UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Monique Aguiar Garcia

Desenvolvimento de Firmware para Testes de Produção de Produtos IoT

Florianópolis
2022

Monique Aguiar Garcia

Desenvolvimento de Firmware para Testes de Produção de Produtos IoT

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Carlos Barros Montez, Dr.
Supervisor: Sandro Alberton Kirchner, Eng.

Florianópolis
2022

Monique Aguiar Garcia

Desenvolvimento de Firmware para Testes de Produção de Produtos IoT

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 22 de março de 2022.

Prof. Hector Bessa Silveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Carlos Barros Montez, Dr.
Orientador
UFSC/CTC/DAS

Sandro Alberton Kirchner, Eng.
Supervisor
Khomp Indústria e Comércio Ltda.

Prof. Odilson Tadeu Valle, Dr.
Avaliador
IFSC

Prof. Eduardo Camponogara, Dr.
Presidente da Banca
UFSC/CTC/DAS

Este trabalho é dedicado aos meus colegas de
graduação que me acompanharam nesta trajetória e
deixaram suas marcas na minha história.

CLÁUSULA REFERENTE À NÃO DIVULGAÇÃO DE INFORMAÇÃO SIGILOSA

Este documento continha informações sobre processos internos da Khomp Indústria e Comércio Ltda., não sendo permitida a publicação da versão original. Apenas à banca avaliadora é permitido o acesso integral a este documento. Assim sendo, o presente relatório apresenta apenas informações gerais sobre os processos internos da empresa, não entrando em detalhes de cunho confidencial.

Florianópolis, março de 2022.

AGRADECIMENTOS

Aos meus pais, por todo o apoio oferecido na minha educação que com certeza foi de extrema importância para que eu chegasse até aqui.

Ao meu orientador e professor, Carlos Montez, pela disposição para que este PFC fosse finalizado e por todos os ensinamentos passados no decorrer da graduação.

A William, namorado, companheiro e amigo, por todo o suporte e compreensão oferecidos durante este período de PFC. Obrigada por me reapresentar ao mundo do desenvolvimento de software, fazendo com que eu desse uma nova chance à área e me descobrisse apaixonada.

Aos meus colegas de trabalho e graduação, Adriano e Thiago, por todo o suporte e disposição para me ajudar no desenvolvimento deste PFC sanando todas as minhas inúmeras dúvidas.

Á Khomp por me permitir desenvolver este projeto me garantindo total autonomia criativa para o desenvolvimento.

Ao meu gestor, Sandro, pelos constantes *feedbacks* e direcionamentos.

E por fim, à comunidade UFSC - professores, servidores, entidades e colegas. Comunidade esta que me acolheu por todos estes anos e me proporcionou um enorme crescimento técnico, profissional e pessoal. Devo à UFSC boa parte da pessoa cidadã que sou hoje.

RESUMO

Este projeto propõe o desenvolvimento de um *firmware* para automatização de um processo de teste de produtos da linha de *gateways* de IoT da empresa Khomp. Os testes de produto na empresa são denominados T2 e buscam validar as funcionalidades de cada módulo do produto quando este já está completamente montado, sendo o último teste realizado antes da expedição ao cliente final. Cada elemento testado demanda o uso de diferentes abordagens e tecnologias. O objetivo que se busca com a implementação deste *firmware* é reduzir significativamente o tempo empregado para testes de produto da linha ITG substituindo um longo processo manual por outro automático e mais simples, também mitigando possíveis falhas humanas. Para isso, são utilizadas técnicas de programação concorrente, testes de *hardware* e *software* e o desenvolvimento de uma interface web para monitoramento dos testes e direcionamento para as etapas guiadas. O *firmware* é implementado utilizando a linguagem Golang e a interface web é desenvolvida utilizando Svelte. Três metodologias ágeis pautam o desenvolvimento deste trabalho: SCRUM para gerenciamento de projeto, FDD para definir o desenvolvimento e TDD para garantir a qualidade do software. Para validação dos resultados, são apresentadas análises comparando o tempo de duração de teste entre os processos e também os *feedbacks* dos usuários.

Palavras-chave: testes de produto, automação de processos, software embarcado, *gateway*.

ABSTRACT

This project proposes the development of a firmware for automating a product testing process from the IoT gateways line from Khomp company. The company's product tests are called T2 and seek to validate the functionality of each product module when it is already fully assembled, the last test being carried out before shipping to the final customer. Each element tested requires the use of different approaches and technologies. The objective pursued with the implementation of this firmware is to significantly reduce the time used for product testing of the ITG line, replacing a long manual process with an automatic and simpler one, also mitigating possible human errors. For this, concurrent programming techniques are used, as well as hardware and software tests and the development of a web interface for monitoring the tests and directing the user to the guided steps. The firmware is implemented using the Golang language and the web interface is developed using Svelte. Three agile methodologies guide the development of this work: SCRUM for project management, FDD to define the development and TDD to guarantee the software quality. To validate the results, analyzes are presented comparing the test duration time between the processes and also the users' feedbacks.

Keywords: product test, process automation, embedded software, gateway.

LISTA DE FIGURAS

Figura 1 – Arquitetura IoT de 3 camadas proposta pelo IEEE. 17
Figura 2 – Definição de IoT (ITU). 17
Figura 3 – Áreas de atuação IoT e principais interessados na implementação da tecnologia. 19
Figura 4 – Sede da Khomp em Florianópolis, SC. 20
Figura 5 – Ciclo de desenvolvimento do *software*. 24
Figura 6 – Aplicação de um ITG 200 *Indoor* com sensores e comunicação com a nuvem. 27
Figura 7 – ITG 200 *Indoor*. 28
Figura 8 – ITG 200 *Outdoor* LoRa. 28
Figura 9 – Vista traseira de um *gateway Indoor* com entradas e placa de *display* e botões em evidência. 29
Figura 10 – Vista superior de um *gateway* Zigbee com modem, módulo de comunicação, elemento seguro e placa ITG-Core em evidência. 30
Figura 11 – Panorama geral do processo de testes de produto da Khomp. 31
Figura 12 – Processo de reparação de itens não conformes. 32
Figura 13 – Entrada e saída do teste T0. 33
Figura 14 – Entrada e saída do teste T1. 34
Figura 15 – Entrada e saída do processo de montagem de produto. 34
Figura 16 – Mapeamento do processo de teste de produto e embalagem. 35
Figura 17 – Teste funcional de um sistema. 38
Figura 18 – Esquema de um teste estrutural evidenciando a implementação do sistema. 38
Figura 19 – Esquema simples de uma árvore de falhas. 40
Figura 20 – Comparativo entre concorrência e paralelismo. 41
Figura 21 – Ciclo de desenvolvimento SCRUM. 44
Figura 22 – Ciclo de desenvolvimento FDD. 45
Figura 23 – Ciclo de desenvolvimento TDD. 47
Figura 24 – Topologia de uma rede LoRaWAN. 49
Figura 25 – Topologias possíveis para uma rede Zigbee. 50
Figura 26 – Esquema simplificado do funcionamento da comunicação via MQTT. 52
Figura 27 – Esquema simplificado do funcionamento dos tópicos MQTT. 53
Figura 28 – Representação de rede móvel infraestruturada e ad hoc. 54
Figura 29 – Requisição e resposta HTTP entre cliente e servidor. 57
Figura 30 – Captura de tela do Gmail, um exemplo de aplicação SPA. 58
Figura 31 – Representação em árvore de um trecho de documento HTML. . . . 59
Figura 32 – Tempo de execução de três programas em Go, Python e C++. . . . 61

Figura 33 – Comunicação entre go rotinas através de go *channels*. 63
Figura 34 – Comparação entre Svelte e outros *frameworks* em relação à performance. 65
Figura 35 – Comparação entre Svelte e outros *frameworks* em relação ao número de linhas de código. 66
Figura 36 – Comparação entre Fiber e outros *frameworks* em relação à performance. 67
Figura 37 – Comparação entre Fiber e outros *frameworks* em relação à concorrência. 67
Figura 38 – Ciclo de vida de uma tarefa no Redmine - GitLab. 69
Figura 39 – Combinação entre SCRUM, FDD e TDD utilizada no desenvolvimento do projeto. 71
Figura 40 – Casos de uso relacionados ao testes de entrada e saída. 71
Figura 41 – Casos de uso relacionados ao *download* do relatório de testes. . . . 72
Figura 42 – Arquitetura do *firmware* de testes. 73
Figura 43 – Arquitetura interna da comunicação LoRa e Zigbee com o Broker MQTT do ITG. 75
Figura 44 – Diagrama de atividades generalizado do testes de módulos de comunicação. 76
Figura 45 – Diagrama de atividades para validação do *slot* Zigbee. 77
Figura 46 – Diagrama de atividades para validação do *slot* LoRa. 77
Figura 47 – Diagrama de atividades do teste de conectividade. 79
Figura 48 – Diagrama sequencial demonstrando a comunicação via RPC com o Go Connect. 80
Figura 49 – Diagrama de atividades do teste de leitura do sensor de contato via RJ11. 81
Figura 50 – Diagrama de atividades do teste de leitura de temperatura *1wire* via RJ11. 82
Figura 51 – Diagrama de atividades do teste display. 83
Figura 52 – Diagrama de atividades do teste de alarme. 85
Figura 53 – Diagrama de atividades do teste de botões. 86
Figura 54 – Diagrama de atividades do teste de leitura e gravação do elemento seguro. 87
Figura 55 – Captura de tela da interface web com o botão ”Gerar Relatório“ desabilitado. 88
Figura 56 – Captura de tela da interface web com o botão ”Gerar Relatório“ habilitado. 89
Figura 57 – *Pop-up* em evidência para *download* do relatório de teste. 89
Figura 58 – Confirmação de *download* no caso de teste não concluído. 90

Figura 59 – Relatório com os resultados. Caso com todos os testes aprovados. . 91
Figura 60 – Tela de atualização de *firmware*. 91
Figura 61 – Esquema geral de um *build* feito pelo kharma. 92
Figura 62 – Roteiro para compilar o código fonte do *firmware* de testes. 93
Figura 63 – Tempos de execução do teste T2 com o uso do *firmware*. 94

LISTA DE TABELAS

Tabela 1 – Comparativos entre as linguagens de programação Golang, Python e C++. 61

LISTA DE ABREVIATURAS E SIGLAS

ABINC	Associação Brasileira de Internet das Coisas
CI/CD	*Continuous Integration/Continuous Delivery*
CTI	*Computer Telephony Integration*
DOM	*Document Object Model*
ETSI	*European Telecommunications Standards Institute*
FDD	*Feature Driven Development*
GPIO	*General Purpose Input/output*
HTTP	*Hypertext Transfer Protocol*
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
IETF	*Internet Engineering Task Force*
IoT	*Internet of Things*
ITG	*IoT Telemetry Gateway*
ITU	*International Telecommunication Union*
LPWAN	*Low Power Wide Area Network*
MQTT	*Message Queue Telemetry Transport*
NIST	*National Institute os Standards ans Technology*
OAS	*Open Automation Software*
OASIS	*Organization for the Advancement of Structured Information Standards*
PAN	*Personal Area Network*
PoE	*Power over Ethernet*
QA	*Quality Assurance*
RAN	*Radio Access Network*
RPC	*Remote Procedure Call*
SPA	*Single Page Web Application*
SWIG	*Simplified Wrapper and Interface Generator*
TDD	*Test-Driven Development*
W3C	*World Wide Web Consortium*

SUMÁRIO

1 INTRODUÇÃO . . . 16
1.1 UMA DEFINIÇÃO PARA IOT . . . 16
1.2 O MERCADO DE IOT . . . 18
1.3 LOCAL DE ESTÁGIO: KHOMP . . . 19
1.3.1 O setor de IoT . . . 21
1.3.1.1 Equipe de desenvolvimento do gateway . . . 21
1.4 MOTIVAÇÃO E DESCRIÇÃO DO PROBLEMA . . . 21
1.4.1 Proposta de solução e requisitos . . . 22
1.5 METODOLOGIAS UTILIZADAS . . . 23
1.6 PLANEJAMENTO . . . 24
1.7 ESTRUTURA DO DOCUMENTO . . . 25
2 PRODUTOS . . . 26
2.1 *GATEWAY* . . . 26
2.1.1 *Gateway* IoT . . . 26
2.2 A LINHA ITG . . . 27
2.2.1 Rede estendida *EveryNet* e elemento de segurança . . . 30
3 PROCESSOS . . . 31
3.1 TESTES . . . 31
3.1.1 Controle de produto não conforme . . . 32
3.2 TESTE T0 - TESTE DA MONTADORA . . . 33
3.3 TESTE T1 - TESTE DE PLACAS . . . 33
3.3.1 Montagem de produto . . . 34
3.4 TESTE T2 - TESTE E EMBALAGEM DE PRODUTO . . . 34
4 FUNDAMENTAÇÃO TEÓRICA . . . 36
4.1 TESTES DE *HARDWARE* . . . 36
4.1.1 Definições utilizadas . . . 36
4.1.2 Técnicas de Teste . . . 37
4.1.2.1 Teste Funcional . . . 37
4.1.2.2 Teste Estrutural . . . 38
4.1.3 Diagnóstico de falhas . . . 39
4.2 PROGRAMAÇÃO PARALELA E CONCORRENTE . . . 40
4.3 METODOLOGIAS ÁGEIS . . . 41
4.3.1 SCRUM . . . 43
4.3.2 Desenvolvimento Orientado a Funcionalidades . . . 44
4.3.3 Desenvolvimento Orientado a Testes . . . 46
4.4 CONCEITOS ESTUDADOS POR MÓDULOS . . . 48
4.4.1 Testes dos módulos de comunicação *wireless* . . . 48

4.4.1.1 LoRaWAN 48
4.4.1.2 Zigbee 49
4.4.1.3 MQTT 51
4.4.2 Teste de conectividade 53
4.4.2.1 Redes móveis 53
4.4.2.2 Rede Ethernet 54
4.4.2.3 Chamada de Procedimento Remoto 55
4.5 INTERFACE *WEB* 55
4.5.1 HTTP 55
4.5.2 *Sigle Page Web Application* 57
4.5.3 *Frameworks* reativos 58
5 FERRAMENTAS E TECNOLOGIAS UTILIZADAS 60
5.1 LINUX EMBARCADO 60
5.2 DEFINIÇÃO DA *STACK* 60
5.2.1 Golang x Python x C++ 60
5.2.2 Golang 62
5.2.2.1 Concorrência em Golang 62
5.2.3 Python 64
5.2.3.1 SWIG 64
5.2.4 *Framework* Svelte para *front-end* 65
5.2.5 Go Fiber 66
5.3 KHARMA 68
5.4 FERRAMENTAS CORPORATIVAS 68
5.4.1 GitLab 68
5.4.1.1 CI/CD 68
5.4.2 Redmine 69
6 PROJETO 70
6.1 METODOLOGIA DE PROJETO 70
6.2 CASOS DE USO 71
6.3 ARQUITETURA DO *FIRMWARE* 72
6.4 IDENTIFICAÇÃO DE DISPOSITIVO 73
6.5 TESTES 74
6.5.1 Módulos de comunicação sem fio 74
6.5.1.1 Mapeamento 75
6.5.1.2 Estrutura do teste 75
6.5.2 Testes de conectividade 77
6.5.2.1 Mapeamento 78
6.5.2.2 Estrutura do teste 78
6.5.3 RJ11 80

6.5.3.1 Mapeamento 80
6.5.3.2 Estrutura do teste 81
6.5.4 Display 82
6.5.4.1 Mapeamento 82
6.5.4.2 Estrutura do teste 83
6.5.5 *Buzzer* 84
6.5.5.1 Mapeamento 84
6.5.5.2 Estrutura do teste 84
6.5.6 Botões 85
6.5.6.1 Mapeamento 85
6.5.6.2 Estrutura do teste 85
6.5.7 Elemento de segurança 86
6.5.7.1 Mapeamento 86
6.5.7.2 Estrutura do teste 86
6.6 INTERFACE WEB 87
6.7 KHARMA 91
7 RESULTADOS 94
8 PROPOSTAS DE MELHORIAS 96
8.1 AUTOCADASTRO *CLOUD* EVERYNET 96
8.2 GERAR DIAGNÓSTICOS DE FALHAS 96
8.3 GERAR RELATÓRIO AUTOMATICAMENTE COM *UPLOAD* PARA SERVIDOR INTERNO KHOMP 96
9 CONCLUSÃO 97
REFERÊNCIAS 99
APÊNDICE A – TESTES UNITÁRIOS 104

1 INTRODUÇÃO

Este capítulo trata sobre o conceito de IoT e o mercado na área. Também é feita uma breve contextualização do problema abordado neste trabalho e a motivação para o mesmo. Ao final, é apresentada a estrutura do documento de maneira sucinta.

1.1 UMA DEFINIÇÃO PARA IOT

A Internet é um rede global de computadores interconectados que executa sobre protocolos padrão de internet (TCP/IP). É uma rede de redes que consiste em milhões de redes privadas, públicas, acadêmicas, empresariais e governamentais ligadas por uma diversidade de tecnologias (MADAKAM; RAMASWAMY; TRIPATHI, 2015).

De acordo com Madakam, Ramaswamy e Tripathi em sua revisão literária sobre *Internet of Things* (IoT), o termo ”coisas“ pode se referir a qualquer objeto ou pessoa distinguível pelo mundo real. Objetos incluem não apenas dispositivos mas também itens como comida, roupas e móveis, monumentos e obras de arte e toda a miscelânea de comércio, cultura e sofisticação.

A partir destes dois conceitos, constrói-se uma boa base para a definição geral de Internet das Coisas. O termo foi cunhado em 1999 por Kevin Ashton, especialista em inovação digital. À época, o objetivo era encontrar um nome para descrever um sistema onde internet e mundo físico estivessem conectados via sensores.

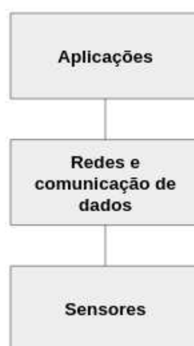
O conceito de IoT, entretanto, não é um consenso. Cada organização lida com o tema de acordo com o próprio contexto e interesse. Em um artigo de (MINERVA; BIRU; ROTONDI, 2015) publicado pelo Instituto de Engenheiros Eletricistas e Eletrônicos (IEEE), é feito um levantamento de definições de IoT fornecidas por organizações que são referência no assunto:

- IEEE: Uma rede de itens - cada um embarcado com sensores - que estão conectados à internet (Figura 1).

- *European Telecommunications Standards Institute* (ETSI): não menciona uma definição diretamente para IoT, mas sim um conceito próximo de ”comunicação M2M - máquina para máquina“ que seria: ”comunicação M2M é a comunicação entre duas ou mais entidades que não necessariamente demandam qualquer intervenção direta humana.“

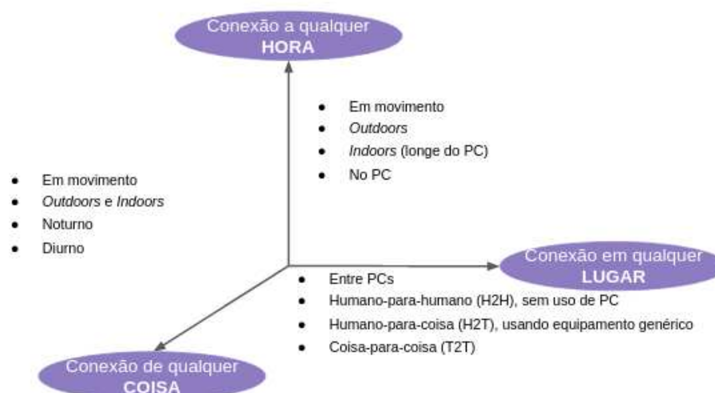
- *International Telecommunication Union* (ITU): uma rede que é “disponível em qualquer lugar, a qualquer hora, por qualquer coisa e por qualquer pessoa.” Nesse contexto, os produtos de consumo podem ser rastreados usando minúsculos rádio transmissores ou *hiperlinks* e sensores incorporados. A Figura 2 ilustra a arquitetura proposta.

Figura 1 – Arquitetura IoT de 3 camadas proposta pelo IEEE.



Fonte – IEEE (2022). Tradução livre.

Figura 2 – Definição de IoT (ITU).



Fonte – IEEE (2022). Tradução livre.

- *Internet Engineering Task Force* (IETF): IoT irá conectar objetos ao nosso redor (eletrônicos, elétricos, não elétricos) para fornecer comunicação perfeita e serviços contextuais fornecidos por eles. O desenvolvimento de *tags* RFID, sensores, atuadores, telefones móveis que interagem e cooperam uns com os outros para tornar o serviço melhor e acessível a qualquer hora e de qualquer lugar faz com que seja possível ”materializar“ o IoT.

- *National Institute os Standards ans Technology* (NIST): utilizando uma abordagem voltada a sistemas cyberfísicos, a entidade define IoT como o próximo grande avanço para o uso da internet. Esta tecnologia permite o desenvolvimento de complexos sistemas de controle capazes de fazer um robô auxiliar um cachorro ou pessoa em uma operação de busca e resgate, ou ajudar profissionais

de saúde a avaliar a recuperação de pacientes após uma alta hospitalar.

- *Organization for the Advancement of Structured Information Standards* (OASIS): um sistema onde a internet está conectada ao mundo físico via sensores onipresentes.

- *World Wide Web Consortium* (W3C): a ”rede das coisas“ é, essencialmente, sobre o papel das tecnologias de rede como facilitadoras do desenvolvimento de aplicações e serviços para IoT, ou seja, sistemas cyberfísicos e suas representações virtuais.

Levando em conta todas estas definições dadas por organizações importantes nas áreas da engenharia e da computação, pode-se resumir IoT na seguinte sentença que será utilizada como referência neste documento:

“Uma rede aberta e abrangente de objetos inteligentes que têm a capacidade de auto-organizar, compartilhar informações, dados e recursos, reagindo e agindo diante de situações e mudanças no ambiente” (MADAKAM; RAMASWAMY; TRIPATHI, 2015).

1.2 O MERCADO DE IOT

Um estudo sobre o IoT na América Latina (COLÓN; NAVAJAS; TERRY, 2019) estimou que até 2023 existirão 416 milhões de dispositivos IoT no Brasil e cerca de 64 bilhões no mundo até 2025. Além disso, segundo a Associação Brasileira de Internet das Coisas (ABINC), 76% das empresas brasileiras acreditam que IoT será importante nos próximos anos.

Praticamente qualquer indústria tem potencial para utilizar IoT de alguma maneira, e muitas já o fizeram, como mostrado na Figura 3. No estudo ”IoT na América Latina e Caribe“, Colón, Navajas e Terry trazem exemplos e dados de diferentes setores que implementaram a tecnologia:

- Manufatura: em 2016, 35% dos fabricantes dos Estados Unidos utilizavam sensores inteligentes;

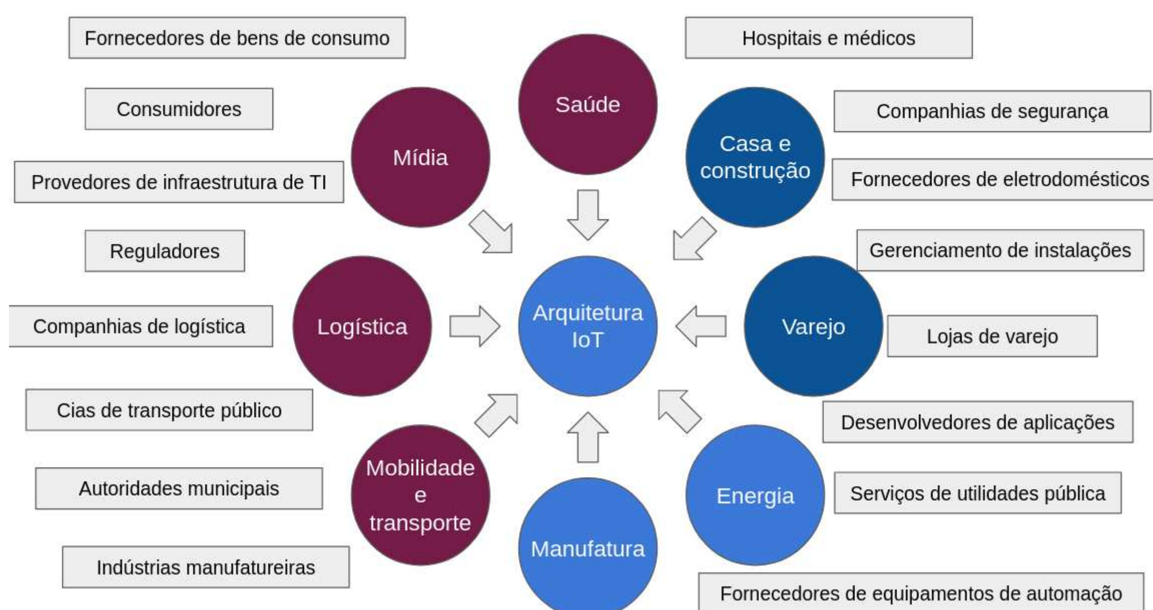
- Logística: sensores para rastreamento colocados em encomendas e contêineres reduzem custos associados a mercadorias perdidas ou danificadas, além de aumentar a velocidade do processamento de pedidos;

- *Smart home*: estima-se que até 2030 a maioria dos dispositivos residenciais estarão conectados à internet;

- Utilidades: é previsto que o uso de IoT para este setor excederá 15 bilhões de dólares até 2024;

- Transporte: o mercado de carros conectados está projetado para atingir 219 bilhões de dólares até 2025;
- Saúde: espera-se um crescimento de US$41 bilhões até 2026. Os dispositivos de assistência médica conectados podem coletar dados, automatizar processos e muito mais;
- Agricultura: este mercado IoT atingirá uma receita de 28 bilhões de dólares até 2023. Os dispositivos de IoT são usados para rastrear a temperatura do solo, níveis de acidez e outras métricas que auxiliam os agricultores no aumento de rendimento das colheitas.

Figura 3 – Áreas de atuação IoT e principais interessados na implementação da tecnologia.



Fonte – Adaptado de IEEE (2022), tradução livre.

1.3 LOCAL DE ESTÁGIO: KHOMP

A Khomp Indústria e Comércio Ltda. é uma empresa de Florianópolis, sediada no bairro Córrego Grande (Figura 4). Fundada em 1996, inicialmente era focada na área de centrais de comutação telefônica pública e privada. Com uma equipe técnica especializada em processamento digital de sinais, rapidamente a empresa passou a atuar no desenvolvimento de projetos personalizados em *hardware* e *software*.

Figura 4 – Sede da Khomp em Florianópolis, SC.

Fonte – (KHOMP, 2021).

Desde então, a empresa apresentou considerável crescimento, tanto em portfólio de produtos quanto em áreas de atuação no mercado. Abaixo são listados alguns dos principais marcos na história da organização (KHOMP, 2021).

