contém um valor de leitura ou um valor de um comando a ser executado.
IsDisposable	Indica se a mensagem pode ser descartada sem a necessidade de persistência no banco de dados e, consequentemente, não são visíveis pelo usuário.
Date	Data e hora de geração da mensagem. No caso de geração por um dispositivo no ambiente, esta informação é gerada pelo *gateway* no momento do processamento.
Processed	Indica que a mensagem já foi processada pelo *gateway* de forma emergencial e um comando emergencial já foi enviado, não sendo necessário reenvio deste pelo servidor.

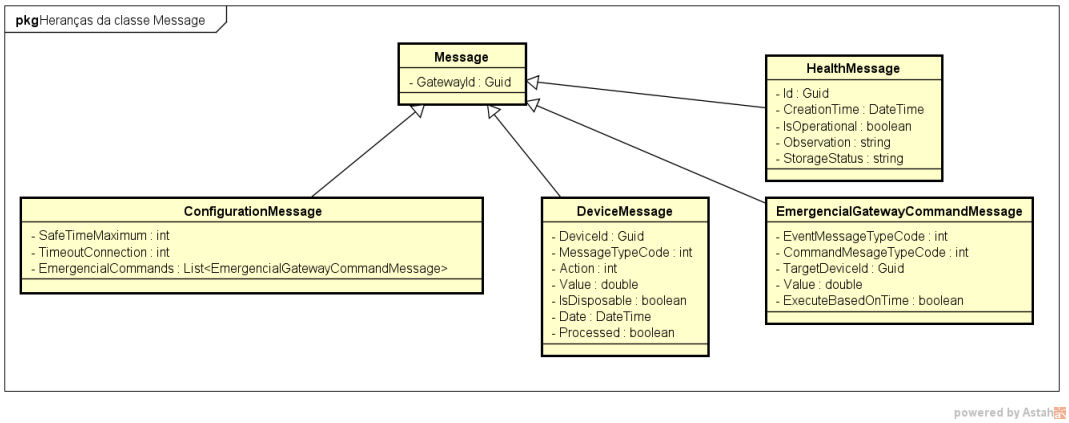
Fonte: Elaborado pelo autor (2018)

Alguns campos da mensagem tem valores fixos e ficam armazenados nos dispositivos para serem inseridos no momento da geração da mesma.

3.1.2.4 Diagrama de tipos de mensagens

Torna-se também necessária a criação de modelos de mensagens para os *softwares* que atuam no servidor e *gateway* da arquitetura. O modelo, representado na Figura 22, utiliza herança para a representação dos diferentes tipos de mensagens.

Figura 22 – Tipos de mensagens



Fonte: Elaborado pelo autor (2019)

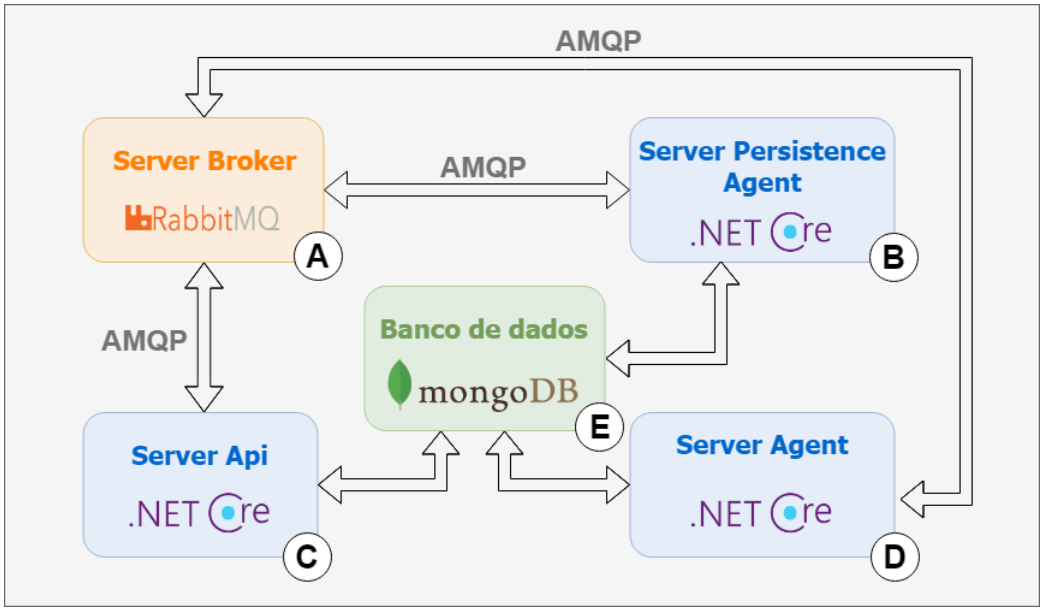
Os tipos de mensagens que herdam de “Message” são:

a) *ConfigurationMessage*: tipo de mensagem utilizada para representar as informações de configuração do *gateway*. Esta mensagem possui as informações de tempo máximo seguro, tempo de *timeout* da conexão e lista com os comandos emergenciais que o *gateway* irá receber.
b) *DeviceMessage*: neste tipo de mensagem são representados os eventos e comandos trafegados pelos dispositivos e sensores que compõem a solução. A mesma mensagem em formato JSON é apresentada na Figura 21.
c) *EmergencialGatewayCommandMessage*: mensagem que representa os comandos emergenciais utilizados pelo *gateway* em casos de indisponibilidade da conexão.
d) *HealthMessage*: mensagem utilizada para controle de disponibilidade dos elementos que compõem a arquitetura.

3.1.3 **Servidor da solução**

A solução também conta com um servidor em ambiente separado do *gateway*, podendo estar hospedado em algum serviço de computação em nuvem ou em local que disponha de estrutura adequada. O servidor utiliza sistema operacional Linux e executa um *broker* com filas de mensagens, um banco de dados, bem como os serviços responsáveis pelo consumo, tratamento e persistência das mensagens, API para comunicação com outros *softwares*, agente de análise e notificação, dentre outros serviços que possam vir a ser necessários.

Figura 23 – Servidor da solução executando remotamente



Fonte: Elaborado pelo autor (2018)

A Figura 23 ilustra a arquitetura do servidor proposto. Nas próximas subseções serão abordados detalhadamente os elementos que o compõem, de acordo com a numeração existente em cada um.

3.1.3.1 Broker

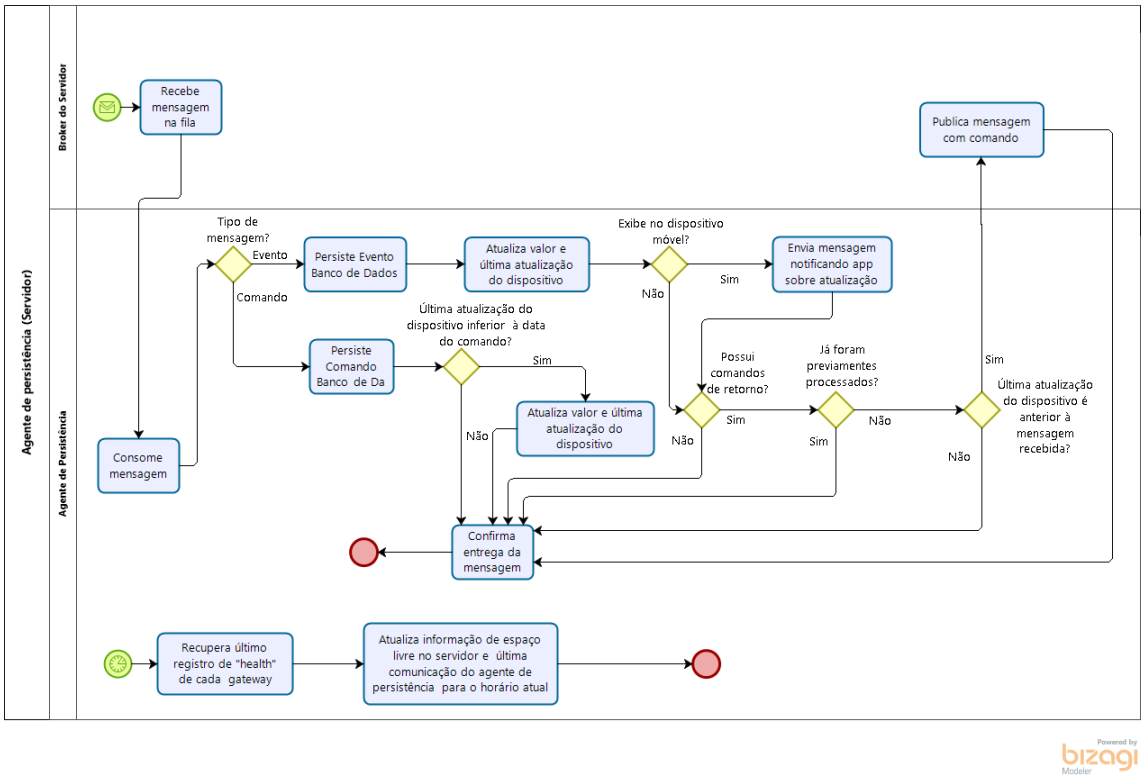
Segundo Girardi (2015) em seu trabalho sobre avaliação de intermediadores de mensagens que suportam AMQP, o intermediador RabbitMQ obteve maior desempenho comparado ao Qpid e ActiveMQ, sendo inferior somente em testes de recebimento de mensagens, no qual o intermediador ActiveMQ se mostrou mais performático. O intermediador Solace PubSub+ não foi avaliado por Girardi, porém como o mesmo utiliza a versão AMQP 1.0, não é de relevância para a escolha do intermediador da solução, uma vez que se optou pela versão AMQP 0.9.1 por conter definições de *broker* e estar a mais tempo no mercado. Com estes argumentos, o intermediador escolhido foi o RabbitMQ.

O *broker* do servidor, Figura 23(A), será responsável pelo recebimento das mensagens vindas do *gateway*, trafegadas através do protocolo AMQP. Além das mensagens oriundas do *gateway*, os agentes presentes no servidor podem realizar publicações AMQP para posterior consumo do *gateway*, bem como pode publicar mensagens de controle da aplicação, informações sobre o status atual de cada dispositivo, ou seja, qualquer mensagem que possa ser utilizada pela solução. Algumas filas de mensagens e *exchanges* devem ser criadas por padrão, as quais são citadas na Subseção 3.1.5 e podem ser construídas no momento da configuração do *broker*.

3.1.3.2 Agente de persistência

O agente de persistência, Figura 23(B), é um *software* com execução no servidor, desenvolvido na linguagem C# com *framework* .NET Core e é responsável pelo consumo das mensagens publicadas no *broker*, bem como realiza uma primeira análise das mensagens, uma vez que alguns tipos precisam de uma resposta imediata, como por exemplo um sensor que detecta vazamento de gás e necessita de uma mensagem solicitando o fechamento da válvula de gás. Tais mensagens com comandos são publicadas no mesmo *broker*, porém em uma outra fila de mensagens. A Figura 24 ilustra o funcionamento deste agente.

Figura 24 – Diagrama de funcionamento do agente de persistência



Fonte: Elaborado pelo autor (2018)

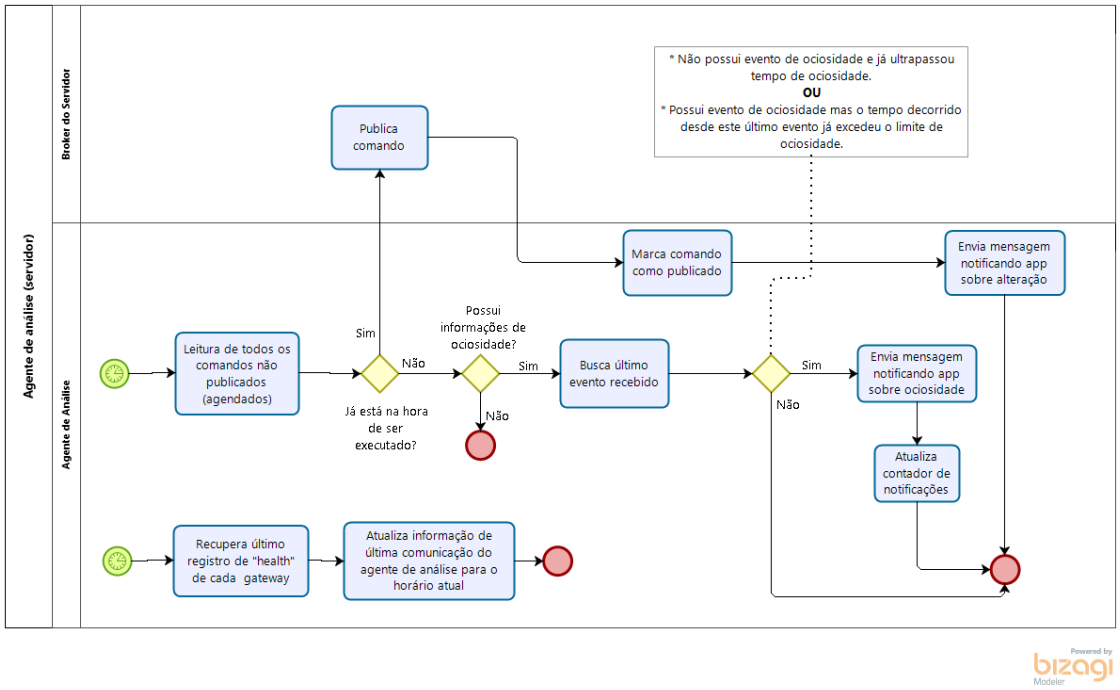
As mensagens consumidas por este agente são persistidas em um banco de dados, citado na Subseção 3.1.3.5. Ressalta-se que o agente pode ser responsável por mais tarefas, dependendo do projeto ao qual faz parte. As mensagens que não requerem ação imediata, quando existirem, podem ser tratadas posteriormente pelo agente de análise.

3.1.3.3 Agente de análise

O agente de análise, Figura 23(D), também é desenvolvido na linguagem C# com *framework* .NET Core e é executado no servidor. Uma de suas tarefas inclui a análise das

mensagens persistidas no banco de dados, a fim de computar, cruzar dados de sensores e dispositivos, persistir resultados, bem como notificar o usuário quando pertinente a respeito dos dados analisados ou alertas gerados, dentre outras tarefas pelas quais pode ser responsável, podendo variar de acordo com o projeto ao qual está inserido. A Figura 25 representa o funcionamento deste agente através de um diagrama.

Figura 25 – Diagrama de funcionamento do agente de análise



Fonte: Elaborado pelo autor (2018)

A execução do agente de análise ocorre em intervalos de tempo, sendo este definido durante a implementação do mesmo.

