UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUIS HENRIQUE PELEGRIN FIGUEIREDO

FRAMEWORK PON C++ 4.0 IOT: PARADIGMA ORIENTADO A NOTIFICAÇÕES PARA AMBIENTE DE INTERNET DAS COISAS

CURITIBA

2022

LUIS HENRIQUE PELEGRIN FIGUEIREDO

FRAMEWORK PON C++ 4.0 IOT: PARADIGMA ORIENTADO A NOTIFICAÇÕES PARA AMBIENTE DE INTERNET DAS COISAS

NOP Framework C++ 4.0 IoT: Notification-Oriented Paradigm for Internet of Things Environment

Dissertação apresentado como requisito para obtenção do título de Mestre em Computação Aplicada do Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná.

Orientador: Profa. Dra. Ana Cristina Barreiras Kochem Vendramin

Coorientador: Prof. Dr. Jean Marcelo Simão

CURITIBA

2022

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Curitiba

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUIS HENRIQUE PELEGRIN FIGUEIREDO

FRAMEWORK PON C++ 4.0 IOT: PARADIGMA ORIENTADO A NOTIFICAÇÕES PARA AMBIENTE DE INTERNET DAS COISAS

Trabalho de pesquisa de mestrado apresentado como requisito para obtenção do título de Mestre Em Computação Aplicada da Universidade Tecnológica Federal do Paraná (UTFPR). Área de concentração: Engenharia De Sistemas Computacionais.

Data de aprovação: 01 de Dezembro de 2022

Dra. Ana Cristina Barreiras Kochem Vendramin, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Carlos Alberto Maziero, Doutorado - Universidade Federal do Paraná (Ufpr)

Dr. Jean Marcelo Simao, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Marcos Talau, Doutorado - Universidade Tecnológica Federal do Paraná

Dr. Robson Ribeiro Linhares, Doutorado - Universidade Tecnológica Federal do Paraná

Documento gerado pelo Sistema Acadêmico da UTFPR a partir dos dados da Ata de Defesa em 01/12/2022.

Dedico este trabalho à minha família e aos meus amigos, por todo apoio incondicional.

AGRADECIMENTOS

Agradeço a todos que de alguma forma contribuíram para a realização deste trabalho, em especial à minha família e aos meus amigos.

Agradeço à minha orientadora Profa. Dra. Ana Cristina Barreiras Kochem Vendramin e ao meu coorientador Prof. Dr. Jean Marcelo Simão, ao Programa de Pós-Graduação em Computação Aplicada (PPGCA) do Departamento Acadêmico de Informática (DAINF), ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial (CPGEI) e à Universidade Tecnológica Federal do Paraná (UTFPR) pela oportunidade de estudo e evolução pessoal e profissional.

Agradeço também aos membros da banca Prof. Dr. Robson Ribeiro Linhares, Prof. Dr. Marcos Talau e Prof. Dr. Carlos Alberto Maziero, por disponibilizar seu tempo na avaliação deste trabalho.

RESUMO

A Internet das Coisas tem transformado o modo como os mundos físico e digital interagem por meio da interconexão de objetos do dia a dia. Esses objetos, equipados com recursos de computação, comunicação, sensores e atuadores, são capazes de interagir de forma colaborativa e realizar uma variedade de tarefas de forma autônoma. Porém, ao mesmo tempo em que promove benefícios para a sociedade, a Internet das Coisas agrava ainda mais a chamada "Crise de Software". Nota-se a dificuldade na produção de softwares devido a características presentes em sistemas da Internet das Coisas, como a arquitetura de comunicação e integração distribuída com componentes e dispositivos heterogêneos. Os atuais paradigmas de programação dominantes apresentam ineficiências para a concepção e execução em ambiente distribuído, sendo geralmente necessárias técnicas de programação e mesmo de engenharia de software adicionais e customizadas para cada aplicação. Como alternativa aos problemas mencionados, o emergente Paradigma Orientado a Notificação (PON) se apresenta como uma nova abordagem para sistemas computacionais construídos por entidades reativas que interagem por meio de notificações pontuais, desacoplando a computação facto-execucional da computação lógico-causal. O PON tem o objetivo de tornar menos árdua a tarefa de desenvolvimento de sistemas por permitir uma concepção em alto nível, tornando o código e sua execução mais eficiente por evitar redundâncias (estruturais e temporais) e, por fim, permitindo a sua execução distribuída por garantir o desacoplamento intrínseco entre as entidades. Atualmente, o PON apresenta várias materializações em software sendo as mais tecnologicamente maduras aquelas que se dão por meio de *frameworks* desenvolvidos em diferentes linguagens de programação. Neste contexto, o presente trabalho apresenta avanços no estado da técnica, implementando o *Framework* PON C++ 4.0 IoT o qual permite a distribuição das entidades constituintes do PON utilizando uma arquitetura *Publish/Subscribe* e o protocolo MQTT (*Message Queuing Telemetry Transport*), ambos comumente utilizados em ambientes IoT. Ademais, são classificados também os tipos de entidades do PON em relação aos seus possíveis modos de distribuição. São apresentados três experimentos utilizando o PON por meio do *Framework* PON C++ 4.0 IoT com o objetivo de verificar os aspectos funcionais e não funcionais do PON e compará-lo com o Paradigma Orientado a Eventos (POE) por meio de implementações em C++ utilizando a arquitetura *Publish/Subscribe* e o MQTT. Os resultados demonstram que o tempo para processamento de mensagens das aplicações desenvolvidas com o *Framework* PON C++ 4.0 IoT foi até 11,2 vezes menor, porém consumiu até 6,7 vezes mais memória que as aplicações em POE. Com relação ao uso de rede, o número de mensagens transmitidas para ambas as implementações foi semelhante. Observou-se em todos os experimentos a maior expressividade e facilidade de distribuição das aplicações desenvolvidas no PON em comparação com o POE. Nas implementações em *Framework* PON C++ 4.0 IoT, houve uma redução na redundância de processamento em até 39,6% ao se utilizar corretamente o compartilhamento de entidades distribuídas.

Palavras-chave: Paradigma Orientado a Notificações; Internet das Coisas; Sistemas Distribuídos; Framework PON C++ 4.0 IoT.

ABSTRACT

The Internet of Things (IoT) has transformed the way the physical and digital worlds interact through the interconnection of everyday objects. These objects with computing, communication, sensing and acting capabilities can interact collaboratively and execute a variety of tasks in an autonomous way. However, while promoting benefits to society, the IoT further aggravates the so-called “Software Crisis”. The difficulty in software development in this context is mainly caused by the IoT system characteristics, such as the communication architecture and distributed integration with heterogeneous components and devices. The current dominant programming paradigms present inefficiencies for designing and executing in a distributed environment, requiring additional and customized software engineering techniques for each application. As an alternative to the mentioned problems, the emerging Notification Oriented Paradigm (NOP) presents itself as a new approach for computer systems built by reactive and decoupled entities that interact through accurate notifications, uncoupling factual-executional from logical-causal computing. NOP aims to make the task of systems development less arduous by allowing a high-level design, making the code and its execution more efficient by avoiding redundancies (structural and temporal) and, finally, enabling its execution to be distributed through its intrinsic entities decoupling model. Currently, NOP presents several materializations in software, the most technologically mature being those developed through frameworks, implemented in different programming languages. In this context, this work presents advances in the state of the technique, implementing the NOP Framework C++ 4.0 IoT, which allows the distribution of the NOP’s entities using, in an innovative way, a Publish/Subscribe architecture and the Message Queuing Telemetry Transport (MQTT) protocol, both of which are commonly used in IoT environments. Furthermore, the types of NOP entities are also classified in relation to their possible distribution modes. Three experiments are presented using NOP Framework C++ 4.0 IoT to verify the functional and non-functional aspects of NOP and compare it with the Event-Driven Paradigm (EDP) through implementations in C++ using the Publish/Subscribe architecture and the MQTT protocol. Results show that the time for processing messages of applications developed with NOP Framework C++ 4.0 IoT is up to 11.2 times smaller, however it consumed up to 6.7 times more memory than the application in EDP. Regarding the use of the network, the number of messages transmitted for both implementations was similar. In all experiments, it was observed a greater expressiveness and ease of application distribution developed with NOP compared to EDP. In the NOP Framework C++ 4.0 IoT implementation, there was a reduction in processing redundancy by up to 39.6% when using distributed entity sharing correctly.

Keywords: Notification Oriented Paradigm; Internet of Things; Distributed Systems; NOP Framework C++ 4.0 IoT.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Nos dias atuais, já é possível observar o crescimento do número de dispositivos computacionais eletrônicos capazes de se conectar e interagir com os seres humanos e entre si (HASAN, 2022). Nessa categoria, incluem-se desde dispositivos como computadores pessoais, *Laptops*, *Smartphones* e *tablets* até os mais variados sensores como uma lâmpada inteligente, medidores de energia inteligentes (*smart meters*) e monitores de batimentos cardíacos (disponíveis, por exemplo, em *smartwatches*) (MISHRA; KERTESZ, 2020).

Neste âmbito, a Internet das Coisas, cujo acrônimo em inglês é IoT (*Internet of Things*), tem transformado a forma como o mundo físico e o digital interagem por meio da interconexão de objetos do dia-a-dia, agora equipados com recursos de computação, comunicação, sensores e/ou atuadores (NIKOUKAR *et al.*, 2018). Esta rede de objetos pode interagir de forma colaborativa e realizar uma variedade de tarefas de forma autônoma (SAMUEL; SIPES, 2019) (DORSEMAINE *et al.*, 2015).

Nas mais diversas áreas como na agricultura, transportes, esportes, saúde, militar, energia ou entretenimento, as possibilidades de aplicação e de crescimento de IoT parecem potencialmente sem fim (SUEDA; SATO; HASUIKE, 2019). Neste contexto, é razoável dizer que seria raro encontrar alguma área que não se beneficie da revolução da IoT. Na indústria, por exemplo, decisões que antes se baseavam em dados mensais ou anuais podem agora se basear em dados precisos em tempo real (MISHRA; KERTESZ, 2020).

Em suma, a lista de dispositivos computacionais capazes de gerar e/ou processar dados e se comunicar cresce a cada dia, sendo que a IoT tem seu papel de destaque neste processo. Com o aumento da quantidade, diversidade, distância física e velocidade da troca de dados entre os dispositivos, a interação entre eles apresenta cada vez mais desafios tecnológicos para o desenvolvimento de aplicações mais eficientes no uso dos recursos disponíveis (SAMUEL; SIPES, 2019) (AL-MASRI *et al.*, 2020).

Neste quadro dado, se por um lado a sociedade se beneficia dessa revolução de aplicações IoT, por outro lado cresce a demanda por desenvolvimento de *softwares* para a IoT, agravando ainda mais a chamada ‘Crise de Software’. Resumidamente, a assim chamada ‘Crise de *Software*’ indica a dificuldade na produção de *softwares* frente ao rápido crescimento na demanda aliada ao aumento da complexidade dos problemas a serem resolvidos (RONSZCKA, 2019) (BAUTSCH, 2007). Em especial no contexto de IoT, observa-se o aumento da demanda e da complexidade das aplicações em virtude das características dos sistemas cada vez mais distribuídos, dos dispositivos muitas vezes limitados em termos de recursos (processamento, memória, energia e/ou largura de banda) disponíveis e das integrações geralmente complexas e envolvendo dispositivos heterogêneos (SUEDA; SATO; HASUIKE, 2019) (HANDOSA; GRAčANIN; ELMONGUI, 2017) (SAMUEL; SIPES, 2019).