- 2001 - a Khomp passou a desenvolver produtos voltados ao mercado de *Computer Telephony Integration* (CTI), junto com os sistemas desenvolvidos pelos clientes Khomp, a linha de placas de circuito impresso fornecidas pela empresa podem ser alocados às mais diversas aplicações que integram computação e telefonia.
- 2009 - a empresa passou a integrar *media gateways* com a mais alta densidade do mercado em seu portfólio, os chamados Kmedias e, posteriormente em 2012, ampliou essa família de produtos com os KMG, *media gateways* de baixa densidade. Nesse mesmo ano, também foi inaugurado o primeiro escritório regional da Khomp, em Buenos Aires, Argentina.
- 2013 - inauguração de um Escritório Regional na Cidade do México, capital mexicana.
- 2014 - ampliação do portfólio de dispositivos IP e lançamento do programa de *workshops* gratuitos "Workoffees Khomp".
- 2018 - lançamento do Khomp Academy, um laboratório de treinamentos presenciais focado na capacitação técnica de parceiros.

Em 2021, completando 25 anos de história e visando fortalecer ainda mais as estratégias nas áreas de telecomunicações e Internet das Coisas, a Khomp ganhou um importante parceiro como sócio: o Grupo Intelbras.

1.3.1 O setor de IoT

A equipe de IoT da Khomp atualmente conta com um time de 20 pessoas distribuídas entre projetistas de *hardware*, analistas de qualidade e testes e desenvolvedores de *firmware*. Essa divisão surgiu em 2017 e é responsável pelos *gateways* de telemetria e também por uma variada gama de sensores inteligentes - os chamados *endpoints* - responsáveis pela coleta e transmissão de informações via protocolo de comunicação MQTT para as mais diversas aplicações.

De forma sucinta, o time de Internet das Coisas da empresa pode ser dividido nas seguintes subáreas:

- Hardware;
- Gateway e Cloud;
- Endpoints LoRa;
- Endpoints Zigbee;
- *Quality Assurance* (QA).

1.3.1.1 Equipe de desenvolvimento do gateway

A subdivisão de *gateway* e *cloud* conta com 4 desenvolvedores responsáveis por todas as tarefas relacionadas aos *gateways* de telemetria da Khomp (linha *IoT Telemetry Gateway* (ITG)) e também aos servidores utilizados para as aplicações de IoT da empresa. Entre as principais responsabilidades do time, pode-se citar o projeto de novos modelos de *gateway*, novas funcionalidades para os produtos já existentes, correção de *bugs* e suporte a clientes.

1.4 MOTIVAÇÃO E DESCRIÇÃO DO PROBLEMA

Testes de produtos são realizados para identificar possíveis erros e corrigí-los antes do envio ao cliente final. Na Khomp, os testes de produto são a última etapa antes da expedição ao comprador, tendo como objetivo realizar uma última aferição do *hardware* do produto já completamente montado.

Apesar de não ser uma tarefa diretamente ligada à equipe de IoT, o teste de produto é uma importante etapa do processo produtivo da Khomp, podendo gerar grandes gargalos. O teste de produto (T2) é, de forma resumida, uma avaliação física do produto, que deve garantir que todos os módulos instalados nele funcionem bem de forma conjunta e que nenhum elemento esteja fora de conformidade (desde módulos de comunicação até botões). O processo de testes de produção, especificamente T2, serão aprofundados no Capítulo 3 deste documento.

No caso da linha ITG, o teste de produto é feito apenas com o auxílio de um *hardware* de testes, sendo todo o processo completamente manual. Tal prática, além de expor o teste a uma alta possibilidade de falha humana, despende bastante tempo: cerca de 10 minutos são gastos para testar um único *gateway* em um procedimento bastante longo. Além disso, o teste também é realizado utilizando o *firmware* original do ITG, um programa que consome bastante recurso computacional, algo desnecessário para este tipo de atividade visto que, para um teste de módulos físicos, não é preciso uma aplicação completa em execução demandando etapas extras de atualização e *reset* do dispositivo.

Dado este contexto, surgiu a ideia de criar um *firmware* de teste mais leve e objetivo que permita a automação do processo de teste de produto, tornando-o mais rápido e confiável.

1.4.1 Proposta de solução e requisitos

A proposta de solução para o problema trata-se da automatização do processo, tornando-o mais rápido e mais confiável ao reduzir a interação do testador que passa a atuar muito mais como um auxiliar durante o teste.

Para eliminar este gargalo, propõe-se que o *firmware* de testes permita a execução completa do teste T2 de um ITG em até 60 segundos, representando uma redução de até 90% no tempo de execução. Além disso, a interação do testador com o *gateway* deve ser minimizada o máximo possível, sendo restrita apenas a casos em que seja realmente necessária alguma ação do testador para validar o teste, como por exemplo testar o funcionamento dos botões.

O *firmware* deverá dar suporte aos seguintes testes de módulos:

- **Modem:** identificar a presença do modem e, caso detectada, avaliar a conexão do mesmo. Além disso, os cartões SIM inseridos no modem também devem ser verificados quanto à presença e configuração;
- **Ethernet:** verificar se a conexão cabeada funciona;
- **Módulos de comunicação sem fio:** o programa deverá ser capaz de identificar se um ITG utiliza o protocolo Zigbee ou LoRa além de averiguar que os módulos estejam funcionais;
- **I/O (*input*/*output*):** verificar o correto funcionamento do *display*, botões e alarmes. Neste caso o *firmware* recebe feedbacks do testador para a realização do teste;
- **Leitura de sensores pelo RJ-11:** os *gateways* da linha ITG contam com duas entradas para conector RJ-11. Para o teste, estarão ligados a estas entradas

um sensor de contato seco e um sensor de temperatura. O *firmware* deverá ser capaz de fazer as leituras destes sensores, garantindo o bom funcionamento das entradas. No caso do contato seco será necessária interação do testador;

- **Elemento de segurança:** verificar existência de elemento de segurança. Além disso, o *firmware* também deverá ser capaz de gravar as informações do elemento seguro ao detectar que o módulo continua com as configurações de fábrica.

Além disso, o projeto também deve conter as seguintes funcionalidades disponíveis para o usuário:

- Haverá o desenvolvimento de uma interface web para acompanhamento do teste e visualização de relatórios. É por essa interface que o testador será instruído, quando necessário, nos momentos em que deverá interagir com o *gateway* para conclusão de todas as etapas.
- A interface web deve informar o tamanho da memória e status da mesma.
- Ao final do teste, o *software* deverá gerar um relatório informando sobre os resultados. Neste relatório também devem constar dados de data e hora de quando o teste foi executado e o número de série do *gateway* testado.
- O *software* deverá permitir o *download* deste relatório em formato legível.
- Após a execução dos testes, se tudo estiver de acordo com o esperado e o dispositivo for aprovado em todos os testes, o *software* oficial deve ser instalado no *gateway*.
- Além disso, o *software* também deve ser genérico o suficiente para que se adapte aos diferentes modelos de ITG.

1.5 METODOLOGIAS UTILIZADAS

Para o desenvolvimento do projeto foram utilizadas metodologias ágeis, visando agregar valor ao projeto o quanto antes de forma iterativa. Os métodos escolhidos para serem utilizados em conjunto foram: *Feature Driven Development* (FDD), *Test-Driven Development* (TDD) e SCRUM.

Cada módulo de teste foi desenvolvido, em uma visão bastante generalista, seguindo a sequência mostrada na Figura 5.

A lógica de negócio de cada teste foi feita com o auxílio de casos de uso e técnicas de diagnósticos de falhas, estudando a função e contexto de cada elemento do *gateway* e observando pontos que poderiam ser utilizados no processo de validação.

Figura 5 – Ciclo de desenvolvimento do *software*.

Fonte – Arquivo pessoal, 2022.

Logo após a validação das regras, a nova funcionalidade passou a ser implementada na aplicação do *firmware* utilizando a metodologia de desenvolvimento orientado a testes, o que garantiu uma baixíssima taxa de retrabalho do código, além de auxiliar no desenvolvimento da aplicação seguindo a arquitetura de *software* planejada.

A última etapa de cada ciclo de desenvolvimento consistiu na implementação do módulo na interface *web*, fazendo com que os valores e resultados obtidos nos testes já pudessem ser lidos pelo usuário de uma forma mais amigável, agregando valor para a entrega desde o início.

1.6 PLANEJAMENTO

Assim que definido o escopo do problema a ser enfrentado, foi dado início à fase de planejamento do projeto. Esta etapa contou com reuniões com o time de produção, responsáveis por realizar os testes de produto da Khomp, o que os caracteriza como “clientes finais” da aplicação.

Nesta reunião foram compreendidas as principais dificuldades do processo implementado até aquele momento e quais deveriam ser as prioridades para a construção do *firmware* de testes, desenhando assim, os primeiros requisitos do sistema.

Outra parte importante desta etapa foi o levantamento da documentação relativa a relatórios de testes e também um estudo inicial de possíveis ferramentas que auxiliassem no desenvolvimento e cumprimento das metas principais. Tendo acesso aos dados de métricas e gargalos do processo foi selecionada a *stack* de *softwares* do projeto, que será explicada mais a frente neste documento.

Durante a etapa de planejamento também foram definidas as metodologias utilizadas no projeto, já explanadas na seção anterior, e esboçada uma arquitetura geral do código.

1.7 ESTRUTURA DO DOCUMENTO

Este capítulo de introdução abordou a ideia geral de IoT, apresentação da empresa e contexto do trabalho. Também apresentou o problema, solução proposta e metodologias utilizadas para desenvolvimento do projeto. O Capítulo 2 trata sobre os produtos, apresentando os *gateways* da linha ITG da Khomp. O Capítulo 3 possui uma explicação sobre o processo de testes de produtos T0, T1 e T2 e os indicadores de cada etapa. O Capítulo 4 traz a fundamentação teórica apresentando os principais conceitos utilizados para o desenvolvimento deste projeto. O Capítulo 5 apresenta as ferramentas e tecnologias de *software* e gestão utilizadas no projeto. O Capítulo 6 apresenta o processo de desenvolvimento do projeto em si, casos de uso e metodologias. O Capítulo 7 discute os resultados obtidos com a implementação do *firmware* de testes. O Capítulo 8 traz uma apresentação de ideias para melhorias e otimizações no *firmware*. Finalmente, o Capítulo 9 apresenta conclusões acerca do processo de desenvolvimento do projeto, melhorias obtidas e conhecimentos adquiridos.

2 PRODUTOS

Este capítulo apresenta a família de *gateways* de telemetria da Khomp, que serão os dispositivos submetidos ao *firmware* de testes.

2.1 *GATEWAY*

A palavra inglesa *gateway* significa, em tradução livre, ”entrada“ ou ”ponte de ligação“. Trazendo para o contexto de redes, *gateways* são tecnologias que servem como ”pontes“ entre protocolos de comunicação, traduzindo pacotes de informações entregues a eles de um formato para outro, ou seja, um conversor de protocolos que pode operar em várias camadas da arquitetura OSI (HUANG, 2018).

Tanenbaum e Wetherall definem duas categorias de *gateways* em seu livro ”Redes de Computadores“. Uma delas descreve os *gateways* de transporte, que serviriam como conexões entre computadores que utilizam diferentes protocolos de transporte orientados a conexões. Um exemplo seria o de um computador que utiliza TCP/IP orientado a conexões e que precise se comunicar com uma segunda máquina que se comunica através do protocolo de transporte ATM, também orientado a conexões. Neste cenário, um *gateway* de transporte copiaria os pacotes de uma conexão a outra fazendo as devidas conversões para que a informação chegue a outra ponta sem problemas ou perdas (TANENBAUM; WETHERALL, 2011).

A segunda categoria definida por (TANENBAUM; WETHERALL, 2011) é a de *gateways* de aplicação. Estes atuam reconhecendo o formato e conteúdo dos dados e convertendo as mensagens para os formatos demandados. Por exemplo, um *gateway* de correio eletrônico poderia converter mensagens da internet em mensagens SMS para telefones móveis.

2.1.1 ***Gateway* IoT**

Com o avanço da tecnologia IoT, tornou-se cada vez mais desafiador integrar o grande número de dispositivos e protocolos com diferentes requisitos de conexão. Para suprir esta demanda, surgiram os *gateways* IoT. Segundo a *Open Automation Software* (OAS), estes dispositivos são essenciais na construção de uma solução IoT robusta e de alto desempenho computacional.

Um *gateway* de IoT permite a comunicação entre dispositivos e também entre dispositivos e nuvem. Trata-se de um *hardware* que contém um *software* embarcado executando uma rotina de tarefas. A funcionalidade mais básica de um *gateway* IoT é facilitar as conexões entre diferentes fontes de dados e destinos. Uma maneira simples de compreender seu papel em uma aplicação IoT é compará-lo com o roteador ou *gateway* tradicional de uma rede doméstica. Estes equipamentos facilitam a comuni-

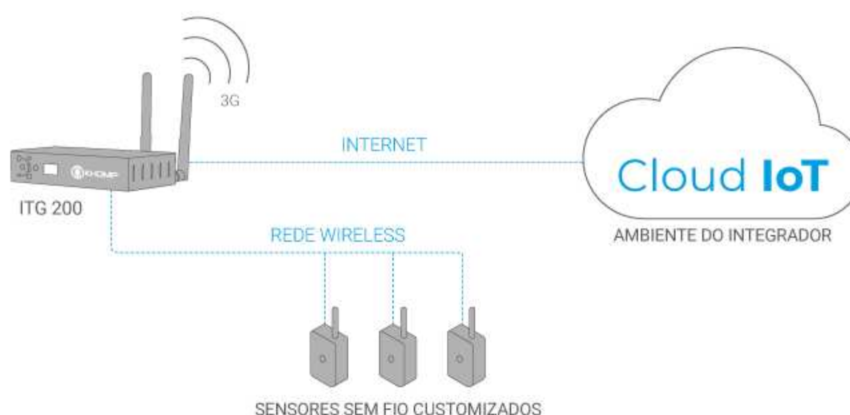
cação entre dispositivos, mantêm a segurança e fornecem uma interface por onde é possível executar funções básicas e de configuração. Um *gateway* IoT faz todas essas tarefas básicas como filtragens de dados até funcionalidades mais especializadas como permitir visualização de dados e realizar análises complexas (OAS, 2021).

2.2 A LINHA ITG

A linha ITG possui *gateways* destinados a aplicações de IoT. O ITG 200, em todas as suas versões, está há alguns anos no mercado e já bastante consolidado.

Os *gateways* ITG são os equipamentos responsáveis por transmitir, via internet, os dados recebidos dos *endpoints* para uma solução em *cloud*. Ou seja, no processo o *endpoint* realiza a medição pré-configurada, envia para o *gateway* que, por sua vez, encaminha para o sistema em nuvem que contabiliza ou aciona determinada função de acordo com a informação recebida. Isto pode ser observado na Figura 6. Por exemplo, ao medir determinada temperatura elevada, um sistema de resfriamento pode ser acionado de forma automatizada (KHOMP, 2021).

Figura 6 – Aplicação de um ITG 200 *Indoor* com sensores e comunicação com a nuvem.



Fonte – (KHOMP, 2021).

Nos *gateways* de telemetria da Khomp, a comunicação com sensores é feita através das tecnologias de comunicação *wireless* Zigbee ou LoRaWAN. O módulo IEEE 802.15.4 (Zigbee) é indicado para projetos de monitoramento de áreas menores, como o ambiente interno de uma indústria ou hospital, por exemplo. Já o módulo LoRa possibilita monitorar áreas mais extensas, podendo ser usado para redes públicas, agronegócios, universidades, entre outros. Estas tecnologias serão melhor descritas do Capítulo 4. O módulo de comunicação sem fio vem de fábrica e é definido pelo

cliente no momento da aquisição, dependendo da aplicação a qual o produto será destinado.

Para a comunicação entre o cliente do *gateway* e o servidor na nuvem, os *gateways* Khomp também contam com duas opções de protocolo: MQTT ou HTTP. A escolha fica a cargo do usuário, sendo facilmente configurável pela interface web do produto. Também há a versões *indoor* para módulos Zigbee ou LoRa (Figura 7) e *outdoor* apenas com LoRa (Figura 8) para melhor adequação ao contexto.

Figura 7 – ITG 200 *Indoor*.

Fonte – (KHOMP, 2021).

Figura 8 – ITG 200 *Outdoor* LoRa.

Fonte – (KHOMP, 2021).

Atualmente está em projeto o ITG 300 que também será lançado nas versões Indoor (Zibee e LoRa) e Outdoor (LoRa). As principais diferenças entre os dois modelos

de ITG estão no *hardware* e na forma como é obtido o binário de cada aplicação embarcada no *gateway*.

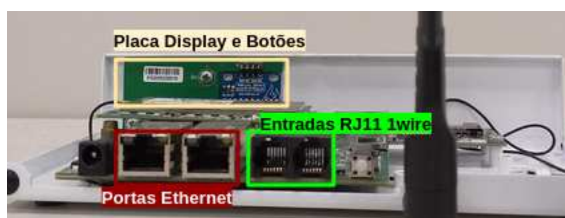
Quanto ao *hardware*, o ITG 300 contará com o dobro de memória e armazenamento em relação ao ITG 200. Além disso, o ITG 300 permitirá alimentação *Power over Ethernet* (PoE) e o módulo de elemento seguro estará sempre presente de fábrica, não sendo um item opcional como no outro modelo.

Por conta do espaço maior para armazenamento, o ITG 300 demandará uma configuração diferenciada para o Linux embarcado no dispositivo, o que impacta no processo de geração (popularmente conhecido como "*build*") do binário da aplicação.

No quesito *hardware*, pode-se especificar os seguintes elementos que compõem os ITGs e são mostrados nas figuras 9 e 10:

- Display OLED com 4 botões;
- 2 portas RJ11 1-Wire para integração de sensores (contato, temperatura e umidade);
- 2 portas RJ45 Fast Ethernet 10/100 Mbps para maior disponibilidade na transmissão dos dados;

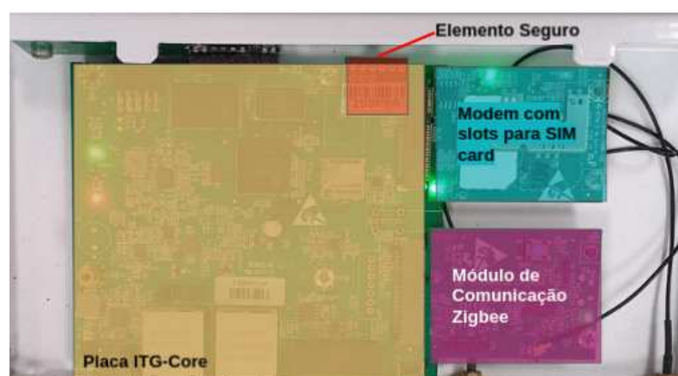
Figura 9 – Vista traseira de um *gateway Indoor* com entradas e placa de *display* e botões em evidência.



Fonte – Arquivo pessoal, 2022.

- Módulo para comunicação sem fio com *endpoints* através do protocolo LoRaWAN ou Zigbee;
- Módulo 3G de dados para até 2 cartões SIM - opcional para ITG 200 *Indoor*. No modelo *outdoor* o modem vem por padrão e o ITG 300 terá suporte apenas para modem 4G;
- Módulo com elemento de segurança criptografado (opcional para ITG 200 Zigbee e LoRa Khomp);
- No caso de falha na rede Ethernet, o módulo de dados 3G provê um sistema de *fallback*, garantindo a entrega das informações.

Figura 10 – Vista superior de um *gateway* Zigbee com modem, módulo de comunicação, elemento seguro e placa ITG-Core em evidência.



Fonte – Arquivo pessoal, 2022.

2.2.1 Rede estendida *EveryNet* e elemento de segurança

EveryNet é uma empresa de *software* subsidiária da *American Power*, empresa de telecomunicações que implanta redes LoRaWAN em território nacional. De forma análoga às operadoras de celular que possuem torres de telefonia e comercializam seu serviço de conectividade móvel, a *American Tower* e a *EveryNet* funcionam, juntas, como uma operadora.

A *American Power* instala as torres e a *EveryNet* fica a cargo de toda a gerência de distribuição de rede LoRaWAN, sendo uma *network server* LoRa com estrutura própria onde qualquer pessoa que pague está apta a utilizar a rede privada *EveryNet* LoRa.

Desde 2021, os *gateways* LoRa da Khomp possuem suporte à rede estendida EveryNet. Esta diferenciação será importante no momento de definir os casos de testes visto que os ITGs *EveryNet* obrigatoriamente devem conter o módulo de elemento de segurança, o que não é o caso em outros modelos de ITG.

O elemento seguro é um microchip que integra protocolo de segurança com autenticação de verificação de sinal para aplicações na área de IoT. Ele armazena, com base em hardware e de forma segura, chaves de criptografia, além de empregar contramedidas que visam eliminar potenciais riscos relacionados a vulnerabilidades de *software* (MICROCHIP, 2018).

Em suma, o elemento de segurança inserido no *gateway* é responsável por realizar a autenticação do dispositivo dentro de uma rede e armazenar a chave de criptografia que permite codificar e decodificar mensagens trocadas entre *gateway* e rede estendida *EveryNet*.

3 PROCESSOS

Este terceiro capítulo abrange todo o processo de testes de produção da Khomp. Os processos T0, T1, T2 e montagem de produto serão citados de forma breve.

Todas as informações deste capítulo foram retiradas de documentos internos da Khomp para mapeamento de processos. Por este motivo, este documento apresenta uma versão adaptada deste capítulo, omitindo informações como indicadores acompanhados em cada processo e também detalhes dos procedimentos de testes. A autora pede desculpas antecipadamente se a omissão de tais informações comprometer o pleno entendimento do texto.

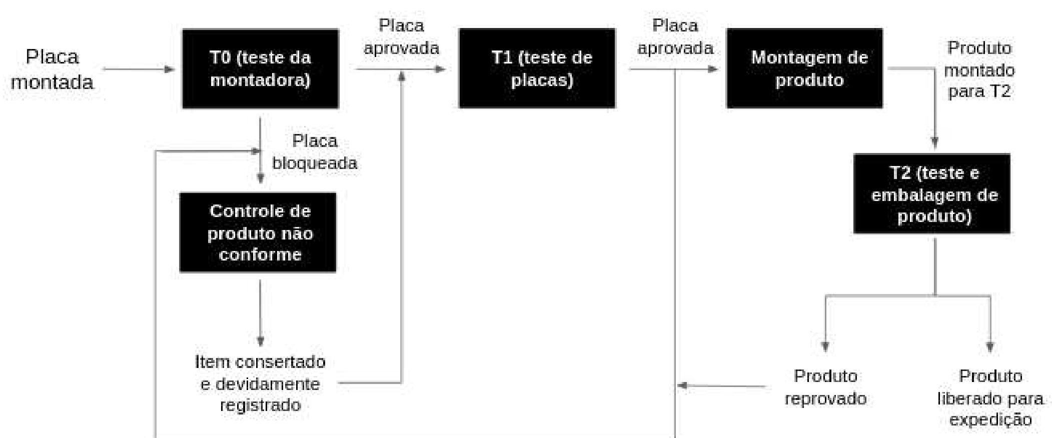
3.1 TESTES

Os produtos da Khomp passam por um processo bastante completo de testes até a chegada no cliente final. O processo abrange desde testes de placa até avaliações mais completas com o produto final já montado.

As duas primeiras etapas - T0 e T1 - são modulares, ou seja buscam avaliar funcionalidades isoladamente de cada placa que vá ser adicionada em qualquer produto mais adiante. Entre T1 e T2 há uma etapa intermediária de montagem de produto. Por fim, em T2 o produto é averiguado em sua totalidade, analisando se todos os módulos funcionam bem no conjunto da obra.

Para cada umas destas etapas há indicadores de desempenho e planos de ação para caso quaisquer não conformidades sejam identificadas. A Figura 11 mostra uma visão macro do processo inteiro.

Figura 11 – Panorama geral do processo de testes de produto da Khomp.



Fonte – Arquivo pessoal, 2022.

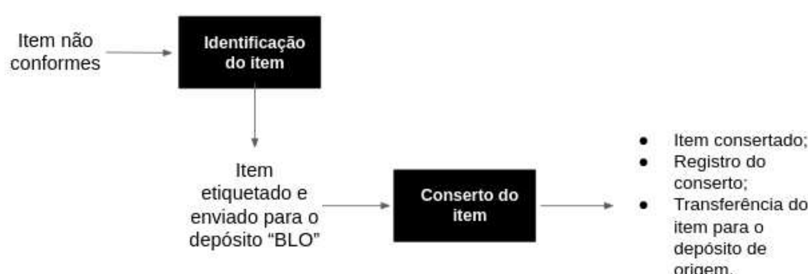
3.1.1 Controle de produto não conforme

Este procedimento descreve o processo adotado pela Khomp para assegurar que produtos que não estejam em conformidade com os requisitos sejam identificados e controlados, para evitar seu uso não intencional, definir o controle, responsabilidades e autoridades relacionadas a este processo.

Os itens são testados e inspecionados (quando aplicável). Caso reprovados, os itens são identificados com a etiqueta de item com defeito. Componentes que apresentarem não-conformidade ou defeito durante os ensaios e testes, deverão ser identificados com uma etiqueta contendo informações como código do item, nome do responsável pela identificação, número de série, data, descrição do defeito apresentado. O responsável por identificar o problema deverá colocar o item em local reservado para este fim.

A Figura 12 mostra um esquemático do processo. As ações tomadas para itens defeituosos dependem da natureza do defeito e do produto. Itens com defeito advindos das montadoras são cadastrados em um formulário e enviados para conserto técnico na produção. O resultado do conserto é inserido em outro documento.

Figura 12 – Processo de reparação de itens não conformes.



Fonte – Arquivo pessoal, 2022.

As demais matérias-primas não conformes serão analisadas pela área de Engenharia de Produção para verificar a necessidade de abertura de plano de ação. As possíveis opções para isso são:

- Retorno ao fornecedor;
- Conserto interno;
- Descarte.

É possível também que a não conformidade do produto não seja detectada na produção, fazendo com que o cliente final receba um produto defeituoso. Nesse caso,

avalia-se a abertura de um RMA e de um plano de ação para correção da origem do problema.

3.2 TESTE T0 - TESTE DA MONTADORA

O teste T0 tem como objetivo garantir a qualidade da montagem terceirizada em cada placa. As placas da Khomp são testadas nas montadoras de acordo com as instruções de trabalho definidas internamente.

As placas que não corresponderem a qualquer item de seus respectivos testes são dadas como reprovadas. Elas então são identificadas com uma etiqueta informando o bloqueio, avaliadas e quando possível consertadas na Khomp. A Figura 13 esquematiza o processo.

Figura 13 – Entrada e saída do teste T0.