3.1.3.4 *Application Programming Interface* (API)

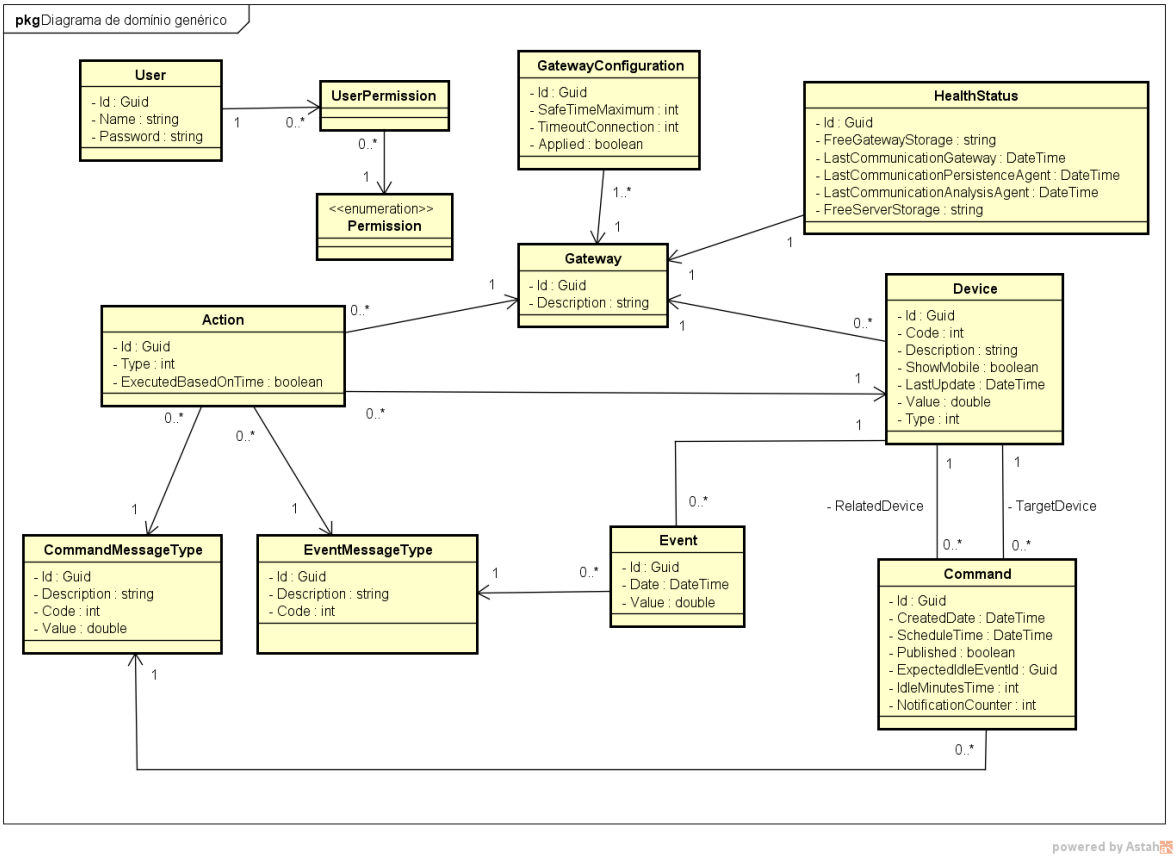
A API da solução, Figura 23(C), tem sua execução no servidor e é desenvolvida na linguagem C# com *framework* .NET Core. Sua principal funcionalidade é fornecer um meio de comunicação com os softwares externos a solução, como o aplicativo móvel e o painel de controle. O diagrama de domínio da aplicação é apresentado na Figura 26 e as entidades envolvidas são:

a) *User, UserPermission e Permission*: representam o usuário vinculado a arquitetura que terá acesso aos *softwares*, vinculado as suas permissões.
b) *Gateway*: entidade que representa o *gateway* da solução.
c) *GatewayConfiguration*: entidade que representa uma configuração de *gateway* que será enviada ao mesmo através do painel de controle.

d) *HealthStatus*: entidade que armazena o *status* geral da arquitetura. Nela estão armazenadas as informações de última comunicação dos agentes, do *gateway* e espaço disponível no servidor e *gateway*.
e) *Device*: representa um dispositivo físico presente no ambiente, suas características e última informação de estado.
f) *Command*: representa um comando enviado a algum dispositivo no ambiente. Nele está contido também o *CommandMessageType* relacionado, possuindo os detalhes do comando. Na entidade *Command* estão contidos também informações sobre horário de agendamento de execução do comando, evento de ociosidade relacionado, dentre outras propriedades de controle.
g) *Event*: entidade que representa um evento de um dispositivo da arquitetura. Nesta entidade está contido um *EventMessageType* com os detalhes do evento.
h) *CommandMessageType*: entidade que representa os detalhes de um determinado comando bem como o valor a ser enviado pelo comando.
i) *EventMessageType*: entidade que representa os detalhes de um evento.
j) *Action*: entidade que representa uma ação da arquitetura. Nela está a relação entre evento e comando relacionado bem como as opções de “Emergencial” e “Executar baseada no tempo” em caso de indisponibilidade.

A API tem acesso ao *broker* do servidor, possibilitando a publicação de mensagens destinadas ao *gateway* e aos dispositivos conectados a ele. As mensagens publicadas podem ser, por exemplo, relacionadas a comandos a serem executados, contendo instruções ou configurações para o *gateway*, de controle da aplicação, dentre outras finalidades. Além do acesso ao *broker*, o *software* também necessita comunicar-se com o banco de dados da solução, citado na Subseção 3.1.3.5, uma vez que pode ser necessário persistir e consultar dados.

Figura 26 – Diagrama de domínio da aplicação



Fonte: Elaborado pelo autor (2019)

Como medida de segurança, durante a comunicação com a API os *softwares* realizam uma autenticação para obter acesso, a qual é realizada através do padrão JWT (*JSON Web Token*), definida através da RFC 7519 (BRADLEY; JONES; SAKIMURA, 2015).

3.1.3.5 Armazenamento de dados

O banco de dados principal da solução, Figura 23(E), é executado localmente no servidor da aplicação . Na escolha do modelo a ser utilizado, foram levados em consideração os modelos relacional e não relacional.

Em um modelo de banco de dados não relacional pode-se encontrar várias diferenças em comparação ao modelo relacional. Sharma e Dave (2012) enumeram que um banco de dados NoSQL não utiliza linguagem SQL, possibilita armazenamento de grande volume de dados e não utiliza os conceitos de ACID, como encontrado em bancos relacionais. Ainda de acordo com Sharma e Dave (2012), bancos de dados NoSQL implementam o conceito BASE: Basicamente disponível, estado leve e eventualmente consistente (*Basically Available, Soft state, Eventual consistency*).

Este paradigma permite escalabilidade horizontal, trazendo benefícios a aplicações com rápido crescimento da base de dados, porém devido a distribuição dos dados, fornece apenas 2 das 3 propriedades do teorema CAP (consistência, disponibilidade e tolerância a falhas).

Este trabalho optou pelo paradigma de banco de dados não relacional, fazendo uso do *software* MongoDB (MONGODB, 2018). Na escolha do banco de dados foram levadas em consideração algumas das principais características dos bancos de dados não relacionais, como a alta taxa de transferência oferecida, o menor custo de *hardware* exigido, suporte a grande quantidade de dados, bem como esquema flexível que permite o armazenamento das mensagens oriundas dos sensores e dispositivos já em formato JSON. O acesso a este banco de dados é realizado através da API e dos dois agentes, os quais possuem execução no servidor.

3.1.4 **Aplicações clientes**

A solução conta com um aplicativo para dispositivos móveis e um painel de controle *web* para gerir a mesma.

3.1.4.1 Aplicativo para dispositivos móveis

O *framework* Ionic foi utilizado neste trabalho uma vez que possui grande difusão, uma comunidade muito unida e excelente documentação, bem como possibilita o desenvolvimento multiplataforma. Tal escolha é também justificada pela plataforma de desenvolvimento escolhida para o desenvolvimento do painel de controle da solução, a plataforma Angular, mesma utilizada pelo *framework Ionic*. Angular já está consolidado no mercado e é mantido pela Google, possuindo suporte para desenvolvimento de grandes soluções com as melhores práticas de programação existentes, comprovado pelos inúmeros projetos de grande porte em seu portfólio.

O aplicativo possui comunicação com o servidor da aplicação fazendo uso do protocolo HTTP, onde é estabelecida a comunicação com a API para acesso aos dados que são exibidos no aplicativo e são enviados os comandos. Alguns dados precisam chegar ao aplicativo de forma assíncrona, fazendo uso assim do *SignalR*, que é uma biblioteca para *real-time web* que possibilita o envio instantâneo de mensagens do servidor para o cliente, oferecendo comunicação em tempo real e assíncrona (MICROSOFT, 2018c).

3.1.4.2 Painel de controle

O painel de controle da solução foi desenvolvido através da plataforma Angular. Importante ressaltar que *Ionic* é construído sobre o *Cordova*, descrito na Subseção 2.7.2, o qual é responsável pela parte estrutural e comunicação com o dispositivo alvo. Através

do painel de controle é possível realizar algumas parametrizações, como configurações relacionadas ao *gateway*, configurações de tipos de mensagem, configurações de ações, parâmetros relacionados à autenticação do sistema, bem como é possível obter informações sobre a disponibilidade da solução. Além destas funções padrão, podem ser implementadas no painel de controle configurações específicas de um determinado projeto, como pode ser visto na Subseção 4.2.2.1.

A comunicação do painel de controle com o servidor é feita através da API. Algumas configurações presentes no painel de controle são relacionadas ao *gateway* e são enviadas ao mesmo através da fila de mensagem de configuração. O *gateway* por sua vez consome a mensagem e aplica as configurações no seu próximo ciclo de execução, que pode demorar até 1 minuto.

3.1.5 **Modelo de referência**

Uma vez que esta estrutura proposta pode ser utilizada em mais de um projeto com aplicações completamente distintas, são inúmeras as possibilidades de sensores e dispositivos diferentes que podem ser utilizados. Nesta arquitetura modular pode-se necessitar da criação de tópicos e filas adicionais tanto no *broker* do dispositivo *gateway* como no *broker* do servidor. Além disso, podem haver outros recursos particulares de cada projeto, como diferentes coleções no banco de dados ou até mesmo pequenas alterações nas implementações dos agentes e da API da solução, desde que não afetem a comunicação e estrutura proposta pelo trabalho.

Muito embora existam estas particularidades de cada projeto, a solução proposta conta com uma arquitetura de tópicos e filas de mensagens padrão, devendo ser seguida para manter a comunicação e compatibilidade dos agentes presentes no *gateway* e servidor, bem como o funcionamento de toda a solução. Dependendo do projeto a ser implementado, pode ser que algumas das filas ou elementos padrão não sejam utilizados, porém já passaram por um planejamento e ficam disponíveis para quando necessário.

3.1.5.1 Tópicos de mensagens padrão

Fazendo uso do protocolo MQTT para a troca de mensagens com o *broker* presente no *gateway* da arquitetura, torna-se necessária a definição de tópicos, os quais são utilizados na publicação e consumo de eventos e comandos. O Quadro 4 contém, respectivamente, o número do elemento relacionado à Figura 27, o nome do tópico, local onde é utilizado e descrição de responsabilidade. A Figura 27 representa um diagrama com as iterações entre os tópicos e os demais elementos da solução.

Quadro 4 – Tópicos padrão contidos no *broker* do dispositivo *gateway*

Nro.	Nome do tópico	Local	Descrição
1	*event/collected*	*Gateway*	Responsável pelas mensagens oriundas dos sensores e dispositivos locais.
2	*cmd/deviceId*	*Gateway*	Responsável pelos comandos a serem consumidos pelos dispositivos locais.

Fonte: Elaborado pelo autor (2018)

3.1.5.2 Filas de mensagens padrão

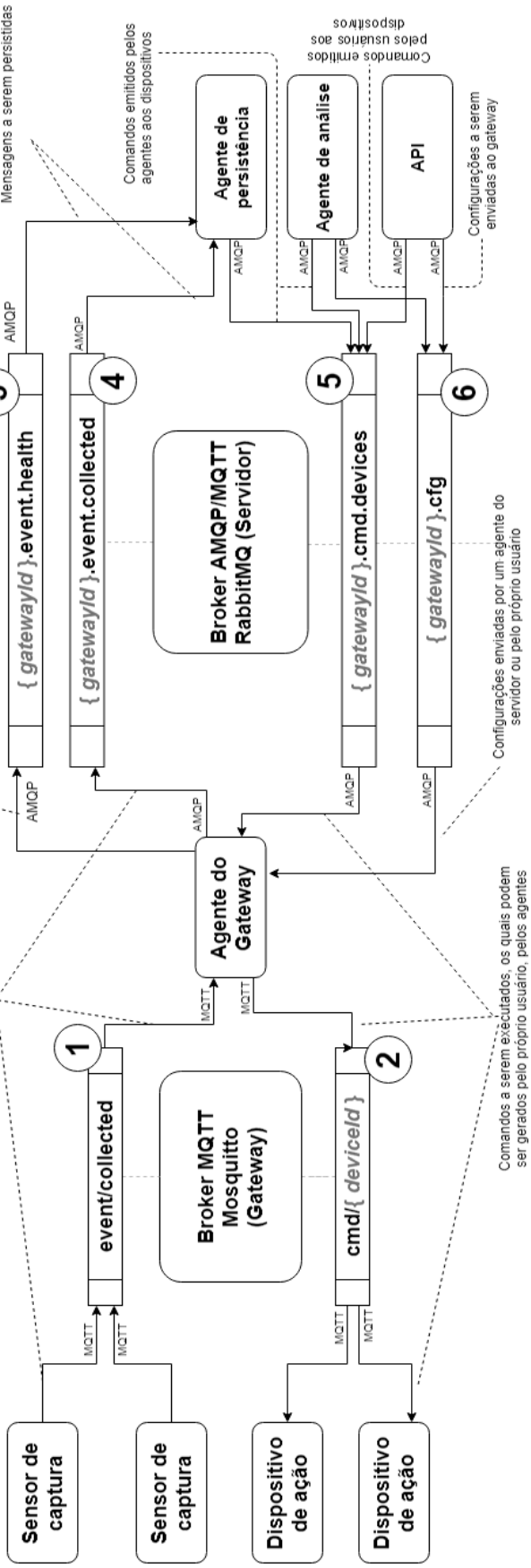
Durante a construção da arquitetura da solução, as filas de mensagens presentes no Quadro 5 devem ser levadas em consideração. A Figura 27 ilustra as filas de mensagens padrão e sua relação com os demais elementos da solução. Importante ressaltar que os agentes não se comunicam diretamente com a fila, sendo neste caso utilizado a *exchange* padrão, criado pelo próprio *broker*, conforme citado no item 1 da Subseção 2.5.3.3. Neste caso, durante o desenvolvimento das aplicações, os comandos de publicação devem conter o parâmetro "*exchange*" vazio.

Quadro 5 – Filas padrão contidas no *broker* do servidor utilizando AMQP

Nro.	Nome da fila	Local	Descrição
3	*{gatewayId}/event.health*	*Servidor*	Responsável pelo recebimento de informações sobre a disponibilidade do *gateway*.
4	*{gatewayId}/event.collected*	*Servidor*	Responsável pelo recebimento das mensagens publicadas pelo *gateway*.
5	*{gatewayId}/cmd.devices*	*Servidor*	Responsável pelo recebimento das mensagens com os comandos a serem executados pelos dispositivos ligados ao *gateway*.
6	*{gatewayId}/cfg*	*Servidor*	Responsável pelo recebimento das mensagens com as configurações a serem consumidas pelo *gateway*.

Fonte: Elaborado pelo autor (2018)

Figura 27 – Filas de mensagens padrão



Fonte: Elaborado pelo autor (2018)

3.2 SEGURANÇA

Como a solução necessita trafegar dados por rede pública, a comunicação deve possuir mecanismos de segurança que criptografem o tráfego de conexão. As conexões com o *broker* do servidor são protegidas pelo protocolo de segurança TLS. Através da biblioteca *OpenSSL* [3] são gerados os certificados que fazem parte da segurança, sendo necessário um certificado raiz para a autoridade de certificação (CA), um certificado para servidor e um para cliente. Certificados de servidor e cliente são assinados pela CA, criando uma cadeia de confiança (PIVOTAL, 2018). Uma vez que os certificados forem validados, a conexão segura é estabelecida, e com isso o tráfego de dados está criptografado.