1.1 Contexto

Diante do cenário supramencionado, é necessário que as técnicas, ferramentas de projeto, análise e implementação de sistemas, em especial de sistemas distribuídos para IoT, tornem-se cada vez mais fáceis e acessíveis aos desenvolvedores de sistemas e conduzam a implementações mais eficientes e eficazes. Porém, observa-se que os paradigmas de programação atualmente dominantes e usuais, derivados dos Paradigmas Imperativos (PI) e dos Paradigmas Declarativos (PD), apresentam ineficiências para o desenvolvimento e execução de programas, principalmente em ambientes concorrentes e/ou distribuídos (GABBRIELLI; MARTINI, 2010) (KAISLER, 2005) (RONSZCKA, 2019) (SIMÃO; STADZISZ, 2009a) (SCHÜTZ *et al.*, 2018).

Essencialmente, o modelo de construção sequencial do PI é orientado a percorrimentos ou buscas sobre elementos passivos, relacionando os dados (variáveis e/ou estrutura de dados) com expressões lógico-causais (estruturas se-então ou similares) (GABBRIELLI; MARTINI, 2010). Na prática, esse modelo favorece a presença de redundância estrutural e redundância temporal. De forma sucinta, a redundância estrutural ocorre por repetição de avaliações de uma mesma variável em partes distintas do código. Por sua vez, a redundância temporal ocorre pela reavaliação desnecessária em expressões lógico-causais, de variáveis cujos estado não foram alterados desde a última avaliação. Tais redundâncias afetam o desempenho das aplicações por gerar processamento desnecessário e, além disso, dificultar o alcance de uma dependência mínima (*i.e.*, desacoplamento) entre os módulos pelo fato de gerar acoplamento implícito entre eles (RONSZCKA, 2019) (GABBRIELLI; MARTINI, 2010) (SIMÃO *et al.*, 2012) (SIMÃO; STADZISZ, 2009a) (PAN; DESOUZA; KAK, 1998).

O PD, por sua vez, proporciona um nível de abstração maior que o PI, facilitando a composição de programas (KAISLER, 2005) (GABBRIELLI; MARTINI, 2010). Além disso, algumas soluções declarativas evitam redundâncias de execução, como é o caso dos motores ou máquina de inferência baseados em Rete e Hal (FORGY, 1982) (LEE; CHENG, 2002). Porém, tais soluções declarativas fazem uso de certas estruturas de dados, as quais são computacionalmente custosas e causam consideráveis sobrecargas de processamento (RONSZCKA, 2019). Assim, mesmo com código redundante, soluções baseadas no PI normalmente apresentam melhor desempenho do que as soluções baseadas no PD. Além disso, tal qual no PI, a programação no PD também gera acoplamento entre os módulos, uma vez que o processo de inferência também se baseia em buscas sobre entidades passivas (RONSZCKA, 2019) (GABBRIELLI; MARTINI, 2010) (SIMÃO *et al.*, 2012) (PAN; DESOUZA; KAK, 1998).

Neste contexto, o Paradigma Orientado a Notificações (PON) surge como uma abordagem alternativa para a concepção de sistemas computacionais visando resolver, ou ao menos amenizar, os problemas destacados nos paradigmas usuais de programação (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009). Ademais, o PON agrega algumas das características e vantagens encontradas no PD como, por exemplo, a representação do conhecimento na forma

de regras, além de algumas das características e vantagens encontradas no PI como, por exemplo, a flexibilidade de expressão e nível apropriado de abstração, via elementos similares aos objetos, ainda que mais aprimorados (SIMÃO; STADZISZ, 2008) (SIMÃO; STADZISZ, 2009a).

Em linhas gerais, o PON propõe a divisão da computabilidade em dois grandes grupos relacionados entre si por meio de notificações de seus constituintes (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009): (i) grupo que trata do processamento facto-execucional por meio de entidades chamadas *Fact-Base-Elements* (*FBEs*). Os *FBEs* são as entidades responsáveis por descrever estados (*Attributes*) e serviços (*Methods*), de forma análoga (mas distinta) aos objetos do Paradigma Orientado a Objetos (POO); (ii) grupo que trata do processamento lógico-causal por meio de entidades chamadas de *Rules*, de forma análoga (mas distinta) aos Sistemas Baseados em Regras (SBR) (RICH; KNIGHT, 1991).

A relação entre as entidades constituintes do PON é ilustrada na Figura 1 por meio de um exemplo no contexto de um sistema de correlação de sensores. Em suma, os *FBEs* permitem representar entidades do mundo real ou abstrato, sendo compostos por meio de entidades notificantes chamadas de *Attributes* e *Methods*. Cada *Attribute* é capaz de notificar as respectivas *Rules* por meio das *Premises* e *Conditions*, enquanto cada *Method* é capaz de ser instigado pelas *Rules* por meio das *Actions* e *Instigations*.

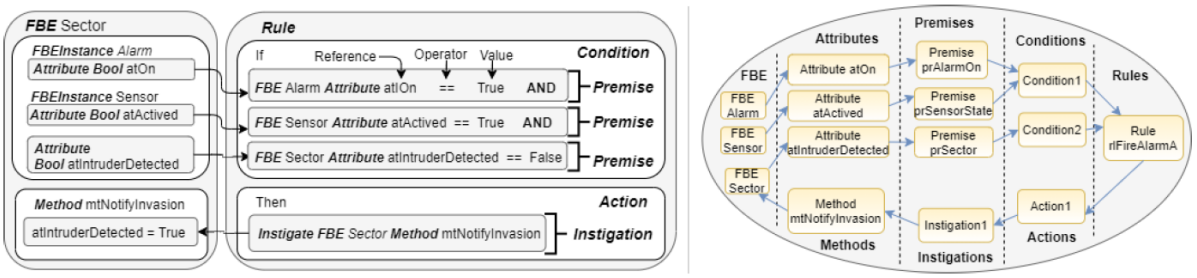


Figura 1 – Exemplo de interação entre as entidades do PON e ciclo de notificações.
Fonte: Oshiro *et al.* (2021)

De forma resumida, as *Rules* apresentam o conhecimento lógico-causal descrito pelas *Conditions* e pelas *Actions*. As *Conditions* são responsáveis pelas decisões e, de forma complementar, as *Actions* são responsáveis pelas ações. O conhecimento lógico das *Conditions* é expresso por meio de operações lógicas (*e.g.*, conjunção ou disjunção) sobre as *Premises* que realizam as avaliações baseadas em um comparador relacional (*e.g.*, igual, diferente, maior e menor) sobre os *Attributes* de um *FBE*. As *Actions*, por sua vez, são relacionadas com uma ou mais *Instigations* que são ativadas quando a *Action* é acionada (pela *Rule*). Quando ativadas, as *Instigations* são responsáveis pela execução dos *Methods* que operam, normalmente, sobre os *Attributes* das entidades *FBEs*, reativando o fluxo de notificações (BANASZEWSKI, 2009) (RONSZCKA, 2019) (OSHIRO *et al.*, 2021).

Esse mecanismo colaborativo e de notificações pontuais entre as entidades do PON promovem, entre outros benefícios, um desacoplamento estrutural ou também chamado de acoplamento mínimo entre os elementos. Esse desacoplamento intrínseco permite (ou ao menos

favorece) o desenvolvimento de aplicações paralelizáveis e/ou distribuídas (RONSZCKA, 2019) (SCHÜTZ, 2019) (KERSCHBAUMER *et al.*, 2015) (LINHARES, 2015) (OLIVEIRA, 2019) (NEGRINI, 2019).

1.2 Motivação

O advento da IoT trouxe uma nova realidade para o mundo de desenvolvimento de sistemas, aumentando a necessidade de desenvolvimento de *softwares* cada vez mais complexos, especialistas e heterogêneos (AL-MASRI *et al.*, 2020). Uma quantidade cada vez maior de sistemas passa a interagir em redes de comunicação cada vez mais interligadas e autônomas. Somado a isso, tem-se a 'Crise de *Software*' e as dificuldades ou ineficiências dos paradigmas de programação dominantes (RONSZCKA, 2019). Nesse contexto, o PON se apresenta como uma alternativa, tão interessante quanto inovadora. O PON, enquanto paradigma de programação/desenvolvimento, tem os objetivos de tornar menos árdua a tarefa do desenvolvimento de sistemas por apresentar abstração e expressividade de regras em alto nível, tornar o código e sua execução mais eficiente por evitar redundâncias (estruturais e temporais) e, por fim, tornar a sua execução paralelizável/distribuível por garantir o desacoplamento de suas entidades (NEVES, 2021) (SIMÃO; STADZISZ, 2009b). Neste contexto, o PON tem se mostrado altamente aderente ao ecossistema distribuído, pois apresenta um mecanismo de colaboração entre as entidades (distribuídas ou não) de forma pontual e objetiva proporcionando um acoplamento mínimo, o que favorece o paralelismo e/ou distribuição fina de processamento (OLIVEIRA *et al.*, 2018) (BARRETO; VENDRAMIN; SIMÃO, 2018) (SIMÃO *et al.*, 2014) (SCHÜTZ, 2019) (KERSCHBAUMER *et al.*, 2015) (LINHARES, 2015) (OLIVEIRA, 2019) (NEGRINI, 2019).

Durante os últimos anos, o PON vem apresentando muitas evoluções, tanto do ponto de vista do estado da arte, com o refinamento dos conceitos que constituem o paradigma ou por meio de linguagens de programação ainda prototipais no âmbito da chamada Tecnologia LingPON, bem como do ponto de vista do estado da técnica, com o desenvolvimento e aprimoramento de *frameworks* do PON sobre linguagens usuais orientada a objetos (como C++) permitindo que essas venham a serem usadas de maneira orientada a notificações (RONSZCKA, 2019) (NEVES, 2021). Porém, o conceito de distribuição das entidades do PON enquanto estado da técnica e estado da arte permanece pouco explorado frente às demandas de desenvolvimento e tecnologias relacionadas com sistemas distribuídos, particularmente para IoT, principalmente em relação aos protocolos e arquiteturas de comunicação geralmente utilizados nesse contexto.

1.3 Justificativa

Devido ao modelo orientado a notificações e ao desacoplamento decorrente desse modelo, as entidades constituintes do PON são distribuíveis por definição. Alguns trabalhos, como os de Talau (2016), Oliveira *et al.* (2018), Barreto, Vendramin e SIMÃO (2018), Xavier (2014), Simão *et al.* (2014), Liao *et al.* (2017) e Mamann *et al.* (2021) já exploraram o PON nesse contexto, porém ainda existem lacunas e conceitos não avaliados em teoria e tecnologias prototipais (estado da arte) e em tecnologias estáveis (estado da técnica).

No estado da técnica do PON, apesar de haver algumas materializações disponíveis (*e.g.*, uma série de arquétipos ou *frameworks* (NEVES, 2021), (CHIERICI, 2020), (MARTINI; SIMÃO; LINHARES, 2018), (NEGRINI *et al.*, 2019) (OLIVEIRA, 2019), (HENZEN, 2015), (VALENÇA, 2012), (RONSZCKA, 2012), (BELMONTE; SIMÃO; STADZISZ, 2012), (BANASZEWSKI, 2009), (SIMÃO; STADZISZ, 2008) e (SIMÃO *et al.*, 2012)) nenhuma delas contempla totalmente a potencialidade das propriedades elementares do PON. Nomeadamente, tais propriedades elementares do PON são (NEVES, 2021): (i) desenvolvimento em alto nível permitindo alguma facilidade de programação; (ii) desacoplamento permitindo paralelismo; (iii) desacoplamento permitindo distribuição; (iv) código não redundante (estruturalmente e temporalmente). Atualmente, dentre os *frameworks* existentes, destaca-se o recente *Framework* PON C++ 4.0 que, dentre outros benefícios, apresenta três das quatro potencialidades das propriedades elementares do PON, carecendo apenas da propriedade de distribuição (NEVES, 2021).