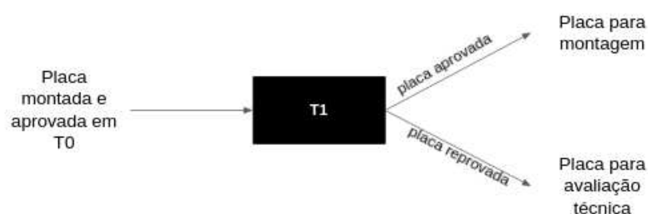
Fonte – Arquivo pessoal, 2022.

3.3 TESTE T1 - TESTE DE PLACAS

A etapa de testes T1 visa garantir a qualidade e perfeito funcionamento das placas para aplicação no produto. Assim como em T0, o teste de placas é feito seguindo as instruções definidas internamente para o processo.

As placas que não corresponderem da maneira esperada em algum aspecto de seus testes são consideradas reprovadas e tratadas conforme o descrito no processo de produto não conforme, explicitado na seção 3.1.1 deste capítulo. A Figura 14 mostra o processo.

Figura 14 – Entrada e saída do teste T1.



Fonte – Arquivo pessoal, 2022.

3.3.1 Montagem de produto

A montagem de produto (Figura 15) é uma etapa intermediária que ocorre entre os testes 1 e 2. Nela o produto é montado completo, já na forma que será enviado ao cliente final. O objetivo deste processo é realizar a montagem dos produtos conforme a ordem de produção, verificando a integridade física dos mesmos.

Figura 15 – Entrada e saída do processo de montagem de produto.

Fonte – Arquivo pessoal, 2022.

3.4 TESTE T2 - TESTE E EMBALAGEM DE PRODUTO

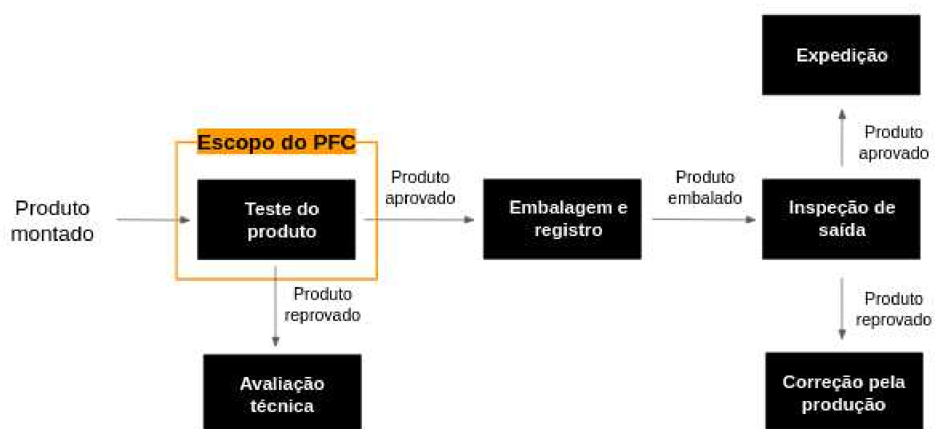
Esta etapa garante a qualidade e perfeito funcionamento do produto, já em sua forma final, pronto para ser entregue ao cliente.

O projeto descrito por este documento consiste na automação de T2 (Figura 16), reduzindo drasticamente o tempo despendido para a execução dos testes e também a possibilidade de falha humana no processo.

Assim como em T0 e T1, o teste de produto possui uma sequência definida internamente e também um protocolo a ser seguido no caso de não conformidades.

Todo este processo dura cerca de 10 a 15 minutos por produto testado, além de ser bastante dependente de interação humana. A proposta do *firmware* automatizado de testes é reduzir o tempo de T2, de forma que o mesmo possa ser executado em até 60 segundos para cada ITG testado. Com o novo *firmware* a interação do testador com o produto também será reduzida apenas aos processos que de fato demandam tal coisa, como testes I/O (botões, alarmes, etc).

Figura 16 – Mapeamento do processo de teste de produto e embalagem.



Fonte – Arquivo pessoal, 2022.

4 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda conceitos importantes para a compreensão geral do contexto do projeto e também para a implementação de cada módulo de teste e da interface *web*. O perfil do projeto é bastante heterogêneo. Por este motivo, este capítulo está dividido da seguinte maneira: inicialmente, são apresentados conceitos gerais do contexto do projeto como testes de *hardware* e programação concorrente. Na sequência são discutidas as metodologias de projeto. Finalmente são apresentados os conceitos utilizados na implementação do projeto por módulo, seguido dos conceitos utilizados no projeto da interface *web*.

4.1 TESTES DE *HARDWARE*

O desafio de testar sistemas eletrônicos cresceu rapidamente nas últimas décadas com dispositivos cada vez mais complexos e multifuncionais (MOURAD; ZORIAN, 2000). O processo de teste busca encontrar falhas em um sistema. Independente de o objetivo ser eliminar erros ou garantir aceitação de funcionalidades, o teste é um elemento fundamental no desenvolvimento de qualquer produto, auxiliando na construção de algo que faça o que foi inicialmente proposto a fazer (MEIRELLES, 2008).

Na subseção 4.1.1 serão apresentadas as terminologias de defeito, falha, erro, mau funcionamento, verificação, validação e teste que foram selecionadas para este trabalho.

4.1.1 Definições utilizadas

Dentre as comunidades técnicas especializadas em sistemas eletrônicos, engenharia de software e tolerância a falhas, há algumas divergências no entendimento de termos importantes como defeito, falha, erro, mau funcionamento, verificação, validação e teste (MEIRELLES, 2008). Por este trabalho se tratar especificamente de testes de módulos de hardware, serão utilizadas as terminologias consideradas pela área de sistemas eletrônicos elencadas por (MEIRELLES, 2008).

- **Defeito:** em um sistema eletrônico, um defeito é a diferença não intencional entre o dispositivo físico real e a especificação de projeto. Defeitos podem aparecer tanto na fase de fabricação de um dispositivo quanto no decorrer do tempo. O aparecimento de um defeito específico frequentemente pode indicar a necessidade de revisão do processo produtivo ou projeto do dispositivo.

- **Falha:** há uma linha bem tênue entre defeito e falha. A falha seria a abstração de um defeito à nível de funcionalidade. Ou seja, defeito trata de problemas relacionados ao hardware (físico) e falha diz respeito ao não cumprimento da função designada para um dispositivo.

- **Erro:** é um valor de saída errado produzido por um componente com falha. Ou seja, um erro é o efeito de uma falha ativa, quando algum circuito não executa a função a qual foi destinado.
- **Mau funcionamento:** são comportamento perceptíveis ao usuário final que resultam de erros não tratados no processo de produção.
- **Verificação:** nome dado ao processo realizado antes da implementação de um circuito, ainda na fase de projeto. Consiste em reproduzir o comportamento do sistema de acordo com o projetado (através de simulações, por exemplo) com o objetivo de assegurar o atendimento às especificações funcionais.
- **Validação:** validação é tida como uma forma de verificação de hardware que utiliza técnicas de testes da engenharia de software.
- **Teste:** o teste busca avaliar se um dispositivo físico está de acordo com as especificações, se apresenta um funcionamento correto (teste funcional) ou ainda se a implementação física se assemelha ao esquemático (teste estrutural). É na fase de testes que se deve garantir que apenas produtos sem defeitos serão comercializados.

4.1.2 Técnicas de Teste

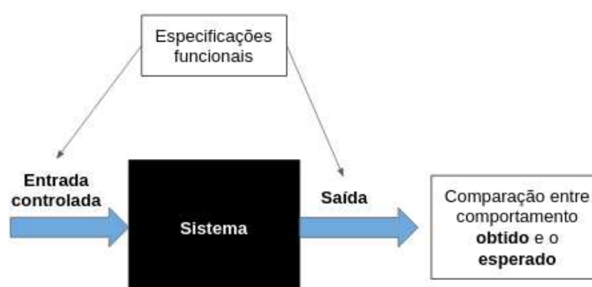
Por se tratar de um sistema embarcado, técnicas de teste de software também são comumente utilizadas para avaliação. As técnicas são classificadas em função da fonte das informações utilizadas para definição dos requisitos de teste (MEIRELLES, 2008). Este documento se aterá a duas técnicas: testes funcionais (caixa preta) e testes estruturais (caixa branca).

4.1.2.1 Teste Funcional

O teste funcional, também conhecido como caixa-preta, é baseado nas especificações do projeto e trata o sistema como uma caixa fechada, onde se conhece apenas o lado externo: as entradas e saídas. Desta forma, o testador é obrigado a utilizar as especificações do produto como um guia para definir casos de testes. Neste tipo de teste, apenas importa se o produto funciona de acordo com o esperado ou não. A forma como as funcionalidades foram implementadas, não entra em questão. Esta técnica pode ser executada de forma implícita, concorrente ou *online* (MEIRELLES, 2008).

A Figura 17 demonstra que em um teste funcional o sistema não é conhecido nos seus pormenores, e que as definições de entrada e saída são fundamentadas de acordo com a especificação funcional do sistema. Para um teste caixa preta o circuito não apresentará falhas se executar satisfatoriamente as funções a ele especificadas.

Figura 17 – Teste funcional de um sistema.



Fonte – Arquivo pessoal, 2022.

4.1.2.2 Teste Estrutural

Testes estruturais também são conhecidos como testes caixa branca pois para estipular os casos de teste é necessário conhecer aspectos internos do sistema, de sua implementação. Esta técnica geralmente faz uso de grafos para representação de um sistema. Do ponto de vista de sistemas embarcados, o objetivo do teste estrutural é avaliar se a estrutura física implementada vai ao encontro da especificada. Depende diretamente da estrutura do circuito (portas lógicas, transistores, por exemplo) para verificar cada um esses elementos (MEIRELLES, 2008). A Figura 18 explicita um grafo representando a importância da implementação do sistema nestes testes.

Figura 18 – Esquema de um teste estrutural evidenciando a implementação do sistema.

Fonte – Arquivo pessoal, 2022.

Um teste de caixa branca, em teoria, deve ser capaz de avaliar todos os possíveis caminhos lógicos que um *software* poderá apresentar ou então todos os estados possíveis, no caso de implementar este teste em um contexto de *hardware* (PRESSMAN, 1995).

4.1.3 Diagnóstico de falhas

Um diagnóstico de falhas busca identificar e isolar, em tempo hábil, a causa de mau funcionamento em um sistema (FENTON; MCGINNITY; MAGUIRE, 2001). Ainda de acordo com Fenton, o processo de diagnóstico pode ser descrito em três passos:

1. **Geração de Informações de Falhas:** devem ser recolhidas informações sobre a origem de uma falha. Estes dados podem ser obtidos de diversas maneiras: análise dos sintomas observados, medições e execução de testes de diagnóstico.

2. **Formulação de Hipóteses de Falha:** a partir das informações coletadas é possível formular hipóteses sobre o que pode estar causando a falha e onde está sua localização no sistema. Essas hipóteses devem ser consistentes com as informações levantadas no primeiro passo.

3. **Distinção de Hipóteses:** se houver mais de uma hipótese de causa para uma falha, então é necessário realizar outros testes ou aplicar uma análise histórica para que haja uma clara distinção entre as ideias. No caso de não haver mais discriminação possível, pode-se utilizar experiência ou tentativa e erro para determinar a melhor forma de endereçar o problema.

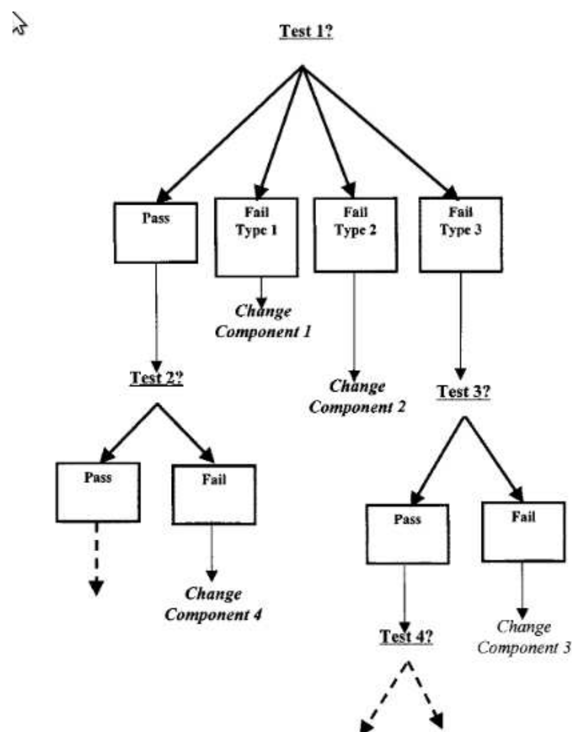
Essencialmente, o processo de diagnóstico pode ser definido como o isolamento de falhas utilizando informações coletadas através de observações e testes do sistema. Tradicionalmente há duas abordagens para o processo de diagnóstico: sistemas baseados em regras e a construção de uma árvore de decisão de falha.

Sistemas de diagnósticos baseados em regras são um compilado da experiência e vivências de profissionais da área de diagnóstico. Geralmente estas regras assumem a forma ”SE <sintoma(s)> ENTÃO <falha(s)>“. Representar o conhecimento acerca de um problema específico pode exigir centenas ou até mesmo milhares de regras.

A inferência de um diagnóstico baseado em regras consiste em pegar informações sobre o domínio do problema e invocar regras correspondentes a essas informações. Isto gera novos dados que são adicionados às informações do problema. Este processo é repetido de forma iterativa até ser encontrada uma solução para o problema.

O método de árvores de decisão de falhas é o mais utilizado para documentação de procedimentos de diagnóstico. Uma árvore de falha (Figura 19) utiliza sintomas observados e resultados de testes como um ponto inicial. Deste nodo pai, ramifica-se uma árvore de decisão com ações, decisões e encaminhamentos (folhas da árvore).

Figura 19 – Esquema simples de uma árvore de falhas.



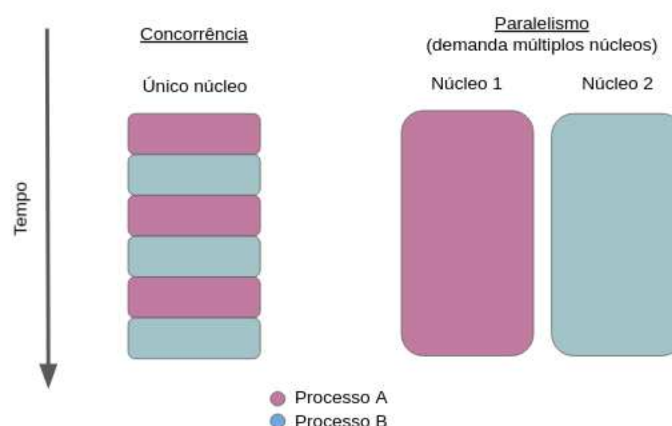
Fonte – (FENTON; MCGINNITY; MAGUIRE, 2001)

4.2 PROGRAMAÇÃO PARALELA E CONCORRENTE

Segundo (BRESHEARS, 2009), um sistema é dito concorrente quando é capaz de suportar dois ou mais processos **em progresso** ao mesmo tempo. Já um sistema que consegue executar duas ou mais ações **simultaneamente** é considerado paralelo. A Figura 20 ilustra como sistemas concorrentes e paralelos lidam com a execução de dois processos ao mesmo tempo.

Uma aplicação concorrente terá duas ou mais *threads* em andamento em algum momento da sua execução. Ou seja, em um processador mononuclear, o sistema operacional estará comutando a execução dos processos. Estas *threads* estão **em andamento** ao mesmo tempo, mas sua execução não acontece de forma simultânea (BRESHEARS, 2009).

Figura 20 – Comparativo entre concorrência e paralelismo.



Fonte – Arquivo Pessoal, 2022.

Na execução paralela, é necessário um processador de vários núcleos disponível. Neste tipo de aplicação, as várias *threads* podem ser executadas cada uma em um núcleo diferente, sem a necessidade de troca entre processos, garantindo uma execução **simultânea** (BRESHEARS, 2009).

É possível desenvolver um sistema concorrente que opere em um processador multinúcleos, utilizando estes *cores* para comutar processos. Porém, é impossível que o conceito de paralelismo seja implementado sem a presença de múltiplos núcleos (BRESHEARS, 2009).

4.3 METODOLOGIAS ÁGEIS

O ”Movimento *Agile*“ ganhou grande força na indústria de software com o ”Manifesto do Desenvolvimento de Software Ágil“, publicado em 2001 por um grupo de profissionais do ramo da programação (ABRAHAMSSON *et al.*, 2017). Quatro pilares sustentam a filosofia expressa no manifesto:

- Indivíduos e interações estão acima de processos e ferramentas;
- O funcionamento do software está acima de uma documentação abrangente;
- Colaboração com o cliente acima de negociações e contrato;
- A capacidade de resposta rápida a mudanças vai acima de um planejamento pré-estabelecido.

É importante frisar que aspectos como processos, ferramentas, uma boa documentação e um contrato bem redigido ainda possui grande valor dentro do processo de

desenvolvimento, mas os outros pontos se tornam mais valorizados à luz da filosofia ágil.

O primeiro pilar indica que o movimento enfatiza o relacionamento e a comunidade de desenvolvedores do software em detrimento a processos institucionais muitas vezes engessados e ferramentas de desenvolvimento. Na prática, este aspecto se mostra presente em um ambiente colaborativo e que inspira o espírito de equipe entre o time de desenvolvimento (ABRAHAMSSON *et al.*, 2017).

Em segundo lugar, a parte mais importante de um software é fazer o que foi designado para fazer, e o quão mais rápido isso acontecer, melhor. Logo, é vital que a equipe de software esteja continuamente mantendo o produto funcionando. Periodicamente, novas funcionalidades vão sendo adicionadas a um software já funcional.

O terceiro ”mandamento“ expressa a necessidade de se manter um bom relacionamento com os clientes, mantendo sempre uma boa comunicação para garantir que todas as partes estão em sintonia quanto ao andamento do projeto e também que o mesmo está sendo desenvolvido de forma a atender à dor do usuário. Um contrato bem elaborado tem grande importância, esta que cresce de acordo com o tamanho do projeto. Do ponto de vista empresarial, o desenvolvimento ágil deve estar focado em entregar valor de negócios imediatamente, reduzindo riscos de incumprimento de contrato.

Por último, o grupo responsável pelo projeto, formando por desenvolvedores e clientes, deve estar ciente da possibilidade de mudanças dentro de um projeto de tecnologia. Um planejamento de software é importante mas não deve ser rígido demais a ponto de não comportar necessidades de ajustes emergentes durante o processo de desenvolvimento. Para o movimento ágil, o ciclo de criação de um software é vivo e deve estar sujeito a melhorias e alterações. Contratos e equipe devem estar preparados para lidar com essas questões de forma rápida e natural.

Abrahamsson ainda enumera as seguintes características de um processo de criação de software *agile*:

1. Processo de desenvolvimento modular;
2. Iterativo e em ciclos curtos, o que permite verificações constantes e correções rápidas;
3. Ciclos de desenvolvimento com duração de uma a seis semanas;
4. Parcimônia, o que reduz a realização de atividades desnecessárias;
5. Adaptável quanto à possibilidade de novos riscos;
6. Abordagem incremental, garantindo uma aplicação funcional construída em pequenas entregas e também minimizando riscos;

7. Orientado a pessoas, ou seja, o processo prioriza pessoas sobre processos e tecnologias;

8. Estilo de trabalho comunicativo e colaborativo.

Dentre os diversos métodos ágeis disponíveis na literatura, este documento se aterá a dois, um focado em gerenciamento de projetos e outro com enfoque no processo de desenvolvimento em si que são, respectivamente, SCRUM e FDD.

Um aspecto importando do ciclo de desenvolvimento FDD engloba a utilização testes de software, o que possibilita a implementação de outra metodologia dentro do projeto, o TDD para garantir a qualidade do software.

As seções a seguir irão explicar o processo por trás de cada uma destas metodologias e como todas atuam em conjunto para garantir um processo de desenvolvimento rápido e de qualidade.

4.3.1 SCRUM

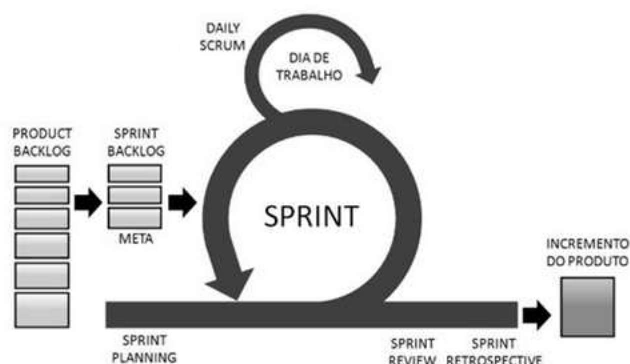
A abordagem SCRUM foi desenvolvida pensando no gerenciamento do processo de desenvolvimento de sistemas. É uma metodologia que aplica conceitos de teoria de controle de processos industriais ao processo de desenvolvimento, entregando uma abordagem que reintroduz ideias de flexibilidade, adaptabilidade e produtividade. SCRUM não define nenhuma técnica específica de desenvolvimento para a implementação de um software, se concentrando apenas em como as pessoas inseridas dentro do processo produtivo devem trabalhar para produzir um sistema flexível em um ambiente de constante mudança (ABRAHAMSSON *et al.*, 2017).

A principal ideia acerca do SCRUM é que o desenvolvimento de um sistema engloba diversas variáveis como requisitos, recursos, tecnologias e prazos, todas estas sujeitas a alterações durante o processo. Isso torna o processo imprevisível e complexo, o que requer flexibilidade para responder a estas mudanças (ABRAHAMSSON *et al.*, 2017).

A Figura 21 ilustra um ciclo de desenvolvimento utilizando SCRUM para o gerenciamento de atividades. Segundo (ABRAHAMSSON *et al.*, 2017), o processo se dá em três fases: *pre-game*, desenvolvimento e *post-game*.

A fase de *pre-game* ilustra a parte mais à esquerda da imagem, que engloba o planejamento da *sprint*, o período que comporta um ciclo de desenvolvimento SCRUM. Esta primeira fase consiste em uma reunião de planejamento - *sprint planning* - onde é analisada a lista de tarefa que devem ser implementadas no produto (*backlog* do *produto*. Após uma discussão que leva em conta o número de pessoas envolvidas no projeto, nível técnico de cada uma e prioridades para o produto são definidas quais destas tarefas serão selecionadas para comporem as tarefas da *sprint* (*backlog* da *sprint*). É também nessa fase que é feito um *design* alto nível da arquitetura, apenas

Figura 21 – Ciclo de desenvolvimento SCRUM.



Fonte – https://evolvemvp.com/o-que-e-scrum-conceito-definicoes-e-etapas/

para ter compreender quais conhecimentos e a complexidade que cada tarefa irá demandar (ABRAHAMSSON *et al.*, 2017).

Feito o planejamento, segue-se para a etapa de desenvolvimento. Esta é a parte *agile* do SCRUM. As diferentes variáveis que, segundo a filosofia do SCRUM, tornam um processo de desenvolvimento imprevisível são aqui observadas e controladas ciclo a ciclo. Ao invés de tratar este assunto apenas no início do projeto de software buscando prever todas as possibilidades, o SCRUM busca estar sempre retornando para esta pauta, controlando as variáveis para que seja possível se adaptar de forma rápida à qualquer necessidade de mudança. Cada etapa de desenvolvimento vai envolver as fases tradicionais da criação de software: requisitos, análise, design, evolução e entrega. A arquitetura do sistema vai evoluindo durante a fase de desenvolvimento (ABRAHAMSSON *et al.*, 2017).

Durante esta segunda fase acontece o desenvolvimento ou aprimoramento do produto, sempre de forma cíclica e incremental, de uma *sprint* para outra.

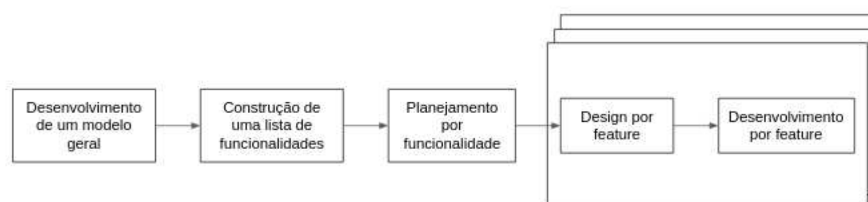
Um ciclo SCRUM encerra-se com o *post-game*. Na reunião de *sprint review*, é revisado o *backlog* da *sprint*, verificando se todas as tarefas foram finalizadas e concluindo a nova versão do produto com as funcionalidades, correções e melhorias planejadas. Esta fase também abrange a redação da documentação técnica do produto e o lançamento da nova versão. Encerrada esta etapa, dá-se início a um novo ciclo SCRUM, com um novo planejamento de *sprint*, desenvolvimento e assim por diante (ABRAHAMSSON *et al.*, 2017).

4.3.2 Desenvolvimento Orientado a Funcionalidades

O método FDD foi criado para trabalhar em conjunto com outras metodologias para desenvolvimento de software visto que não cobre todo o processo de desenvolvi-

mento, se encarregando apenas da parte relativa ao projeto e construção do software. Além disso, é bastante flexível visto que não requer nenhuma técnica específica para desenvolvimento de software. FDD enfatiza aspectos de qualidade durante todo o processo e converge com o SCRUM na filosofia de entregas frequentes e tangíveis, com um monitoramento preciso do processo (ABRAHAMSSON *et al.*, 2017).

Figura 22 – Ciclo de desenvolvimento FDD.



Fonte – (ABRAHAMSSON *et al.*, 2017)

Há cinco etapas que constituem um ciclo de desenvolvimento FDD, como ilustrado na Figura 22. Cada uma delas conta com diretrizes e técnicas necessárias para a entrega do sistema.

1. **Desenvolver modelo geral**: nesta etapa, os desenvolvedores já têm ciência do escopo, contexto e requisitos do sistema documentados como casos de uso, especificações funcionais, etc. A equipe, com cada especialidade, se reúne e é apresentado um passo-a-passo que descreve o funcionamento do sistema em alto nível. A partir disso, o domínio geral é dividido em diferentes áreas ou módulos e é feito um passo-a-passo detalhado para cada parte. A partir destes esquemáticos mais específicos, constroem-se modelos de objetos (ou outro paradigma semelhante) para cada módulo e então se discute acerca dos modelos mais apropriados. Desta forma, um modelo geral do sistema é construído simultaneamente ao processo.

2. **Construir lista de funcionalidades**: junto com uma documentação de requisitos, o modelo criado na etapa anterior forma uma boa base para a construção de uma lista de recursos para o sistema a ser desenvolvido. Na lista, a equipe de desenvolvimento apresenta cada uma das funcionalidades de valor para o cliente final que serão incluídas no sistema. Antes de passar para a próxima etapa, a lista ainda é revisada pelos usuários e desenvolvedores para que seja validada.