Na conexão entre os dispositivos e o *gateway* local optou-se pela autenticação através de usuário e senha com o *broker*. Foram executados testes de criptografia das mensagens através do protocolo TLS embarcando os certificados nos dispositivos, porém como a comunicação local também estará com proteção de senha da rede sem fio, esta criptografia não foi utilizada para evitar atrasos na comunicação com o *broker.*

As comunicações do aplicativo móvel e painel de controle através do protocolo HTTP são validadas com o padrão JWT, citado na Subseção 3.1.3.4, o qual garante a segurança através de *tokens*. A comunicação assíncrona através do *SignalR* também é autenticada com o *token* gerado pelo JWT.

Neste capítulo foi apresentada a arquitetura modular que pode ser utilizada em diversos projetos IoT. Foram apresentados também os itens padrão que devem ser implementados, como filas, tópicos de mensagens e segurança envolvida. No Capítulo 4 é apresentado um problema a ser resolvido fazendo uso desta arquitetura, a fim de validá-la.

[3] *OpenSSL:* Biblioteca *open source* que implementa os protocolos SSL e TLS e fornece funções básicas relacionadas a criptografia.

4 VALIDAÇÃO DA SOLUÇÃO

A fim de validar a arquitetura, foi construída uma solução para a prevenção de incêndios e acidentes causados por distrações ou vazamentos de gás utilizando fogões. Através do uso de diversos sensores que coletam dados a serem analisados pelos agentes, podem ser percebidos possíveis descuidos ou mal-uso do fogão através do cruzamento das coletas e dos dados inseridos manualmente para comparação.

Durante a pesquisa sobre trabalhos relacionados à detecção de fogo e gás, como citado na Seção 2.8, foram encontrados alguns trabalhos voltados a detecção e ação em casos de incêndio, porém percebeu-se uma carência de solução que possa prevenir incêndios ocasionados por erros comuns do dia-a-dia, como o esquecimento de uma panela no fogo ou até mesmo erros no preparo dos alimentos.

O objetivo desta solução foi a prevenção de acidentes e não a automação de um fogão. O fluxo de gás é interrompido por completo e não individualmente por queimador, bem como não há dispositivos para acionamento remoto dos mesmos, uma vez que isso oferece vários riscos e pode colocar o usuário em perigo.

Conforme citado no Capítulo 3, os agentes presentes no *gateway* e no servidor foram desenvolvidos na linguagem C# através do *framework* .Net Core. O aplicativo móvel e o painel de controle foram desenvolvidos, respectivamente, através dos *frameworks* Ionic e Angular, como descrito no Capítulo 3, embora a arquitetura permita o uso de outras tecnologias. Os detalhes da solução desenvolvida são descritos ao longo deste capítulo.

4.1 ESTATÍSTICAS

Pode-se considerar o fogão a gás um utensílio doméstico de primeira prioridade, estando presente em praticamente todas as casas. Normalmente é alimentado através de gás GLP, Gás Liquefeito de Petróleo.

De acordo com Sindigás (2012), o gás GLP pode ser obtido através do refino do petróleo ou através de gás natural, sendo distribuído em recipientes de diversos tamanhos e pesos que passam por processos de requalificação em determinados intervalos de tempo, sendo submetidos a diversos testes de qualidade.

Acidentes envolvendo gás normalmente são de grande proporção e podem envolver várias vítimas devido a fácil propagação e alto poder de combustão. Sindigás (2012) menciona que em condições normais de trabalho um botijão de gás não explode e, devido a uma válvula de segurança presente no mesmo, quando elevado a altas temperaturas a mesma se abre e parte da pressão é liberada para evitar acidentes. Porém, se o botijão

continuar exposto a tais temperaturas, o mesmo pode explodir e colocar vidas em risco.

Muito embora falhas nos botijões possam causar acidentes, Sindigás (2017) cita que a maior parte dos acidentes ocorre por erros na instalação e uso inapropriado. A Tabela 1, utilizando a metodologia DPMO[1], aponta os acidentes reportados à Sindigás e que tiveram confirmação das causas.

Tabela 1 – Acidentes com recipiente de gás P13

P13		2013			
		Quant. Acidentes	Nível Sigma	Defeitos Por Milhão	Botijões Engarrafados no Período
Motivo do Acidente	Instalação	94	6,54	0,23	400.065.329
	Recipiente	30	6,75	0,07	
	Uso inapropriado	39	6,7	0,10	
	Impossibilidade de apuração	20	6,83	0,05	
TOTAL	Total de acidentes	183	6,41	0,457	

Fonte: Adaptado de SINDIGÁS (2017)

As próximas seções descrevem uma solução que tem por objetivo a prevenção de acidentes envolvendo gás e fogões, a fim de reduzir os causados por distrações ou descuido.

4.2 SOLUÇÃO IMPLEMENTADA

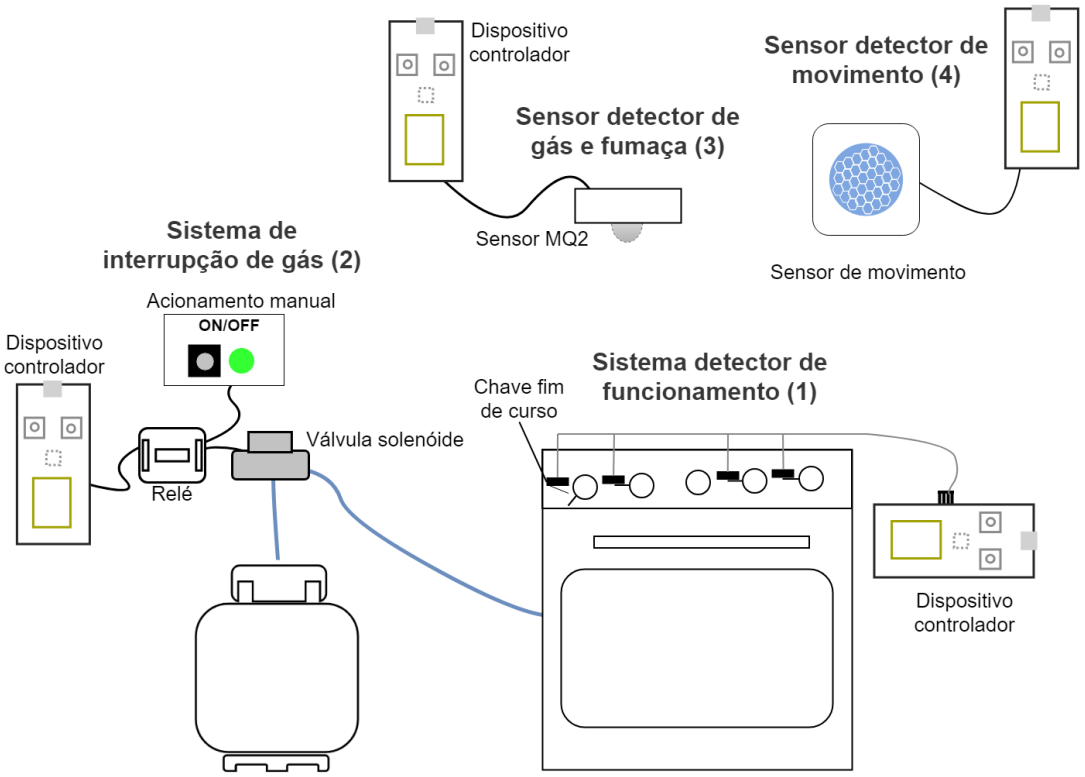
Esta seção explica individualmente as características dos dispositivos que foram montados e utilizados, bem como os softwares que foram construídos.

4.2.1 **Dispositivos utilizados**

A Figura 28 ilustra os componentes que foram montados e utilizados, sendo individualmente descritos nesta subseção com base na identificação numérica ao lado de cada um.

[1] DPMO: Defeitos por milhão de oportunidade (número de acidentes x 1.000.000 / botijões engarrafados no período)

Figura 28 – Modelo da solução implementada



Fonte: Elaborado pelo autor (2018)

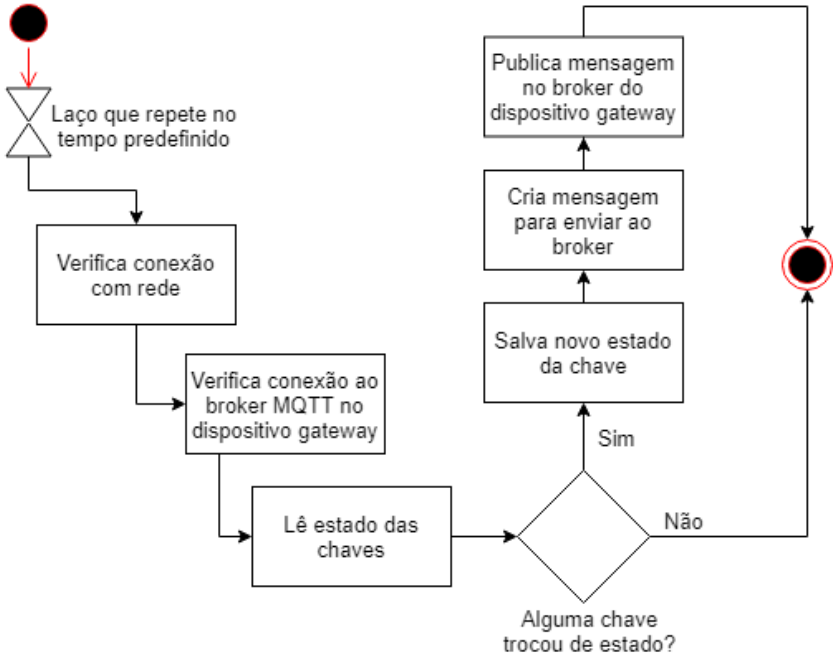
4.2.1.1 Sistema detector de funcionamento (1)

Este componente da solução é responsável pela leitura do estado dos botões correspondentes aos queimadores do fogão e posterior envio ao *gateway*. A detecção de funcionamento é realizada através do estado das chaves de fim de curso instaladas junto aos botões dos queimadores. Uma haste é instalada em cada botão, a fim de acompanhar o movimento do mesmo. O estado que representará o fim de curso da chave é quando o respectivo botão do fogão estiver desligado. Durante o desenvolvimento, um fogão com 4 queimadores foi utilizado para fins de teste, porém mais chaves poderiam ser adicionados para atender a fogões com maior número de queimadores.

O diagrama da Figura 29 representa o fluxo de funcionamento deste subsistema. A cada iteração do laço principal ocorre a verificação da conexão com a rede e com o *broker*, seguidos da leitura das chaves e, em casos de alterações de estado, realiza-se o envio de mensagens com os eventos ao *broker* do dispositivo *gateway*. A placa utilizada para a

leitura das chaves foi uma NodeMCU (TEAM, 2015), citada na Seção 2.2.

Figura 29 – Diagrama de funcionamento do detector de funcionamento



Fonte: Elaborado pelo autor (2018)

4.2.1.2 Sistema de interrupção de fluxo de gás (2)

O sistema de interrupção de gás pode ser acionado de forma automática em casos de detecção de riscos através dos agentes localizados no servidor, ou pode haver acionamento através de comandos enviados pelos usuários da solução.

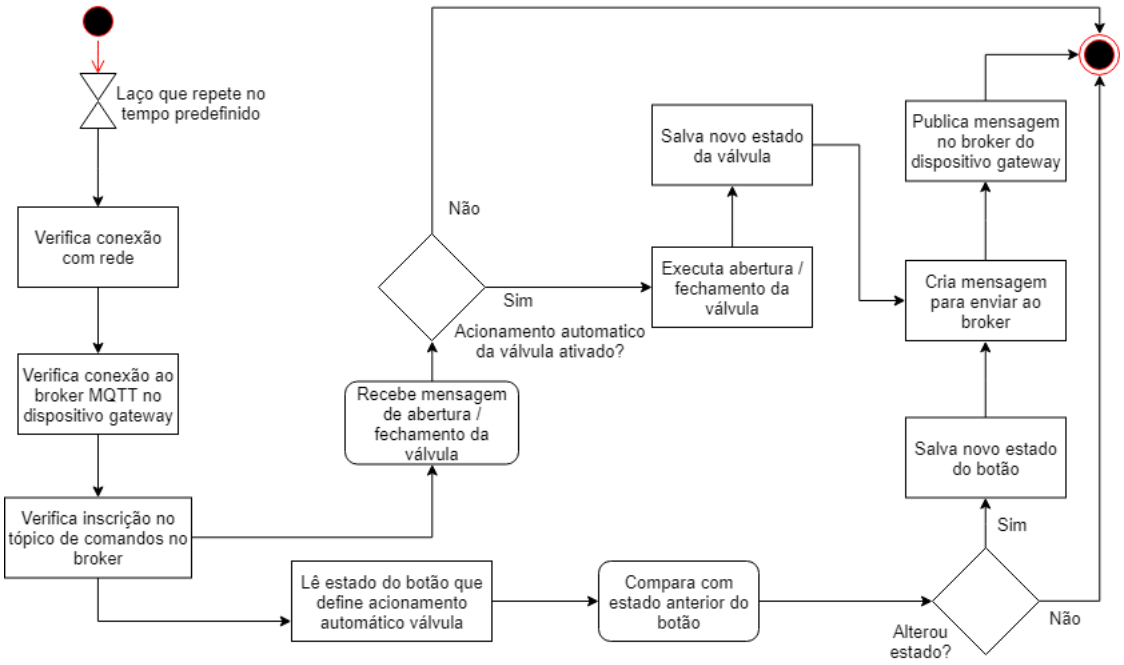
Os comandos automáticos de acionamentos podem ser originados dos agentes presentes no servidor, durante o consumo das mensagens para persistência, ou posteriormente, durante as análises realizadas pelo agente responsável. Além disso, caso a comunicação entre servidor e *broker* seja perdida, o *broker* também tem autonomia para enviar comandos emergenciais ao receber mensagens críticas e que podem indicar riscos, conforme citado na Subseção 3.1.2.2, bem como quando o limite de segurança for ultrapassado.

Os comandos enviados pelo aplicativo são recebidos pela API presente no servidor, convertidos em mensagens de comandos e publicadas no *broker* do servidor para posterior consumo do *gateway*, explicado na Subseção 3.1.2.2.

Para reestabelecer o fluxo de gás, o usuário pode utilizar o botão presente no aplicativo. Caso a comunicação do *gateway* com o servidor esteja inativa, o usuário não poderá utilizá-lo para enviar comandos, sendo necessário desativar o controle de fluxo manualmente utilizando um botão presente neste subsistema. Após a comunicação ser

reestabelecida, o usuário deve religar o subsistema novamente para reativar a segurança. A Figura 30 ilustra o fluxo descrito.