Ainda, orbitando a esfera dos *frameworks*, tem-se alguns trabalhos e técnicas existentes que abordam a distribuição do PON: Talau (2016), Barreto, Vendramin e SIMÃO (2018), Simão *et al.* (2014), Oliveira *et al.* (2018), Mendes *et al.* (2019) e Mamann *et al.* (2021). Porém, esses trabalhos apresentam propostas de distribuição ainda não completas por não permitirem a distribuição de todos os elementos do PON. Além disso, os trabalhos abordando o PON no contexto de sistemas distribuídos divergem em protocolos e modelos de distribuição, sendo necessário um estudo amplo com o objetivo de convergir os esforços teóricos e práticos em uma mesma direção, particularmente no tocante à IoT.

Destaca-se que os sistemas no contexto de IoT utilizam geralmente um protocolo de comunicação padronizado por entidades internacionais (AL-MASRI *et al.*, 2020). Ainda, é pertinente destacar também que cada protocolo em IoT apresenta características específicas, o que os torna adequados ao uso em determinados contextos. Dito isso, para permitir a ampliação do escopo de experimentos do PON no ambiente IoT, faz-se necessário também a adequação do PON para o uso de protocolos padronizados e utilizados para IoT de forma a permitir a integração também com sistemas heterogêneos. Ademais, considerando-se o contexto em que os sistemas IoT estão inseridos, é importante a utilização de um protocolo que seja leve, escalável, interoperável e extensível.

1.4 Objetivo Geral

O objetivo geral deste trabalho é promover avanços ao Paradigma Orientado a Notificações (PON) no contexto de sistemas distribuídos para IoT de forma a permitir a distribuição das entidades constituintes do PON e, ao mesmo tempo, prover a integração do PON com aplicações IoT por meio do uso de arquitetura e protocolo de comunicação padronizado.

1.5 Objetivos Específicos

Para atingir o objetivo geral, propõe-se os seguintes objetivos específicos:

- Conformar o *Framework* PON C++ 4.0 para permitir a implementação de aplicações distribuídas no PON, por meio da distribuição potencial de suas entidades, alcançando assim um *Framework* PON C++ 4.0 IoT.
- Criar e implementar testes unitários e de integração para o *Framework* PON C++ 4.0 IoT com o objetivo de facilitar a manutenção e, eventualmente, a expansão do *framework*.
- Permitir a integração entre aplicações distribuídas PON via *Framework* PON C++ 4.0 IoT com outras aplicações, por meio de um protocolo padronizado, nomeadamente o MQTT.
- Realizar experimentos de aplicações do PON em ambiente IoT com o *Framework* PON C++ 4.0 IoT.
- Comparar aspectos funcionais e não funcionais do PON e do POE por meio de experimentos em ambiente de IoT.

1.6 Organização do Trabalho

Este documento está dividido em quatro capítulos, além da presente Introdução. No Capítulo 2 é apresentada uma revisão dos aspectos gerais de IoT, dos paradigmas de programação e, particularmente, do PON, relatando em geral as materializações existentes do PON e particularmente os trabalhos realizados com o PON no contexto de sistemas distribuídos. No Capítulo 3, por sua vez, é apresentado o desenvolvimento realizado neste trabalho, mais precisamente a proposta de distribuição do PON na forma de um *Framework* PON C++ 4.0 IoT. No Capítulo 4 são apresentados os experimentos e resultados. Por fim, o Capítulo 5 apresenta as conclusões e os trabalhos futuros.

2 REVISÃO DA LITERATURA

Neste capítulo são apresentados os conceitos e tópicos utilizados no desenvolvimento deste trabalho. A Seção 2.1 apresenta os conceitos de Internet das Coisas, destacando as principais propriedades, desafios e protocolos utilizados. Em seguida, a Seção 2.2 aborda os conceitos gerais de paradigmas de programação e a Seção 2.3 detalha o emergente Paradigma Orientado a Notificações (PON). Sobre o PON, são apresentados os conceitos e as materializações desenvolvidas com o objetivo de facilitar e promover o desenvolvimento de aplicações segundo esse paradigma. Na sequência, a Seção 2.4 apresenta os trabalhos relacionados à esta pesquisa no tocante ao PON. Dentre os trabalhos relacionados, destaca-se o *Framework* PON C++ 4.0, apresentado em detalhes na Seção 2.4.8.

2.1 Internet das Coisas

A Internet das coisas, cujo acrônimo em inglês é IoT (*Internet of Things*), pode ser definida como um conjunto de dispositivos (físicos e virtuais) conectados entre si e/ou com serviços em nuvem com o objetivo de compartilhar dados, compartilhar funcionalidades ou aprimorar operações, geralmente de forma autônoma e com base em tecnologias de comunicação interoperáveis (SAMUEL; SIPES, 2019) (DORSEMAINE *et al.*, 2015) (ITU-T STANDARD, 2012). Os principais objetivos para a adoção da IoT são as novas possibilidades de produtos e funcionalidades que podem ser adicionados para transformar as atividades e negócios em determinada área, adicionando eficiência no uso dos recursos e potencialmente incrementando ou até mesmo criando novas formas de monetização para as empresas por meio da venda de novos produtos e/ou serviços (SAMUEL; SIPES, 2019).

Na literatura, o termo IoT é muitas vezes usado como sinônimo para Redes de Sensores sem Fio (RSSF). Porém, apesar de IoT e rede de sensores sem fio se relacionarem, do ponto de vista conceitual existem diferenças notáveis. A IoT pode ser entendida como um novo paradigma, integrando tecnologias já existentes como as RSSF, RFID (*Radio Frequency IDentification*), computação em nuvem, *Middlewares* e aplicações para interação com usuários (MANRIQUE; RUEDA-RUEDA; PORTOCARRERO, 2016). Manrique, Rueda-Rueda e Portocarrero (2016) consideram a IoT como a terceira era da Internet. Enquanto a primeira e a segunda eras se caracterizaram pela conexão de pessoas à Internet por meio de computadores pessoais e dispositivos móveis, respectivamente, o desafio atual é a conexão de objetos cotidianos, criando uma conexão entre o mundo físico e o digital. Nesse sentido, as RSSFs podem ser consideradas parte integral da IoT promovendo a conexão dos objetos (MANRIQUE; RUEDA-RUEDA; PORTOCARRERO, 2016).

O trabalho de Manrique, Rueda-Rueda e Portocarrero (2016) também apresenta distinções a partir da origem dos termos IoT e de outras áreas de estudo. Conforme mostrado na Figura 2, no conceito etimológico, o termo IoT surge como uma tendência: (i) da Internet, sendo

considerada uma subárea da computação distribuída e das Telecomunicações (mais especificamente das Redes de Computadores); (ii) das comunicações 'Máquina para Máquina' (*Machine to Machine* ou M2M), sendo considerada uma subárea dos sistemas ciber-físicos e embarcados. Neste contexto, os sistemas IoT usam a Internet para garantir que cada objeto ou 'coisa' tenha um endereço identificável (MANRIQUE; RUEDA-RUEDA; PORTOCARRERO, 2016).

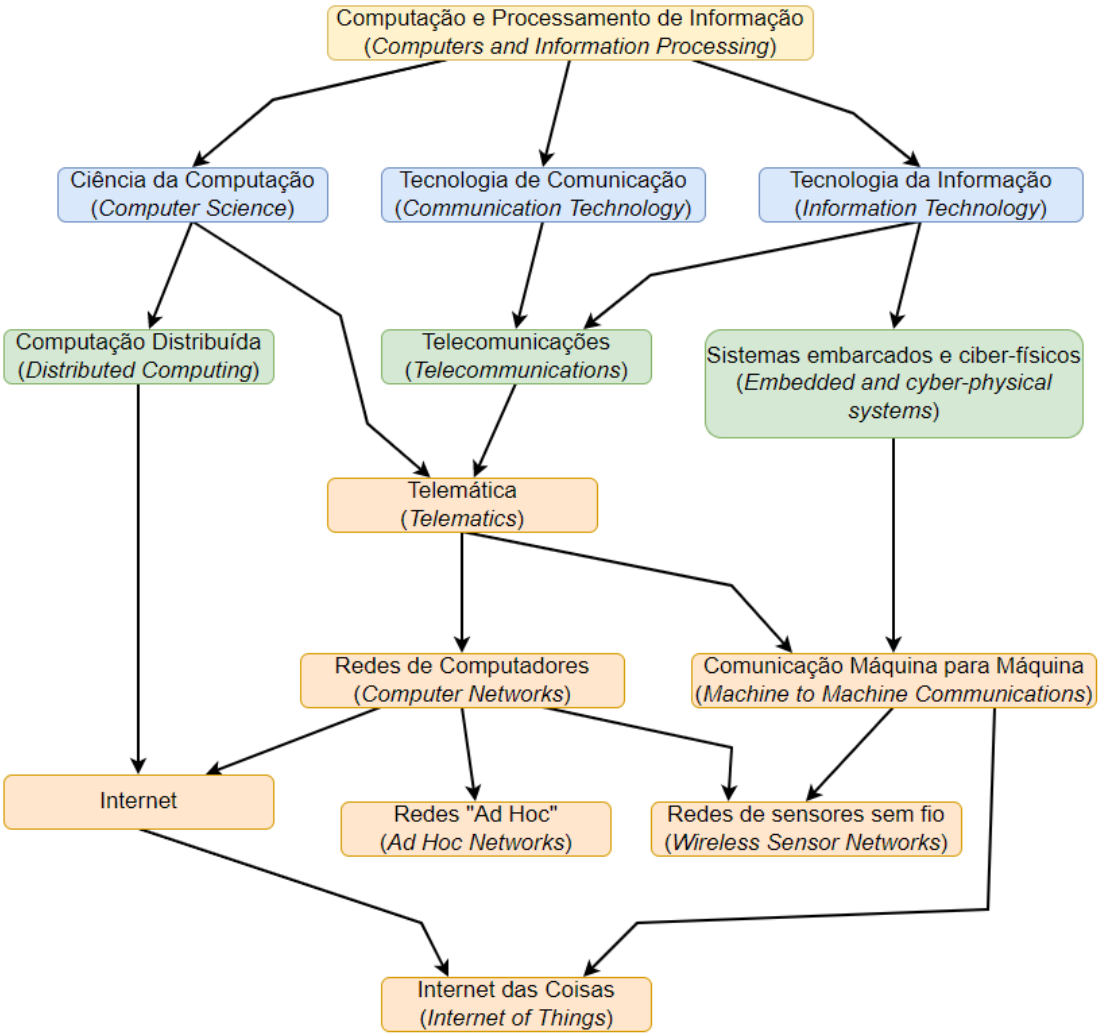


Figura 2 – Origem etimológica do termo IoT.

Fonte: Adaptado de Manrique, Rueda-Rueda e Portocarrero (2016).

As aplicações IoT possuem requisitos específicos, dentre os quais destacam-se (COULOURIS; DOLLIMORE; KINDBERG, 2011) (SAMUEL; SIPES, 2019) (MANRIQUE; RUEDA-RUEDA; PORTOCARRERO, 2016) (ITU-T STANDARD, 2012): (a) a Heterogeneidade, pois os dispositivos conectados podem ser diferentes em termos de sistemas operacionais, fabricantes, linguagens de programação, redes de comunicação e recursos de hardware; (b) a Segurança e Privacidade, incluindo todos os níveis, incluindo os dispositivos, rede de comunicação, infraestrutura e serviços; (c) a Escalabilidade, para suportar o aumento de recursos e usuários; e (d) a Qualidade do Serviço, pois o sistema deve atender às expectativas principalmente em termos de confiabilidade, segurança e desempenho.