3. **Planejar funcionalidades**: fase de planejamento propriamente dito do ciclo. Consiste na criação de um plano de alto nível, priorizando funcionalidades que são

mais importantes para que o usuário tenha acesso o quanto antes. Para isso é definido um cronograma com os principais marcos do projeto. Neste planejamento também são avaliadas possíveis relações de dependências entre módulos. É nesta fase que são distribuídas entre os desenvolvedores as tarefas desenhadas na etapa 1.

4. **Design e Desenvolvimento da *feature***: essas duas etapas são iterativas e uma iteração não deve durar mais que alguns dias, estendendo-se em no máximo duas semanas. Uma ou mais funcionalidades são selecionadas juntamente com os resposáveis por cada uma, equipes/profissionais encarregados de diferentes módulos podem trabalhar de forma simultânea entre o desing e implementação das funcionalidades. Esta etapa engloba todo o processo de desenvolvimento: inspeção de projeto, programação, testes unitários, integração e revisão de código. Após cumpridas as tarefas de um ciclo, as novas funcionalidades implementadas passam a compor a aplicação principal enquanto um novo ciclo se inicia com um novo conjunto de funcionalidades.

4.3.3 Desenvolvimento Orientado a Testes

TDD é um processo iterativo de desenvolvimento. Segundo Hardik Shah, no TDD o desenvolvedor escreve os testes baseado nas especificações do projeto que ditam como um ditam como o programa deve se comportar. É uma metodologia de ciclo bastante rápido que consiste em teste, escrita de código e refatoração (SHAH, 2019).

Segundo (BECK, 2010), desenvolvimento guiado a testes baseia-se em duas regras simples:

- Escrever código apenas se necessário, ou seja, no caso de um teste automatizado falhar;
- Eliminar duplicatas.

O projeto deve ser composto por componentes bastante coesos porém fracamente acoplados, a fim de tornar os testes mais fáceis (BECK, 2010). Estas características fazem com TDD seja uma metodologia facilmente trabalhada em conjunto com outros métodos ágeis, como FDD e SCRUM, por exemplo, por terem filosofias parecidas e atuarem em campos diferentes no contexto de um projeto de software.

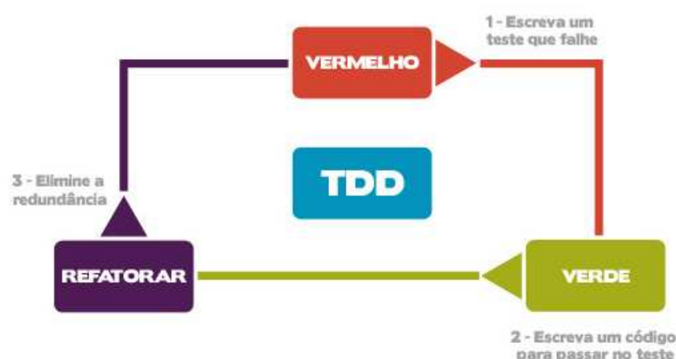
As fases que compõem um ciclo TDD são listadas por Beck e apresentadas na Figura 23:

1. Vermelho - Escrever um pequeno teste que não funcione e que talvez nem mesmo rode inicialmente.

2. Verde - Escrever apenas o código necessário para que o teste seja aprovado. Sem se preocupar em implementar uma lógica real na função.

3. Refatorar - Implementar a lógica e eliminar duplicatas.

Figura 23 – Ciclo de desenvolvimento TDD.



Fonte – https://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151

Na fase "**vermelho**", são escritos os testes unitários da função que será implementada. Funções e seu comportamento esperado são definidos a partir do projeto de software. Um teste unitário deve ser simples e focado em uma única função. Escritos os testes, o próximo passo é esboçar a função. Nesta primeira etapa busca-se apenas um código que compile, ou seja, sem erros de sintaxe. Após implementar o esboço da função, deve-se rodar os testes e é esperado que estes falhem. Neste contexto a falha é algo bastante significativo e até mesmo positivo. A partir dela é possível concluir que o programa está sendo devidamente testado pelo o que deve de fato entregar e apresenta o comportamento esperado dentro de um ambiente controlado (SHAH, 2019).

Na segunda etapa, "**verde**", o objetivo principal é fazer com que o teste passe, não importando os meios utilizados para isto. Aqui, escreve-se apenas o suficiente de código para que o teste relacionado seja aprovado· Assim que todos os testes forem executados e aprovados, considera-se que a função está funcionando da forma esperada a nível de entrada e saída, mesmo que o código implementado não siga as boas regras de programação (SHAH, 2019).

Na última etapa, "**refatorar**", o código feito para a função pode ser melhorado sem preocupação com possíveis erros de código. Aqui, deve-se revisar o programa e implementar as melhorias possíveis para mantê-lo limpo e legível. Após cada alteração, os testes devem ser executados novamente de forma a garantir que todos continuam passando (SHAH, 2019).

Este ciclo é feito para cada funcionalidade mapeada na etapa de definição de requisitos. Isto garante um desenvolvimento fluido que diminui o risco de falhas simples avançarem no processo, causando um retrabalho evitável e atrasos (SHAH, 2019).

4.4 CONCEITOS ESTUDADOS POR MÓDULOS

Esta seção trata de conceitos importantes utilizados na implementação dos principais módulos de teste do ITG: conectividade e comunicação com rede de sensores sem fio.

4.4.1 Testes dos módulos de comunicação *wireless*

Quanto à comunicação com *endpoints*, os *gateways* ITG podem ser classificados em Zigbee ou LoRa. Esta seção explora estas duas tecnologias e também se aprofunda no conceito de MQTT, outra tecnologia de comunicação utilizada extensamente pelos *gateways* e em especial para o teste de módulos de comunicação.

Comunicação sem fio consiste na transferência de informações entre dispositivos sem a necessidade de uma interface física que os conecte, como cabos, por exemplo. Segundo (FENG; YAN; LIU, 2019), essas tecnologias são divididas, de acordo com a distância de transmissão, em três categorias: tecnologias de curta distância (distâncias inferiores a 10m), média distância (a distância varia de 10m a 100m) e tecnologias de comunicação sem fio de longa distância (distâncias superiores a 100m).

Abaixo serão explicadas duas destas tecnologias: Zigbee (curta-média distância) e LoRaWAN (longa distâncias).

4.4.1.1 LoRaWAN

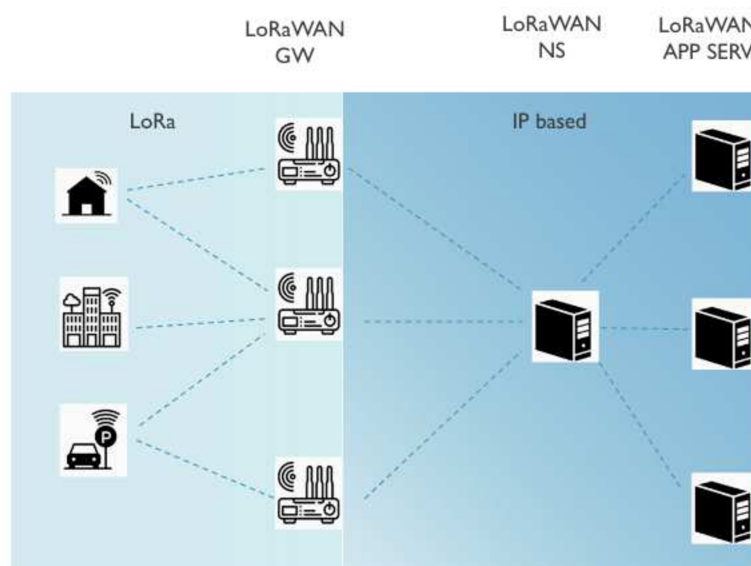
LoRaWAN é uma tecnologia de rede de área ampla e baixa potência (*Low Power Wide Area Network* (LPWAN)) que tem recebido atenção significativa nos últimos anos. Tem como características o baixo consumo de energia, uma comunicação com baixa taxa de dados em uma ampla cobertura.

Em grande parte dos cenários, uma rede IoT termina em sensores alimentados por bateria distribuídos áreas extensas. Estes fatores demandam tecnologias que ofereçam um baixo consumo de operação e comunicação sem fio a longa distância e o protocolo LoRaWAN se encaixa perfeitamente neste contexto (HAXHIBEQIRI *et al.*, 2018).

Para entender a comunicação através da rede LoRaWAN é importante conhecer um pouco do conceito da camada física LoRa. Patenteada pela Semtech em 2014, é uma tecnologia de modulação de rádio para redes longa distância e baixa potência (como a LoRaWAN). O nome LoRa vem do inglês ***lo****ng*-***ra****nge*, referência ao alcance extremamente longo que a tecnologia permite para conexão. Sinais LoRa podem che-

gar a percorrer 700km, porém comunicações LoRa geralmente são limitadas em cerca de 5km para áreas urbanas e 15km em regiões rurais. LoRa opera na camada física permitindo que dispositivos traduzam dados em sinais RF. O padrão LoRaWAN é um protocolo de software assíncrono que utiliza a camada física LoRa (SEMTECH, 2021).

Figura 24 – Topologia de uma rede LoRaWAN.



Fonte – (HAXHIBEQIRI *et al.*, 2018).

A rede LoRaWAN é estruturada com uma topologia estrela, de forma que dispositivos LoRa só podem se comunicar através de *gateways* LoRa, sem trocas diretas entre si. Um *gateway* LoRa atua apenas no envio de pacotes de dados brutos dos *endpoints* para um servidor através do protocolo UDP. Um *gateway* IoT, como o ITG com módulo de comunicação LoRa, não apenas repassa estas mensagens como também as traduz para formatos de dados legíveis para humanos e que podem ser manipulados mais facilmente, como por JSON por exemplo. Também são suportados comandos vindos do servidor para o *endpoint* LoRa, estes são chamados "*downlinks*". Além disso, a comunicação termina em aplicações que podem pertencer a terceiros. No contexto da Khomp, estes terceiros seriam os clientes integradores. Unindo os *endpoints*, *gateway* LoRa, servidor de rede e aplicações, temos como resultado uma rede LoRa como a ilustrada na Figura 24 (HAXHIBEQIRI *et al.*, 2018).

4.4.1.2 Zigbee

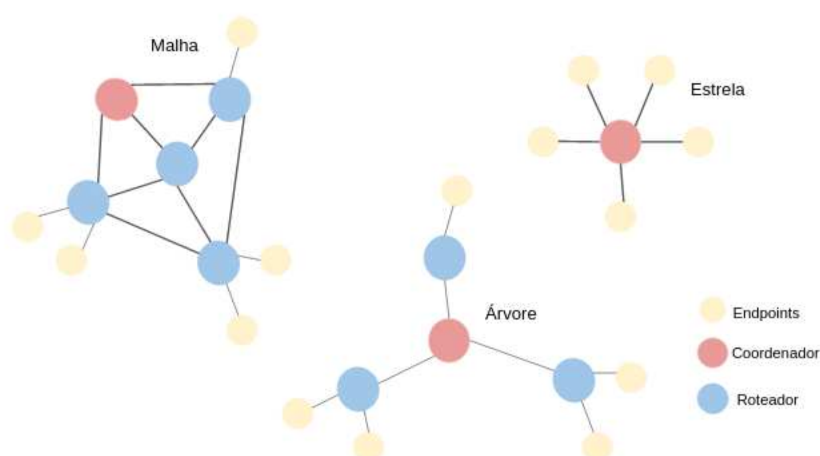
O protocolo LoRaWAN supre muito bem as necessidades de aplicações IoT que demandam comunicação a longas distâncias. Porém, muitas vezes o cenário não precisa de uma cobertura tão ampla. Neste caso, o uso de LoRa continua sendo uma opção, mas exagerada já que uma rede de comunicação poderosa com alcance de

centenas de metros ou quilômetros estaria sendo utilizada para troca de dados entre dispositivos com, no máximo, 100 metros de distância entre si.

Para casos assim o protocolo Zigbee se encaixa perfeitamente, principalmente como solução mais barata. Com foco em dispositivos de baixa potência, o Zigbee assim como o LoRaWAN, também tem como características o baixo consumo de energia e baixa taxa de transferência de dados, porém com alcance máximo de 100 metros entre dispositivos.

O protocolo Zigbee é baseado na camada física definida pelo padrão IEEE 802.15.4 (da mesma forma que a rede LoRaWAN com a camada física LoRa). Este padrão busca justamente especificar a camada física para suporte a redes sem fio que focam em baixo custo, baixa velocidade de comunicação entre dispositivos próximos (KUZMINYKH; SNIHUROV; CARLSSON, 2017).

Figura 25 – Topologias possíveis para uma rede Zigbee.



Fonte – Arquivo Pessoal, 2022.

Zigbee suporta diversas topologias de rede como estrela, malha e árvore. Uma rede Zigbee é composta por dois tipos de dispositivos: FFD (funcionalidade total) e RFD (funcionalidade reduzida). Dependendo do papel de um *device* na rede este poderá ser classificado como coordenador, roteador ou *endpoint*. Coordenadores são necessariamente FFD visto que inicializam a rede e precisam gerenciá-la. Roteadores também precisam ser FFD pois conectam-se a outros dispositivos transmitindo dados entre eles. Os *endpoints* podem ser FFD ou RFD já que sua única função é receber e transmitir mensagens. Dispositivos RFD apenas adquirirem e transmitem as informações para o nó pai. Não são usados para tarefas como descoberta e manutenção de rotas (WANG; HE; WAN, 2011).

A Figura 25 ilustra as três possíveis topologias para uma rede Zigbee. Na arquitetura estrela, coordenador é o dispositivo central, conectando-se a todos os demais da

rede. É o esquema mais simples porém pode facilmente sobrecarregar o coordenador visto que este é responsável por rotear todas as mensagens da rede.

Uma topologia de árvore apresenta o coordenador no topo e roteadores nos níveis intermediários da árvore. Os dispositivos Zigbee encerram cada ramo da árvore, inclusive alguns *endpoints* podem estar diretamente ligados ao coordenador. Neste cenário, um *endpoint* que seja FFD pode se conectar a outro *dispositivo* e expandir a árvore. Apesar de não sobrecarregar tanto o coordenador, a topologia de árvore mantém a característica de apresentar apenas um caminho possível para cada *endpoint*.

No caso da rede *mesh*, ou malha, os roteadores são interligados criando várias rotas possíveis entre dispositivos. Esta topologia torna a rede bem robusta, garantindo que as mensagens serão enviadas pela melhor rota possível. Esta é a topologia implementada pelos *gateways* IoT Zigbee da Khomp, onde o ITG cumpre o papel de coordenador da rede.

4.4.1.3 MQTT

Message Queue Telemetry Transport (MQTT) é um protocolo de transporte de mensagens entre cliente e servidor que utiliza um sistema baseado em publicação e assinatura de tópicos. É uma tecnologia simples e projetada para ser de facilmente implementada, tornando-a ideal para uso em muitas situações como para comunicação em *Internet of Things* (IoT), onde não são necessárias habilidades avançadas de programação ou quando a largura de banda da rede é um recurso importante (BANKS; GUPTA, 2015).

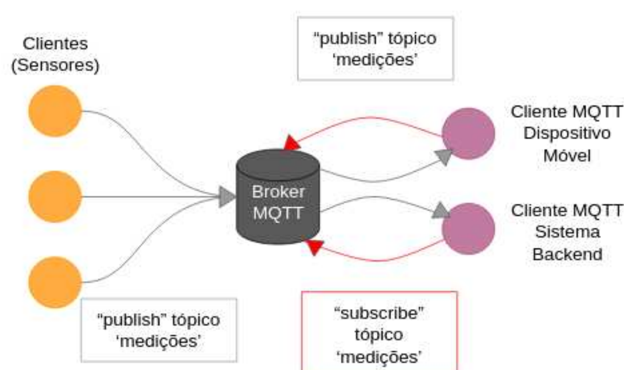
Segundo a especificação oficial do MQTT, o protocolo é executado utilizando TCP/IP - assim como HTTP - ou outros protocolos de rede que forneçam dados ordenados, sem perdas e em conexões bidirecionais. Seus recursos incluem:

- Uso do padrão de mensagem publicar/assinar, que fornece distribuição de mensagens um-para-muitos (1-n) e dissociação de aplicações.
- Transporte de mensagens independentemente do conteúdo.
- Três opções para qualidade do serviço (QoS) no que diz respeito à entrega de mensagens: **"no máximo um vez"** onde as mensagens são entregues de acordo com os melhores esforços do ambiente operacional. Este cenário é suscetível a perda de mensagens então seu uso é aconselhado para ambientes em que a perda de leituras individuais não representa grande impacto. Outra forma é **"pelo menos uma vez"** onde há a garantia de que as mensagens serão entregues porém, com a chance de duplicatas e, por último **"exatamente uma vez"** onde há a garantia de entrega da mensagem exatamente uma vez. Esta configuração é indicada para casos em que duplicatas ou a não entrega de uma mensagem possa gerar danos maiores como, por exemplo, cobranças indevidas.

- Redução do tráfego da rede.
- Mecanismo para notificação quando ocorre uma desconexão não conforme.

Como definido por (DINCULEANĂ; CHENG, 2019), os clientes da aplicação não trocam mensagens diretamente entre si, o *broker* é encarregado desta tarefa. A Figura 26 apresenta um esquemático simplificado do funcionamento de um *broker* MQTT.

Figura 26 – Esquema simplificado do funcionamento da comunicação via MQTT.

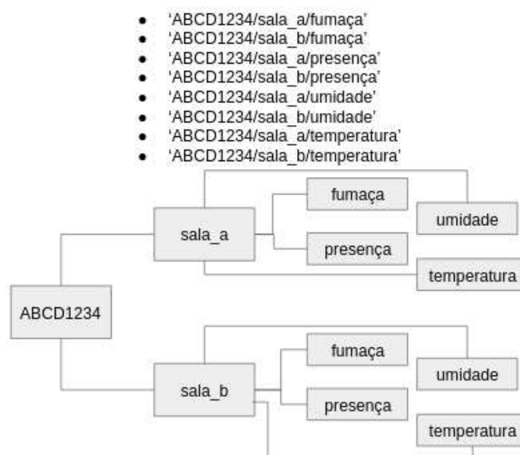


Fonte – Arquivo Pessoal, 2022.

Cada mensagem MQTT contém um tópico, organizado em estrutura de árvore, a qual os clientes podem publicar ou assinar - enviar ou receber mensagens do tópico, respectivamente. O *broker* recebe mensagens com determinado valor ou comando publicadas por clientes e as retransmite para cada cliente inscrito no tópico em específico. O protocolo MQTT foi projetado para comunicação assíncrona, com assinaturas e/ou publicações para diferentes entidades ocorrendo paralelamente (DINCULEANĂ; CHENG, 2019).

A Figura 27 explica a estrutura em árvore do tópico de uma mensagem MQTT. Na imagem, um *gateway* é identificado pela *string* 'ABCD1234' e está conectado a diferentes sensores em duas salas diferentes. Cada sala está equipada com um sensor de fumaça, presença, temperatura e umidade. Na imagem são explicitados 8 tópicos, cada um referente a um sensor. Um sistema que implementa uma aplicação para monitoramento dos ambientes *sala_a* e *sala_b* pode escolher receber informações apenas do sensor de temperatura da *sala_b*, neste caso, o cliente MQTT desta aplicação deve assinar o tópico $'ABCD1234/sala_b/temperatura'$. Em um cenário em que se deseja receber informações de todos os sensores da *sala_a*, o tópico deve ser $'ABCD1234/sala_a/\#'$, onde o operador # sinaliza que todos os subtópicos são de interesse do cliente. O uso do # funciona de forma análoga para acessar todas as informações recebidas pelo *gateway*. Outro operador importante é o '+' que pode ser

Figura 27 – Esquema simplificado do funcionamento dos tópicos MQTT.



Fonte – Arquivo Pessoal, 2022.

utilizado para ”pular“ um nível de especificação da árvore, por exemplo, um cliente inscrito no tópico $'ABCD1234/+/umidade'$ receberá todas as informações relativas aos sensores de umidade conectados ao *gateway*, independente da sala ou quaisquer outros subtópicos que possam estar entre estes dois níveis.

4.4.2 Teste de conectividade

A conectividade de um *gateway* ITG diz respeito a sua capacidade de conexão com as interfaces de rede disponíveis: cabeada (conexão à rede Ethernet) ou sem fio (conexão 3G ou 4G via modem). Esta seção explora os conceitos de redes móveis sem fio e rede Ethernet. Além disso, também aborda RPC, uma tecnologia utilizada neste teste.

4.4.2.1 Redes móveis

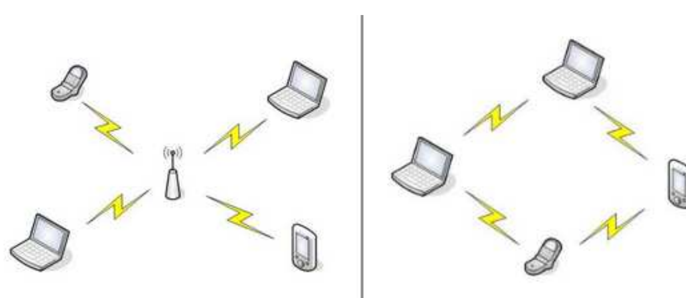
De forma bem simples, uma tecnologia é denominada ”móvel“ quando pode ser utilizada por um usuário em movimento. Uma rede móvel é projetada para permitir que um ou mais dispositivos móveis possam se comunicar. Além de permitir a mobilidade dos equipamentos, redes móveis também são flexíveis e representam uma diminuição nos custos de infra-estrutura (BEZERRA, 2009).

Ainda segundo (BEZERRA, 2009), estas redes podem ser classificadas entre infraestruturadas e ad hoc. No caso da infraestruturada, há a presença de um ponto de acesso (estação base) responsável por intermediar a comunicação entre todos os dispositivos. Ou seja, nessas redes, os equipamentos não são capazes de estabelecer comunicação direta entre si. Este tipo de rede móvel pode ser facilmente exemplificado com o funcionamento de telefones celulares, por exemplo, que demandam uma torre

de telefonia como ponto de acesso para outros aparelhos. GSM (2G), UMTS (3G) e LTE (4G) são exemplos de redes móveis infraestruturadas.

Já as redes móveis ad hoc não apresentam o elemento central ”ponto de acesso“. Essas redes são caracterizadas pela comunicação direta entre os aparelhos e cresceram bastante nos últimos anos com o avanço de padrões de protocolo como *bluetooh* e wi-fi. Por exemplo, através do *bluetooth* é possível conectar um aparelho celular, a um *smart watch* e a fones de ouvido formando uma rede de curto alcance conhecido por *Personal Area Network* (PAN) (BEZERRA, 2009).

Figura 28 – Representação de rede móvel infraestruturada e ad hoc.



Fonte – (BEZERRA, 2009).

A Figura 28 mostra, à esquerda, um exemplo de rede móvel sem fio infraestruturada, com a presença de um ponto de acesso para a comunicação entre dispositivos. À direita da figura, há um exemplo de rede que utiliza topologia ad hoc, permitindo troca de informações direta entre aparelhos.

4.4.2.2 Rede Ethernet

A rede Ethernet consiste em um barramento onde múltiplos computadores compartilham um único meio de transmissão. É um sistema que interliga dispositivos fisicamente, permitindo o compartilhamento de arquivos e recursos entre eles. Segundo (TANENBAUM; WETHERALL, 2011), é provavelmente a rede de computação mais utilizada no mundo.

Há dois tipos de Ethernet: a clássica que resolve problemas de múltiplo acesso, e a comutada que é utilizada atualmente. Na Ethernet comutada, *switches* são utilizados para conectar diferentes computadores (TANENBAUM; WETHERALL, 2011).

A Ethernet clássica é a forma original, que operava a uma velocidade de 3 a 10Mbps. Foi criada por Bob Metcalfe e David Boggs em 1976. Eles utilizaram um cabo coaxial grosso para conectar as máquinas criadas pelos pesquisadores do Xerox PARC, um centro de pesquisa em Palo Alto, que funcionavam de forma isolado. Assim foi criada a primeira rede local de computadores. O nome ”Ethernet“ foi dado

pelos criadores em referência ao éter transmissor de luz (*luminuferous ether*). À época acreditava-se que a radiação eletromagnética se propagava por este meio (TANENBAUM; WETHERALL, 2011).

A Ethernet comutada seria a evolução da clássica, operando a velocidades bem mais altas de 100, 1000 e 10.000Mbps. Na comutada, não há mais a arquitetura de cabo único (o éter), aqui cada estação tem um cabo único dedicado que é conectado a um *hub* central ou *switches* (TANENBAUM; WETHERALL, 2011).

4.4.2.3 Chamada de Procedimento Remoto

Em um mecanismo tradicional de chamada de um procedimento (por exemplo, um método ou função), quem chama (o *caller*) e quem recebe a chamada (*callee*) pertencem à mesma unidade de um programa, ou seja, estão na mesma *thread* de execução. Essa característica se mantém presente mesmo em linguagens que suportam a implementação de múltiplas *threads* (SOARES, 1992).

Em aplicações distribuídas baseadas em *Remote Procedure Call* (RPC) - as chamadas de procedimento remoto - a comunicação *interthread* é permitida. Mesmo quando *caller* e *callee* estão inseridos em contextos completamente diferentes, as *threads* trocam informações através de parâmetros passados para identificação do procedimento desejado. Uma chamada de procedimento remoto pode ser feita entre *threads* de um mesmo programa e até entre unidades executadas em máquinas diferentes (SOARES, 1992).

4.5 INTERFACE *WEB*

A interface *web* auxilia o usuário na utilização do sistema. Esta seção trata conceitos e tecnologias importantes para a construção de uma interface simples e eficiente.

4.5.1 HTTP

Hypertext Transfer Protocol (HTTP) é um protocolo de aplicação simples que roda sobre TCP e é utilizado para o transporte de informações entre servidores e clientes *web*. Atualmente, (TANENBAUM; WETHERALL, 2011) cita que o HTTP também vem sendo amplamente utilizado como um protocolo de transporte para comunicação de informações entre limites de diferentes redes, neste caso não envolvendo necessariamente um servidor *web* ou ou navegador.

Servidores web utilizam HTTP como principal protocolo de comunicação sendo comumente conhecidos como "servidores HTTP". Estes servidores armazenam dados e os enviam quando requisitados por clientes web. Esta comunicação se dá entre requi-

sições HTTP por parte do cliente e respostas HTTP enviadas pelo servidor (GOURLEY *et al.*, 2002).