Figura 30 – Diagrama de funcionamento do sistema de interrupção de gás



Fonte: Elaborado pelo autor (2018)

4.2.1.3 Sensor detector de gás e fumaça (3)

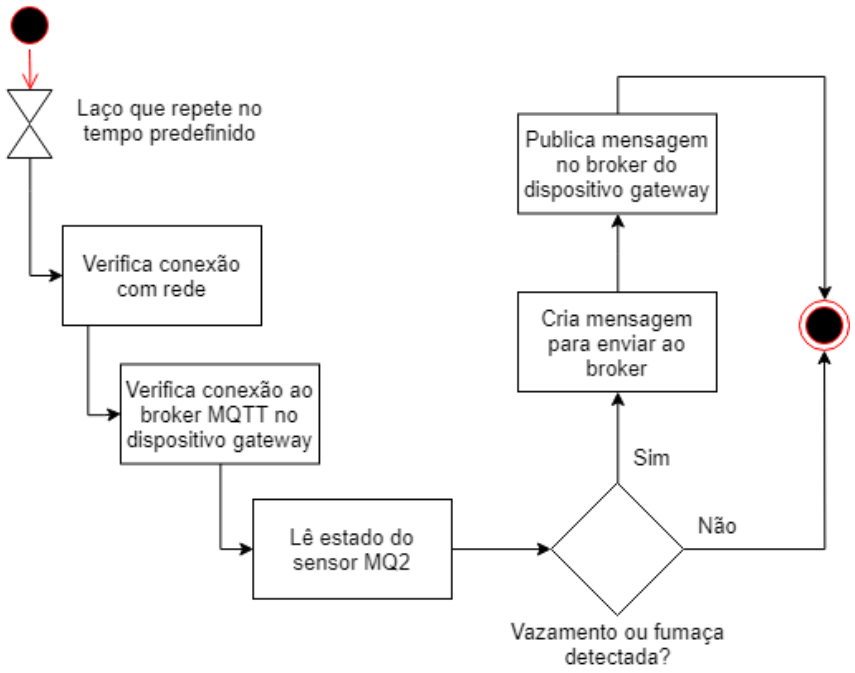
Este subsistema é instalado próximo ao fogão e é responsável por detectar e notificar a presença de gases inflamáveis e fumaça, também enviando os dados coletados ao *gateway*.

O funcionamento é dado por um sensor MQ2 que, através de material sensível, aumenta a condutividade quando ocorre a detecção de gases ou fumaça. O sensor MQ2 pode detectar fumaça e gases como GLP, Gás Natural, Propano, Butano, dentre outros inflamáveis (FILIPEFLOP, 2018).

Os dados gerados pelo sensor são coletados por uma placa NodeMCU e ao ocorrer detecção de algum gás e/ou fumaça, uma mensagem é enviada ao *broker* do dispositivo *gateway*.

Esta mensagem é crítica e deve ser tratada mesmo se não houver conexão entre o *gateway* e o servidor. Para que este tratamento seja executado, o usuário deve informar no painel de controle esta relação entre evento e ação. A Figura 37 mostra esta configuração. O diagrama presente na Figura 31 ilustra o fluxo de leitura do sensor e envio da mensagem.

Figura 31 – Diagrama de funcionamento do sistema de interrupção de gás



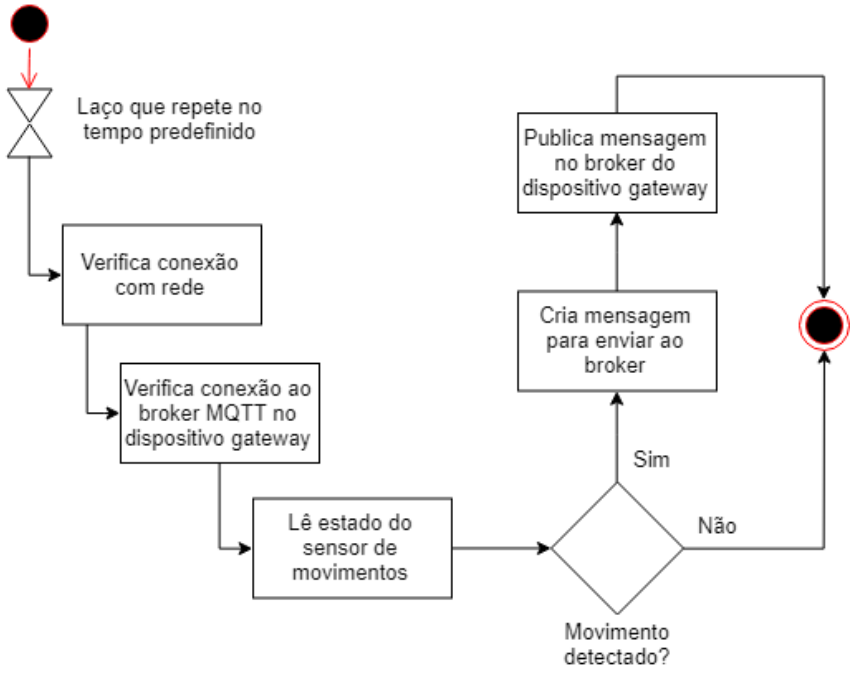
Fonte: Elaborado pelo autor (2018)

4.2.1.4 Detector de presença (4)

O sensor de presença DYP-ME003 é um sensor baseado na tecnologia infravermelho, capaz de detectar movimentos a uma distância máxima de 7 metros. Possui duas formas básicas de operação: na primeira forma, ao detectar um movimento, a saída fica em nível alto, isto é, fornecendo tensão máxima no pino de sinal. Quando o tempo de atraso da detecção chega ao final, o pino torna-se nível baixo novamente, desativando a tensão no pino de sinal. Na segunda forma de operação, quando um movimento for detectado, o pino será configurado para nível alto e, enquanto houver movimento durante o tempo de atraso da detecção, o pino permanecerá em nível alto, sendo que o tempo de atraso será sempre baseado na última ocorrência. Se ao final do tempo nada for detectado, o pino volta a ser nível baixo. (SHENZHEN, 2018). O sensor suporta também regulagem de distância, variando entre, aproximadamente, 3 e 7 metros e regulagem de tempo de atraso, variando entre 5 segundos e 200 segundos.

Através da placa NodeMCU, mensagens são enviadas ao *broker* do dispositivo *gateway* a cada detecção ocorrida. A solução pode conter mais de um detector de presença, possibilitando assim maior precisão na tomada de decisões. A Figura 32 mostra o fluxo realizado por este subsistema.

Figura 32 – Diagrama de funcionamento do sistema detector de presença



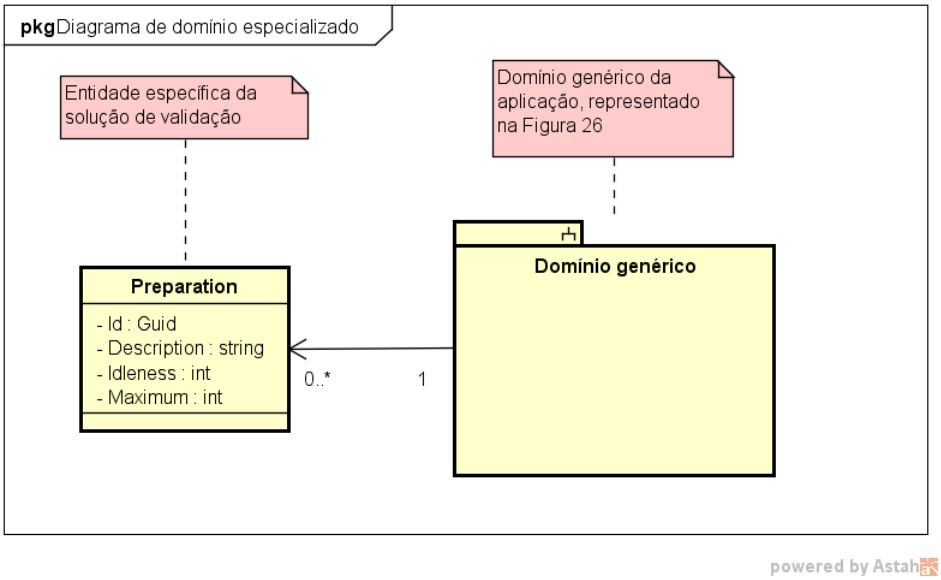
Fonte: Elaborado pelo autor (2018)

4.2.2 *Softwares* construídos

Os *softwares* clientes implementados consistem de um painel de controle e um aplicativo móvel, estes comunicando-se com a API em execução no servidor. A API foi construída com base nas especificações feitas na Subseção 3.1.3.4, necessitando de algumas adequações para atender a este cenário de uso. Dentre as alterações necessárias, pode-se destacar a criação de uma entidade específica da solução de validação, como pode ser visto na Figura 33.

A Figura 33 apresenta a entidade *Preparation*, responsável pela representação dos preparos dos alimentos, contendo informações sobre tempo de ociosidade, tempo máximo de preparo, além de um identificador único e uma descrição. Ainda conforme ilustra a Figura 33, o modelo de domínio proposto na Figura 26 fica inalterado, mantendo-se todas as características da arquitetura proposta.

Figura 33 – Modelo de domínio da solução de validação



Fonte: Elaborado pelo autor (2019)

Nesta Subseção são descritas as características que os *softwares* clientes desenvolvidos possuem.

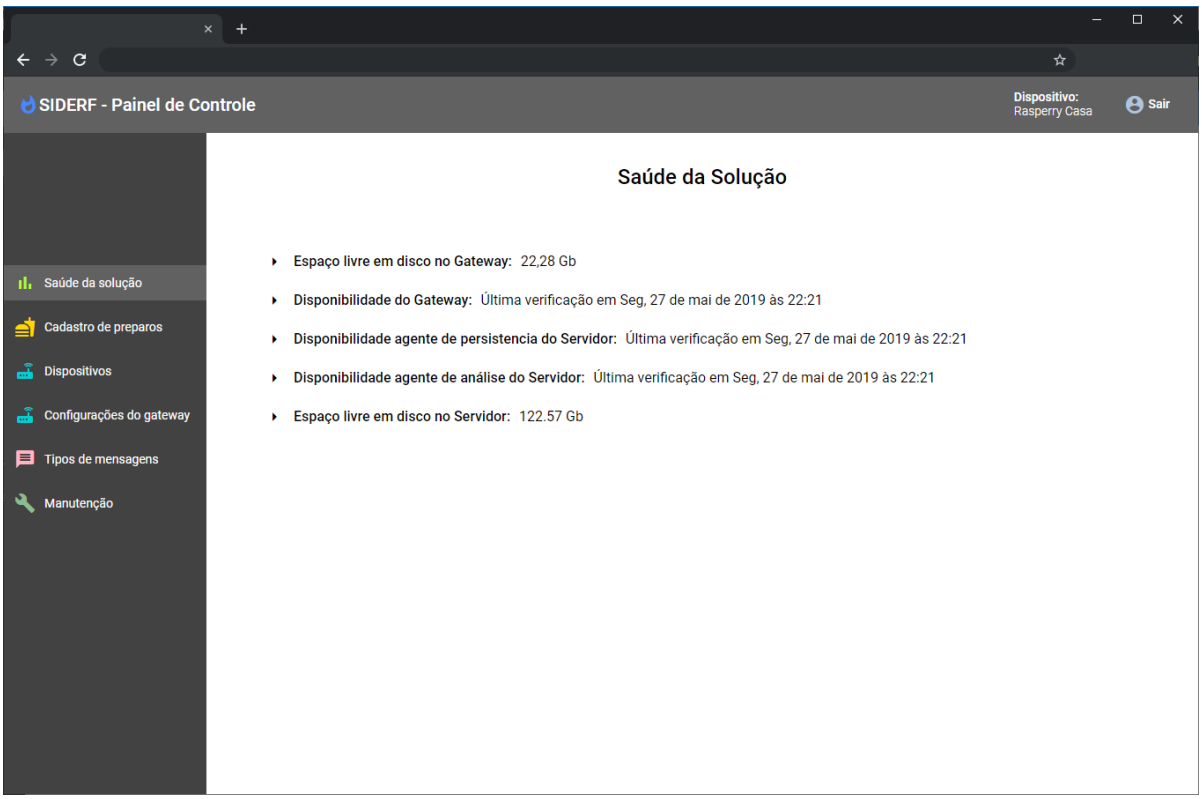
4.2.2.1 Painel de controle

O painel de controle desta solução oferece, além dos recursos básicos de configuração citados na Subseção 3.1.4.2, uma maneira de inserir dados referentes a tempos de preparo de determinados tipos de alimentos. Através destes tempos é possível, através do aplicativo móvel, escolher um tipo de alimento que será preparado em determinado queimador do fogão e, a partir disso, gerar notificações ao usuário com base nos dados inseridos cruzados aos dados de movimento coletados. Ações preventivas também podem ser tomadas com base nestas informações inseridas.

As figuras presentes nesta subseção ilustram os recursos do painel de controle. Configurações relacionadas ao *gateway* devem ser confirmadas antes de serem aplicadas. A Figura 35 mostra a mensagem alertando sobre as alterações no topo da página. Após a confirmação, as mesmas serão enviadas e aplicadas em até 1 minuto.

A Figura 34 ilustra as informações que estão disponíveis no menu “Geral” do painel de controle. Nele estão disponíveis os dados referentes ao status geral de alguns dos elementos que compõem a solução.

Figura 34 – Tela geral do painel de controle



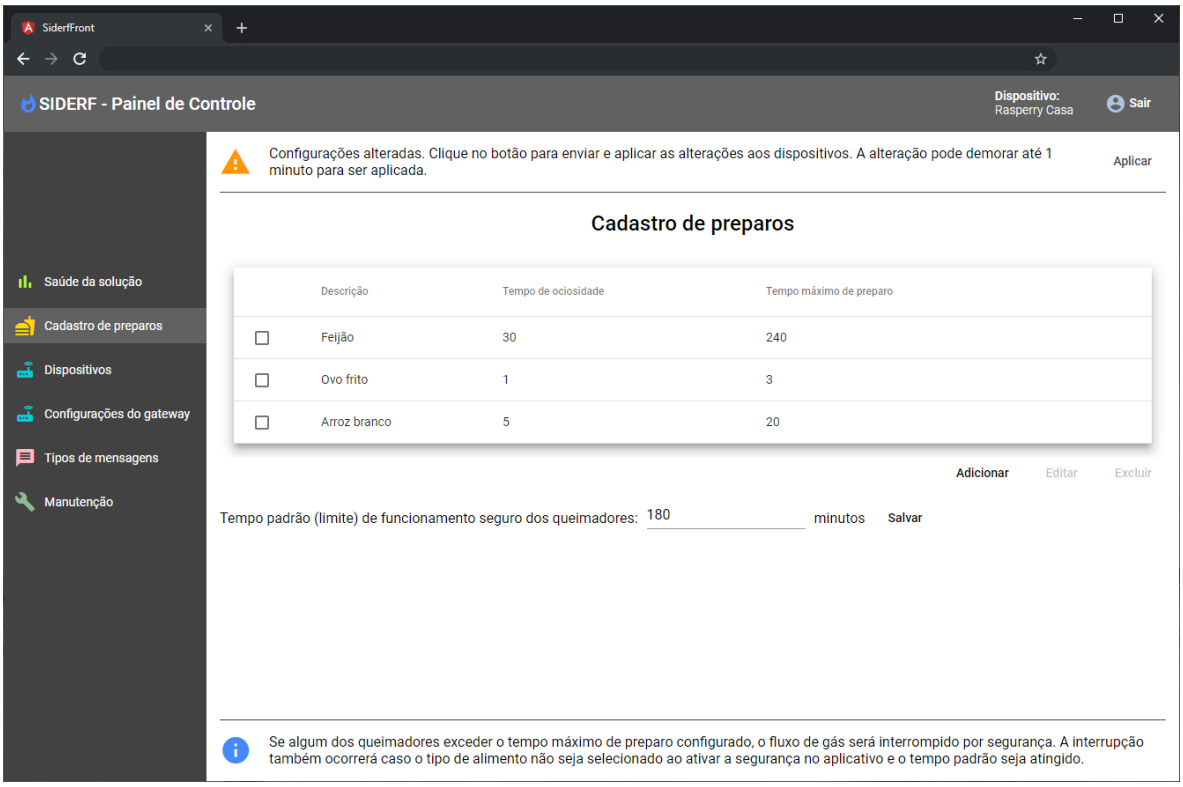
Fonte: Elaborado pelo autor (2019)

Na Figura 35 pode ser observado o menu desenvolvido especificamente para a solução de validação, permitindo o cadastrado dos tempos de preparo que são utilizados como parâmetro no momento do uso do fogão.