Alguns autores destacam também as características ou os desafios normalmente envolvidos nos cenários e dispositivos de IoT (SUEDA; SATO; HASUIKE, 2019) (DORSEMAINE *et al.*, 2015) (TAIVALSAARI; MIKKONEN, 2018) (HANDOSA; GRAčANIN; ELMONGUI, 2017). Dentre as principais características, geralmente estão presentes as limitações de recursos computacionais (tais como, processamento, memória, energia e largura de banda) e/ou a mobilidade dos dispositivos.

Como forma de atender às demandas crescentes e desafios das aplicações para IoT, existe uma grande variedade de protocolos de aplicação. Tais protocolos promovem a padronização na troca de mensagens entre dispositivos heterogêneos, além de cobrirem, ao menos em parte, os desafios de segurança, escalabilidade e qualidade de serviço (AL-MASRI *et al.*, 2020) (SUEDA; SATO; HASUIKE, 2019). Os protocolos mais comuns nesse contexto são apresentados na Seção 2.1.1.

2.1.1 Protocolos de Comunicação

O advento da IoT promoveu o rápido aumento da quantidade e da velocidade de geração, consumo e processamento de dados (HASAN, 2022). Porém, para que a troca de dados entre os dispositivos IoT aconteça, é necessário a definição e o uso de protocolos de comunicação ou *frameworks* que padronizem e estabeleçam regras de criação e troca de mensagens entre os sistemas. Atualmente, existem vários protocolos utilizados em aplicações IoT, cada qual com suas particularidades e uso apropriado. Dentre os principais protocolos utilizados, pode-se citar (AL-MASRI *et al.*, 2020): *HyperText Transfer Protocol* (HTTP) - e sua versão com uma camada adicional de segurança *HyperText Transfer Protocol Secure* (HTTPS), *Message Queuing Telemetry Transport* (MQTT), *Advanced Message Queuing Protocol* (AMQP), *Constrained Application Protocol* (CoAP), *Extensible Messaging and Presence Protocol* (XMPP), *Data Distribution Service* (DDS), entre outros.

Conforme apresentado na Figura 3, dentre os vários protocolos utilizados em aplicações IoT, destacam-se principalmente o HTTPS e o MQTT, sendo atualmente os dois protocolos mais utilizados e que apresentam maior suporte em plataformas comerciais (AL-MASRI *et al.*, 2020). Os detalhes dos protocolos MQTT e o HTTP são explicados nas Seções 2.1.1.1 e 2.1.1.2, respectivamente.

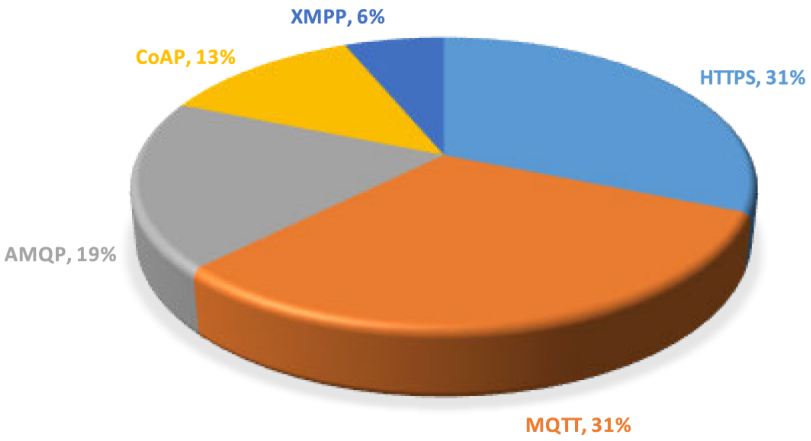


Figura 3 – Protocolos de comunicação suportados por aplicações comerciais.
Fonte: Al-Masri *et al.* (2020).

2.1.1.1 *Message Queuing Telemetry Transport* - MQTT

O *Message Queuing Telemetry Transport*, também conhecido pelo acrônimo MQTT, é um protocolo de aplicação definido para a troca de mensagens entre sistemas (OASIS STANDARD, 2014). O MQTT se destaca por ser computacionalmente leve, ser um padrão aberto, simples e fácil de ser implementado. O protocolo foi desenvolvido para ser usado na comunicação de diversos tipos de dispositivos, incluindo os que possuem poucos recursos computacionais (memória, processamento, energia), como é o caso de dispositivos geralmente usados em aplicações para IoT.

O MQTT foi desenvolvido por Dr. Andy Stanford Clark da IBM e Arlen Nipper da Arcom em 1999 sendo que os seus princípios de *design* se concentraram em minimizar a largura de banda da rede e os requisitos de recursos dos dispositivos, garantindo entrega confiável de mensagens. Em 2013, o protocolo foi padronizado pela OASIS (OASIS STANDARD, 2014) (AL-MASRI *et al.*, 2020).

O MQTT funciona segundo o padrão *publish/subscribe* o qual permite o desacoplamento das entidades e a comunicação ‘um para um’ ou ‘um para muitos’, isto é, uma única mensagem de um produtor pode ser entregue para um ou vários consumidores/assinantes. Conforme exemplificado na Figura 4, no padrão *publish/subscribe*, os clientes MQTT (produtores e consumidores) interagem uns com os outros por intermédio de tópicos disponibilizados em um servidor de mensagens centralizado, conhecido como *broker*. Os consumidores registram no *broker* o seu interesse em um determinado tópico e tão logo um produtor publique alguma mensagem no respectivo tópico, o *broker* envia uma notificação para os consumidores interessados, com o conteúdo da mensagem publicada.

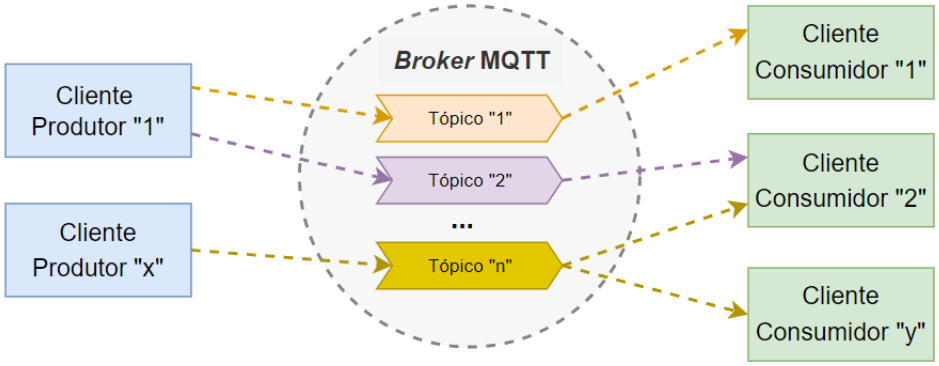


Figura 4 – Arquitetura de comunicação entre clientes produtores e consumidores no protocolo MQTT.

Conforme exemplificado na Figura 5, um cliente que deseja receber informações de temperatura de um sensor envia uma mensagem ao *broker* solicitando conexão (*connect*) e, tendo sua solicitação confirmada (*connack*), subscreve no tópico de interesse (*Subscribe /temp*). Uma vez que o sensor publique uma nova mensagem no respectivo tópico (*Publish /temp*), o *broker* envia uma notificação para o cliente previamente subscrito nesse tópico (MISHRA; KERTESZ, 2020).

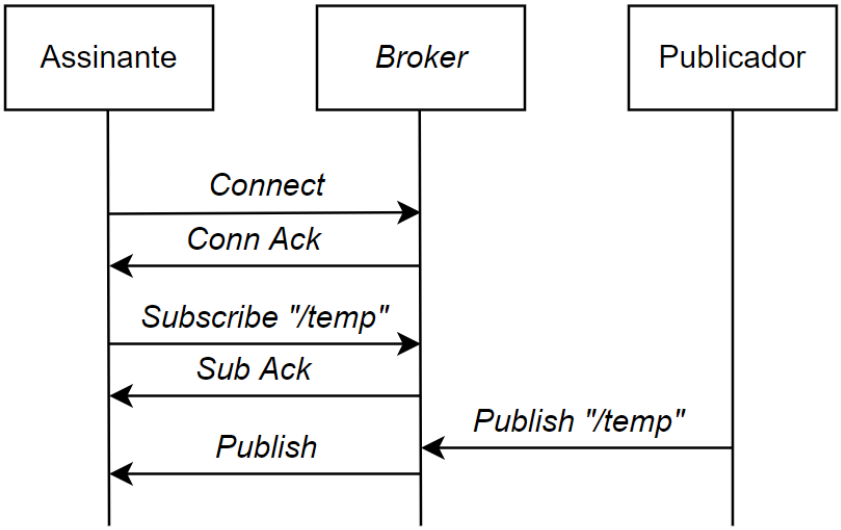


Figura 5 – Exemplo de fluxo de mensagens MQTT

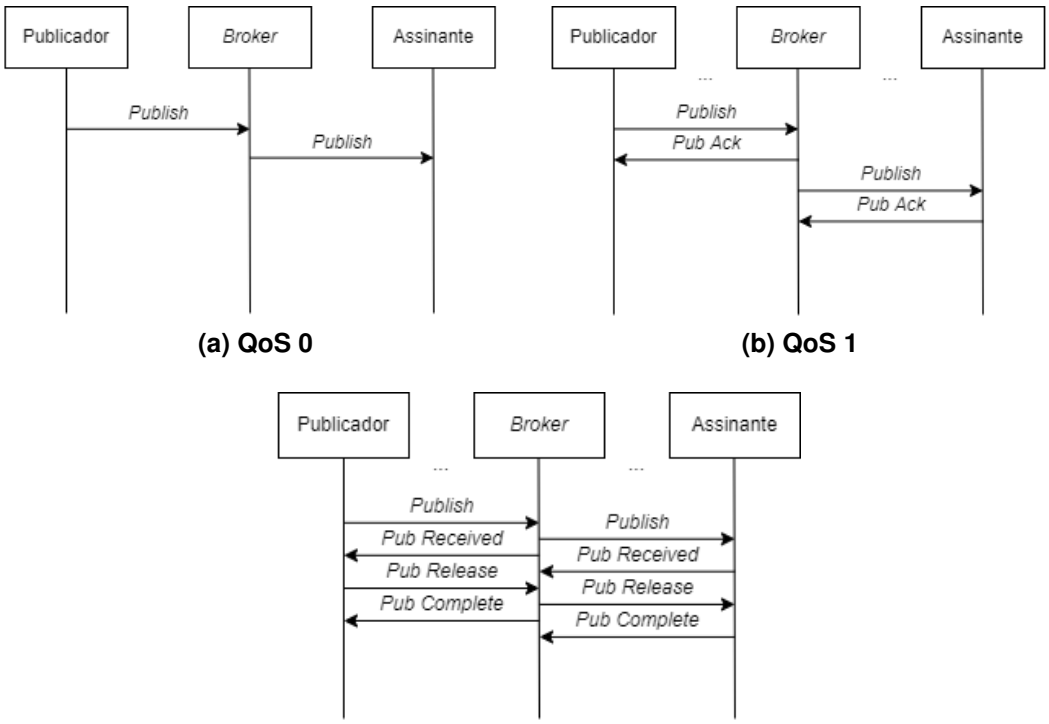
Fonte: Adaptado de Al-Masri *et al.* (2020).