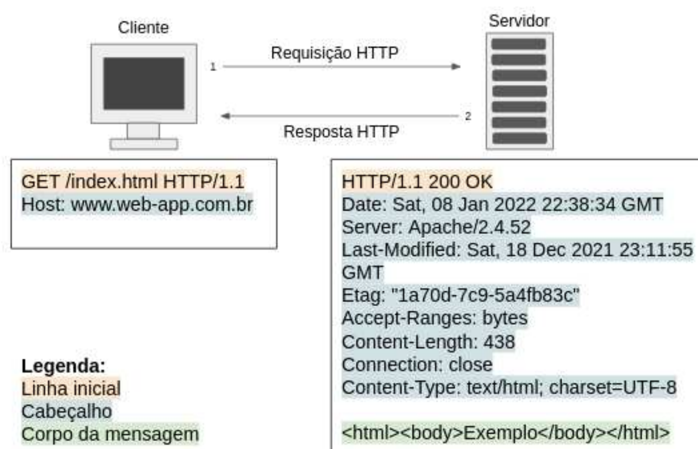
Requisições HTTP podem ser feitas através de diversos métodos suportados pelo protocolo. Métodos HTTP moldam como será feita a comunicação entre cliente e servidor. (TANENBAUM; WETHERALL, 2011) lista oito métodos. A lista a seguir cita a definição de todos os oito segundo os autores, mas este documento citará apenas os métodos GET e POST mais adiante, largamente utilizados na construção da interface web.

- GET - o cliente requisita ao servidor para que este envie um objeto ou página (TANENBAUM; WETHERALL, 2011). O servidor então envia uma resposta HTTP ao cliente com os dados solicitados.
- HEAD - neste caso o cliente solicita apenas o cabeçalho da mensagem, sem a página/objeto propriamente ditos.
- POST - nesta chamada, o cliente envia dados ao servidor, em resposta, o servidor retorna uma página com o resultado da operação.
- PUT - de forma contrária ao GET, o PUT escreve os dados na página passada como parâmetro.
- DELETE - cliente requisita ao servidor a exclusão de uma página.
- TRACE - método bastante utilizado para depuração visto que instrui o servidor a enviar a solicitação de volta.
- CONNECT - permite que um usuário se conecte a um servidor web através de um dispositivo intermediário.
- OPTIONS - permite que o cliente consulte informações sobre uma página, obtendo os métodos e cabeçalhos que podem ser utilizados nela.

Uma mensagem HTTP é composta por uma linha inicial, cabeçalho e corpo da mensagem, sendo que uma linha em branco obrigatoriamente deve sinalizar o fim do cabeçalho. O corpo da mensagem é uma informação opcional.

A Figura 29 ilustra o fluxo que ocorre entre cliente e servidor HTTP quando uma solicitação é feita. Primeiramente, o cliente envia uma requisição de um método GET ao servidor, a mensagem enviada nesta requisição possui uma linha inicial que explicita qual o método solicitado, o endereço da página e qual protocolo utilizado. Em seguida, destacado em azul, há o cabeçalho da requisição, que informa qual o *host* desejado. Esta mensagem não possui corpo, mas fica explícita a linha em branco logo após o cabeçalho, indicando o fim do mesmo.

Figura 29 – Requisição e resposta HTTP entre cliente e servidor.



Fonte – Arquivo Pessoal, 2022.

Ao receber a requisição e estando todos os dados corretos, o servidor enviará de volta ao cliente uma resposta HTTP. Na imagem, destacado em laranja, a linha inicial da resposta mostra novamente a versão do protocolo e agora o status da requisição, neste caso o código 200 indica que ocorreu tudo certo. Logo depois, o longo trecho em azul contém as informações sobre a página que está sendo retornada compondo o cabeçalho, seguido da linha em branco indicando seu término. Por fim, em verde, o corpo da mensagem com o código HTML que descreve provavelmente uma página em uma aplicação web.

4.5.2 *Sigle Page Web Application*

Aplicações web dinâmicas atualizadas em tempo real sem atualização de página, com interfaces escaláveis de alto desempenho estão dominando cada vez mais o mercado de desenvolvimento web com uma nova maneira de desenvolver conteúdo web, chamada aplicação web de página única ou *Single Page Web Application* (SPA) (MONTEIRO, 2014).

SPA é um aplicativo da web, ou site, que cabe em uma única página da web com o objetivo de fornecer uma experiência de usuário mais fluida e uma interface rica. Além disso, apenas uma página não está limitada a apenas um arquivo. Podemos ter muitos modelos em muitos arquivos diferentes. A principal característica deste tipo de aplicação web é poder atualizar partes de uma interface sem enviar ou receber uma requisição de página inteira. Em aplicativos de várias páginas, há o recarregamento de página em cada requisição, tornando a navegação mais lenta. Produtos como Gmail (Figura 30), Trello e Outlook são exemplos de desenvolvimento bem sucedido de SPA (MONTEIRO, 2014).

Figura 30 – Captura de tela do Gmail, um exemplo de aplicação SPA.

Fonte – Arquivo pessoal, 2022.

4.5.3 *Frameworks* reativos

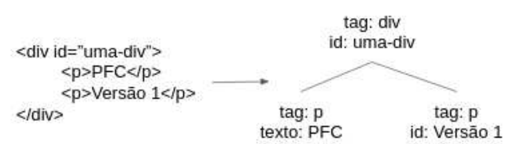
Programação reativa, segundo (BAINOMUGISHA *et al.*, 2013), é um paradigma de programação baseado na noção de valores contínuos variantes no tempo com propagação de mudança. Esta abordagem facilita o desenvolvimento de aplicações orientadas a eventos já que o programador se preocupa apenas com as funcionalidades do software, deixando para que linguagem e *frameworks* gerenciem as mudanças de estado do sistema de forma automática.

Frameworks são ferramentas e bibliotecas que auxiliam desenvolvedores a programar de forma mais simples e ágil, implementando funcionalidades, comandos e estruturas que reduzem o escopo de preocupações que um desenvolvedor deve ter ao projetar um software. Do ponto de vista de aplicações web, *frameworks* reativos são responsáveis por manter elementos de documentos HTML e XML atualizados de forma dinâmica e automática sem que o programador precise interagir diretamente com o *Document Object Model* (DOM) (MOUSQUER, 2021).

DOM é uma interface para desenvolvimento e programação de aplicações em documentos XML ou HTML. Como uma plataforma cruzada, o DOM fornece uma interface padrão em diferentes ambientes e aplicativos, que pode ser implementada em qualquer linguagem (XU, 2021). Com DOM é possível acessar e atualizar dinamicamente o conteúdo de documentos e objetos exibidos por um navegador de web. Como HTML e XML possuem hierarquia, o DOM processa estes documentos como se fossem árvores (MOUSQUER, 2021).

Ao iterar sobre um documento HTML, por exemplo, uma árvore é criada para representar este documento (Figura 31). Quando alguma parte do documento é alterada, os nodos da árvore também serão alterados (MOUSQUER, 2021), mantendo a aplicação web atualizada em tempo real.

Figura 31 – Representação em árvore de um trecho de documento HTML.



Fonte – Adaptado de (MOUSQUER, 2021).

5 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

Este capítulo trata de tecnologias e ferramentas utilizadas para desenvolvimento do projeto e também descreve o processo e critérios de escolha quando se aplica.

5.1 LINUX EMBARCADO

Antes de discorrer sobre a escolha de ferramentas é importante entender o contexto geral do ambiente em que o projeto será executado. A placa central do ITG possui tem um *kernel* embarcado para execução da aplicação *core* do *gateway*. Por já ser uma característica "nativa" do produto, todo o projeto do *firmware* de testes foi pensado no fato de que a aplicação seria executada em um sistema operacional Linux embarcado.

Um software embarcado, ou incluído, é uma categoria de softwares executados em máquinas que não são de fato um computador, por exemplo, servem para sistemas microprocessados em aviões, máquinas industriais, brinquedos, telefones, entre outros. A expressão "embarcado" sugere exatamente que o software é executado sobre um sistema microprocessado interno, ou seja, embarcado em um sistema maior (STADZISZ; RENAUX, 2007).

"Linux" pode fazer referência tanto ao "*kernel linux*" ou ao sistema operacional que é executado com base neste núcleo. O *kernel* foi desenvolvido por Linus Torvalds, inspirado no sistema Minix. Possui seu código-fonte aberto à comunidade sob a licença GPL. A partir deste código-fonte, várias distribuições Linux são utilizadas como sistemas operacionais em computadores pessoais e diversos outros sistemas (STALLMAN, 2021). Estas distribuições vêm sendo inseridas em uma grande gama de microprocessadores e microcontroladores.

Não há, de fato, qualquer *kernel* Linux conhecido como "Linux Embarcado". Uma melhor descrição seria uma distribuição Linux, bibliotecas e ferramentas para executar o sistema operacional em um sistema embarcado (MACHA, s.d.).

5.2 DEFINIÇÃO DA *STACK*

Esta seção discorrerá sobre a decisão acerca dos *softwares* e ferramentas para desenvolvimento do *firmware* e interface *web*.

5.2.1 Golang x Python x C++

Inicialmente foram analisadas três possíveis linguagens de programação para o desenvolvimento do *firmware* de testes: Golang, Python e C++. Na Tabela 1 é apresentado um breve comparativo entre elas com os principais fatores que influenciaram na decisão.

Tabela 1 – Comparativos entre as linguagens de programação Golang, Python e C++.

	Golang	Python	C++
Complexidade da sintaxe	Médio	Fácil	Difícil
Domínio prévio	Nenhum	Bom domínio	Pouco domínio
Suporte à concorrência	Alto	Sem suporte nativo	Sem suporte nativo

Fonte – Autoria própria, 2022.

”Complexidade da sintaxe“ está relacionado ao quão difícil é ler um código implementado em determinada linguagem de programação. Por exemplo, para uma pessoa leiga em Lua - linguagem de programação criada no Brasil - mas com certo domínio de lógica de programação no geral, quanto esforço ela precisa empregar para compreender o que um trecho de código implementado em Lua faz?

Já o campo ”Domínio prévio“ é subjetivo e completamente pessoal, representando o quanto de experiência e segurança eu possuía para desenvolver código nas três linguagens. Por último, e extremamente importante para projeto levando em conta o objetivo de redução massiva de tempo de execução do processo de testes, foi analisado o suporte à programação concorrente das três linguagens, se este suporte é nativo e inerente à linguagem ou se é necessário o uso de bibliotecas e pacotes.

Um quarto atributo, relacionado a performance também foi analisado de acordo com a Figura 32. Nela, são mostrados os tempos de execução de três programas distintos escritos em cada uma das três linguagens.

Figura 32 – Tempo de execução de três programas em Go, Python e C++.

	Go	Python	C++
fasta	1.26 secs	37.32 secs	0.77 secs
special-norm	1.43 secs	120.99 secs	0.72 secs
n-body	6.38 secs	567.56 secs	2.12 secs

Fonte – https://codilime.com/blog/go-vs-python-and-cpp-main-differences/

Traçando um paralelo com os objetivos definidos para o projeto, a linguagem GO foi escolhida por ser mais performática que Python e ao mesmo tempo mais simples que C++. Por opinião pessoal, a opção escolhida foi usar GO por ser mais vantajosa a

relação performance/complexidade da sintaxe em comparação com as demais, o que permitiria a cumprimento das especificações de projeto sem muitas dificuldades de aprendizado da *stack*.

Além disso, para alguns pontos específicos do projeto foram necessários *scripts* em python, mas é importante frisar que são realmente trechos bem específicos e que em praticamente nada impactam no tempo de execução do código, não comprometendo o objetivo de tempo máximo de execução, principal justificativa para o uso de Golang.

5.2.2 Golang

Golang é uma linguagem de programação criada pela empresa Google com a contribuição de muitos desenvolvedores da comunidade *open source*. Comumente chamada apenas de ”Go“, é uma linguagem pensada especialmente para concorrência.

Go não é orientada a objetos como JAVA e C++, não internalizando conceitos como classes e herança. Porém, apresenta estruturas como interfaces, permitindo polimorfismo.

A construção de um código Golang se dá principalmente pelas funções, além disso apresenta aspectos fundamentais de uma linguagem funcional como funções lambda (ZHHUTA, 2015).

5.2.2.1 Concorrência em Golang

O principal motivo para a escolha da linguagem Golang foi seu alto nível de suporte à programação concorrente, tornando a atividade de desenvolvimento de código mais simples e intuitiva. Para entender o porquê de a linguagem ter sido escolhida para compor o *core* do projeto e também a arquitetura geral do *firmware* é necessário entender os elementos que compõem o suporte à concorrência na linguagem.

Um conceito bastante familiar da programação concorrente é o de ”*threads*“. Uma *thread* consiste em uma cadeia de processos que serão executados ordenadamente de acordo com determinado comando. Cada *thread* executa apenas uma linha de processos por vez, não permitindo o processamento paralelo real de mais de um comando simultaneamente. Porém, um sistema multitarefas permite que uma CPU, mesmo com apenas um núcleo (*thread*), **simule** um paralelismo de tarefas ao alternar a execução dos diferentes processos em uma fila, dando uma sensação de que há várias *threads* rodando simultaneamente, quando há apenas uma.

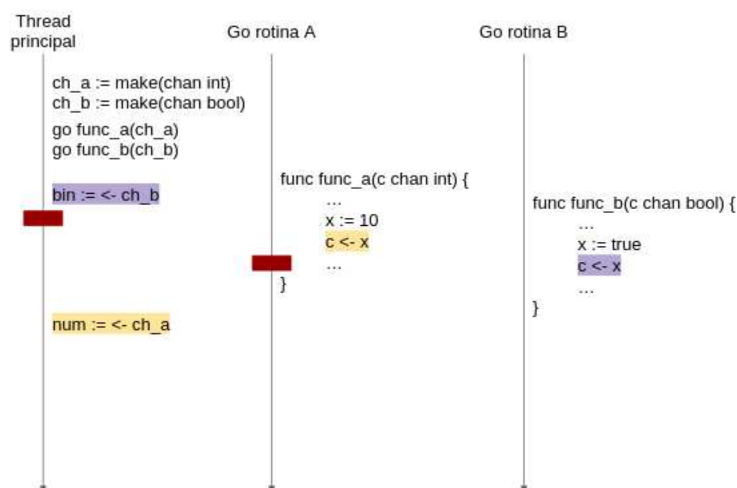
Em Golang, as *threads* são chamadas de ”*goroutines*“ - ou go rotinas. Em relação a outras linguagens do programação, é bastante simples declarar uma nova *thread* em Go, basta utilizar a *keyword* ”go“. Ao inserir ”go“ antes da chamada de uma função em Golang, o compilador Go entende que todas as tarefas englobadas

por aquela função devem ser executadas paralelamente com a *thread* principal do programa. Desta forma, o processador irá alternar a execução de tarefas da *thread* principal e as demais go rotinas declaradas.

Porém, um problema bastante comum de ocorrer é a finalização da *thread* principal antes do processamento de uma go rotina. Isso se dá pois go rotinas, a princípio, são independentes entre si, incluindo a go rotina principal. Dessa forma, nada impede a *thread* principal de encerrar a execução do programa sem a conclusão de outras *threads*.

Para resolver esse problema, a ferramenta utilizada em Go para garantir a comunicação entre *threads* é a implementação de "*channels*". Um *channel* é um ponteiro com um *mutex*. São tipados, ou seja, na declaração de um *channel*, deve-se definir qual o tipo de variável será informada pelo mesmo. A Figura 33 ilustra o comportamento de go rotinas que comunicam-se através de um *channel*.

Figura 33 – Comunicação entre go rotinas através de go *channels*.



Fonte – Arquivo Pessoal, 2022.

No exemplo ilustrado, há três go rotinas sendo executadas paralelamente, a *thread* principal e as go rotinas A e B. A *thread* principal deseja ler informações de um channel em dois momentos. No primeiro deles, a informação desejada é esperada pelo *channel* ch_b, como este *channel* está vazio em um primeiro momento, a *thread* principal é bloqueada até que alguma informação seja escrita em ch_b. Enquanto isso, as outras duas *threads* (A e B) seguem sendo executadas, na expectativa de alguma delas envie dados através de ch_b. Seguindo a linha temporal, em sequência, a go rotina A faz uma operação de escrita em ch_a. Logo após esta operação, a *thread* A é também bloqueada até que alguma outra go rotina leia as informações enviadas por ch_a. Neste ponto, temos duas *threads* bloqueadas: a principal aguardando a chegada

de dados através de ch_b e a go rotina A aguardando a leitura das informações enviadas em ch_a. Neste ínterim, a *thread* B segue executando. Avançando um pouco mais no tempo, B envia uma informação para o *channel* ch_b, logo após, a *thread* principal é desbloqueada, visto que conseguiu receber os dados que esperava. Um pouco mais adiante, a go rotina principal tenta ler de dados de ch_a. Este *channel* já não está mais vazio, visto que foi realizada uma operação de escrita na *thread* A que até o momento, estava bloqueada aguardando a leitura de ch_a.

Desta forma, com uma forma bastante simplificada para declaração de go rotinas e o uso de *channels* para garantir a comunicação entre *threads*, a linguagem Golang mostra porque é tão visada para aplicações que demandam programação concorrente. Para o desenvolvimento do *firmware* de testes, estes conceitos foram extensamente utilizados a fim de alcançar o objetivo de redução de tempo dos testes de produto.

5.2.3 Python

Python é uma linguagem de programação de propósito geral. É de alto nível, interpretada e orientada a objetos. Possui tipagem dinâmica, exceções e gerenciamento automático de memória, contando com uma sintaxe simples e elegante. A linguagem possui diversas bibliotecas padrão e diversas outras criadas por terceiros oferecem suporte à linguagem (SANNER *et al.*, 1999).

Apesar de não ser selecionada como a linguagem para desenvolvimento do projeto, *scripts* em python foram necessários em determinados pontos do código que demandavam o uso de bibliotecas e ferramentas que não oferecem suporte a Golang.

A seção 5.2.3.1 trata um exemplo de tecnologia utilizada que demandou o uso de python mesmo buscando uma aplicação Golang pura.

5.2.3.1 SWIG

Simplified Wrapper and Interface Generator (SWIG) é uma ferramenta de desenvolvimento de software que conecta programas C e C++ com linguagens de programação de alto nível como Python, Golang e Ruby. De forma geral, SWIG cria um *script* equivalente ”traduzindo“ C/C++ para a linguagem de alto nível desejada (SWIG, 2022).

Para este projeto foi utilizado o SWIG C++-Python para o teste de *display* e também para coletar as informações de placa do *gateway*.

O SWIG gera um programa em uma linguagem pré-definida a partir do arquivo original em C/C++, implementando as mesmas funcionalidades e buscando manter estruturas de dados análogas entre uma linguagem e outra (como classes em C++ e interfaces em Golang). O arquivo gerado, no geral, não é de fácil legibilidade, sendo bastante complexo para se entender, mas para o uso em funcionalidades simples, a

dificuldade de compreensão do *script* gerado não afeta de forma grave o ritmo de desenvolvimento.

5.2.4 *Framework* Svelte para *front-end*

A escolha do *framework* para implementação da interface web foi baseada no *benchmark* realizado por Wenqing Xu (XU, 2021) que faz uma comparação entre os principais *frameworks* JavaScript.

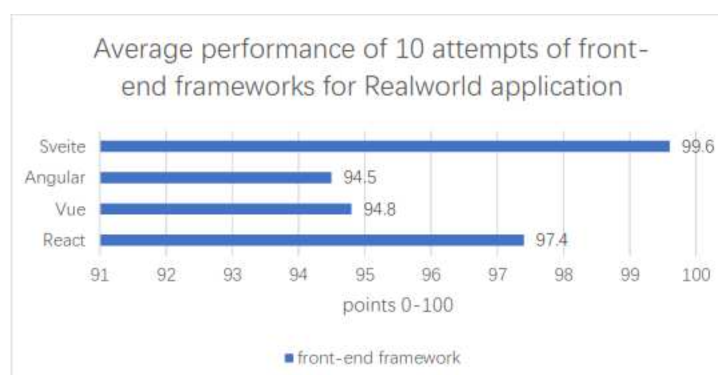
Svelte é uma ferramenta baseada em TypeScript, uma linguagem desenvolvida pela Microsoft que consiste em um superconjunto do JavaScript. Svelte é utilizado para desenvolver aplicações rápidas e possui várias similaridades com outros *frameworks* como o objetivo de construir com facilidade interfaces altamente interativas (XU, 2021).

A principal diferença, que traz um grande destaque ao Svelte, é o fato de converter a aplicação para JavaScript durante o tempo de compilação ao invés de interpretar o código durante a execução. O principal impacto desta forma de implementação é que não há custo algum de performance para a abstração do *framework* e nenhuma penalidade na primeira vez em que a aplicação é carregada (XU, 2021).

O artigo analisa diversos aspectos entre os *frameworks* Svelte, React, Angular e Vue. Duas características foram utilizadas para a tomada de decisão em relação a ferramenta para *frontend*: performance (Figura 34) e simplicidade (Figura 35). Em ambos os casos o Svelte apresentou melhor desempenho frente aos demais.

Performance é uma qualidade que permeia todas as fases deste projeto, sendo natural que a escolha de ferramentas seja feita levando tal propriedade em consideração. Xu definiu como regra para esta análise a média de 10 tentativas de conexão de uma aplicação.

Figura 34 – Comparação entre Svelte e outros *frameworks* em relação à performance.

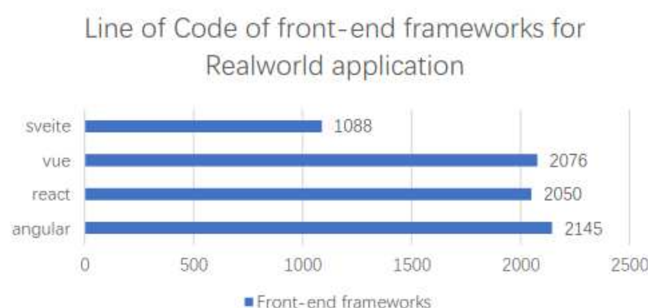


Fonte – (XU, 2021).

Já o critério ”simplicidade“ diz respeito à facilidade de implementação que a ferramenta traz. Uma sintaxe simples que possibilite um desenvolvimento rápido e fluido

permitindo uma curva de aprendizado acelerada é importante. No artigo, o autor utilizou como parâmetro para esta análise a quantidade de linhas de código necessárias para implementar uma mesma aplicação.

Figura 35 – Comparação entre Svelte e outros *frameworks* em relação ao número de linhas de código.



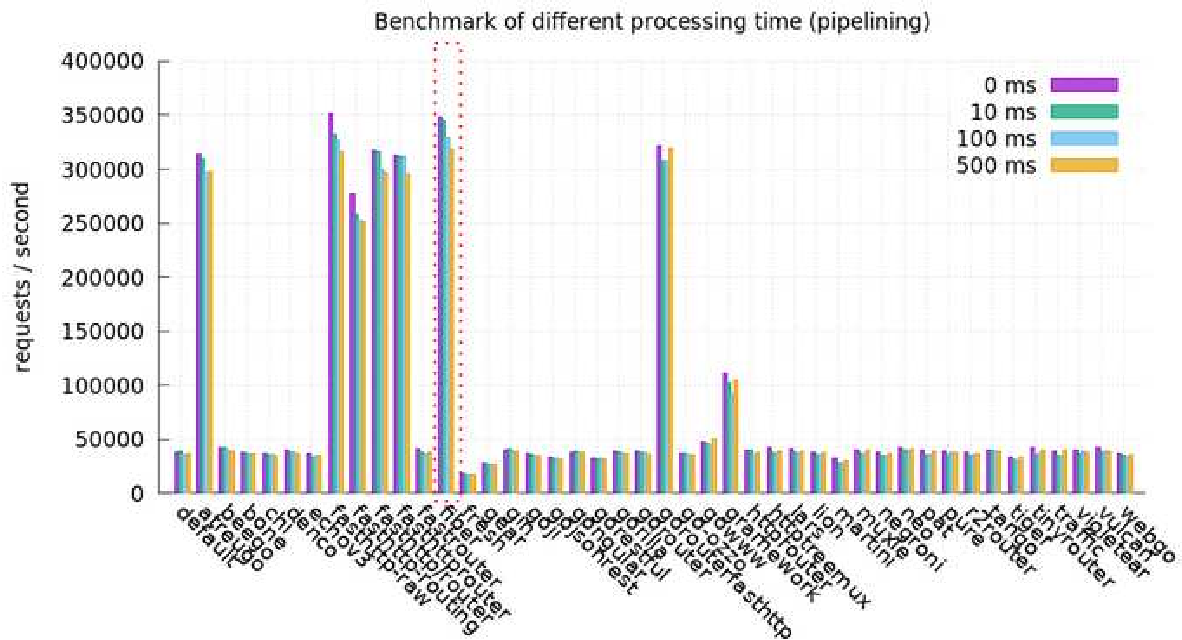
Fonte – (XU, 2021).

5.2.5 Go Fiber

Fiber é um *framework web* para Golang baseado em Fasthttp, mecanismo HTTP mais rápido para Go. A proposta da ferramenta é possibilitar um desenvolvimento rápido e sem preocupações quanto à alocação de memória e performance (FIBER, 2021).

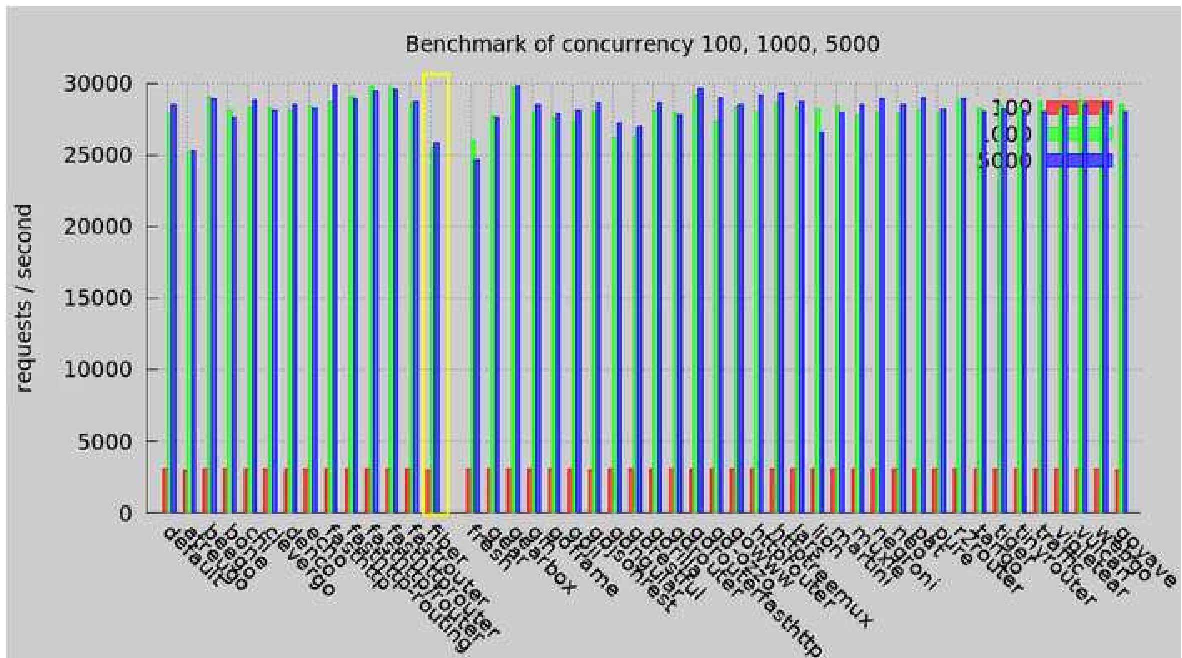
A escolha do Go Fiber foi baseada em um levantamento feito por vários desenvolvedores independentes que compara diversos *frameworks web* para Golang. O comparativo leva em conta todo o processo englobado em uma requisição HTTP (VÁRIOS, 2020). Para a decisão foram escolhidos dois testes, alinhados com o objetivo de desempenho do projeto (Figura 36) e também ao uso de programação concorrente (Figura 37):

Figura 36 – Comparação entre Fiber e outros *frameworks* em relação à performance.