Durante o cadastro de um tempo de preparo, o usuário pode definir uma descrição para identificar o mesmo, bem como pode cadastrar um tempo de ociosidade e um tempo máximo de preparo. O tempo de ociosidade é utilizado para a geração de notificações junto aos dados do sensor de movimentos, onde pode ser possível identificar e avisar o usuário de possíveis esquecimentos e evitar assim riscos ao mesmo. O tempo máximo de preparo serve como um tempo de segurança. Se o tempo de funcionamento do queimador ultrapassar o valor estipulado neste campo, ocorre a interrupção do fluxo de gás preventivamente.

Ainda nesta tela há a configuração de tempo limite considerado seguro no uso dos queimadores do fogão. Este valor serve de referência caso um queimador seja utilizado sem indicação do tipo de alimento que está sendo preparado. Se este tempo for excedido, também há a interrupção do gás preventivamente.

Figura 35 – Tela de cadastro de tempos do painel de controle

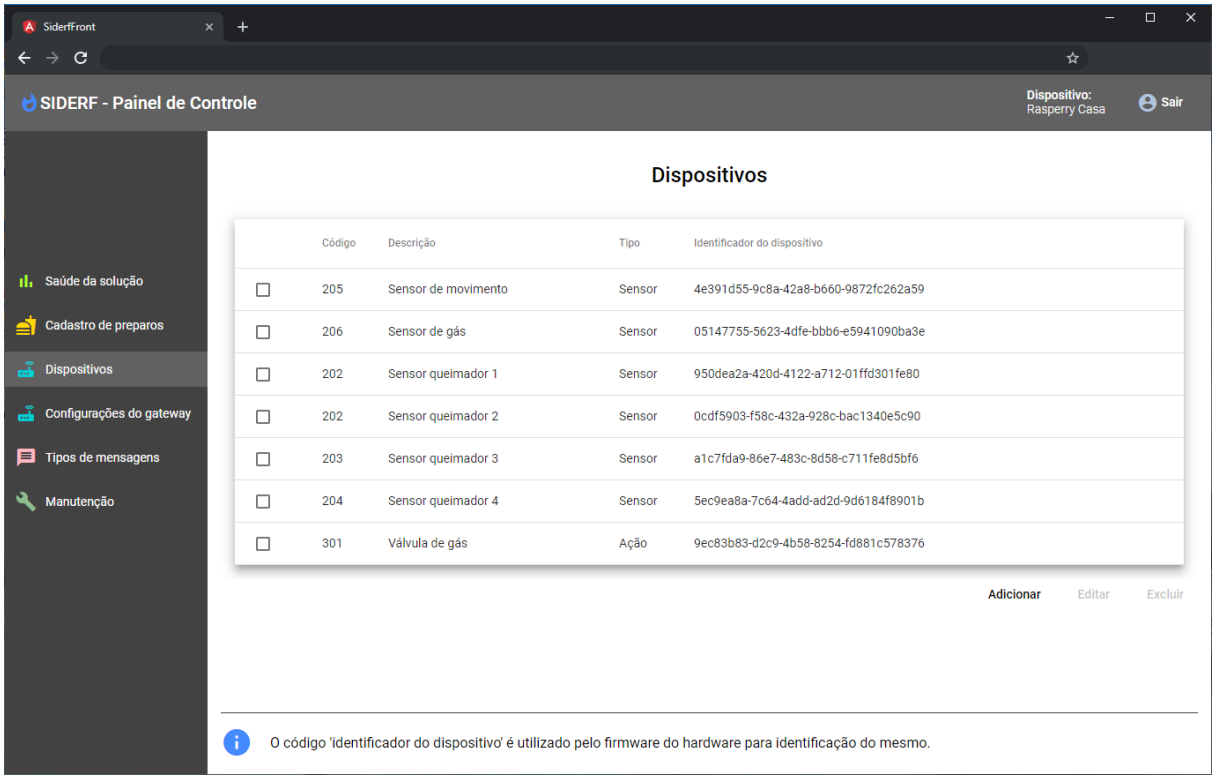


Fonte: Elaborado pelo autor (2019)

A Figura 36 mostra o cadastro de dispositivos do *software*. Nesta tela, após o cadastro de um novo dispositivo, é possível visualizar o identificador do mesmo, utilizado como identificação única em toda a solução.

Esta sequência de caracteres também é utilizada no *software* dos dispositivos, sendo inserida durante o seu desenvolvimento. Através deste menu também é possível realizar alterações de algumas informações previamente cadastradas ou remover o dispositivo.

Figura 36 – Tela de configurações dos dispositivos no painel de controle



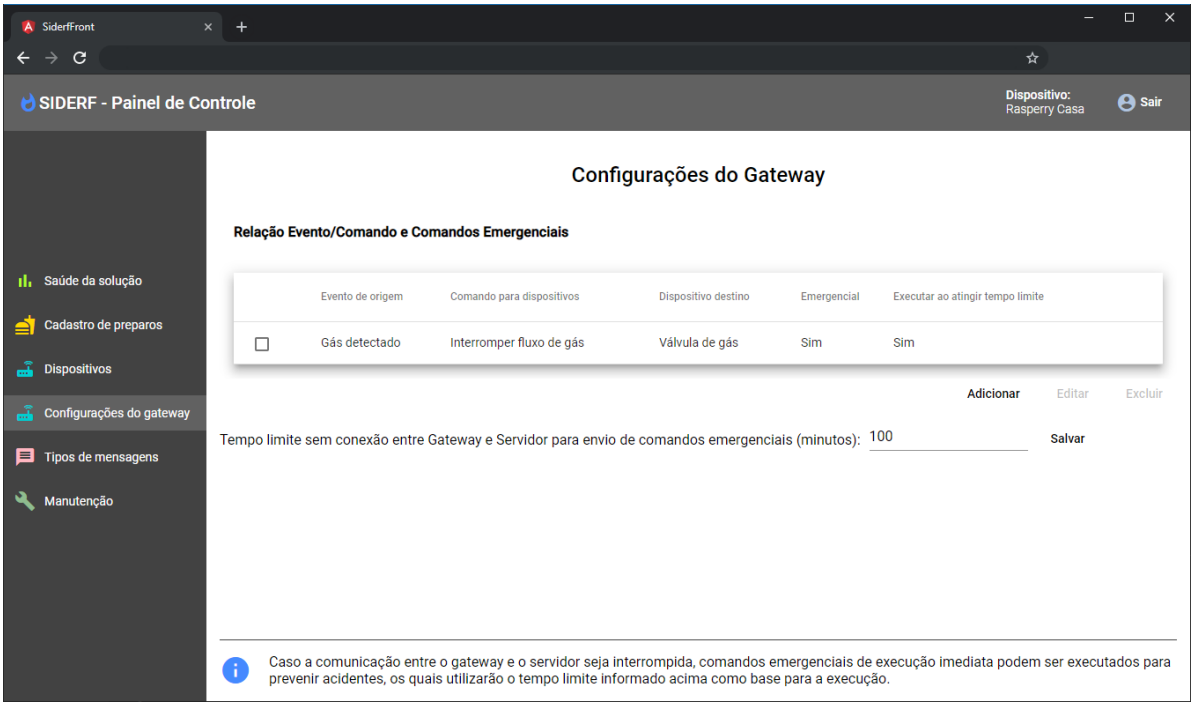
Fonte: Elaborado pelo autor (2019)

A Figura 37 mostra o menu do *software* responsável pelo cadastro de ações da arquitetura. Neste menu é possível cadastrar gatilhos que possibilitam à arquitetura enviar comandos quando determinados eventos forem recebidos. Durante o cadastro de uma relação *evento X comando*, é possível informar se a mesma é emergencial e se ela executará ao atingir o tempo limite.

Se a relação for emergencial, ela será enviada ao gateway da solução para que possa ser executada mesmo se a conexão entre ele e o servidor estiver inativa, sendo assim, caso o gateway esteja *offline* e receber uma mensagem com o evento informado, irá executar o comando relacionado. Caso a relação seja configurada para executar ao atingir o tempo limite, o *gateway* irá executar o comando configurado independentemente de ter recebido o evento relacionado, usando assim o tempo limite informado no campo da tela representada pela Figura 37.

Através desta configuração é possível, por exemplo, informar ao *gateway* que caso ocorra instabilidade na conexão e uma mensagem de gás detectado for recebida, o mesmo deve enviar um comando solicitando o fechamento do fluxo de gás do fogão. Pode-se ainda configurar para para que o fluxo de gás seja interrompido após o *gateway* ficar *offline* por determinado tempo.

Figura 37 – Tela de configurações do gateway no painel de controle

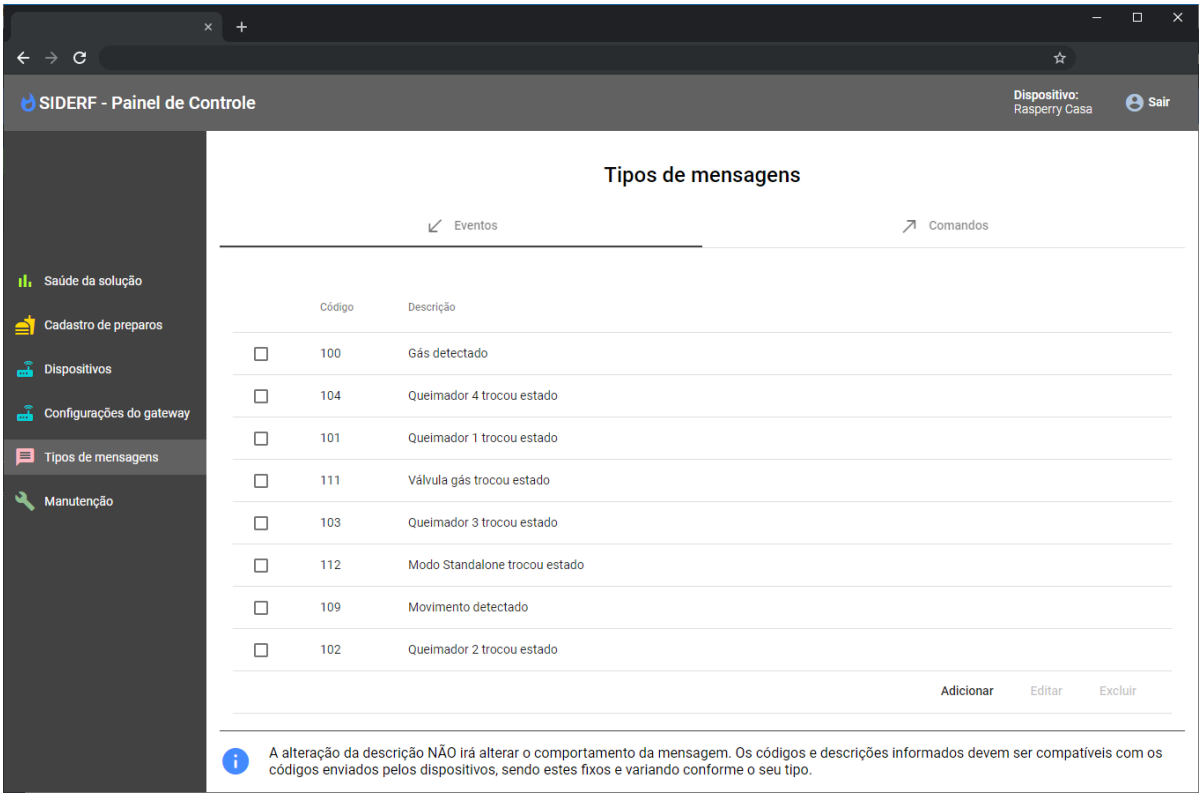


Fonte: Elaborado pelo autor (2019)

A Figura 38 tem como objetivo possibilitar o cadastro de uma descrição para os códigos de mensagens que trafegam pelo sistema. Nela é possível observar que existem duas abas, a aba com as mensagens de eventos e a aba com as mensagens de comandos.

Este cadastro serve apenas para facilitar a identificação, uma vez que a definição de códigos deve ser feita no momento da programação dos dispositivos. A troca de nomes não alterará a função a qual está relacionada com o dispositivo.

Figura 38 – Tela de configurações de tipos de mensagens no painel de controle



Fonte: Elaborado pelo autor (2019)

O menu “Manutenção” contém as opções relacionadas a autenticação e manutenção do sistema.

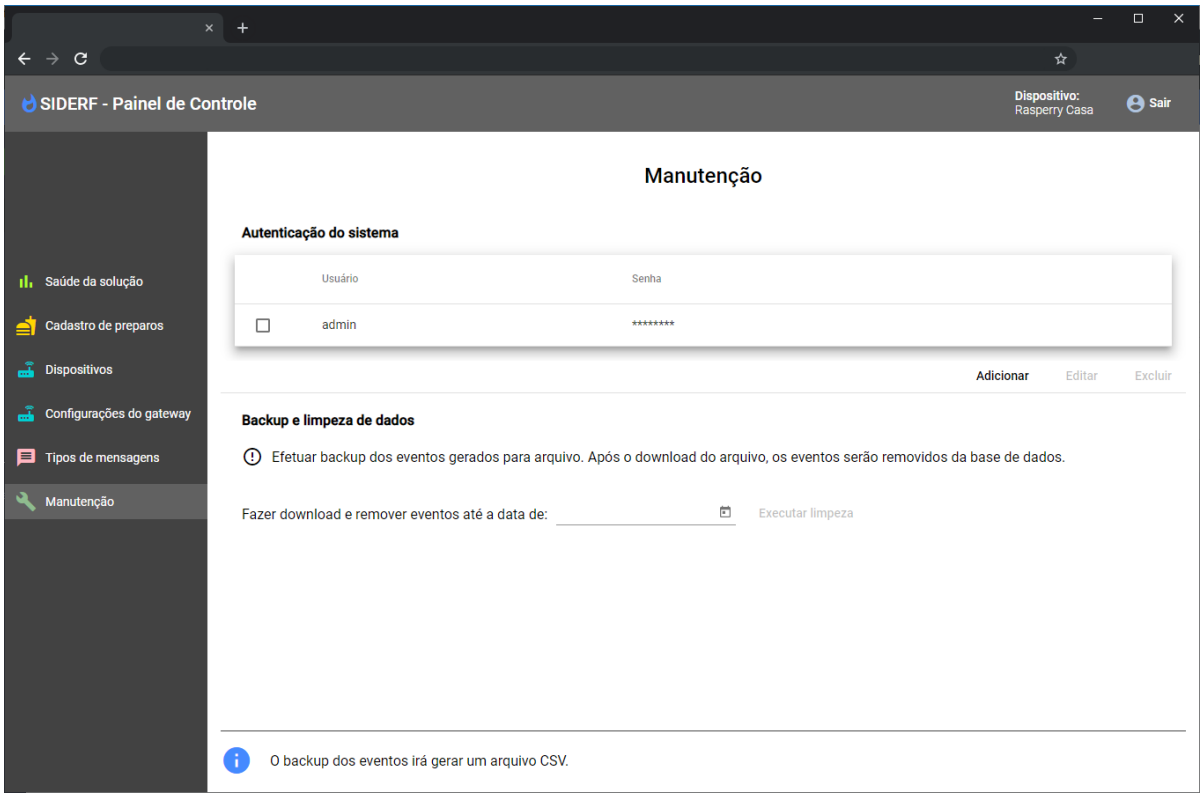
A autenticação dos usuários ao sistema é feita através de usuário e senha, sendo este cadastro realizado pelo painel de controle, conforme mostra a Figura 39.