O MQTT opera geralmente sobre o protocolo de transporte TCP (*Transmission Control Protocol*), mas pode operar sobre outro protocolo desde que sua entrega seja confiável, ordenada e que permita fluxo de dados bidirecional. É pertinente ressaltar que existe a versão MQTT para rede de sensores (MQTT-*Sensor Networks* ou MQTT-SN) que pode operar sobre outras camadas de transporte, principalmente as utilizadas em redes sem fio, geralmente susceptíveis a perdas de pacotes.

A camada de segurança provê o uso de *Transport Layer Security* (TLS) (MISHRA; KERTESZ, 2020) (AL-MASRI *et al.*, 2020). Em linhas gerais, o TLS, em sua versão mais recente

1.2 (DIERKS; RESCORLA, 2008), permite que duas aplicações se comuniquem de uma forma concebida para evitar escutas, adulteração ou falsificação de mensagens. A especificação do TLS define duas partes principais: o protocolo de registro TLS; e o *handshake*. O protocolo de registro utiliza criptografia simétrica (por exemplo, AES - *Advanced Encryption Standard*) e verificação de integridade utilizando funções *hash* (por exemplo, SHA-1 - *Secure Hash Algorithm 1*). O *handshake*, por sua vez, define a versão do TLS utilizada (TLS 1.0, 1.2, 1.3 etc.), o algoritmo de criptografia e a autenticidade dos clientes, por meio de chaves assimétricas e assinatura digital (DIERKS; RESCORLA, 2008).

Conforme apresentado na Figura 6, o MQTT permite também a definição de padrões de Qualidade de Serviço (QoS - *Quality of Service*) referentes à entrega das mensagens, podendo ser definido como (OASIS STANDARD, 2014) (AL-MASRI *et al.*, 2020) (MISHRA; KERTESZ, 2020):



(a) QoS 0 **(b) QoS 1**

(c) QoS 2

Figura 6 – Sequência de publicação de mensagens MQTT para diferentes níveis de QoS.
Fonte: Adaptado de Sueda, Sato e Hasuike (2019)

- QoS0 (‘no máximo uma vez’): usado para mensagens que podem ser eventualmente perdidas sem prejuízo ao sistema. Por exemplo, os dados de algum sensor que são publicados frequentemente e o não recebimento de uma das publicações não prejudicaria a funcionalidade do sistema, pois uma nova publicação será recebida em breve. Conforme apresentado na Figura 6a, o QoS0 não usa nenhum tipo de confirmação para o recebimento de mensagens.

- QoS1 (‘pelo menos uma vez’): garante o recebimento de uma mensagem pelo menos uma vez, podendo eventualmente receber mensagens duplicadas. Conforme exemplificado na Figura 6b, o QoS1 utiliza uma mensagem de confirmação de recebimento de notificação de uma publicação (*Publish Ack*).

- QoS2 (‘exatamente uma vez’): como o próprio nome sugere, garante o recebimento de somente uma mensagem pelo destinatário. Conforme apresentado na Figura 6c, o QoS2 utiliza um *handshake* de quatro etapas que se inicia com a publicação (*Publish*) de um pacote com QoS2 contendo um identificador, a partir do qual é gerado, pelo receptor, uma confirmação de recepção, chamada de PUBREC (*Publish Received*). Ao receber o PUBREC, o publicador gera uma notificação de recebimento PUBREL (*Publish Release*) a qual é novamente confirmada por meio da mensagem PUBCOMP (*Publish Complete*) enviada pelo receptor. A mensagem de PUBCOMP finaliza o processo garantindo que a mensagem tenha sido entregue exatamente uma vez para o cliente assinante.

Outro aspecto padronizado pelo protocolo MQTT é a estrutura de uma mensagem, conforme apresentada na Figura 7. Os dois primeiros bytes estão sempre presentes e representam o cabeçalho da mensagem. O restante do conteúdo é variável conforme o tipo da mensagem e o conteúdo enviado. O limite máximo por mensagem é de 256 MB (AL-MASRI *et al.*, 2020).

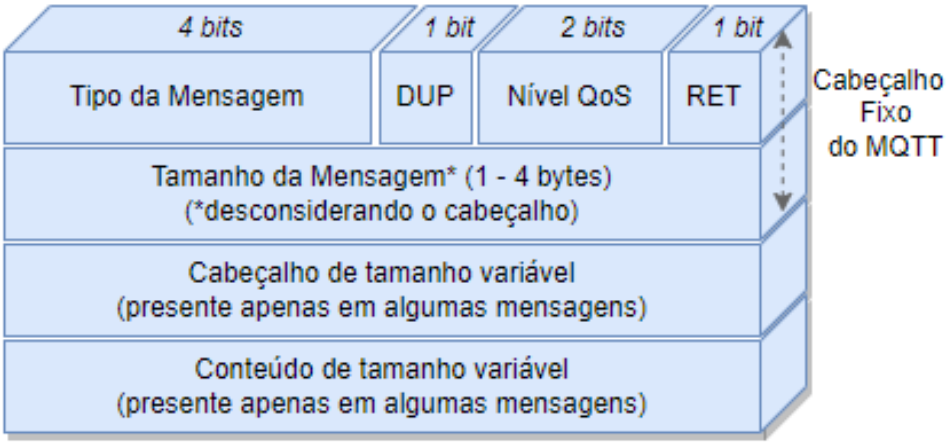


Figura 7 – Estrutura de uma mensagem MQTT.

Fonte: Adaptado de Al-Masri *et al.* (2020).

O primeiro byte do cabeçalho é composto por 4 bits representando o tipo de mensagem (também conhecida como *control byte*). Os tipos de mensagens são apresentados no Quadro 1. Os campos subsequentes representam as *flags* da mensagem. Comumente, as *flags* são utilizadas na mensagem de *Publish*, apresentando um bit correspondente à *flag* de duplicação, seguido por dois bits representando o nível de QoS esperado na comunicação e mais um bit representando a *flag* de retenção. Neste contexto, a *flag* de duplicação com valor 0 indica que é a primeira tentativa de publicação da mensagem em questão, enquanto o valor 1 indica uma

nova tentativa de envio. Por sua vez, a *flag* de retenção com valor 1 indica ao *broker* que a mensagem em questão deve ser armazenada e enviada para novos clientes que se inscreverem no respectivo tópico. Somente uma mensagem pode ser armazenada por tópico, mantendo-se a última recebida com a *flag* habilitada (OASIS STANDARD, 2014).

Quadro 1 – Tipos de mensagens descritas no MQTT.

Nome	Valor	Sentido da mensagem	Descrição
Reservado	0	Proibida	Reservado
CONNECT	1	Cliente para o *Broker*	Requisição de conexão
CONNACK	2	*Broker* para o Cliente	*Acknowledgment* ou *Ack* (Conhecimento) de conexão
PUBLISH	3	Cliente para o *Broker* ou *Broker* para Cliente	Publicação de mensagem
PUBACK	4	Cliente para o *Broker* ou *Broker* para Cliente	*Ack* de Publicação
PUBREC	5	Cliente para o *Broker* ou *Broker* para Cliente	*Publish received* (Publicação Recebida), utilizada juntamente com o QoS 2
PUBREL	6	Cliente para o *Broker* ou *Broker* para Cliente	*Publish release* (Liberação de Publicação), utilizada juntamente com o QoS 2
PUBCOMP	7	Cliente para o *Broker* ou *Broker* para Cliente	*Publish complete* (Publicação completa), utilizada juntamente com o QoS 2
SUBSCRIBE	8	Cliente para o *Broker*	Requisição de inscrição em determinado tópico
SUBACK	9	*Broker* para o Cliente	*Ack* de inscrição
UNSUBSCRIBE	10	Cliente para o *Broker*	Cancelamento de inscrição
UNSUBACK	11	*Broker* para o Cliente	*Ack* do cancelamento de inscrição
PINGREQ	12	Cliente para o *Broker*	Requisição de PING
PINGRESP	13	*Broker* para o Cliente	Resposta de PING
DISCONNECT	14	Cliente para o *Broker* ou *Broker* para Cliente	Notificação de desconexão
AUTH	15	Cliente para o *Broker* ou *Broker* para Cliente	Autenticação entre cliente e *Broker*

Fonte: Adaptado de OASIS Standard (2014)

O segundo byte do cabeçalho representa o tamanho da mensagem enviada, desconsiderando-se o tamanho do cabeçalho em si. Os demais campos variam de acordo com o tipo da mensagem (OASIS STANDARD, 2014).

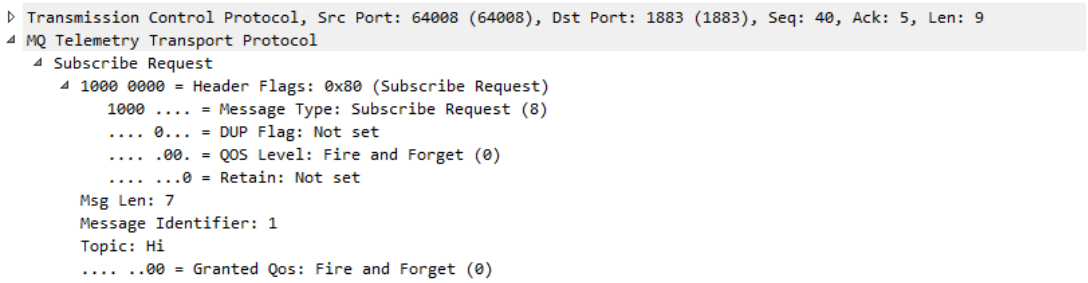


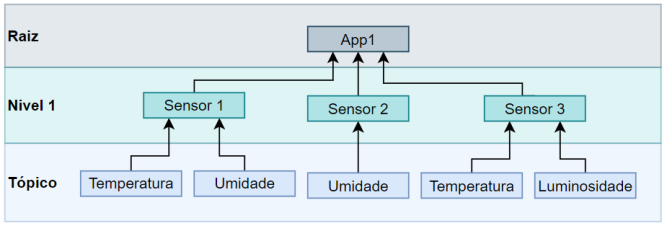
Figura 8 – Exemplo de uma mensagem *Subscribe* MQTT.

Fonte: Balaji (2017).

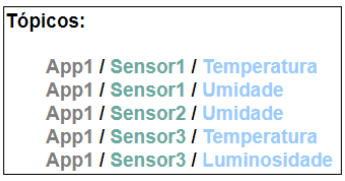
Como exemplo, a Figura 8 apresenta um exemplo de mensagem de inscrição (*Subscribe*) interpretada pelo software *Wireshark*. Nota-se que o cabeçalho identifica o tipo de mensagem (8 - *Subscribe Request*) além da sinalizadora ou *flag* de duplicação (neste caso com

valor zero, significando que é a primeira tentativa de envio deste pacote), o nível de QoS e a *flag* de retenção. Observa-se em seguida o tamanho da mensagem, seguida por um identificador e o tópico correspondente à inscrição em questão (OASIS STANDARD, 2014).

Além da definição do conteúdo das mensagens, a especificação do MQTT descreve a possibilidade de criação de uma estrutura hierárquica em árvore para os tópicos conforme exemplificado na Figura 9. De forma sucinta, os tópicos permitem aos clientes do *broker* trocarem informações dentro de um determinado campo semântico, ou seja, representam um grupo de mensagens referentes a um mesmo contexto. Os tópicos permitem também que sejam aplicados filtros nas mensagens para cada cliente conectado. A Figura 9a apresenta um exemplo de tópicos para um sistema com múltiplos sensores.



(a) Exemplo gráfico de tópicos com o MQTT.



(b) Exemplo textual de tópicos com o MQTT.

Figura 9 – Exemplo de uma estrutura hierárquica de tópicos com o MQTT.