Fonte – (VÁRIOS, 2020).

Figura 37 – Comparação entre Fiber e outros *frameworks* em relação à concorrência.



Fonte – (VÁRIOS, 2020).

5.3 KHARMA

É o sistema da Khomp de compilação de plataformas embarcadas. Para que uma aplicação seja compilada pelo kharma, é necessária a criação de um roteiro que define quais os pacotes necessários para o funcionamento do programa, como estes devem ser baixados e instalados (clone de repositório do git ou então por alguma ferramenta de gerenciamento de pacotes) e também para quais diretórios do Linux embarcado abrigará os novos *scripts*, arquivos de configuração e binários aplicações.

É também através do roteiro do kharma que é definida a inicialização do ambiente operacional como quais aplicações devem começar a execução assim que o dispositivo for ligado, a configuração inicial e etc. Com o uso de CI/CD suportado pelo GitLab, o kharma é capaz de automaticamente selecionar as versões mais atualizadas de aplicações internas da Khomp para *build*.

O resultado do processo de compilação do kharma é um arquivo .bin com todos os pacotes necessários compactados pronto para execução em alguma placa com a arquitetura especificada para cross compilação.

5.4 FERRAMENTAS CORPORATIVAS

Nesta seção serão brevemente apresentadas as ferramentas corporativas utilizadas pela Khomp para gestão de projetos e desenvolvimento.

5.4.1 GitLab

GitLab é a plataforma de controle de versão de códigos git utilizada pelas equipes de desenvolvimento da Khomp. Todos os códigos escritos na empresa, incluindo novas funcionalidades, correção de *bugs* e melhorias internas são versionados pelo GitLab. Antes de um trecho de código ser oficialmente incorporado à versão final da aplicação, este é submetido à revisão de outro desenvolvedor da equipe.

O GitLab também oferece suporte a *Continuous Integration/Continuous Delivery* (CI/CD), ferramenta utilizada no processo de *build* das aplicações e apresentada abaixo.

5.4.1.1 CI/CD

CI/CD é uma abordagem utilizada para automatização da geração de versões toda vez que um código-fonte é atualizado. Para que o processo funcione, é necessária a integração com uma ferramenta para gerenciamento de código fonte como o GitLab (SINGH *et al.*, 2019).

CI é um conjunto de práticas que relacionadas a princípios de desenvolvimento de software. Afirma-se que todo o código escrito para uma aplicação deve ser con-

centrado em um repositório comum para que sempre que houver atualização no código deste, é acionado um *script* responsável por identificar as alterações recentes e integrá-las ao código existente, executando casos de teste projetados de acordo com a aplicação (SINGH *et al.*, 2019).

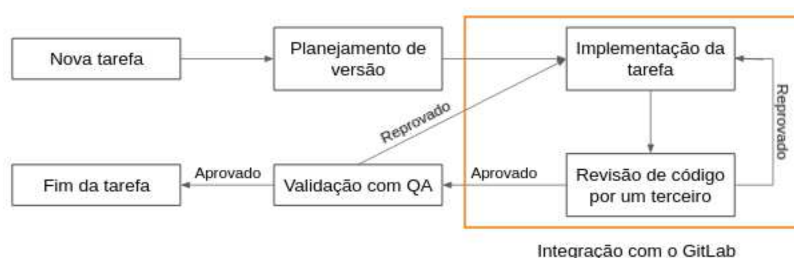
Seguido de um CI bem sucedido acontece o processo de CD onde novos recursos são implementados no momento e na forma que forem requisitados no servidor ativo. O *script* executado durante o processo de CD compila o código do ramo principal da ferramenta de versionamento e a executa em um ambiente de testes/produção, permitindo que desenvolvedores testem a aplicação através da atualização de diversos módulos do sistema(SINGH *et al.*, 2019).

5.4.2 Redmine

Redmine é a ferramente utilizada pela Khomp para gestão de projetos e processos. Todas as tarefas realizadas dentro de um projeto devem ser reportadas na plataforma, independentemente de serem atividades de desenvolvimento de software ou apenas estudo de alguma tecnologia.

O Redmine possui integração com o GitLab, permitindo um acompanhamento mais preciso das atividades por parte dos gestores e equipes de projetos. A Figura 38 mostra um esquemático sobre o fluxo de projetos de software reportados e acompanhados na ferramenta.

Figura 38 – Ciclo de vida de uma tarefa no Redmine - GitLab.



Fonte – Arquivo Pessoal, 2022.

6 PROJETO

Este capítulo aborda o desenvolvimento do sistema. Também são explorados assuntos como planejamento do projeto, metodologias utilizadas e engenharia de software. Inicialmente é explicado como as metodologias exploradas no Capítulo 4 foram implementadas para o desenvolvimento deste projeto. Em seguida é apresentada a arquitetura do *firmware* de testes. Na seção de testes é detalhado cada módulo de teste realizado pelo *firmware*. Todos possuem uma estrutura bastante metódica:

- **Mapeamento**: detalha o módulo testado quanto à sua funcionalidade (qual o comportamento esperado?) e também identifica condição para aprovação e possíveis motivos de falhas;

- **Estrutura do teste**: apresenta pseudo-código e/ou diagramas do teste proposto para validação do módulo. A estrutura do teste é feita de acordo com as informações descritas na seção de mapeamento;

Por fim, o projeto da interface do usuário e o roteiro de compilação para o kharma são apresentados;

6.1 METODOLOGIA DE PROJETO

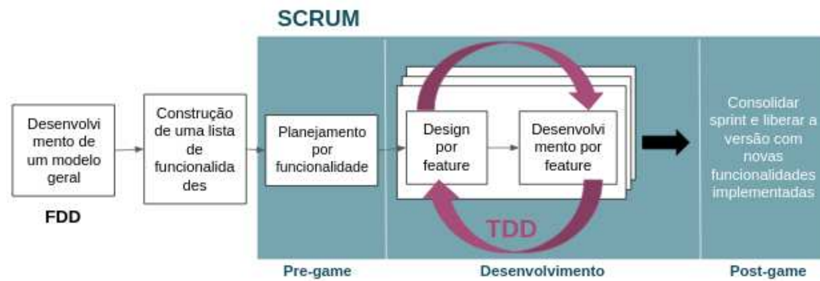
O *firmware* foi desenvolvido com o uso de metodologia ágil, buscando fazer pequenas entregas que agregassem algum valor à aplicação final do ponto de vista do usuário.

Dentre os vários métodos ágeis descritos na literatura e utilizados no mercado de tecnologia, este projeto foi desenvolvido utilizando uma combinação de SCRUM e FDD, visto que se mostrou a mais adequada para o perfil do projeto e prazo de entrega. Tal combinação busca trazer o foco do SCRUM no gerenciamento do projeto enquanto o FDD pauta o desenvolvimento do software propriamente dito.

Em conjunto com estes dois métodos que ditam o processo geral de desenvolvimento, foi utilizada uma metodologia orientada a testes dentro das *sprints* para garantir a qualidade do software durante o desenvolvimento do *firmware* e da interface web.

A Figura 39 mostra um esquemático de como foi utilizada a união das três metodologias para o desenvolvimento do projeto. O ciclo produtivo em si é pautado utilizando FDD, desde o projeto inicial do software como levantamento de requisitos. O SCRUM é responsável por ditar o gerenciamento das tarefas do projeto, definindo prioridades e cronogramas para as entregas, consolidando o que foi implementado no fechamento de um ciclo FDD. E dentro do processo de desenvolvimento, a qualidade do software é garantida utilizando TDD, possibilitando uma evolução mais fluida do software e mitigando riscos de pequenos *bugs* que poderiam causar retrabalho desne-

Figura 39 – Combinação entre SCRUM, FDD e TDD utilizada no desenvolvimento do projeto.



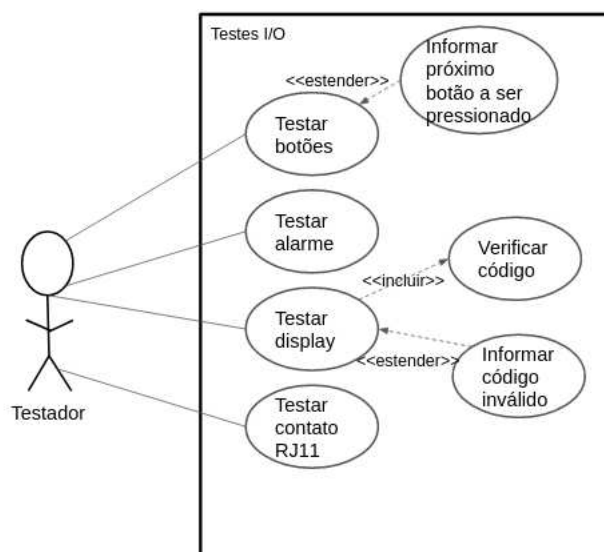
Fonte – Arquivo pessoal, 2022.

cessário. No apêndice deste documento constam os pseudocódigos de testes unitários implementados.

6.2 CASOS DE USO

O caso de uso a seguir (Figura 40) mostra apenas o cenário referente aos testes de entrada e saída que demandam alguma interação do testador. Os demais testes serão representados por meio de diagramas sequenciais e de atividades já que estas representações ilustram melhor os testes totalmente automatizados.

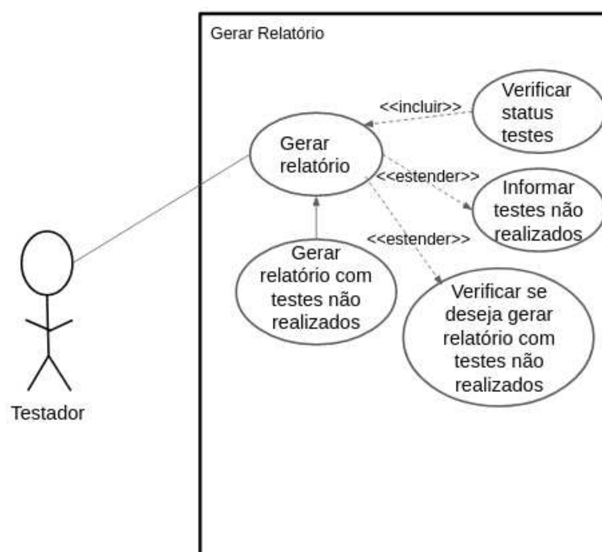
Figura 40 – Casos de uso relacionados ao testes de entrada e saída.



Fonte – Arquivo pessoal, 2022.

Um outro caso de uso importante é gerar relatórios de teste via interface web, mostrado na Figura 41.

Figura 41 – Casos de uso relacionados ao *download* do relatório de testes.



Fonte – Arquivo pessoal, 2022.

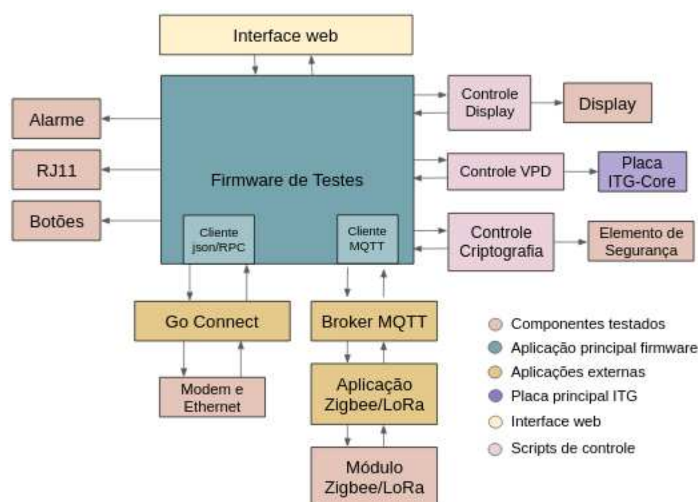
6.3 ARQUITETURA DO *FIRMWARE*

A Figura 42 mostra um esquemático que representa a arquitetura de software feita para o projeto. Alguns módulos de testes são mais complexos que outros e terão suas arquiteturas detalhadas mais à frente neste documento.

Esta arquitetura foi projetada após a fase de mapeamento e estudo de cada um dos requisitos de projeto. Cada módulo de teste é feito de forma concorrente com os demais, em uma *thread* diferente. Alguns inclusive executam *threads* internas.

Em destaque no centro, o *firmware* de testes consiste na aplicação principal em Golang que interage com todos os componentes de teste direta ou indiretamente. Alguns testes demandam que outras aplicações sejam executadas, já outros utilizam um *script* intermediário para controlar e obter informações dos módulos.

Figura 42 – Arquitetura do *firmware* de testes.



Fonte – Arquivo pessoal, 2022.

O *firmware* lida com testes de módulos sem nenhuma interdependência entre eles, o que justifica o uso de concorrência a fim de agilizar o processo buscando reduzir ao máximo o tempo de execução. Para uma aplicação em Golang, a forma canônica para concorrência é o uso de go rotinas e *channels*. A estrutura do *software* baseia-se em um único *channel* que centraliza todas as informações de testes geradas nas inúmeras *threads* do programa, visto que cada módulo é testado em sua própria go rotina.

6.4 IDENTIFICAÇÃO DE DISPOSITIVO

O *firmware* será utilizado para realização de testes T2 de duas linhas de produto dentro dos *gateways* IoT da Khomp: o ITG 200 e o ITG 300.

Para fazer esta diferenciação, se utiliza um *script* em python gerado através do SWIG C++. Este código coleta os dados vitais do produto (VPD) como número e ano da serial, nome do produto e da placa base.

É a informação ”nome do produto“ que identifica a versão da placa principal do ITG. Como explicado no Capítulo 2, uma das principais diferenças entre os ITGs 200 e 300 é a placa *core* do produto. Dessa forma, a detecção do modelo em teste é feita a partir desse dado.

Para a etapa de teste e gravação de elemento seguro, o resultado deve ser interpretado de maneira diferente dependendo de qual dispositivo está sendo testado. ITGs 300 e qualquer ITG LoRa que tenha suporte à rede EveryNet saem com o elemento de segurança gravado de fábrica, sendo que é obrigatória a aprovação deste módulo no teste.

Já para os casos de ITGs 200 Zigbee ou LoRa (sem suporte à EveryNet) o elemento se torna opcional. Desta forma, caso o dispositivo ”reprove“ no teste ao não detectar, o resultado não é visto como uma falha.

6.5 TESTES

De acordo com as terminologias levantadas no Capítulo 4, o *firmware* irá substituir um processo de **teste** que busca **validar** que componentes essenciais do sistema não apresentam **falhas** no seu funcionamento. Pode-se dizer que, em uma visão mais alto nível, o *firmware* de testes é como um *teste estrutural* composto por diversos módulos de *testes funcionais* que buscam validar cada elemento.

A implementação de cada funcionalidade (módulo de teste) foi feita de acordo com a priorização definida durante a fase de construção da lista de tarefas - segunda etapa do ciclo FDD. As tarefas foram ordenadas na lista de prioridade analisando a função que cada módulo desempenha no *gateway*, *hardwares* com uma funcionalidade mais crítica para o funcionamento do ITG foram listados nas primeiras colocações:

1. *Slots* Zigbee e LoRa;
2. Conectividade via modem e Ethernet;
3. Display;
4. Botões;
5. Alarme;
6. RJ11;
7. Elemento de Segurança.

Ainda foram consideradas as tarefas de criar o roteiro para compilação do *firmware* através do kharma e validar o novo *firmware* junto ao time de produção.

6.5.1 Módulos de comunicação sem fio

Como já citado em outras seções anteriores deste documento, os *IoT gateways* da Khomp suportam duas tecnologias para comunicação sem fio com sensores: Zigbee e LoRaWAN.

O teste deste componente é o mais importante do *firmware* visto que trata da função mais básica do *gateway*: garantir a comunicação entre dispositivos que utilizam diferentes protocolos.

6.5.1.1 Mapeamento

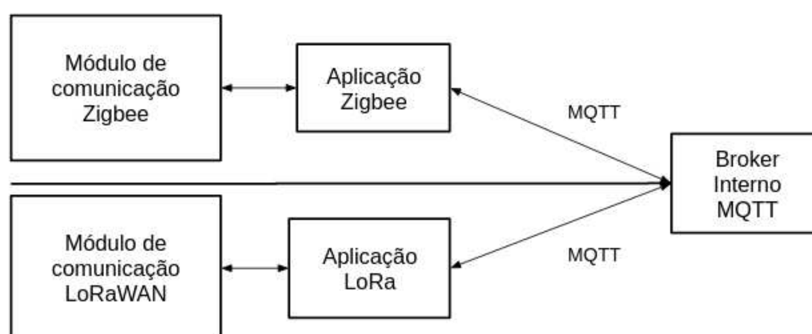
Para os módulos de comunicação sem fio têm-se:

- **Função:** receber mensagens de dispositivos que utilizam tecnologia de comunicação compatível (Zigbee ou LoRa) repassando-as ao *gateway*;
- **Condição de aprovação:** conseguir detectar o recebimento de alguma informação por parte do *gateway* vinda do *slot* Zigbee ou LoRa;
- **Possíveis falhas:** não localizar os arquivos de execução das aplicações Zigbee/-LoRa e não receber nenhuma informação de dispositivo externo em determinado período de tempo.

6.5.1.2 Estrutura do teste

Para propor um modelo de teste para *slots* Zigbee ou LoRa, foi necessário estudar um pouco mais a fundo o contexto destes elementos na arquitetura interna do *gateway*. A Figura 43 mostra o caminho que uma informação enviada por *endpoint* percorre dentro do ITG.

Figura 43 – Arquitetura interna da comunicação LoRa e Zigbee com o Broker MQTT do ITG.



Fonte – Arquivo pessoal, 2022.

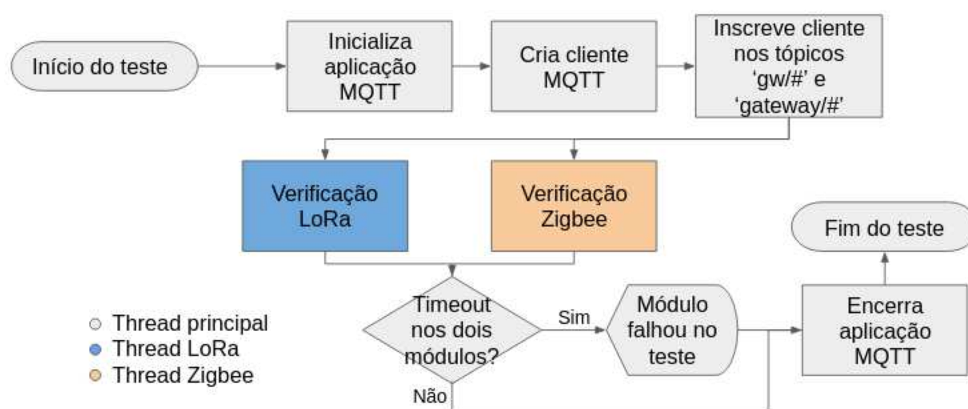
A aplicação *core* do ITG, por intermédio de outras aplicações, recebe mensagens de *endpoints* e também transmite informações para estes através do protocolo MQTT.

No caso do Zigbee, o *slot* se comunica via serial com uma aplicação de rede Zigbee que troca mensagens MQTT com o *Broker* interno do *gateway*. O módulo LoRaWAN envia dados para um gerenciador de pacotes LoRa que, através de uma ponte UDP/MQTT, se comunica com o *Broker*.

Apesar de não haver interação direta com os módulos de comunicação sem fio, é possível afirmar que uma forma de validar este componente é averiguar o recebimento de mensagens no Broker interno.

Testar os módulos de comunicação sem fio consiste em identificar qual o módulo utilizado pelo *gateway* em questão, LoRa ou Zigbee, e confirmar que o Broker Interno do ITG recebe mensagens nos tópicos padrão ’gw/#’ para Zigbee e ’gateway/#’ para LoRaWAN. O teste depende diretamente da existência de um cliente MQTT inscrito nestes tópicos, para isso é necessário uma aplicação MQTT em execução. A Figura 44 ilustra o esquemático do teste sem ainda entrar em detalhes quanto a implementação de cada *thread*.

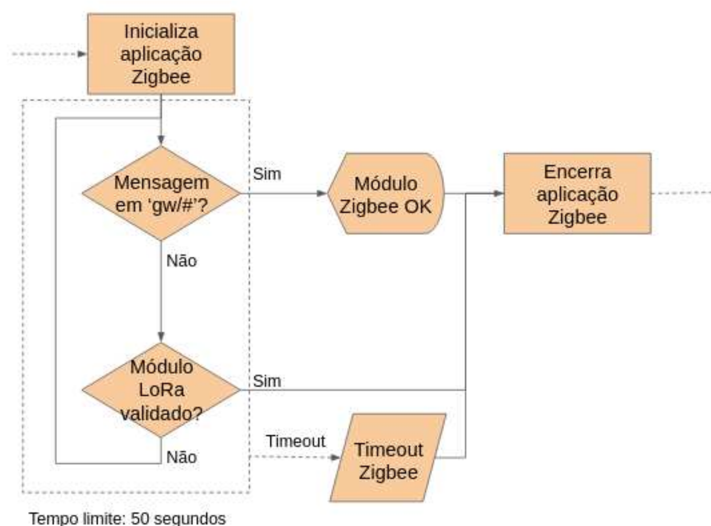
Figura 44 – Diagrama de atividades generalizado do testes de módulos de comunicação.



Fonte – Arquivo pessoal, 2022.

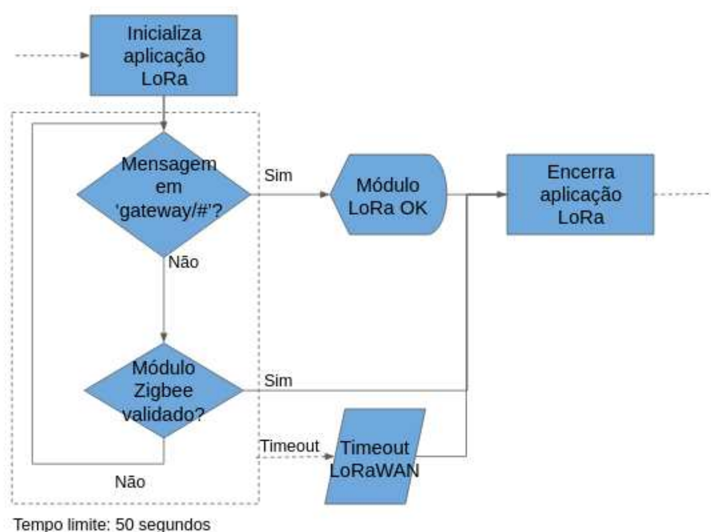
Além disso, buscando o objetivo de tempo máximo de 60 segundos de todo o processo de validação do produto, é necessário definir um período limite razoável para a espera de mensagens, sendo 50 segundos o tempo máximo definido para espera de mensagens nos tópicos. As figuras 45 e 46 mostram os diagramas de atividade para o teste proposto de *slots* Zigbee e LoRa, respectivamente.

Figura 45 – Diagrama de atividades para validação do *slot* Zigbee.



Fonte – Arquivo pessoal, 2022.

Figura 46 – Diagrama de atividades para validação do *slot* LoRa.



Fonte – Arquivo pessoal, 2022.

Após o teste, as aplicações Zigbee, LoRa e MQTT não serão mais utilizadas e poderão ser encerradas.

6.5.2 Testes de conectividade

Por um lado, um *gateway* IoT serve para comunicações com protocolos IoT como LoRaWAN e Zigbee. Porém, é característica fundamental também ser capaz de

se conectar a internet. Os testes de conectividade via modem e Ethernet devem validar a capacidade do ITG de utilizar os protocolos da internet para se comunicar com a rede.

6.5.2.1 Mapeamento

Para os testes relacionados à conectividade do *gateway* têm-se:

- **Função:** o modem e o cabo Ethernet devem garantir a comunicação do *gateway* com os protocolos da internet (TCP, UDP, IP).
- **Condição de aprovação:** detectar conexão com a internet via modem e também por interface cabeada.
- **Possíveis falhas:** impossibilidade de uma ou ambas interfaces de se conectar à internet.

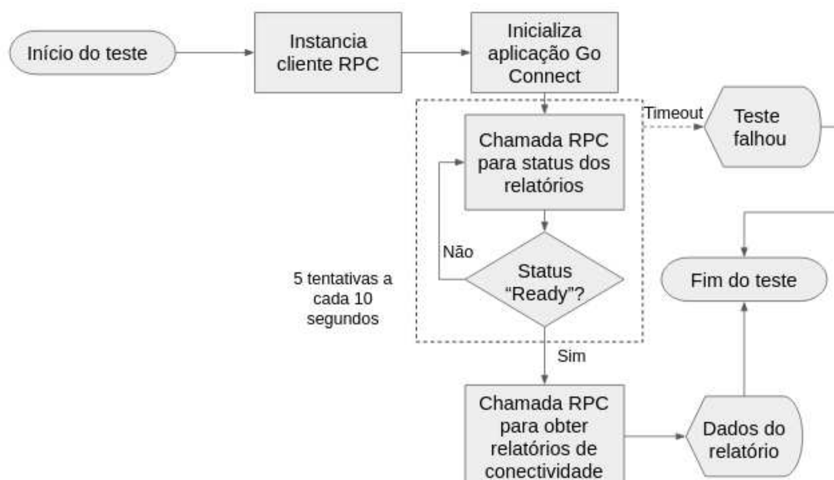
6.5.2.2 Estrutura do teste

Este módulo de testes tem como objetivo avaliar as interfaces de rede do *gateway*: se há ou não modem e se este está conectado, verificação do cartões SIM (de 1 a 2), qualidade do sinal (RSSI) e também analisa o ping TCP e ICMS do modem. Além disso, também avalia a interface de rede cabeada (Ethernet) analisando ping TCP e ICMS.

Os teste de conectividade foram realizados utilizando uma aplicação para gerência de conectividade criada na Khomp, o Go Connect.