Durante o uso da aplicação muitos eventos podem ser persistidos na base de dados e como reflexo é requerido maior armazenamento de disco no servidor.

Conforme mostra a Figura 39, através das opções presentes nesta tela torna-se possível solicitar a execução da limpeza da base de dados até a data informada. A partir disso, o sistema gera um arquivo com todos os eventos até a data definida e os remove do banco de dados. O arquivo resultante é então baixado pelo usuário.

O usuário pode consultar o estado atual do disco a partir dos indicadores presentes no menu “Geral” do painel de controle, ilustrado na Figura 34.

Figura 39 – Tela de manutenção no painel de controle



Fonte: Elaborado pelo autor (2019)

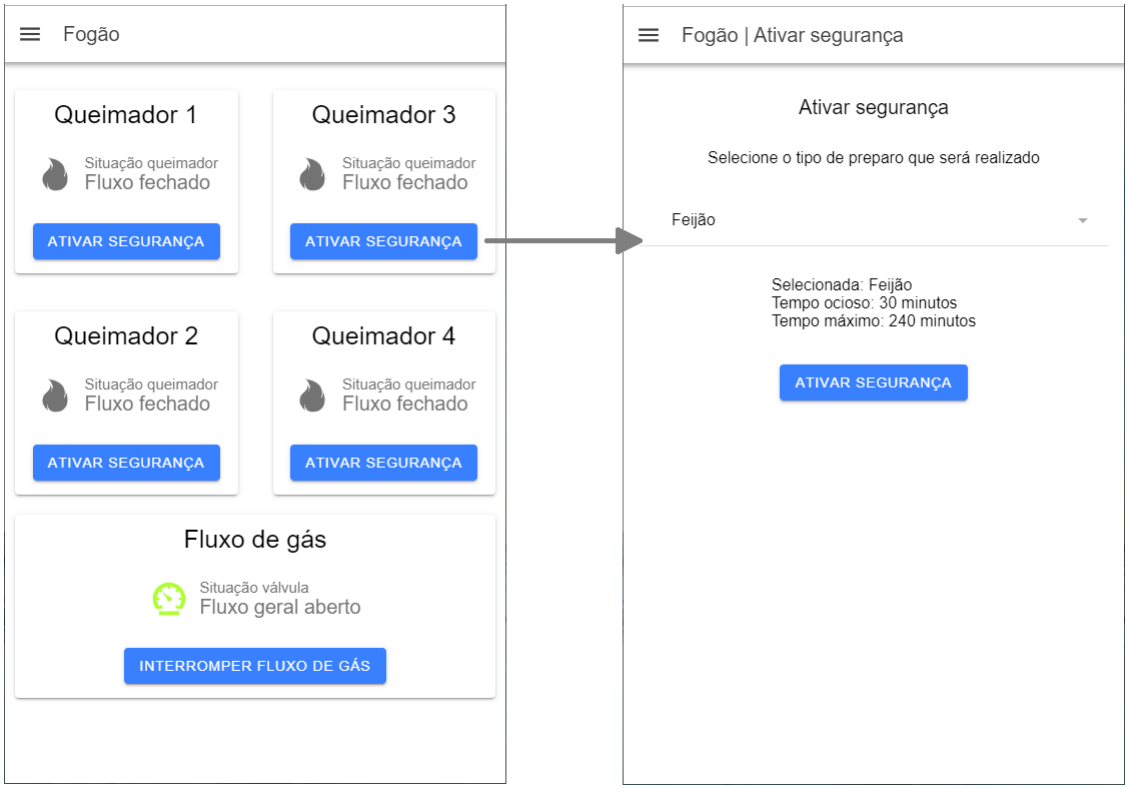
4.2.2.2 Aplicativo móvel

O aplicativo móvel foi desenvolvido seguindo a tecnologia de comunicação citada na Subseção 3.1.4.1. As suas funcionalidades podem ser observadas através das figuras presentes nesta subseção, junto as explicações correspondentes.

A Figura 40 permite uma visualização de como é a tela com as informações sobre os queimadores do fogão e do estado do fluxo de gás. Esta tela possibilita ao usuário acompanhar quais queimadores estão ativos no momento, bem como configurar a segurança para os mesmos.

A tela de configuração de segurança exibe os tipos de preparo cadastrados previamente pelo painel de controle, conforme exibido na Figura 35. Ao selecionar o tipo de cozimento, o usuário pode observar o tempo de ociosidade em que será notificado sobre a mesma e o tempo máximo de cozimento, servindo como base para a interrupção do fluxo de gás.

Figura 40 – Tela do aplicativo com visualização dos queimadores e configurações para ativar segurança dos mesmos



Fonte: Elaborado pelo autor (2019)

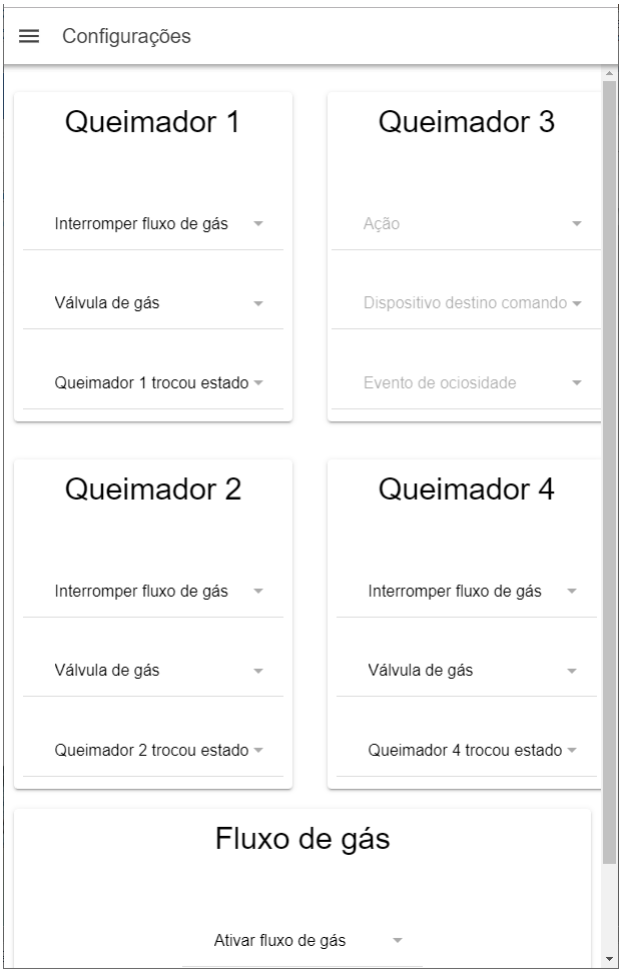
A Figura 41 exibe a tela de configuração do aplicativo, onde são definidas as relações entre os dispositivos com os tipos de eventos e comandos existentes. Quando o dispositivo configurado for um sensor, deverão ser informados:

a) Ação: ação a ser executada como medida de segurança caso o tempo do preparo for excedido.
b) Dispositivo destino do comando: dispositivo que receberá o comando.
c) Evento de ociosidade: evento de ociosidade relacionado ao queimador que servirá como comparação no momento do envio das notificações.

Quando o dispositivo configurado for do tipo “ação”, deverão ser informados:

a) Comando ligar: comando a ser enviado para acionar o dispositivo.
b) Comando desligar: comando a ser enviado para desativar o dispositivo.

Figura 41 – Configurações dos controles no aplicativo

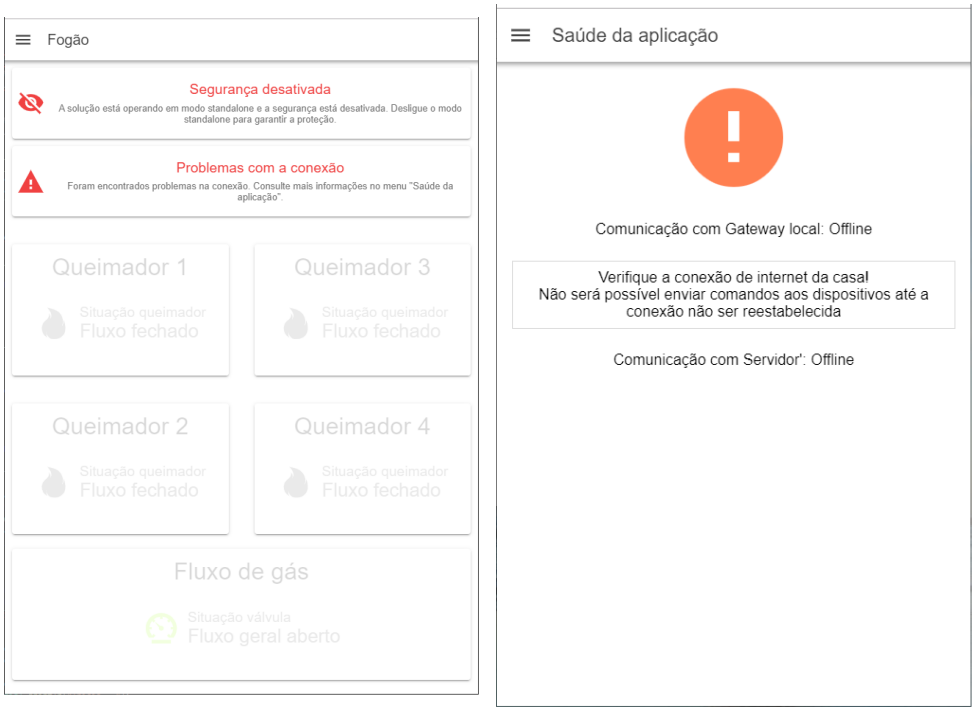


Fonte: Elaborado pelo autor (2019)

Na Figura 42a e Figura 42b pode ser observado o estado do aplicativo quando o *gateway* perder a conexão com o servidor. Na parte superior da Figura 42a é exibida a mensagem informando problemas na conexão.

Ainda na Figura 42a é possível observar que nenhuma ação pode ser realizada enquanto o *gateway* estiver sem conexão, sendo necessário reestabelecer a mesma para o envio de comandos. Caso o usuário configure um comando emergencial para interrupção do gás na ausência de conexão, conforme mostrado na Figura 37, ainda é possível reativar o fluxo manualmente através de um botão presente no *hardware* que compõe a solução. Quando o acionamento manual ocorrer, a solução não intervém mais no fluxo de gás e logo a proteção é desativada. A partir do momento em que for reestabelecida a conexão com a *internet*, o usuário deve acionar o botão novamente para reativar a proteção. A Figura 42b exibe o menu com a disponibilidade da aplicação, mostrando que existem problemas entre o *gateway* e o servidor, impossibilitando o envio de comandos e recebimento de eventos dos dispositivos.

Figura 42 – Aplicativo com estado de falha



(a) Tela principal

Fonte: Elaborado pelo autor (2019)

(b) Tela de *status*

Fonte: Elaborado pelo autor (2019)

Após a interrupção do fluxo de gás, caso a comunicação entre *gateway* e servidor esteja ativa, a liberação do mesmo é feita pelo aplicativo através do botão “Reativar fluxo de gás”. Ao pressionar o botão, uma segunda tela é mostrada orientando o usuário a conferir todos os botões do fogão antes de liberar o fluxo, deixando-os todos desligados, caso contrário vazamentos poderão oferecer riscos ao usuário e a solução irá detectar e interromper o fluxo novamente. A Figura 43 mostra este recurso.

Figura 43 – Aplicativo exibindo opção para liberar fluxo de gás

Fonte: Elaborado pelo autor (2019)

4.3 Métricas coletadas

A fim de gerar métricas para obter os limites suportados pela arquitetura, foram executados testes de exaustão através de um *software* desenvolvido, o qual simula a geração de eventos representando detecção de movimento em um ambiente. Durante sua execução, foram realizados, para cada quantidade presente na coluna 1 da Tabela 2, 4 execuções com mesmo número de eventos, a fim de obter uma média aritmética da diferença entre o tempo de persistência do primeiro e último evento no servidor. A Tabela 2 mostra os resultados obtidos.

Ao serem executados os testes, nenhuma mensagem foi perdida durante a execução, confirmando a estabilidade da solução, que levou em média 17 milissegundos para persistir cada mensagem. Além dos testes com as quantidades mencionadas na Tabela 2, também foram executados testes de envio de mensagens simultâneas de múltiplos clientes, através

do mesmo *software* desenvolvido, porém através do uso de *threads*. Durante este teste foram iniciadas 1000 *threads*, sendo que cada *thread* criou uma nova conexão e enviou 10 mensagens à arquitetura, obtendo sucesso na execução.

Tabela 2 – Tempo de consumo de eventos

Quantidade	Tempo médio
10	00:00:00,1527 horas
100	00:00:01,7020 horas
1.000	00:00:17,1154 horas
10.000	00:02:52,3096 horas
100.000	00:27:33,2965 horas
1.000.000	05:03:39,1146 horas

Fonte: Elaborado pelo autor (2019)

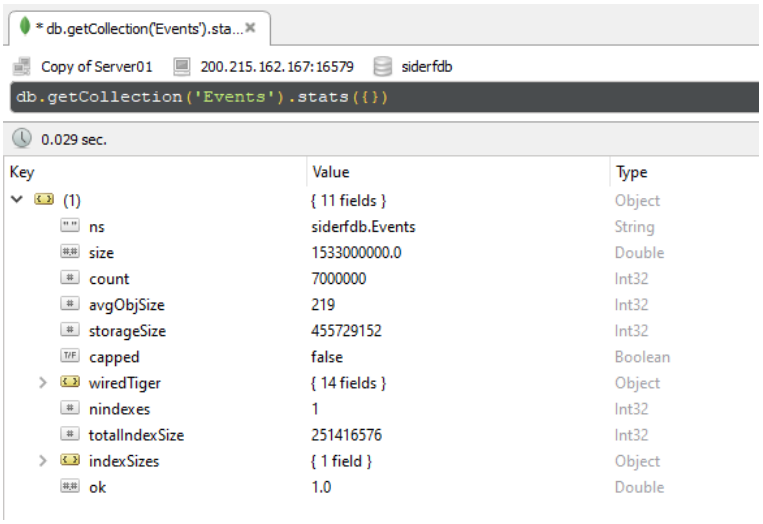
Ainda fazendo uso do *threads*, foram iniciadas 1.000 conexões simultâneas com o envio de 1.000 mensagens em cada conexão. Durante esta execução, constatou-se atrasos no consumo das mensagens à medida em que estas acumulavam-se no *broker* do servidor, atraso este devido a uma limitação de desempenho do *hardware Raspberry Pi 3B* do *gateway*.