A Figura 9b apresenta em formato de texto a mesma estrutura descrita na Figura 9a, porém utilizando os separadores apropriados, isto é, o caractere / (barra oblíqua) como separador de níveis. É importante salientar que os nomes dos tópicos precisam ter pelo menos um caractere, podem conter espaço entre os identificadores e diferenciam letras maiúsculas e minúsculas. Não há limite para o número de níveis de tópicos e nem limite no número de raízes da estrutura hierárquica de tópicos (OASIS STANDARD, 2014).

A partir da estrutura hierárquica de árvore criada pelos tópicos é possível aos clientes realizarem a inscrição no *broker* por meio de caracteres reservados, conhecidos como *wildcards*. O uso dos *wildcards* nas operações de inscrição permitem a seleção de um conjunto de tópicos em uma única requisição. Os dois modos de uso de inscrição com *wildcards* disponíveis são: o modo multinível; e o modo nível único. É pertinente ressaltar que os *wildcards* estão disponíveis apenas para inscrições, não se aplicando à publicações (OASIS STANDARD, 2014).

O modo de inscrição com *wildcard* multinível permite ao cliente se inscrever em todos os tópicos a partir de determinado nível, marcado pelo caractere # (cardinal) (OASIS STANDARD, 2014). Por exemplo, considerando-se a estrutura de tópicos apresentada na Figura 9b, uma inscrição em *App1/#* seria o equivalente a uma inscrição em todos os tópicos disponíveis a partir de *App1/*, o que engloba *Sensor1*, *Sensor2* e *Sensor3* e seus respectivos tópicos */Temperatura*, */Umidade* e */Luminosidade*. Esse tipo de subscrição pode se aplicar em diversos níveis, podendo ser, por exemplo, *App1/#* ou mesmo *#*, representando uma inscrição em todos os tópicos disponíveis no *broker*. Esse mecanismo de inscrições com *wildcard* multinível em *App1/#*

pode ser utilizado, por exemplo, por uma central de controle que recebe todas as informações de todos os sensores do ambiente para, eventualmente, executar uma ação pré-determinada.

O segundo modo de *wildcard* disponível é o modo de nível único, representado pelo caractere + (sinal de adição). Nesse modo, é possível se inscrever em todos os tópicos de um respectivo nível (OASIS STANDARD, 2014). Como exemplo, utilizando-se da estrutura de tópicos apresentada na Figura 9b, uma inscrição em '*App1/Sensor3/+*' corresponde a uma inscrição em '*App1/Sensor3/Temperatura*' e '*App1/Sensor3/Luminosidade*'. Outra opção, é a utilização do *wildcard* de nível único entre a descrição dos tópicos selecionados. Por exemplo, uma inscrição em '*App1/+/Temperatura*' representa uma inscrição em '*App1/Sensor1/Temperatura*' e '*App1/Sensor3/Temperatura*'. Esse mecanismo de inscrições com *wildcard* nível único em '*App1/+/Temperatura*' pode ser utilizado, por exemplo, por uma central de controle que recebe apenas as informações de temperatura de todos os sensores do ambiente para, eventualmente, executar uma ação pré-determinada.

Além das inscrições com *wildcards*, a especificação do MQTT v5 define o conceito de inscrições compartilhadas (*shared subscriptions*) (OASIS STANDARD, 2014). Uma inscrição compartilhada é realizada utilizando-se um prefixo especial, definido como '*$share/Nome Compartilhado/*', no qual o texto '*$share*' identifica que a inscrição será compartilhada e o texto '*Nome Compartilhado*' identifica um nome único para o grupo de clientes. Após o prefixo especial, a inscrição segue os mesmos padrões definidos para inscrições não compartilhadas (incluindo a possibilidade de uso dos *wildcards*). Diferentemente das publicações não compartilhadas na qual cada cliente recebe uma cópia das mensagens que atendam os filtros da inscrição, nas inscrições compartilhadas a mensagem é publicada somente para um dos clientes de cada grupo, de forma alternada. Esse modo permite que o processamento das mensagens seja balanceado (*load balancing*) entre vários clientes que realizem a mesma avaliação, ainda que com uma frequência menor de notificações, permitindo ao sistema escalar horizontalmente (OASIS STANDARD, 2014).

2.1.1.2 *HyperText Transfer Protocol* - HTTP

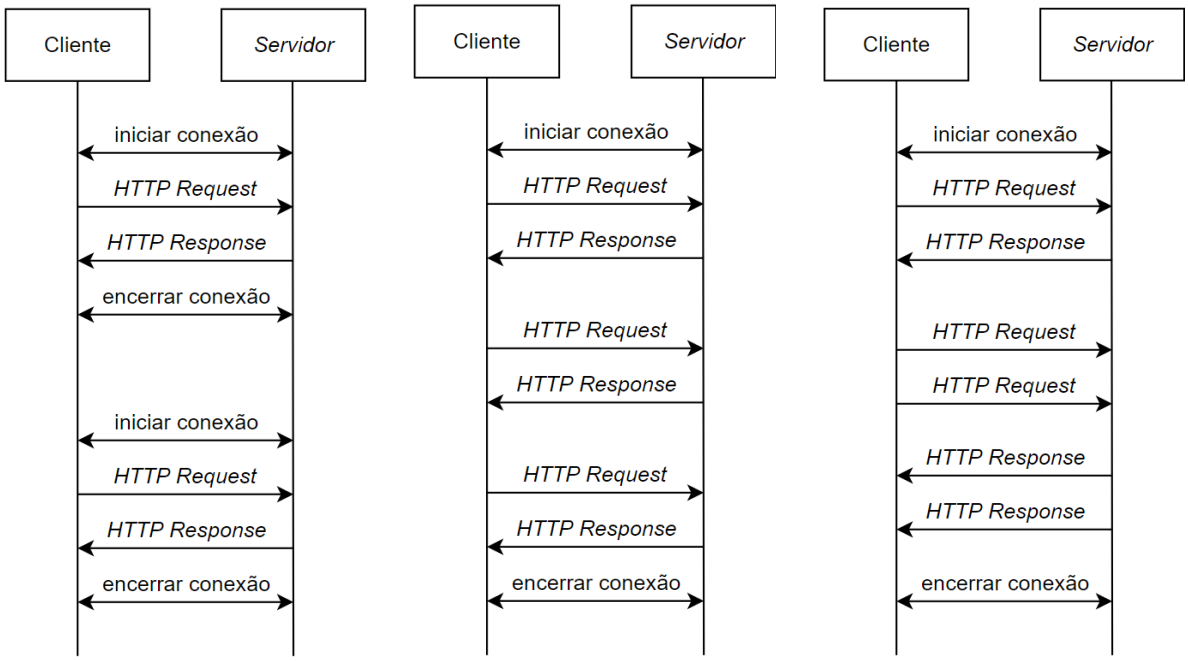
O *HyperText Transfer Protocol* também conhecido pelo acrônimo HTTP é um protocolo de aplicação genérico usado amplamente para comunicação dentro da *World Wide Web* (WWW) (AL-MASRI *et al.*, 2020). O HTTP é um protocolo baseado em perguntas e respostas (*request-response*) no qual um cliente envia uma requisição e um servidor gera uma resposta correspondente. Por conta dessa arquitetura, a comunicação ocorre apenas no modelo 'um para um'. Uma das funcionalidades chave do HTTP é a negociação de conteúdo, no qual os agentes envolvidos negociam qual a melhor representação dos dados para determinado usuário. O HTTP é um protocolo sem estado (*stateless*), ou seja, cada conjunto de requisição-resposta é tratado de forma independente e não relacionado com os demais (FIELDING *et al.*, 1999).

A especificação do HTTP descreve os *tokens* dos métodos permitidos para execução nos recursos. Os métodos apresentados são: *"OPTIONS", "GET", "HEAD", "POST", "PUT", "DELETE", "TRACE", "CONNECT"*, além da possibilidade de definição de métodos estendidos. De forma sucinta, os métodos podem ser descritos como (FIELDING *et al.*, 1999):

- GET: o método GET é usado para recuperar informações de um determinado recurso. Solicitações usando GET devem apenas retornar dados e não devem ter nenhum outro efeito sobre os dados.
- HEAD: o funcionamento do método HEAD é semelhante ao método GET, porém transfere apenas a linha de status e a seção de cabeçalho. Mais precisamente, o método HEAD solicita uma resposta de forma parecida ao método GET, diferenciando-se por não conter o corpo da resposta.
- POST: uma solicitação POST é usada para enviar dados ao servidor como, por exemplo, informações do cliente ou *upload* de arquivos.
- PUT: o método PUT substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.
- DELETE: o método DELETE remove um recurso específico.
- CONNECT: o método CONNECT estabelece um túnel para o servidor identificado pelo recurso de destino.
- OPTIONS: o método OPTIONS é usado para solicitar informações sobre as opções disponíveis no recurso identificado pelo URI. Este método permite que o cliente determine as opções e/ou requisitos associados a um recurso, sem implicar em uma ação sobre ele.
- TRACE: o método TRACE executa um teste de chamada *loopback* junto com o caminho para o recurso de destino.

O protocolo HTTP possui algumas versões sendo as mais comuns a 1.1 e 2.0. A última versão disponível é a 3.0, disponibilizada em 2018. Dentre as diferenças entre as versões do HTTP, destaca-se o fluxo de controle de mensagens, conforme apresentado na Figura 10. Enquanto na versão 1.0 (Figura 10a) a conexão é aberta e encerrada (*handshake*) em cada comunicação, a versão 1.1 (Figura 10b) permite o envio e recebimento de múltiplas requisições e respostas. De forma similar à versão 1.1, a partir da versão 2.0 (Figura 10c) é permitido o envio e recebimento de múltiplas requisições por conexão, podendo inclusive ocorrer mais de uma em paralelo (AL-MASRI *et al.*, 2020).

O endereçamento no HTTP ocorre por meio do uso de *Uniform Resource Identifier* (URI). A URI é um texto que identifica um recurso via nome, localização ou outra característica



(a) HTTP 1.0 **(b) HTTP 1.1** **(c) HTTP 2.0**

Figura 10 – Fluxo de mensagens HTTP em diferentes versões.

Fonte: Adaptado de Al-Masri *et al.* (2020)

(FIELDING *et al.*, 1999). Um dos exemplos mais comuns no contexto da Internet é o *Uniform Resource Locator* (URL), o qual utiliza-se da identificação dos recursos por meio do mecanismo de acesso primário (por exemplo, a sua localização de rede). Como exemplo de uma URL pode-se citar o site da UTFPR (*https://www.utfpr.edu.br/*) ou a operação de busca da palavra 'pesquisa' no site de buscas Google (*https://www.google.com/search?q=pesquisa*).

O HTTP opera geralmente sobre o TCP na camada de transporte e pode prover uma camada de segurança por meio do uso TLS (já explicado brevemente na Seção 2.1.1.1). Caso a camada de segurança esteja presente, o HTTP passa a ser chamado de HTTPS (*Hyper Text Transfer Protocol Secure*) (AL-MASRI *et al.*, 2020).

2.1.2 Considerações sobre os protocolos de comunicação para IoT

Em relação aos protocolos de comunicação, apresentados na Seção 2.1.1, observa-se que, na prática, cada um possui benefícios e limitações que os tornam adequado a determinado escopo de aplicações. Considerando-se os protocolos mais populares no contexto de IoT (AL-MASRI *et al.*, 2020), nomeadamente HTTP/HTTPS e MQTT, o Quadro 2 apresenta as principais propriedades e características observadas.

Quadro 2 – Comparação dos protocolos de comunicação.