Para isso, o *firmware* de testes se comunica com o Go Connect através um cliente RPC, chamando os procedimentos do Go Connect que reportam as informações de conectividade do *gateway*. Toda a comunicação com modem e rede cabeada é feita através do gerenciador de conectividade, o *firmware* de testes apenas se preocupa em tratar os resultados obtidos com as chamadas de procedimento remoto e ”interpretá-los“. A Figura 47 mostra o esquema alto nível deste módulo de teste.

Figura 47 – Diagrama de atividades do teste de conectividade.



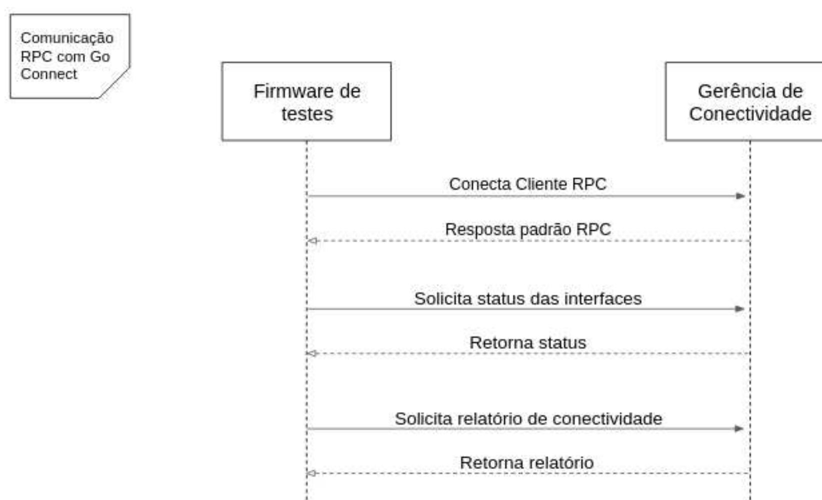
Fonte – Arquivo pessoal, 2022.

O *firmware* de teste cria uma instância de cliente RPC e inicializa o Go Connect. Então, entra em um *loop* aguardando o *status* ”Ok“ do procedimento remoto que indica que o relatório de conectividade do *gateway* está pronto. O *firmware* faz uma nova requisição RPC para obter os dados do relatório e os exibe na interface web.

Assim como no testes de módulos de comunicação, há um tempo limite para que a aplicação receba os dados solicitados. Caso o relatório não esteja disponível em até 50 segundos o teste falha.

A Figura 48 ilustra um diagrama sequencial que representa a comunicação via RPC entre a aplicação do *firmware* de testes e a aplicação Go Connect.

Figura 48 – Diagrama sequencial demonstrando a comunicação via RPC com o Go Connect.



Fonte – Arquivo pessoal, 2022.

6.5.3 RJ11

Os *gateways* da linha ITG possuem duas entradas para conector RJ11, comumente utilizadas para sensoreamento.

RJ11 é um tipo de conector bastante utilizado no ramo de telecomunicações. Faz parte da família dos ”RJ“ (*registered jack*), assim como o cabo Ethernet. Conectores RJ são interfaces físicas padronizadas para conexão de equipamentos de telecomunicação e rede de computadores. O número ”11“ representa a padronização dos pinos do conector.

6.5.3.1 Mapeamento

Para os testes relativos às entradas RJ11 tem-se:

- **Função:** garantir que periféricos possam ser integrados ao *gateway*, inclusive a partir de conexão *1-wire*;
- **Condição de aprovação:** fazer leituras, com e sem *1-wire*, das portas RJ11 que possuem valores condizentes com o esperado dentro de um contexto;
- **Possíveis falhas:** não detectar alteração de estado esperada ou realizar leitura considerada anômala e não localizar os arquivos gerados pelos sensores conectados.

6.5.3.2 Estrutura do teste

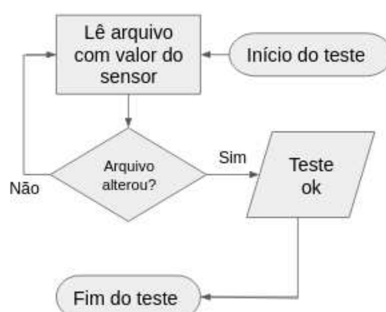
O teste para entradas RJ11 busca validar o funcionamento correto das portas e também da leitura *1-wire* pelo conector. Seguindo o processo T2 já consolidado na empresa que utiliza sensores de contato e temperatura para validar as entradas RJ11, a proposta do *firmware* é manter o uso destes sensores. Desta forma foram realizados dois tipos de testes para RJ11, um com sensor de contato e outro com uma sonda de temperatura, permitindo atestar o *1-wire*.

Para realizar a leitura do sensor de contato, foi utilizada de uma interface *General Purpose Input/output* (GPIO) por meio da entrada RJ11. GPIO é uma interface de pinos programáveis que provê acesso fácil a propriedades de dispositivos. Os pinos podem ser definidos como entradas que recebem dados de alguma fonte externa e os repassam para o sistema (no caso o *gateway*) para serem manipulados. Pinos podem também atuar como saídas de dados, transmitindo informação para outros dispositivos (BALACHANDRAN, 2009).

Devido ao uso de GPIO para o sensor de contato seco, existem dois diretórios no *gateway* com arquivos referentes às propriedades das duas portas RJ11 quando este tipo de sensor está conectado a elas. O *firmware* de testes permite monitorar a todo momento estes arquivos até que seja detectada alteração no valor de uma das portas, indicando que o sensor foi ativado.

O teste reporta falha caso não seja identificada alteração nos arquivos e o testador reporte ao sistema que a ação de contato do sensor foi realizada. A Figura 49 mostra a lógica de negócio implementada para o teste de RJ11 com sensor de contato.

Figura 49 – Diagrama de atividades do teste de leitura do sensor de contato via RJ11.



Fonte – Arquivo pessoal, 2022.

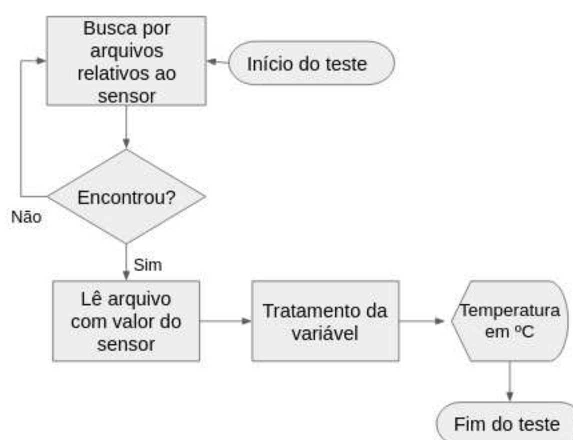
A medição de temperatura é realizada por um sensor de temperatura DS18B20, com interface *1-wire* para envio de leitura. Um dispositivo *1-wire* possui apenas dois cabos, um terra e um para transmissão de dados. Estes sistemas utilizam um diodo retificador e um capacitor para alimentar os dispositivos, desta forma dispensando

fontes externas de alimentação e outros cabos (AWTREY; SEMICONDUCTOR, 1997).

Para cada sensor de temperatura conectado ao ITG - de 0 a 2 unidades - um arquivo com informações, entre elas a leitura de temperatura, é armazenado em um diretório no *gateway*.

O *firmware* de testes, assim que inicializado, irá procurar por estes arquivos e, no caso de encontrar um, o acessará e filtrará a informação de medição de temperatura. Este valor então será tratado para ser exibido na tela da interface de forma legível para o testador. Dada esta sequência, o teste de entrada RJ11 com leitura *1-wire* estará completo e aprovado, como demonstrado na Figura 50.

Figura 50 – Diagrama de atividades do teste de leitura de temperatura *1wire* via RJ11.



Fonte – Arquivo pessoal, 2022.

O teste reporta falha se a leitura obtida pelo sensor estiver muito divergente do esperado, que é próximo da temperatura ambiente.

6.5.4 Display

O display é uma das interfaces do usuário disponibilizadas pelo ITG que possibilita algumas das configurações do produto. O teste da tela é um teste I/O que demanda interação humana para a validação do módulo.

6.5.4.1 Mapeamento

Para o teste de display tem-se:

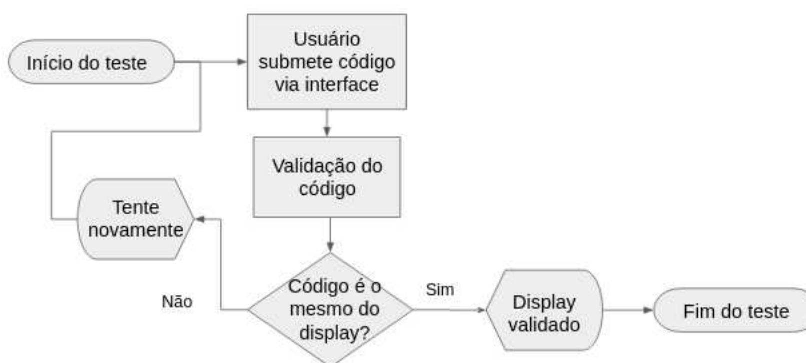
- **Função:** atuar como uma interface para o usuário que permite a **visualização** de informações relativas ao *gateway* e sua configuração.
- **Condição de aprovação:** o display mostra de forma legível as informações pré definidas.

- **Possíveis falhas:** display não mostrar dado algum ou informações diferentes do esperado ou ainda o conteúdo da tela não ser facilmente legível.

6.5.4.2 Estrutura do teste

O teste de *display* deve assegurar que o usuário seja capaz de ler as informações na tela facilmente. Desta forma, a proposta é que o testador deva inserir alguma entrada na interface web que cumpra este critério. Foi optado por um código numérico de 4 dígitos que será mostrado na tela assim que a aplicação for iniciada. O testador deve inserir este código no campo adequado da interface web para validação. Caso o código informado esteja errado, o usuário será avisado e deverá tentar novamente. Esta lógica é ilustrada na Figura 51.

Figura 51 – Diagrama de atividades do teste display.



Fonte – Arquivo pessoal, 2022.

Para manipular o display, foi utilizado um *script* para controle da tela. Este código foi gerado através do SWIG a partir da biblioteca C++ original do display, da mesma forma como é feito para a aplicação final do ITG.

Houve bloqueios nesta parte do desenvolvimento. O SWIG suporta Golang, mas não foi possível utilizar esta funcionalidade devido a necessidade de utilizar CGO, uma biblioteca padrão que permite a criação de pacotes Golang capazes de chamar códigos em C/C++. Porém, o uso de CGO acabaria criando uma camada de complexidade a mais que tornaria todo o desenvolvimento do projeto mais lento (seria necessário ter a *toolchain* da arquitetura disponível - o OMAP3 - para cross-compilar dependências).

Uma opção ao SWIG seria o uso de RPC, fazendo uso da mesma técnica utilizada no teste de conectividade. Porém, foi escolhido manter o uso do SWIG pois esta ferramenta já era utilizada para controle do *display* no produto final, sendo uma solução já consolidada. Além disso, a utilização de chamadas de procedimento remoto adicionaria uma camada de complexidade a mais no projeto, não sendo algo necessário para uma primeira versão. No caso do teste de conectividade, a aplicação utilizada

para coletar as informações - o Go Connect - já implementava uma interface com uma camada de abstração para uso de RPC.

Dado este cenário, optou-se por utilizar o SWIG para gerar o *script* em python, da mesma forma como na aplicação *core* do *gateway*. Por ser uma operação simples com o *display*, o uso de um código em python em uma *thread* concorrente a outras, não afetaria a performance do código de maneira preocupante.

6.5.5 *Buzzer*

Os ITGs são equipados com um *buzzer* para emissão de alertas sonoros. Assim como no caso do *display*, o teste de alarme também é caracterizado como teste IO e demanda interação com o usuário.

6.5.5.1 Mapeamento

Para o teste de alarme tem-se:

- **Função:** emitir sinal sonoro quando solicitado.
- **Condição de aprovação:** emissão de sinal sonoro.
- **Possíveis falhas:** não soar o alarme.

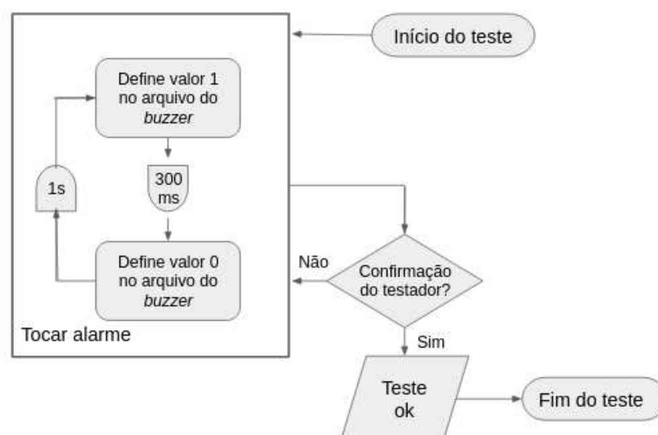
6.5.5.2 Estrutura do teste

O teste do alarme proposto é bastante simples, demandando apenas a manipulação do arquivo com o valor binário para a saída do *buzzer* e uma confirmação dada pelo testador indicando se o alarme foi soado ou não.

A Figura 52 descreve o processo de testes de alarme. O *firmware* de teste soará o alarme por 300 milissegundos a cada 3 segundos a partir do momento em que é executado. O alarme apenas irá parar de soar quando o testador confirmar pela interface web que ouviu o som. Neste cenário, este elemento será aprovado no teste.

Caso contrário, a condição de falha do teste será a não confirmação por parte do testador, indicando que há algum problema com o *buzzer*. Ficará registrado no relatório a falha da funcionalidade do alarme.

Figura 52 – Diagrama de atividades do teste de alarme.



Fonte – Arquivo pessoal, 2022.

6.5.6 Botões

Os *gateways* possuem quatro botões do tipo ”*push buttons*“ que permitem a navegação pelo visor. Assim como no caso do alarme e *display*, a verificação dos botões também depende da interação do usuário.

6.5.6.1 Mapeamento

Para o teste dos botões tem-se:

- **Função:** alterar um estado do produto, no caso da aplicação final do ITG, permite a navegação pelo display.
- **Condição de aprovação:** causar mudança de estado quando pressionado.
- **Possíveis falhas:** não provocar alteração alguma em seu próprio estado.

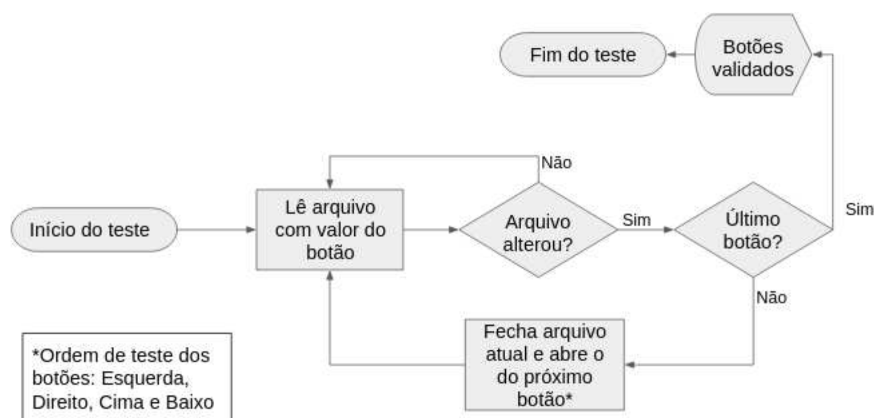
6.5.6.2 Estrutura do teste

Da mesma forma que os testes de *display* e alarme, os botões também demandam interação do testador. Porém, nesse caso, o usuário não será responsável pelo '*feedback*' mas sim servirá como agente, pressionando o botão solicitado enquanto o *firmware* se monitorará os arquivos relativos aos botões, detectando alguma alteração de estado.

Para cada botão na ordem ESQUERDO, DIREITO, CIMA e BAIXO, o testador receberá a instrução de pressioná-lo. Quando todos os botões forem pressionados, na ordem solicitada, o *firmware* informará o sucesso do teste. Caso contrário, será

reportado que o teste falhou ou não foi realizado. O processo pode ser observado na Figura 53.

Figura 53 – Diagrama de atividades do teste de botões.



Fonte – Arquivo pessoal, 2022.

6.5.7 Elemento de segurança

O elemento de segurança é um circuito para armazenamento de dados criptografados, sendo imune a várias técnicas de invasão existentes. Este componente é **obrigatório** para os *gateways* com suporte à rede estendida da EveryNet e também em todos os ITG 300. Para ITG 200 (sem ser EveryNet) a presença do elemento de segurança é opcional.

6.5.7.1 Mapeamento

Para o elemento de segurança tem-se:

- **Função:** armazenar chaves de criptografia de maneira segura;
- **Condição de aprovação:** ao verificar *slots* de dados encontrar informações e não um campo vazio;
- **Possíveis falhas:** falha na leitura ou escrita do elemento seguro indicando que há alguma configuração errada ou que o elemento não está inserido no *gateway* correto. Também há falha se não houver um elemento seguro inserido.

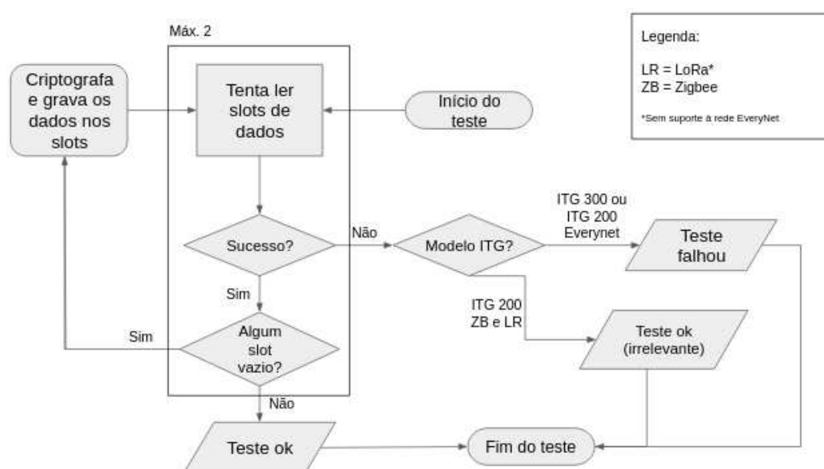
6.5.7.2 Estrutura do teste

Este teste tem a singularidade de possuir duas funções: validar o elemento seguro e, caso necessário, gravar as informações criptografadas nele. A fabricante do

elemento de segurança fornece uma biblioteca python para manipulação do componente.

Desta forma, a proposta para validação do elemento seguro é escrever um código em python que execute o ciclo de ler e gravar o componente, sendo este *script* chamado pela aplicação Golang principal do *firmware*. A Figura 54 evidencia a lógica utilizada para validação do elemento.

Figura 54 – Diagrama de atividades do teste de leitura e gravação do elemento seguro.



Fonte – Arquivo pessoal, 2022.

Primeiramente o programa tentará verificar as informações nos *slots* de dados, caso estejam vazios, o software deverá escrever as informações necessárias que serão criptografadas. Após esse processo, será feita mais uma leitura dos *slots*, a fim de validar que a operação de escrita obteve sucesso. Se os *slots* continuarem vazios então o teste será reportado como falha. Caso contrário, o elemento de segurança estará aprovado.

6.6 INTERFACE WEB

A interface web foi desenvolvida utilizando o *framework* Svelte. Consiste em um conjunto de requisições HTTP para envio e recebimento de dados em formato JSON e também o documento HTML que descreve a aparência da página web. A interface atualiza os dados a cada 1 segundo utilizando uma técnica de *pooling*, ou seja, a cada 1 segundo requisições GET são enviadas ao servidor buscando novas informações.

Na parte superior da tela, são mostradas informações sobre o uso de memória de *gateway*, qual modelo de ITG está sendo testado e o tempo de execução da aplicação.

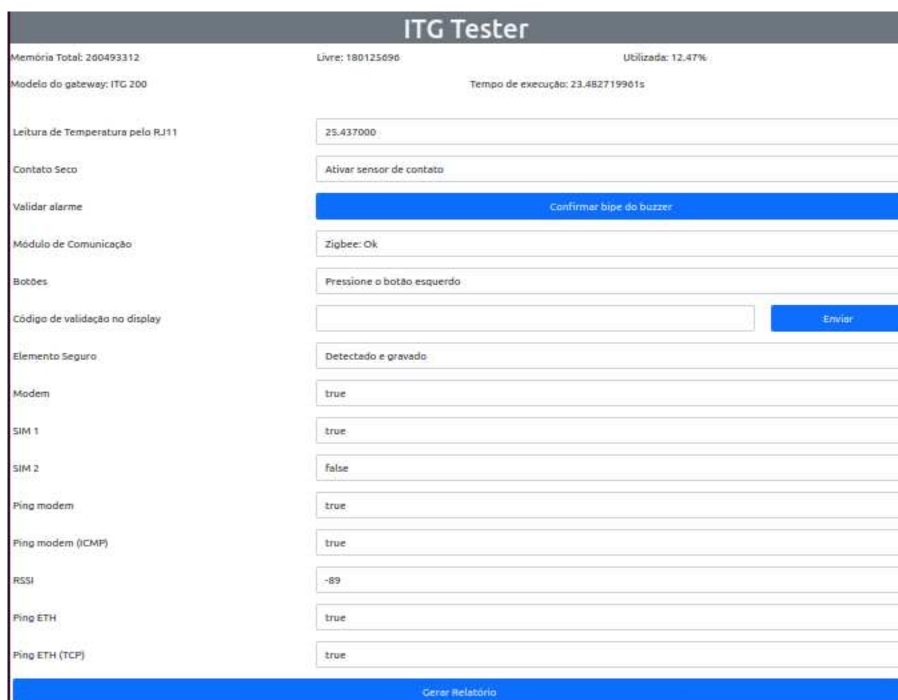
A maioria dos campos da interface apenas retornam o estado dos testes automatizados. No caso dos testes I/O, ela serve como um guia. No teste dos botões, a página informa qual botão deve ser pressionado. Também é pela interface que o testador informa que ouviu o alarme e insere o código de validação do *display*.

Figura 55 – Captura de tela da interface web com o botão ”Gerar Relatório“ desabilitado.

Fonte – Arquivo pessoal, 2022.

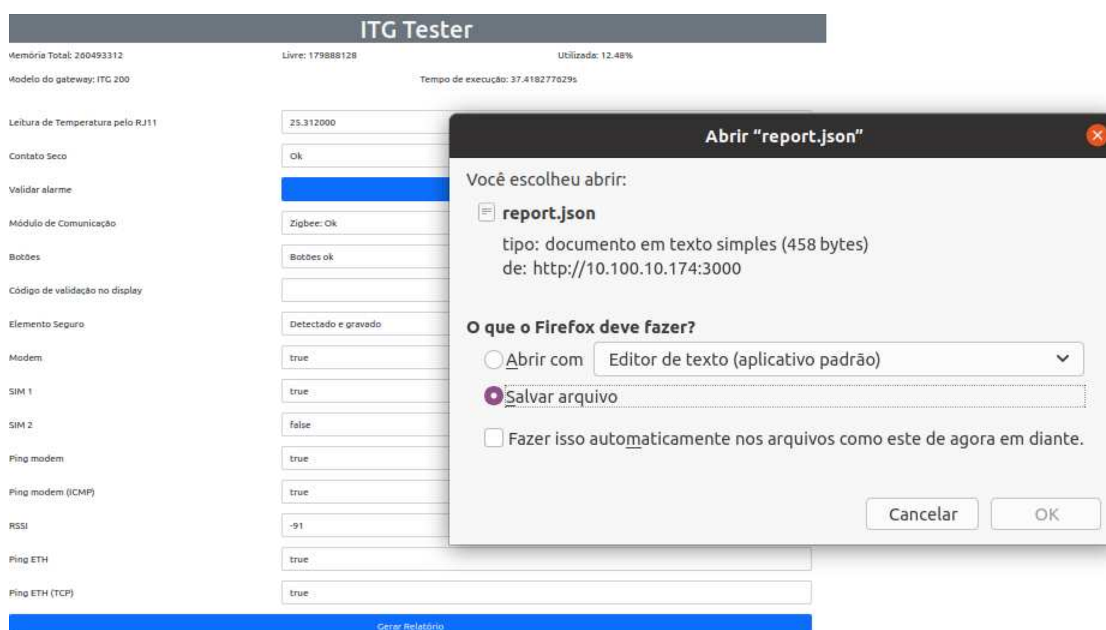
Ao final da página, há um botão ”Gerar Relatório“ que abre uma janela para fazer o *download* do relatório de testes (Figura 57). Este botão fica habilitado para clique apenas quando todos os testes automatizados foram concluídos. Na Figura 55 é possível observar o botão desabilitado, sinalizado por um tom mais fraco de azul. Também é mostrado o estado ”Carregando“ no teste de módulo de comunicação, o que indica que ao menos um teste automático ainda não foi finalizado. Já na Figura 56, todos os testes automáticos foram encerrados e o botão para relatórios está disponível.

Figura 56 – Captura de tela da interface web com o botão "Gerar Relatório" habilitado.



Fonte – Arquivo pessoal, 2022.

Figura 57 – *Pop-up* em evidência para *download* do relatório de teste.

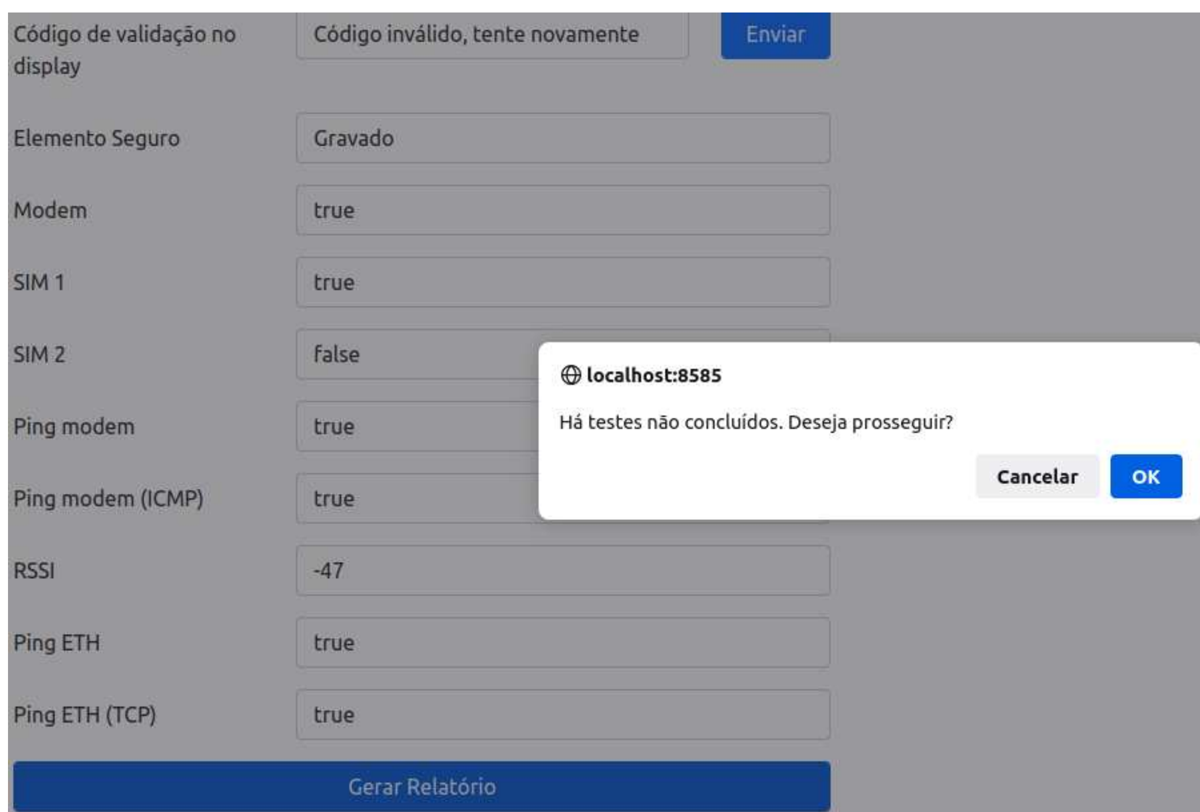


Fonte – Arquivo Pessoal, 2022.