Embora 1000 conexões com 1.000 mensagens acarretaram em atraso, cabe salientar que este teste de exaustão foi feito com o intuito de conhecer os limites suportados pelo *hardware* utilizado e a aplicação proposta como prova de conceito demanda poder computacional muito menor aos testes realizados.

Devido a modularidade da arquitetura proposta por este trabalho, soluções que possam vir a implementar esta e que gerem uma quantidade massiva de mensagens a ponto de atrasar o consumo das mesmas, podem substituir o *hardware* utilizado como *gateway* por um com maior capacidade computacional a fim de maximizar o desempenho.

Visando mensurar o espaço necessário pela aplicação no servidor, 7 milhões de eventos foram gerados a fim de coletar o espaço em disco requerido para armazenamento. Conforme mostra a Figura 44, o espaço necessário foi de 455,73 MB. A partir deste resultado, pode-se mensurar que uma mensagem de evento ocupará em disco, em média, 65,10 *bytes*.

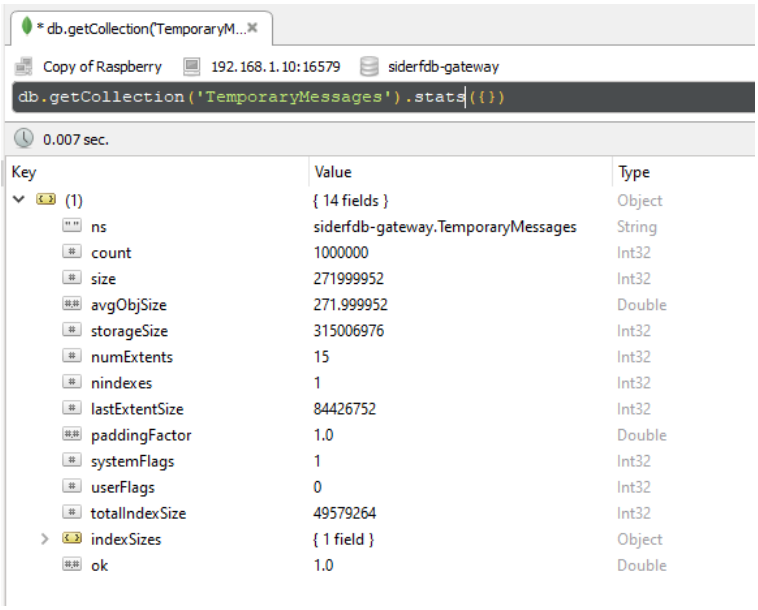
Figura 44 – Estatística de armazenamento MongoDB



Fonte: Elaborado pelo autor (2019)

Devido ao sistema de contingência proposto pela arquitetura, foi necessário também analisar a quantidade de mensagens *offline* suportadas pelo *gateway* em caso de indisponibilidade da conexão com a internet. Durante os testes foram persistidas 1.000.000 mensagens de movimento em situação de contingência. A Figura 45 mostra a estatística gerada pelo banco de dados do *gateway*. O espaço de armazenamento utilizado foi de 300,41 MB, portanto pode-se afirmar que uma mensagem de contingência ocupa, em média, 315,01 bytes.

Figura 45 – Estatística de armazenamento em modo de contingência



Fonte: Elaborado pelo autor (2019)

O *gateway Raspberry Pi 3B* utilizado na solução possui um cartão *MicroSD* com capacidade de 32 GB, sendo que após a instalação do sistema operacional e *softwares* necessários ficou com 22,91 GB de espaço disponível. Com isso, pode-se estimar que a arquitetura suporte 70 milhões de mensagens *offline*. Se a solução que faz uso da arquitetura gerar uma mensagem por segundo, 24 horas por dia, poderiam ser armazenados dados resultantes de 810 dias ininterruptos.

Neste capítulo foi descrito um problema que pôde ser resolvido através da arquitetura proposta no Capítulo 3. Os detalhes da implementação foram apresentados e, ao final do capítulo, foram realizados testes de desempenho com a finalidade de encontrar os limites da arquitetura. No Capítulo 5 são apresentadas as conclusões do trabalho e são citados trabalhos futuros que podem ser desenvolvidos visando a expansão deste.

5 **CONCLUSÕES**

Neste trabalho foi desenvolvida uma arquitetura modular para possibilitar o monitoramento e intervenção remota através de sensores e dispositivos, implementando mecanismo de contingência capaz de executar ações emergenciais em casos de falha na comunicação entre o servidor e o *gateway* local. Com foco no baixo custo, optou-se pelo uso de um hardware limitado e compacto, a *Raspberry Pi 3B*, sendo esta responsável pela execução de um *broker* que recebeu mensagens dos sensores e dispositivos.

Durante pesquisa sobre tecnologias disponíveis no mercado, optou-se por utilizar o formato *publish/subscribe* na troca de mensagens dos dispositivos locais com o *gateway*, bem como do *gateway* com o servidor, uma vez que este tipo de comunicação é assíncrona e indicada para este tipo de aplicação. Na escolha dos *brokers*, optou-se pelo uso do Eclipse Mosquitto com protocolo MQTT na *Raspberry*, uma vez que este *broker* é leve e exige poucos recursos de *hardware* e o protocolo MQTT possui baixa latência e trafega mensagens de cabeçalho compacto. No servidor, o *broker* utilizado foi o RabbitMQ, visto que o mesmo possui suporte ao protocolo AMQP e já foi analisado em um trabalho anterior a este por Girardi (2015), obtendo o melhor resultado dentre os testados.

Durante o projeto da arquitetura percebeu-se a necessidade de um painel de controle para a administração da mesma, de modo a possibilitar algumas parametrizações que anteriormente ficariam fixas no *software*.

Ao pesquisar por uma solução a ser implementada como prova de conceito que tivesse como objetivo a segurança e diminuição de riscos domésticos, foram encontradas várias soluções voltadas a detecção de gás e fogo em um determinado ambiente, porém percebeu-se uma carência de produtos com a capacidade de analisar e predizer possíveis riscos através da leitura de sensores presentes em um determinado ambiente. Através desta carência, optou-se por implementar uma solução que pudesse alertar por meio de coletas de dados de alguns sensores, sobre possíveis distrações no uso de fogões caracterizados por excesso de funcionamento de um determinado queimador, bem como pudesse intervir e interromper o fluxo de gás caso algum tempo fosse excedido ou risco fosse detectado. A solução implementada baseou-se na arquitetura apresentada no Capítulo 3.

Após a implementação da solução, algumas métricas de desempenho e armazenamento foram coletadas para poder mensurar a capacidade da arquitetura, as quais foram apresentadas na Seção 4.3. Os resultados obtidos foram satisfatórios, sendo possível afirmar que a contingência seria capaz de armazenar até 800 dias de mensagens *offline*, muito acima do necessário, visto que a contingência da arquitetura tem por objetivo o armazenamento temporário das informações.

Durante os testes de exaustão, os quais simularam condições de trabalho muito acima das exigidas pela solução de validação, comprovou-se que mesmo com uma elevada carga de mensagens simultâneas, nenhuma foi perdida. O envio simultâneo de 10 mensagens em 1.000 conexões distintas pôde comprovar que a arquitetura suportaria um número alto de dispositivos enviando mensagens paralelamente.

Quando a arquitetura foi submetida ao consumo de 1.000 mensagens simultâneas oriundas de 1.000 conexões diferentes, houve atraso no consumo, porém não houve interrupção do funcionamento do *gateway*. Mesmo após um limite ser estabelecido, uma solução com maior demanda poderia fazer uso da arquitetura simplesmente substituindo o *hardware* do *gateway* por um de maior capacidade de processamento e com isso a capacidade poderia ser aumentada.

O uso dos protocolos de comunicação AMQP e MQTT em conjunto tornou a solução resiliente e garantiu estabilidade à mesma, tornando possível o funcionamento da solução mesmo com um *hardware* limitado operando como *gateway*, demonstrando a eficiência da arquitetura projetada, que foi o objetivo deste trabalho.

5.1 Trabalhos futuros

Como sugestão de trabalhos futuros, pode-se listar:

a) Tornar o aplicativo para dispositivos móveis mais modular a fim de suportar outros tipos de soluções sem a necessidade de alterações de código, onde o conceito de queimador e válvula poderia ser generalizado e o cadastro destas descrições poderia vir do painel de controle, junto aos ícones e demais características necessárias para representar os dispositivos no aplicativo;
b) Realizar mais testes de desempenho visando otimizar o consumo das mensagens lidas dos *brokers*, considerando a possibilidade de paralelizar algumas partes do processo;
c) Na solução implementada como prova de conceito, substituir a forma de detecção de funcionamento dos queimadores, onde as chaves fim de curso poderiam ser substituídas por termômetros, a fim de obter maior precisão na leitura dos estados dos mesmos;
d) Em casos onde a solução implementada como prova de conceito viesse a ser utilizada pelo público em geral, tornar-se-ia importante criar uma página *"wizard"* de configuração inicial, permitindo que o usuário fosse capaz de ingressar em uma rede e inserir os dados necessários para configurar o produto.

REFERÊNCIAS

ADOBE. **PhoneGap Documentation | PhoneGap Docs**. 2018. Disponível em: <http://docs.phonegap.com>. Acesso em: 02 out. 2018.

ALBERTIN, A. L.; ALBERTIN, R. M. d. M. A internet das coisas irá muito além das coisas. **GV-Executivo**, São Paulo, v. 16, p. 12–17, mar./abr. 2017.

ALBORS, J. **Segurança em dispositivos IoT: Ainda temos tempo para vencer a batalha?** 2018. Disponível em: <https://www.welivesecurity.com/br/2018/07/31/seguranca-em-dispositivos-iot>. Acesso em: 16 ago. 2018.

ANATEL. **Portal Institucional - Página Inicial - Agência Nacional de Telecomunicações**. 2019. Disponível em: <http://www.anatel.gov.br/institucional>. Acesso em: 08 jun. 2019.

ANGULAR. **Angular - Architeture overview**. 2018. Disponível em: <https://angular.io/guide/architecture>. Acesso em: 24 set. 2018.

APACHE. **Components - Apache Qpid**. 2018. Disponível em: <https://qpid.apache.org/components/index.html>. Acesso em: 22 set. 2018.

APACHE. **Overview - Apache Qpid**. 2018. Disponível em: <https://qpid.apache.org/overview.html>. Acesso em: 22 set. 2018.

ASHTON, K. That “internet of things” thing. **RFID Journal**, p. 1, 2009.

ASUSTEK, C. I. **Tinker Board | Placas Mãe | ASUS Brasil**. 2018. Disponível em: <https://www.asus.com/br/Motherboards/Tinker-Board/specifications>. Acesso em: 12 set. 2018.

BIEGELMEYER, A. **Desenvolvimento e aplicação de uma casa inteligente**. 2015. Monografia (Engenharia de Controle e Automação) - Universidade de Caxias do Sul. Caxias do Sul, 2015.

BRADLEY, J.; JONES, M.; SAKIMURA, N. **JSON Web Token (JWT)**. Fremont: Internet Engineering Task Force (IETF), 2015.

CHENG, F. **Build Mobile Apps with Ionic 2 and Firebase: Hybrid Mobile App Development**. 1. ed. New Zealand: Apress, 2017.

CLARK, J. **What is the Internet of Things?** 2016. Disponível em: <https://www.ibm.com/blogs/internet-of-things/what-is-the-iot>.

COELHO, P. **Internet das Coisas Introdução Prática (Portuguese Edition)**. Lisboa: FCA, 2017.

CONWAY, A. et al. **AMQP - Advanced Message Queuing Protocol - Protocol Specification version 0-9-1**. [S.l.]: Cisco Systems, 2008.

CORDOVA. **Documentation - Apache Cordova**. 2018. Disponível em: <https://cordova.apache.org/docs/en/latest>. Acesso em: 02 out. 2018.

DUNKA, B.; EMMANUEL, E.; OYERINDE, D. Hybrid mobile application based on ionic framework technologies. **International Journal of Recent Advances in Multidisciplinary Research**, Plateau State, v. 04, p. 3121–3130, dez. 2017.

ECLIPSE. **Eclipse Mosquitto**. 2018. Disponível em: <https://mosquitto.org>. Acesso em: 12 dez. 2018.

ECLIPSE, F. **Eclipse Mosquitto | projects.eclipse.org**. 2018. Disponível em: <https://projects.eclipse.org/projects/technology.mosquitto>. Acesso em: 12 dez. 2018.

ECMA. **Standard ECMA-404 The JSON Data Interchange Syntax**. 2. ed. Geneva: Ecma International, 2017.

EVANS, D. **A Internet das Coisas. Como a próxima evolução da Internet está mudando tudo**. 2011. Disponível em: <https://www.cisco.com/c/dam/global/pt_br/assets/executives/pdf/internet_of_things_iot_ibsg_0411final.pdf>. Acesso em: 16 ago. 2018.

FIELDING, R. **Architectural Styles and the Design of Network-based Software Architectures**. Donald Bren School of Information and Computer Sciences – University of California - Irvine, 2010. Disponível em: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Acesso em: 17 ago. 2018.

FILIPEFLOP. **Sensor de Gas MQ-2 Inflamável e Fumaça**. 2018. Disponível em: <https://www.filipeflop.com/produto/sensor-de-gas-mq-2-inflamavel-e-fumaca>. Acesso em: 06 nov. 2018.

GIRARDI, M. M. **Avaliação de intermediadores de mensagens que suportam o protocolo AMQP**. 2015. Monografia (Ciência da Computação) - Universidade de Caxias do Sul. Caxias do Sul, 2015.

GOURLEY, D. et al. **HTTP: The Definitive Guide**. 1. ed. Sebastopol: O'Reilly Media, 2002.

HEATH, N. **What is the Raspberry Pi 3? Everything you need to know about the tiny, low-cost computer**. 2017. Disponível em: <https://www.zdnet.com/article/what-is-the-raspberry-pi-3-everything-you-need-to-know-about-the-tiny-low-cost-computer/>. Acesso em: 15 ago. 2018.

IONIC. **Welcome to Ionic - Ionic Framework**. 2018. Disponível em: <https://ionicframework.com/docs/v1/guide/preface.html>. Acesso em: 28 set. 2018.

IPV6.BR. **ZigBee usa agora 6loWPAN! Sua próxima lâmpada terá IPv6?** 2013. Disponível em: <http://ipv6.br/post/zigbee-usa-agora-6lowpan-sua-proxima-lampada-tera-ipv6>. Acesso em: 22 set. 2018.

J., A.; K, T. B. **Internet of things (IoT) : technologies, applications, challenges and solutions**. 1. ed. Boca Raton: CRC Press;Taylor & Francis, 2018. (IoT).

JAFFEY, T. **MQTT and CoAP, IoT Protocols**. 2014. Disponível em: <https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php>. Acesso em: 22 set. 2018.

LOPEZ, M. **Tesla e carros elétricos: se essa é a nova tendência, por que Musk não entrega os resultados prometidos?** 2018. Disponível em: <https://www.infomoney.com.br/blogs/investimentos/investimentos-internacionais/post/7320959/tesla-carros-eletricos-essa-nova-tendencia-por-que-musk-nao>. Acesso em: 08 jun. 2019.

MAGRANI, E. **A Internet das Coisas**. 1. ed. Rio de Janeiro, RJ: FGV Editora, 2018.