Funcionalidade	HTTP	MQTT
Padrão de Mensagens	requisição / resposta	*Publish / Subscribe*
Transporte	TCP	TCP (exceto MQTT-SN)
Camada de rede	IPv4 ou IPv6	IPv4 ou IPv6
Mensagens Assíncronas	Não	Sim
Suporte à QoS	Não	Sim
Endereçamento	URI	Tópicos
Segurança	TLS	TLS
Distribuição de dados	1 para 1	1 para N, N para N
Entidade padronizadora	IETF, W3C	OASIS

Fonte: Adaptado de Al-Masri *et al.* (2020)

Além das características e propriedades, encontram-se na literatura alguns trabalhos que abordam a comparação dos diferentes protocolos existentes no contexto de IoT. Destacam-se os trabalhos com foco em IoT cuja comparação inclui o MQTT e o HTTP em relação à características e propriedades geralmente presentes nesse contexto:

- Wukkadada *et al.* (2018) apresentam uma comparação em relação ao consumo de energia e taxa de sucesso de transmissões, apontando que o MQTT consegue uma taxa de sucesso maior e um consumo de energia menor que o HTTP.

- Yokotani e Sasaki (2016a) e Yokotani e Sasaki (2016b) apresentam uma análise comparativa do HTTP e MQTT em relação ao uso da largura de banda em múltiplos cenários. Os cenários apresentados possuem como foco principal a IoT nos quais geralmente ocorre a transferência de uma grande quantidade de mensagens contendo pequenos blocos de dados em cada. O trabalho aponta que o MQTT é mais eficiente que o HTTP desde que os nomes dos tópicos no MQTT não sejam muito grandes (isto é, maiores que 690 bytes).

- Naik (2017) apresenta uma análise comparativa de múltiplos protocolos, incluindo MQTT e HTTP. Considerando-se apenas o HTTP e MQTT, os resultados apresentados mostram que o MQTT é mais eficiente que o HTTP. Em suma, o MQTT apresenta menor latência, menor uso de largura de banda, maior confiabilidade/QoS (*Reliability*), menor consumo de Energia e menor demanda por recursos do dispositivo. Destaca-se também que o MQTT utiliza mensagens menores representando menos *overhead* na comunicação. O trabalho destaca ainda que o MQTT apresenta também maior uso em sistemas M2M e IoT em relação ao HTTP. Por sua vez, o HTTP apresenta maior segurança, padronização/interoperabilidade e serviços (*provisioning*, como multiplexação, priorização e compressão de cabeçalho) em relação ao MQTT.

- Sueda, Sato e Hasuike (2019) apresentam uma análise comparativa do MQTT, *Websocket* e HTTP em relação ao *overhead* de comunicação dos protocolos. Os resultados

mostram que o *overhead* apresentado pelo MQTT tende a ser menor, desde que o tamanho dos tópicos no MQTT seja cuidadosamente determinado.

Considerando-se os trabalhos analisados e o Quadro 2 que resume as características do MQTT e do HTTPS, identifica-se que o MQTT se mostra mais adequado ao contexto de IoT avaliado. Em suma, além do MQTT ser um protocolo desenvolvido para dispositivos e redes no qual os recursos computacionais são escassos, ele também permite uma arquitetura desacoplante *Publish/Subscribe* escalável para comunicação entre vários clientes (1 para N ou N para N) de forma assíncrona (AL-MASRI *et al.*, 2020). De antemão, essas características são pertinentes e sinérgicas também ao Paradigmas Orientado a Notificações (PON), conforme discutido ao longo deste presente trabalho.

Além dos protocolos de comunicação em IoT, outro ferramental necessário para o desenvolvimento dos sistemas são os paradigmas de programação, detalhados na Seção 2.2, servindo de prólogo ao PON justamente, o qual é apresentado na seção 2.3.

2.2 Paradigmas de Programação

Conforme descreve Peter Van Roy (ROY, 2012), um paradigma de programação é uma abordagem para programar um computador com base em um conjunto coerente de princípios. Neste âmbito, cada paradigma suporta um conjunto de conceitos e isso o faz mais adequado para a solução de determinados tipos de problema. No seu livro, Peter Van Roy (ROY, 2012) apresenta uma taxonomia para classificação dos paradigmas de programação conforme o conjunto de conceitos que permitem distinguir e organizar os paradigmas uns em relação aos outros.

Os conceitos são, basicamente, elementos primitivos que compõem um determinado paradigma. Como exemplo de conceitos, pode-se citar os quatro principais (ROY, 2012) (XAVIER, 2014):

- registro (*record*) sendo uma estrutura de dados, ou seja, uma referência para um grupo de dados com um índice para acesso.
- encapsulamento de escopo léxico (*lexically scoped closures*) que representa os 'módulos' ou 'entidades' com suas referências externas como, por exemplo, procedimentos e funções.
- independência/concorrência (*independency/concurrency*) que ocorre quando a execução das instruções não possui dependência.
- estados nomeados (*named state*) que representa a capacidade de armazenar informação em um componente computacional com nome, ou seja, uma variável, de nome único, que registra e mantém diferentes dados durante um período de execução. Dessa

> forma, introduz-se uma abstração de tempo que afeta o estado das entidades. Diferentemente de funções que uma vez chamadas com os mesmos argumentos retornarão sempre o mesmo resultado, entidades com estado nomeado possuem memória que faz com que a resposta seja diferente dependendo do momento de sua execução, isto é, baseado nas modificações ocorridas por eventos prévios.

Peter Van Roy (ROY, 2012) destaca ainda que os paradigmas de programação são independentes das linguagens de programação que os implementam. Enquanto existe um grande número de linguagens de programação, existem relativamente poucos paradigmas de programação. Ademais, os paradigmas existentes possuem muitas semelhanças e conceitos em comum. Por fim, uma linguagem pode também implementar um ou mais paradigmas de programação.

De forma geral, pode-se resumir os paradigmas de programação considerando-se dois grandes grupos, conforme apresentado na Figura 11 (RONSZCKA, 2012): o Paradigma Imperativo (PI) e o Paradigma Declarativo (PD) (GABBRIELLI; MARTINI, 2010). De forma sucinta, o PI se caracteriza pela flexibilidade de programação e pela forma explicitamente sequencial pela qual as instruções são executadas. O PD apresenta um modelo menos flexível, porém mais simplificado de programação no qual o programador se concentra na organização do conhecimento sobre a resolução do problema computacional em si e em alto nível ao invés da implementação técnica propriamente dita. Em suma, enquanto o PI impõe pesquisas orientadas a laços de repetições sobre elementos passivos, relacionando dados a expressões causais, o PD possui soluções declarativas que trabalham por recursões e/ou mecanismos de inferência. É importante destacar que mesmo esses dois grandes grupos (PI e PD) possuem intersecções ou similaridades em alguns aspectos (BANASZEWSKI, 2009) (GABBRIELLI; MARTINI, 2010) (RONSZCKA, 2019) (NEVES, 2021).

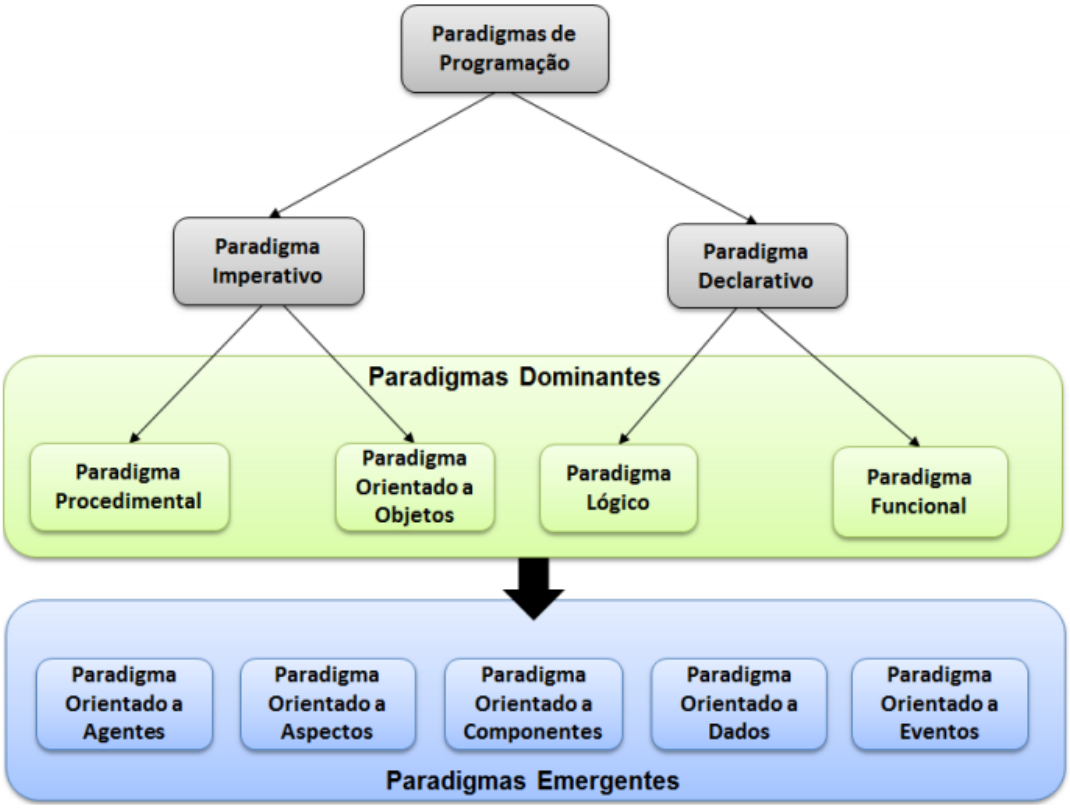


Figura 11 – Classificação simplificada dos paradigmas de programação.
Fonte: Ronszcka (2012).

Cada grupo de paradigmas apresentado na Figura 11 engloba um subconjunto de paradigmas com características próximas. Alguns desses paradigmas possuem atualmente maior relevância nas linguagens de programação vigentes e, por isso, são chamados de Paradigmas dominantes (SCOTT, 2000) (WATT, 2004) (BROOKSHEAR, 2012). Os paradigmas de programação emergentes, por sua vez, são estabelecidos geralmente na forma de uma materialização, como um arquétipo ou *framework* sobre uma linguagem de programação de um paradigma dominante, antes de terem suas próprias linguagens de programação (BANASZEWSKI, 2009) (RONSZCKA, 2019).

2.2.1 Paradigmas Imperativos

Dentre os Paradigmas Imperativos, definem-se como dominantes o Paradigma Procedimental (PP) e o Paradigma Orientado a Objetos (POO) (BROOKSHEAR, 2012) (GABBRIELLI; MARTINI, 2010) (BANASZEWSKI, 2009) (SIMÃO; STADZISZ, 2008). Em linhas gerais, o PP se baseia na organização sequencial de variáveis e comandos, na qual as variáveis representam os estados das entidades e os comandos executam ações que alteram estes estados.

Como exemplo do PP, pode-se considerar o Código 2.2.1, o qual apresenta uma implementação simples, feita em C, de uma aplicação para verificação de sensores (NEVES, 2021). Essa aplicação é responsável pela verificação do estado de um sensor e pelo acionamento de

um alarme em caso de presença detectada. Uma vez ativado o sensor (eventualmente por uma ação externa do ambiente), a aplicação executa o processamento correspondente, acionando um alarme respectivo ao setor. Neste exemplo, o Sensor é implementando como uma estrutura (*struct*) a qual possui duas variáveis membros. Ademais, são declaradas funções auxiliares para alterar os valores destas variáveis da estrutura utilizando ponteiros. O laço de repetição ou *loop* principal, dentro da seção *main*, executa continuamente a verificação do estado do sensor (*Sensor.isActive*) e estado da leitura do sensor (*sensor.isPresenceDetected*).