Caso o testador faça o *download* de um relatório em que algum teste I/O não

tenha sido realizado, está ação deverá ser confirmada pelo testador através de um alerta e será reportado que tal validação não foi realizada, como mostra a captura de tela na Figura 58.

Figura 58 – Confirmação de *download* no caso de teste não concluído.



Fonte – Arquivo pessoal, 2022.

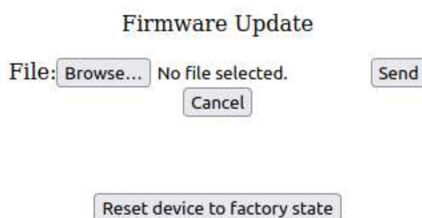
Abaixo são mostradas capturas de tela do relatório gerado no formato JSON (Figura 59) e também da tela para *upload* utilizada para instalação do binário do *firmware* de testes antes do processo e para a atualização para a aplicação final do ITG assim que o produto for validado e aprovado (Figura 60).

Figura 59 – Relatório com os resultados. Caso com todos os testes aprovados.


```
{
        "mac_gateway": "F8033201A934",
        "serial": "108852",
        "data_hora": "2022-02-15 19:40:28.227176076 +0000 UTC m=+35.146910046",
        "c1": "Não ativado",
        "c2": "Ativado",
        "temperatura": "Ok: 25.375000",
        "alarme": "Ok",
        "protocolo": "Zigbee: Ok",
        "botoes": "Ok",
        "display": "Ok",
        "elemento_seguranca": "Gravado",
        "modem": {
                "status": "3G: true",
                "sim1": "true",
                "sim2": "false",
                "ping_tcp": "true",
                "ping_icmp": "true",
                "rssi": "-91"
        },
        "ethernet": {
                "ping_tcp": "true",
                "ping_icmp": "true"
        }
}
```


Fonte – Arquivo Pessoal, 2022.

Figura 60 – Tela de atualização de *firmware*.



Fonte – Arquivo pessoal, 2022.

6.7 KHARMA

Para obter um arquivo binário com todos os pacotes, arquivos e aplicações necessários para o pleno funcionamento do *fimrware* de testes, é necessário descrever o passo a passo do processo em um arquivo .sh que será utilizado para cross-compilar

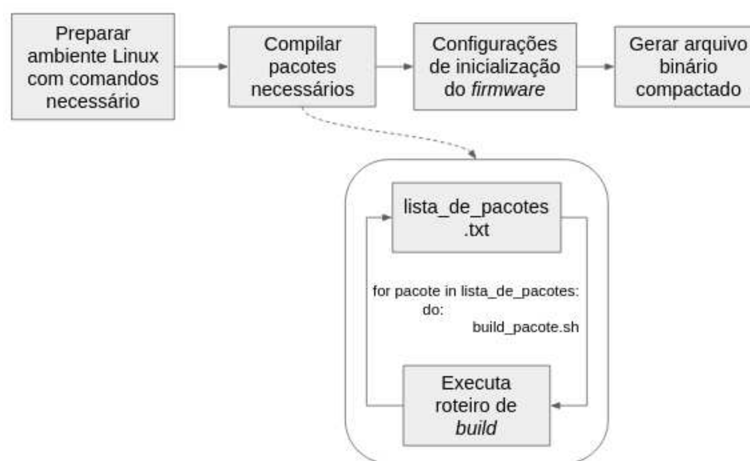
toda a aplicação que está no diretório principal do GitLab em sua versão mais recente para a arquitetura compatível com a do *gateway*, ARM64.

Um arquivo .sh pode ser executado por um terminal de comandos, como o do Linux, por exemplo. Portanto, o roteiro de compilação do kharma para o *firmware* de testes consiste em um conjunto de comandos que instalam pacotes, enviam arquivos para seus diretórios corretos dentro do *gateway*, compilam aplicações, etc, além de tratar os possíveis erros de cada passo. Aliado ao roteiro, há um arquivo de texto (.txt) que descreve uma lista de pacotes necessários para a aplicação, ou seja outras aplicações que devem ser executadas para o funcionamento correto do *firmware*. Cada pacote listado possui seu próprio roteiro de cross-compilação. Alguns dos pacotes que merecem destaque são: Go Connect, aplicações LoRa, Zigbee e MQTT.

Além disso, o próprio projeto do *firmware* possui outro arquivo com a lista de dependências que descreve os pacotes python necessários para o teste de elemento seguro. Estes pacotes também serão instalados durante o processo de cross-compilação.

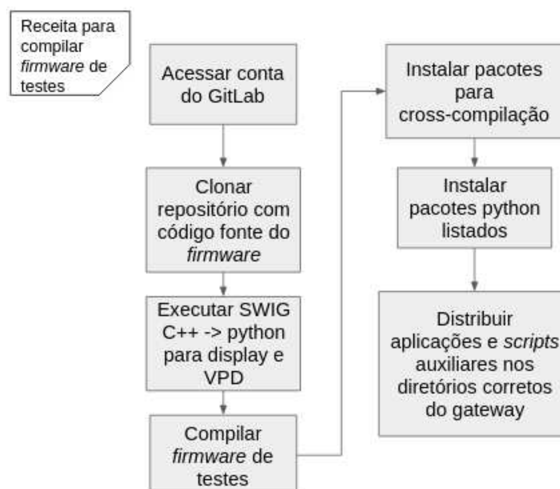
As figuras abaixo descrevem o funcionamento geral de um processo de *build* feito pelo kharma (Figura 61) e também especificam os passos necessários para compilar o código-fonte do *firmware* de testes (Figura 62).

Figura 61 – Esquema geral de um *build* feito pelo kharma.



Fonte – Arquivo pessoal, 2022.

Figura 62 – Roteiro para compilar o código fonte do *firmware* de testes.



Fonte – Arquivo pessoal, 2022.

7 RESULTADOS

Neste capítulo são consolidados os resultados obtidos com os testes iniciais do *firmware*. São apresentados um comparativo entre processos em relação ao tempo despendido para a realização de cada e as percepções do usuário final durante os testes.

O primeiro destaque é para a grande redução de tempo obtida em T2 com o uso do *firmware*. O processo T2 tradicional leva em média 10 minutos. Um testador do time de produção foi convidado a reproduzir o teste utilizando o *firmware*. Apenas foi explicado que a proposta do projeto é substituir o processo manual atualmente realizado. Não foi detalhada nenhuma informação quanto a procedimento de testes para que também fosse avaliada a usabilidade da interface web. Um mesmo testador realizou a operação 10 vezes para que fosse possível observar a curva de aprendizado em relação ao uso da interface. O gráfico abaixo (Figura 63) mostra a média de tempo em cada iteração.

Figura 63 – Tempos de execução do teste T2 com o uso do *firmware*.

Fonte – Arquivo pessoal, 2022.

No primeiro teste, onde há uma barreira a ser vencida quanto ao aprendizado da interface, já é perceptível o ganho de tempo do novo processo proposto. Um processo com duração de 165 segundos já representa 27.5% do tempo usualmente gasto no procedimento documentado no Capítulo 3.

As demais iterações, mesmo com possíveis distrações do testador, demonstram uma familiaridade maior com a interface, acelerando ainda mais o processo. Desconsiderando os valores limítrofes - 165 segundos em T1 e 39 segundos em T8 - e calculando a média, temos uma estimativa de 59,75 segundos por teste. O que se enquadra dentro da especificação de possibilitar um ciclo de testes dentro de 60 segundos. Isto mostra uma redução de cerca de 90% no tempo.

É importante frisar que este tempo está fortemente condicionado à realização dos testes de I/O pelo usuário, pois, no geral, os testes automatizados são encerrados em até 50 segundos.

A primeira versão do projeto recebeu sugestões para melhorias em questão de usabilidade. Os usuários pediram mais elementos para interação como a opção de reiniciar um teste através do botão ao invés de recarregar a página e fornecer mais *feedbacks* sobre o andamento do teste pelo interface web. Estas melhorias estão sendo implementadas de acordo com as metodologias ágeis seguidas.

Porém, mesmo com pontos a acrescentar, o *firmware* de testes impressionou positivamente e mostra um grande potencial para implementação na linha de produção pela sua agilidade e capacidade de simplificar o processo, como dito no testemunho abaixo de um colaborador da empresa:

> O teste desenvolvido é funcional, simples e objetivo. Apresenta pontos de melhorias, porém está em um bom nível levando em conta o objetivo proposto. Com a sua finalização há um grande potencial em ser utilizado no setor industrial da empresa, trazendo mais assertividade e rapidez para o processo. - SANCHES, B. Testador do time de produção.

Com a implementação do *firmware* para o teste T2 será possível observar se o pré-requisito de segurança será cumprido, ou seja, se a automatização de boa parte do teste realmente diminuirá a incidência de falha humana. A expectativa é que esse objetivo se mostre concluído a médio-longo prazo visto que será implantado um processo mais simples e os testadores terão suporte e auxílio da aplicação para as etapas que dependem de alguma interação humano-máquina.

8 PROPOSTAS DE MELHORIAS

A seguir, serão levantados pontos de melhoria a ser implementados nos próximos ciclos do projeto. Algumas questões foram listadas ainda na etapa de levantamento de requisitos, outras surgiram no decorrer dos ciclos de desenvolvimento e ainda há aquelas que foram solicitadas pelos usuários finais após os primeiros testes com a primeira versão.

8.1 AUTOCADASTRO *CLOUD* EVERYNET

Gateways com suporte à rede EveryNet precisam de registro na *Radio Access Network* (RAN) da operadora, um serviço para gerenciamento de *gateways* na nuvem. Sem este registro é impossível trocar dados com a rede EveryNet.

Atualmente, o cadastro é feito no momento em que o *gateway* é ligado com sua aplicação final. A proposta é que este autocadastro seja feito em T2 com o *firmware* de testes, logo após validado o elemento de segurança.

8.2 GERAR DIAGNÓSTICOS DE FALHAS

No Capítulo 4 que aborda a fundamentação teórica para o desenvolvimento do projeto, são listadas técnicas para diagnósticos de falhas. A proposta é que futuramente o *firmware* seja capaz de fornecer possíveis diagnósticos para os casos de falha e não apenas retorne se um componente falhou no teste ou não.

Para isso, árvores de falha já começaram a ser esboçadas para os módulos apresentados neste documento. A estimativa é de que até o final de 2022 o *firmware* suporte mais essa funcionalidade.

8.3 GERAR RELATÓRIO AUTOMATICAMENTE COM *UPLOAD* PARA SERVIDOR INTERNO KHOMP

Ainda buscando tornar o processo mais automatizado e simples, esta proposta busca substituir a operação ”manual“ de *download* e armazenamento do arquivo por uma automática onde o relatório do teste é enviado diretamente para um servidor interno da Khomp, facilitando o acesso e mitigando ainda mais a possibilidade de falha humana.

9 CONCLUSÃO

Este PFC propôs o desenvolvimento de um *firmware* de testes embarcado para a otimização do processo de teste de produto da linha ITG de *gateways* IoT da Khomp. O processo, denominado T2, levava um tempo considerável de execução e possuía muitos passos, sendo bastante suscetível a falhas humanas.

O projeto foi desenvolvido com base em técnicas de teste de *hardware* e *software* e também nos passos descritos do processo tradicional. Utilizando Golang, uma linguagem pensada para projetos de concorrência e paralelismo, o *firmware* apresentou um desempenho bastante satisfatório, reduzindo em até 90% o tempo de execução do processo.

Com a implementação do *firmware* na linha de produção do ITG, o novo processo T2 passará a ter os seguintes passos:

1. Instalar o *firmware* de testes no ITG;
2. Abrir interface web em um navegador com o número de IP e porta mostrados no *display*;
3. Observar se o modelo do *gateway* testado é mostrada corretamente na interface;
4. Realizar os testes dos quatro botões apertando-os na ordem solicitada;
5. Confirmar se o alarme está sendo soado ou não;
6. Testar entrada RJ11 acionando sensor de contato conectado;
7. Validar o *display* inserindo o código mostrado na interface web;
8. Se necessário, aguardar a conclusão dos testes automatizados: conectividade, módulo de comunicação, *1-wire*, elemento de segurança;
9. Gerar relatório e armazená-lo em local previamente definido pelo time de produção;
10. Caso aprovado em todos os testes: Acessar a página de atualização de *firmware* e instalar a aplicação final do ITG na versão mais recente. Enviar produto para o setor de expedição;
11. Caso produto seja reprovado em ao menos um teste: encaminhar produto para diagnóstico e correção.

O trabalho desenvolvido foi validado pelos usuários finais da aplicação e, com mais alguns ajustes, estará pronto para ser finalmente implementado na linha de produção, promovendo uma grande economia de tempo do processo e reduzindo gargalos relacionados à falha humana.

Do ponto de vista da autora, poder desenvolver um projeto com caráter tão heterogêneo e abrangente foi uma grande oportunidade para conhecer todos os aspectos relacionados ao produto. Tal projeto também promoveu um crescimento profissional significativo, visto que várias tecnologias e conceitos novos precisaram ser internalizados junto à consolidação de assuntos vistos durante a graduação em Engenharia de Controle e Automação como: programação concorrente, redes de computadores e metodologias para desenvolvimento de software.

REFERÊNCIAS

ABRAHAMSSON, Pekka; SALO, Outi; RONKAINEN, Jussi; WARSTA, Juhani. Agile software development methods: Review and analysis. **arXiv preprint arXiv:1709.08439**, 2017.

AWTREY, Dan; SEMICONDUCTOR, Dallas. Transmitting data and power over a one-wire bus. **Sensors-The Journal of Applied Sensing Technology**, [Peterborough, NH: North American Technology, Inc.], c1984-c2006., v. 14, n. 2, p. 48–51, 1997.

BAINOMUGISHA, Engineer; CARRETON, Andoni Lombide; CUTSEM, Tom van; MOSTINCKX, Stijn; MEUTER, Wolfgang de. A survey on reactive programming. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 45, n. 4, p. 1–34, 2013.

BALACHANDRAN, Sasang. General Purpose Input/Output (GPIO). **Michigan State University College of Engineering. Published**, p. 08–11, 2009.

BANKS, Andrew; GUPTA, Rahul. mqtt-v3.1.1-plus-errata01, 2015. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.pdf. Acesso em: 14 dez. 2021.

BECK, Kent. **TDD Desenvolvimento Guiado por Testes**. 1. ed. Porto Alegre: [*s.n.*], 2010.

BEZERRA, Rafael Lopes. Análise da conectividade em redes móveis utilizando dados obtidos da mobilidade humana. **Universidade Federal do Rio de Janeiro (UFRJ)**, 2009.

BRESHEARS, Clay. **The art of concurrency: A thread monkey's guide to writing parallel applications**. [*S.l.*]: "O'Reilly Media, Inc.", 2009.

COLÓN, Rafael Pérez; NAVAJAS, Sergio; TERRY, Elizabeth. IoT IN LAC 2019: Taking the Pulse of the Internet of Things in Latin America and the Caribbean, 2019. Disponível em: https://publications.iadb.org/publications/english/document/IoT_IN_LAC_2019_Taking_the_Pulse_of_the_Internet_of_Things_in_Latin_America_and_the_Caribbean_en.pdf. Acesso em: 21 ago. 2021.

DINCULEANĂ, Dan; CHENG, Xiaochun. Vulnerabilities and limitations of MQTT protocol used between IoT devices. **Applied Sciences**, Multidisciplinary Digital Publishing Institute, v. 9, n. 5, p. 848, 2019.

FENG, Xiang; YAN, Fang; LIU, Xiaoyu. Study of wireless communication technologies on Internet of Things for precision agriculture. **Wireless Personal Communications**, Springer, v. 108, n. 3, p. 1785–1802, 2019.

FENTON, William G; MCGINNITY, T Martin; MAGUIRE, Liam P. Fault diagnosis of electronic systems using intelligent techniques: A review. **IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)**, IEEE, v. 31, n. 3, p. 269–281, 2001.

FIBER. **Go Fiber**. 2021. Disponível em: `https://docs.gofiber.io/`. Acesso em: 23 nov. 2021.

GOURLEY, David; TOTTY, Brian; SAYER, Marjorie; AGGARWAL, Anshu; REDDY, Sailu. **HTTP: the definitive guide**. [*S.l.*]: "O'Reilly Media, Inc.", 2002.

HAXHIBEQIRI, Jetmir; DE POORTER, Eli; MOERMAN, Ingrid; HOEBEKE, Jeroen. A survey of LoRaWAN for IoT: From technology to application. **Sensors**, Multidisciplinary Digital Publishing Institute, v. 18, n. 11, p. 3995, 2018.

HUANG, July. Gateway vs Router: What's the Difference?, 2018. Disponível em: `https://medium.com/@july.huang666/gateway-vs-router-whats-the-difference-fb010ee3b5cc`. Acesso em: 7 set. 2021.

KHOMP. **A história da Khomp**. Disponível em: `https://www.khomp.com/pt/institucional/`. Acesso em: 30 ago. 2021.

KUZMINYKH, Ievgeniia; SNIHUROV, Arkadii; CARLSSON, Anders. Testing of communication range in ZigBee technology. *In*: IEEE. 2017 14th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM). [*S.l.*: *s.n.*], 2017. P. 133–136.

MACHA, Uday Shankar. Embedded linux operating system. **Corpus ID**, v. 38636247, n. 1, p. 10.

MADAKAM, Somayya; RAMASWAMY, R.; TRIPATHI, Siddharth. Internet of Things (IoT): A Literature Review, 2015. Disponível em: https://www.scirp.org/pdf/JCC_2015052516013923.pdf. Acesso em: 21 ago. 2021.

MEIRELLES, Paulo Roberto Miranda. Teste Integrado de Software e Hardware: Reusando Casos de Teste de Software em Teste de Microprocessadores. **UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL**, 2008. Disponível em: https://www.lume.ufrgs.br/bitstream/handle/10183/25520/000751158.pdf. Acesso em: 15 set. 2021.

MICROCHIP. **ATECC608A - Network and Accessories secure authentication**. 2018. Disponível em: https://www.microchip.com/en-us/product/ATECC608A. Acesso em: 4 mar. 2022.

MINERVA, Roberto; BIRU, Abyi; ROTONDI, Domenico. Towards a Definition of Internet of Things (IoT), 2015. Disponível em: https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf. Acesso em: 21 ago. 2021.

MONTEIRO, Fernando. **Learning single-page web application development**. [*S.l.*]: Packt Publishing Ltd, 2014.

MOURAD, Samiha; ZORIAN, Yervant. **Principles of testing electronic systems**. [*S.l.*]: John Wiley & Sons, 2000.

MOUSQUER, Gabriel Blank Stift. **Vue.js e os frameworks reativos**. Disponível em: https://medium.com/@gabrielblanksm/vue-js-e-os-frameworks-reativos-fa4afd319b70. Acesso em: 23 nov. 2021.

OAS, Open Automation Software -. **What is an IoT Gateway?** Disponível em: https://openautomationsoftware.com/open-automation-systems-blog/what-is-an-iot-gateway/. Acesso em: 7 set. 2021.

PRESSMAN, Roger S. **Engenharia de Software**. 3. ed. [*S.l.*]: Pearson, 1995. ISBN 8534602379.

SANNER, Michel F *et al.* Python: a programming language for software integration and development. **J Mol Graph Model**, v. 17, n. 1, p. 57–61, 1999.

SEMTECH. **What is LoRa?** Disponível em:
https://lora-developers.semtech.com/learn/get-started/what-is-lora/.
Acesso em: 15 set. 2021.

SHAH, Hardik. **What is TDD? [Roadmap to Implement TDD in Your Organization]**. 2019. Disponível em: https://www.simform.com/blog/what-is-tdd/. Acesso em: 15 jul. 2021.

SINGH, Charanjot; GABA, Nikita Seth; KAUR, Manjot; KAUR, Bhavleen. Comparison of different CI/CD tools integrated with cloud platform. *In*: IEEE. 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence). [*S.l.*: *s.n.*], 2019. P. 7–12.

SOARES, Patricia Gomes. On remote procedure call. *In*: PROCEEDINGS of the 1992 conference of the Centre for Advanced Studies on Collaborative research-Volume 2. [*S.l.*: *s.n.*], 1992. P. 215–267.

STADZISZ, Paulo Cézar; RENAUX, Douglas Paulo Bertrand. Software Embarcado. **XIV Escola Regional de Informática SBC**, v. 1, p. 107–155, 2007.

STALLMAN, Richard. **Linux e o Sistema GNU**. 2021. Disponível em:
https://www.gnu.org/gnu/linux-and-gnu.pt-br.html. Acesso em: 12 out. 2021.

SWIG. **SWIG**. Disponível em: http://www.swig.org/. Acesso em: 10 jan. 2022.

TANENBAUM, Andrew S.; WETHERALL, David. **Redes de Computadores**. 5. ed. São Paulo: [*s.n.*], 2011. ISBN 978-85-7605-924-0.

VÁRIOS. **Go Web Framework Benchmark**. 2020. Disponível em:
https://github.com/smallnest/go-web-framework-benchmark. Acesso em: 23 nov. 2021.

WANG, Wei; HE, Guangyu; WAN, Junli. Research on Zigbee wireless communication technology. *In*: IEEE. 2011 International Conference on Electrical and Control Engineering. [*S.l.*: *s.n.*], 2011. P. 1245–1249.

XU, Wenqing. Benchmark Comparison of JavaScript Frameworks React, Vue, Angular and Svelte, 2021.

ZHHUTA, V. Go Programing Language (GoLang). ’ - **FOSS Lviv 2015**, p. 29–29, 2015.

APÊNDICE A – TESTES UNITÁRIOS

Este apêndice elenca os pseudocódigos dos testes unitários escritos durante o desenvolvimento dos módulos do *firmware* de testes.

```
// Teste dos modulos de comunicacao sem fio
// "protocol" e modulo go que trata dos testes dos slots LoRa e Zigbee

func TestMQTTClient():
    protocol.MQTT()
    client <- protocol.MQTTClient()

    assert client.IsConnected() == true

func LoRaExecution():
    protocol.RunLoRaController()
    process <- exec.Output('path/to/lora-controller/bin', 'status')

    assert string.Contains(process, 'running') == true


func TestMessageHandler(): \\ LoRa case
    protocol.MQTT()
    mqttClient <- protocol.MQTTClient()
    message <- protocol.client.Publish(gatewayIP, 'gateway/teste', 1, 'message')

    assert protocol.LROK == true
    assert protocol.ZBOK == false


func ZigbeeExecution():
    protocol.RunZigbeeCoordinator()
    process <- exec.Output('path/to/zigbee-coord/bin', 'status')

    assert string.Contains(process, 'running') == true


func TestMessageHandler(): \\ Zigbee case
    protocol.MQTT()
    mqttClient <- protocol.MQTTClient()
    message <- protocol.client.Publish(gatewayIP, 'gw/teste', 1, 'message') \\
```

```
       Zigbee topic

    assert protocol.ZBOK == true
    assert protocol.LR == false


// Teste de conectividade
// "connectivity" e o modulo Go que trata dos testes de conectividade

func GoConnectExecution():
    connectivity.RunGoConnect()
    process <- exec.Output('path/to/go-connect/bin', 'status')

    assert string.Contains(process, 'running') == true


func TestRPCClientConnection():
    client <- connectivity.RPCClient()
    connectivity.RunGoConnect()
    responseJson <- ""
    callError <- client.Call("Listener.GetConnStatus", &responseJson)

    assert callError == nil
    assert responseJson != nil


// Teste de alarme
// "io" e o modulo Go que trata dos testes de entrada e saida
// como display, botoes e alarme

func TestandoAlarme()
   io.FuncAlarm()
   filePath <- "path/to/file"
   file <- OpenFile(filePath)
   value <- ReadFile(file)
   expected_value <- "0"

   assert value == expected_value


// Teste botoes
func TestWrongButton()
```

```
    btn <- io.FuncButtons() // expect test btn A but tests btn B
    filePath_b <- "path/to/file_b"
    file_b <- OpenFile(filePath_b)
    file_b.Write('0')
    file_b.Close()

    assert btn.currentState == false


func TestExpectedButton()
    btn <- io.FuncButtons() // expect test btn A
    filePath_a <- "path/to/file_a"
    file_a <- OpenFile(filePath_a)
    file_a.Write('0')
    file_a.Close()

    assert btn.currentState == true


// Teste display
func TestDisplayCallingPythonScript()
    defaultCode <- "FFFF"
    codeSize <- 4
    macGw <- "AAA123BBB456"
    ipGw <- "10.10.10.200"
    validationCode <- io.GetCode()

    assert validationCode == ""

    io.DisplayInit(macGw, ipGw)
    validationCode <- io.GetCode()

    assert validationCode != ""
    assert len(validationCode) == codeSize
    assert validationCode != defaultCode


// Teste RJ11
func TestTemperaturaOk():
    filePath <- 'path/to/file'
    file <- OpenFile(filePath)
    file.Write("t=250000")
```

```
    file.Close()
    temperatureRead <- sensors.Rj11Read1Wire()

    assert temperatureRead == "OK"


func TestTemperaturaNotOk():
    filePath <- 'path/to/file'
    file <- OpenFile(filePath)
    file.Write("t=550000")
    file.Close()
    temperatureRead <- sensors.Rj11Read1Wire()

    assert temperatureRead == "Out of expected range"


func TestTemperaturaNoSensor():
    temperatureRead <- sensors.Rj11Read1Wire()

    assert temperatureRead == "No sensor found"


func TestContactSensor():
    cs <- sensors.FuncReadCsRj11()
    filePath <- "path/to/file"
    file <- OpenFile(filePath)
    file.Write('0')
    file.Close()

    assert cs.currentState == true
```