MCKAY, J. **MQTT ON TESSEL**. Tessel Project, 2014. Disponível em: <https://tessel.io/blog/98339010407/mqtt-on-tessel>. Acesso em: 15 set. 2018.

MCTIC. **MCTIC**. 2019. Disponível em: <http://www.mctic.gov.br>. Acesso em: 08 jun. 2019.

METZGAR, D. **.Net Core in Action**. Shelter Island: Manning Publications, 2018.

MICHAEL, Y. **Conhecendo o MQTT**. IBM developerWorks, 2017. Disponível em: <https://www.ibm.com/developerworks/br/library/iot-mqtt-why-good-for-iot/index.html>. Acesso em: 15 set. 2018.

MICROSOFT. **About .NET Core**. 2018. Disponível em: <https://docs.microsoft.com/en-us/dotnet/core/about>. Acesso em: 27 out. 2018.

MICROSOFT. **C# 6.0 draft language specification**. 2018. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification>. Acesso em: 27 out. 2018.

MICROSOFT. **Introduction to ASP.NET Core SignalR | Microsoft Docs**. 2018. Disponível em: <https://docs.microsoft.com/pt-br/aspnet/core/signalr/introduction?view=aspnetcore-2.2>. Acesso em: 09 dez. 2018.

MONGODB. **MongoDB Documentation**. 2018. Disponível em: <https://docs.mongodb.com>. Acesso em: 09 dez. 2018.

OASIS, S. **OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0**. [S.l.]: OASIS Standard, 2012.

OASIS, S. **OASIS MQTT Version 3.1.1 Plus Errata 01**. [S.l.]: OASIS Standard, 2015.

OLSSON, J. **6LoWPAN demystified**. Dallas: Texas Instruments, 2014.

OSAKO, H.; GIMENEZ, J. C. Internet das coisas: o desafio de estabelecer um padrão único de protocolo. **IV SRST – SEMINÁRIO DE REDES E SISTEMAS DE TELECOMUNICAÇÕES**, p. 1–7, ago. 2016.

PALANISAMY, M. **NodeMCU ESP8266 with Arduino IDE**. 2016. Disponível em: <http://mohanp.com/nodemcu-esp8266-with-adruino-ide/>. Acesso em: 09 set. 2018.

PELLENZ, A. **Sistema de automação residencial**. 2017. Relatório de estágio (Tecnólogo em Automação Industrial) - Universidade de Caxias do Sul. Caxias do Sul, 2017.

PEREIRA, L. F. **C.E.R.B.E.R.U.S. Robô Bombeiro**. 2016. Monografia (Engenharia de Computação) - Universidade de São Paulo: Escola de Engenharia de São Carlos. São Paulo, 2016.

PILOTTI, J. S. **Sistema de automação residencial: Acessibilidade no controle doméstico**. 2014. Monografia (Bacharel em Tecnologias Digitais) - Universidade de Caxias do Sul. Caxias do Sul, 2014.

PINE, M. I. **ROCK64 - PINE64**. 2018. Disponível em: <https://www.pine64.org/?page_id=7147>. Acesso em: 12 set. 2018.

PIPER, A. **Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP**. VMware Inc., 2013. Disponível em: <https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html>. Acesso em: 16 set. 2018.

PIVOTAL. **TLS Support - RabbitMQ**. 2018. Disponível em: <https://www.rabbitmq.com/ssl.html>. Acesso em: 05 out. 2018.

RABBITMQ, C. **AMQP 0-9-1 Model Explained**. 2018. Disponível em: <https://www.rabbitmq.com/tutorials/amqp-concepts.html>. Acesso em: 20 set. 2018.

RABBITMQ, D. **Clients & Developer Tools**. 2018. Disponível em: <https://www.rabbitmq.com/devtools.html>. Acesso em: 22 set. 2018.

RABBITMQ, S. **Compatibility and Conformance**. 2018. Disponível em: <https://www.rabbitmq.com/specification.html>. Acesso em: 20 set. 2018.

REIS, F. d. **Introdução aos Microcontroladores**. 2015. Disponível em: <http://www.bosontreinamentos.com.br/eletronica/eletronica-geral/introducao-aos-microcontroladores>. Acesso em: 09 set. 2018.

RICHARDSON, A. **RabbitMQ in Action: Distributed messaging for everyone**. Shelter Island: Manning Publications, 2012.

ROY, G. M. **RabbitMQ in Depth**. Shelter Island: Manning Publications, 2017.

SALEIRO, M.; EY, E. **ZigBee uma abordagem prática**. 2018. Disponível em: <https://lusorobotica.com/ficheiros/Introducao_ao_Zigbee_-_por_msaleiro.pdf>. Acesso em: 25 out. 2018.

SAMI, S. **RabbitMQ, AMQP, MQTT Rest of the world**. 2017. Disponível em: <https://medium.com/@sakibsami/rabbitmq-amqp-mqtt-rest-of-the-world-74433c5ff8c7>. Acesso em: 10 nov. 2018.

SANTOS, J. S. **Detector de vazamento de gás com aviso por SMS**. 2012. Monografia (Engenharia de Computação) - Centro Universitário de Brasília: Faculdade de tecnologia e ciências sociais aplicadas. Brasília, 2012.

SAUDATE, A. **REST: Construa API's inteligentes de maneira simples**. São Paulo: Casa do Codigo, 2014.

SERRANO, T. M.; NUNEZ, R. **AMQP - Protocolo de Comunicação para IoT**. Embarcados, 2018. Disponível em: <https://www.embarcados.com.br/amqp-protocolo-de-comunicacao-para-iot>. Acesso em: 20 set. 2018.

SHARMA, V.; DAVE, M. Sql and nosql databases. **International Journal of Advanced Research in Computer Science and Software Engineering**, Jaunpur, p. 20–27, ago. 2012.

SHENZHEN, D. S. C. L. **PIR Sensor Module instructions**. 2018. Disponível em: <http://f03.s.alicdn.com/kf/HTB1GSxSGXXXXXbbaXXX.PRXFXXX9.pdf>. Acesso em: 11 nov. 2018.

SINDIGÁS. **Gás LP no Brasil - Energia para o desenvolvimento e o bem-estar social**. 1. ed. Rio de Janeiro: Sindicato Nacional das Empresas Distribuidoras de Gás Liquefeito de Petróleo, 2012. v. 6.

SINDIGÁS. **Panorama do setor de GLP em movimento**. 26. ed. Rio de Janeiro: Sindicato Nacional das Empresas Distribuidoras de Gás Liquefeito de Petróleo, 2017. v. 1.

SOLACE. **AMQP - Solace Developer Portal**. 2018. Disponível em: <https://dev.solace.com/tech/amqp>. Acesso em: 27 set. 2018.

SOLACE. **Solace PubSub+ Software Message Broker**. 2018. Disponível em: <https://docs.solace.com/Solace-SW-Broker-Set-Up/Setting-Up-SW-Brokers.htm>. Acesso em: 27 set. 2018.

STALLINGS, W. **Arquitetura e organização de computadores**. 8ª. ed. São Paulo: Pearson Prentice Hall, 2010.

SYSTEMS, E. **ESP8266EX Datasheet**. 2018. Disponível em: <https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf>. Acesso em: 03 nov. 2018.

TANENBAUM, A. S. **ORGANIZAÇAO ESTRUTURADA DE COMPUTADORES**. São Paulo: Pearson Education, 2007.

TEAM, N. **NODE MCU DEVKIT V1.0**. 2015. Disponível em: <https://github.com/nodemcu/nodemcu-devkit-v1.0/blob/master/NODEMCU_DEVKIT_V1.0.PDF>. Acesso em: 09 set. 2018.

TESLA. **Model S | Tesla**. 2019. Disponível em: <https://www.tesla.com/models>. Acesso em: 12 jun. 2019.

TSCHOFENIG, H. et al. **Architectural Considerations in Smart Object Networking**. Fremont: IETF - Internet Engineering Task Force, 2015.

VASTERS, C. **From MQTT to AMQP and back**. 2017. Disponível em: <http://vasters.com/blog/From-MQTT-to-AMQP-and-back>. Acesso em: 12 out. 2018.

WAHER, P. **Learning Internet of Things: Explore and learn about Internet of Things with the help of engaging and enlightening tutorials designed for Raspberry Pi**. Birmingham: Packt Publishing, 2015.

WILKEN, J. **Angular in Action**. 1. ed. Shelter Island: Manning Publications, 2018.

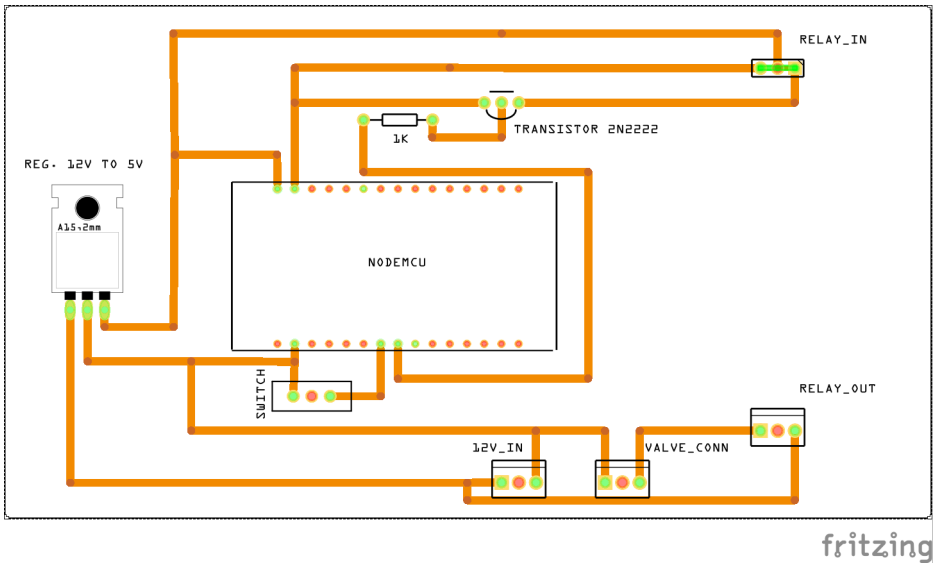
XUNLONG, S. C. L. **Orange PI - Orangepi**. 2018. Disponível em: <http://www.orangepi.org/>. Acesso em: 11 set. 2018.

ZIEMANN, V. **A hands-on course in sensors using the Arduino and Raspberry Pi**. Boca Raton: CRC Press, 2018. (Sensors series).

APÊNDICE A – CIRCUITOS

Nesse apêndice podem ser vistos os circuitos construídos para a solução de validação, fazendo uso do dispositivo NodeMCU citado na Seção 2.2 bem como os demais dispositivos e sensores necessários para o funcionamento da mesma.

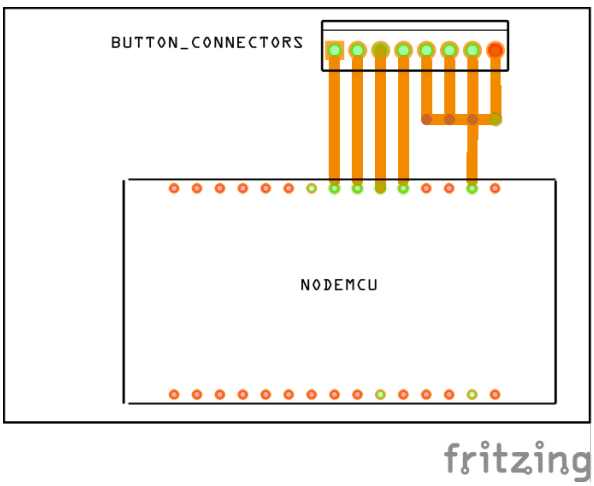
Figura 46 – PCB sistema válvula



Fonte: Elaborado pelo autor

A Figura 46 ilustra o sistema de válvula da solução de validação, sendo esta responsável pelo controle da válvula de gás.

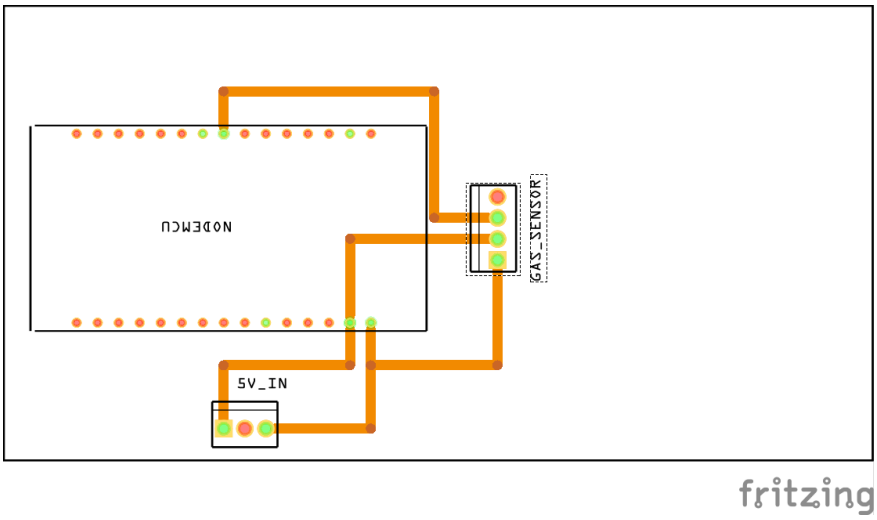
Figura 47 – PCB sistema de botões queimadores



Fonte: Elaborado pelo autor

A Figura 47 ilustra o sistema responsável pela leitura da posição dos botões dos queimadores do fogão.

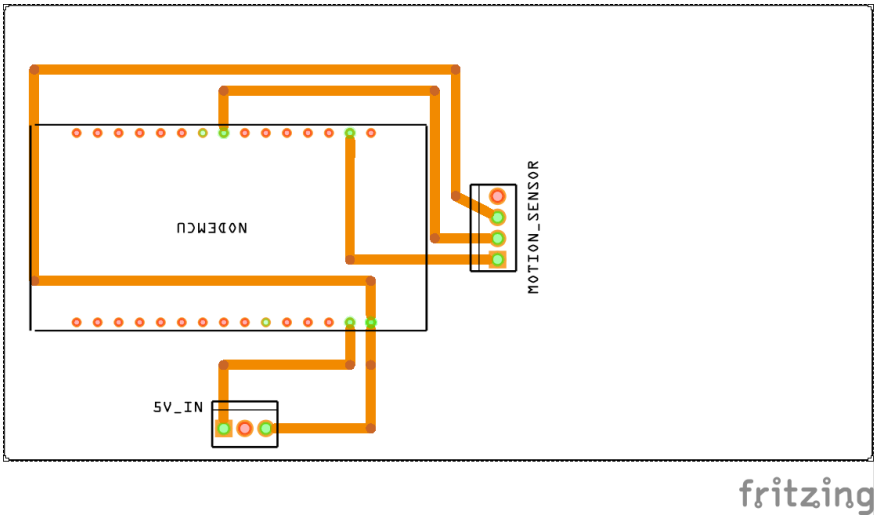
Figura 48 – PCB sistema de sensor de gás



Fonte: Elaborado pelo autor

A Figura 48 representa o sistema responsável pela leitura do sensor de gás.

Figura 49 – PCB sistema detecção de movimento



Fonte: Elaborado pelo autor

A Figura 49 representa o sistema responsável pela leitura do sensor de movimento.