Pode-ser observar, na linha 20 do Código 2.2.1, que a avaliação do estado do sensor (*sensorA.isActive == true*) se repete a cada ciclo iterativo, mesmo que não haja alterações em relação às avaliações anteriores. Isto caracteriza uma redundância temporal pelo desperdício de avaliação ao longo do tempo. Observa-se também, nas linhas 26 e n-5, um exemplo de redundância estrutural, onde uma mesma avaliação lógica (*sensorB.isPresenceDetected == true*) é estruturalmente repetida no código e, portanto, avaliada duas vezes a cada ciclo. Ainda, estas redundâncias podem estar dispersas no código, com n linhas de distância, criando dificuldades mesmo para desenvolvedores experientes tratá-las ou mitigá-las. Isto posto, cumulativamente, tais redundâncias podem representar desperdícios de processamento significativas (OSHIRO, 2021) (PAN; DESOUZA; KAK, 1998) (SIMÃO; TACLA; STADZISZ, 2009).

Código 2.1 Exemplo de uma aplicação de sensor utilizando o Paradigma Procedimental.

```
1 struct Sensor {
2   bool isActive;
3   bool isPresenceDetected;
4
5   Sensor() {
6     isActive = false;
7     isPresenceDetected = false;
8   }
9  };
10
11 void activate_sensor(Sensor* p_sensor) {
12     p_sensor->isActive = true;
13 }
14
15 int main() {
16 Sensor sensorA;
17 Sensor sensorB;
18
19   while (true) {
20     if ( sensorA.isActive == true ) &&
21        ( sensorA.isPresenceDetected == true ){
22        //Dispara alarme relacionado ao sensor A
23     }
24
25     if ( sensorB.isActive == true ) &&
26        ( sensorB.isPresenceDetected == true ){
27        //Dispara alarme relacionado ao sensor B
28     }
...  // ...
n-9
n-8    if ( sensorA.isActive == true ) &&
n-7       ( sensorA.isPresenceDetected == true ) &&
n-6       ( sensorB.isActive == true ) &&
n-5       ( sensorB.isPresenceDetected == true ){
n-4       //Dispara alarme relacionado ao ambiente inteiro
n-3     }
n-2   }
n-1   return 0;
n    }
```

Fonte: Adaptado de Oshiro (2021) e Neves (2021)

Por sua vez, o POO apresenta a organização dos programas compostos por entidades modulares denominadas objetos, que podem ser entendidas como abstrações de uma entidade real ou imaginária, contendo as características pertinentes para sua implementação computacional. Esses objetos agrupam atributos (similar às variáveis do PP) e métodos (similar às funções

do PP), além dos possíveis relacionamentos com outros objetos. Em resumo, o PP e o POO se diferenciam pela forma como os elementos e as instruções são organizados, sendo o POO considerado mais rico e estruturado em termos de abstração e expressão do código do que o PP (WATT, 2004) (RONSZCKA, 2019). Como exemplo de POO, pode-se considerar o Código 2.2.2 que implementa, em C++ e utilizando objetos, a mesma aplicação de sensores apresentada em PP no Código 2.2.1 (NEVES, 2021). Neste código, é possível observar o agrupamento dos atributos e métodos de um sensor em uma única classe (classe *Sensor*). Observa-se também, de forma similar ao PP, a redundância da avaliação dos atributos na linha 21 mesmo quando estes não tenham sido alterados desde a última avaliação.

Código 2.2 Exemplo de uma aplicação de sensor utilizando o Paradigma Orientado a Objetos.

```
1   class Sensor {
2       public:
3           bool isRead{false};
4           bool isActivated{false};
5
6           void Activate() {
7               isActivated = true;
8               isRead = false;
9           }
10
11          void Process() {
12              isActivated = false;
13              isRead = true;
14          }
15  };
16
17  int main() {
18      Sensor sensor;
19
20      while (true) {
21          if (sensor.isActivated && !sensor.isRead) {
22              sensor.Process();
23          }
24      }
25      return 0;
26  }
27
```

Fonte: Neves (2021)

Embora sejam paradigmas dominantes, o PP e o POO apresentam ineficiências intrínsecas aos paradigmas imperativos. Essas ineficiências são principalmente derivadas dos modelos de construção que, de maneira geral, são sequenciais e baseado em avaliações sobre elementos passivos. Essa característica favorece o surgimento de redundâncias temporais, isto

é, pode ocorrer uma repetição de avaliações desnecessárias de uma variável que pode não ter seu estado ou valor alterado desde a última (re)avaliação. Conforme os exemplos apresentados, observa-se que em ambos os casos (Código 2.2.1 e 2.2.2), estão presentes principalmente a redundância temporal, visto que as variáveis/atributos relativas aos estados do sensor são continuamente reavaliadas mesmo sem ter ocorrido alteração em seu estado.

Destaca-se também que os paradigmas imperativos dificultam o reaproveitamento de avaliações lógicas em diferentes contextos, sendo que uma avaliação utilizada por diferentes contextos tem de ser repetida em diferentes partes do código, gerando redundância estrutural. As redundâncias degradam o desempenho das aplicações e também causam acoplamentos que dificultam a distribuição do código em módulos, núcleos de processamento ou computadores distintos. A distribuição de código dentro dos paradigmas imperativos é normalmente uma atividade complexa e que exige uma análise específica de cada código (RONSZCKA, 2012) (RONSZCKA, 2019) (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009).

2.2.2 Paradigmas Declarativos

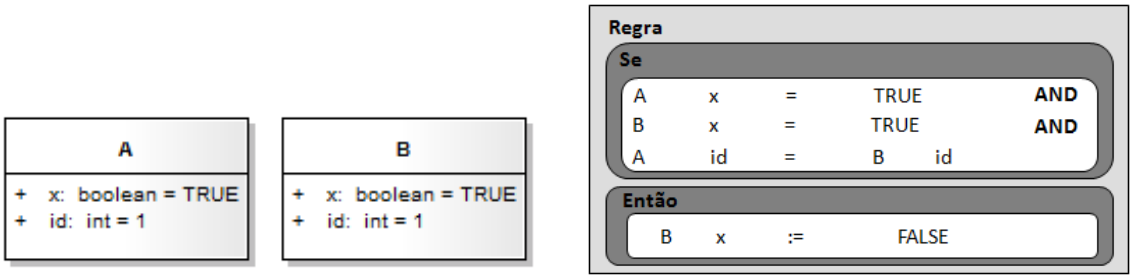
No grupo dos Paradigmas Declarativos, por sua vez, definem-se como dominantes o Paradigma Funcional (PF) e o Paradigma Lógico (PL) (SCOTT, 2000) (WATT, 2004) (BANASZEWSKI, 2009). De forma sucinta, o PF se baseia no conceito de funções computacionais calculáveis por meio de cálculo *lambda*. Os programas em PF são construídos a partir da manipulação de dados por meio de funções, em um modelo no qual as funções podem invocar outras funções ou até mesmo serem passadas como parâmetros. Além disso, as funções do PF são ditas puras, pois não possuem estado compartilhado interno sendo que seu resultado depende apenas de suas entradas (SCOTT, 2000) (NEVES, 2021). Como exemplo, considera-se o Código 2.2.3 de autoria de Gustavo Brunholi Chierici, o qual implementa, por meio da linguagem Clojure, a mesma aplicação de processamento de sensores apresentado para os Paradigmas Imperativos, mas agora utilizando o Paradigma Funcional (NEVES, 2021).

Código 2.3 Exemplo de uma aplicação de sensor utilizando o Paradigma Funcional.

```
1  ;; Autoria: Gustavo Brunholi Chierici - Data 22/06/2021
2  Grupo de Pesquisa: PON - UTFPR
3
3  (defrecord FPSensor [is-read is-activated])
4  ;; Record ("struct") do sensor
5
6  (defn create-sensor [] (FPSensor. false false))
7  ;; Funcao que retorna um sensor
8
9  (defn activate-sensor ;; Funcao que "ativa" o sensor
10  [sensor]
11 (assoc sensor :is-activated true :is-read false))
12
13 (defn deactivate-sensor ;; Funcao que "desativa" o sensor
14 [sensor]
15 (assoc sensor :is-activated false))
16
17 (defn read-sensor ;; Funcao que verifica o estado do sensor
18 [sensor]
19 (assoc sensor :is-read true))
20
21 (defn check-sensor ;; Funcao que verifica se o sensor
22 ;; esta ativado e o verifica em caso positivo
23 [sensor]
24 (if (and (:is-activated sensor) (not (:is-read sensor)))
25 (deactivate-sensor (read-sensor sensor)) sensor))
```

Fonte: Neves (2021)

Por sua vez, o PL, diferentemente do PF, é construído a partir da expressão de dados (ou elementos factuais) e regras, conforme apresentado na Figura 12. A Figura 12a apresenta duas entidades factuais A e B enquanto que a Figura 12b apresenta um exemplo de regra.



(a) Entidades. **(b) Regras.**

Figura 12 – Conhecimento representado em regras no PL.

Fonte: Banaszewski (2009)

Conforme apresentado na Figura 13, a partir dos dados e das regras é possível a realização de consultas nas quais, por meio de um processo de inferência lógica, serão produzidos os resultados (SCOTT, 2000) (BANASZEWSKI, 2009).

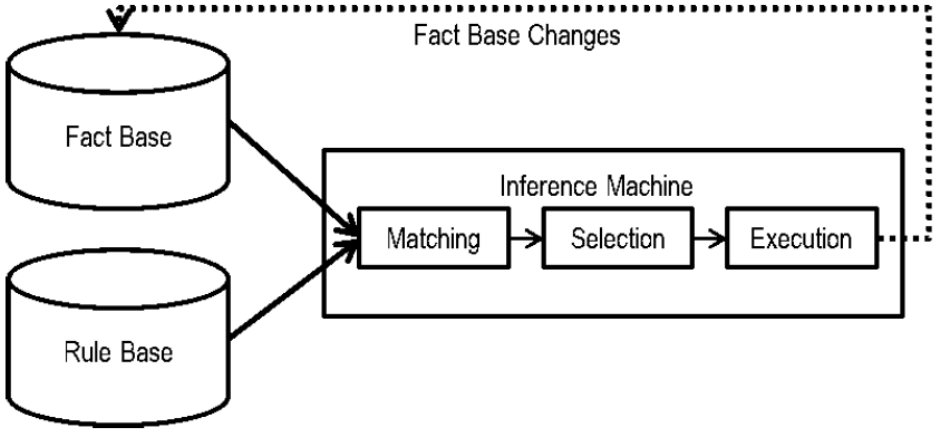


Figura 13 – Arquitetura interna de um Sistema Baseado em Regras.
Fonte: Banaszewski (2009)

Como exemplo do PL, observa-se o Código 2.2.4 o qual apresenta a implementação da aplicação de sensores desenvolvido segundo o PL, por meio da linguagem RuleWorks. Observa-se que as expressões utilizadas são muito similares à descrição em si das regras (NEVES, 2021).

Código 2.4 Exemplo de uma aplicação de sensor utilizando o Paradigma Lógico.

```
1  (object-class sensor
2   ^is-read
3   ^is-activated
4  )
5
6  (rule process-sensor:sensor
7    (sensor ^$ID <the-sensor> ^is-read <false>
8        ^is_activated <true>)
9    -->
10   (bind <the-sensor ^is-read> (false))
11   (bind <the-sensor ^is-activated> (false))
12  )
```

Fonte: Neves (2021)

Observa-se que os Paradigmas Declarativos, no geral, apresentam um maior nível de abstração (quando comparados com o PI) facilitando a compreensão dos programas. Porém, geralmente as soluções geradas pelos paradigmas declarativos fazem uso de estrutura de dados que são computacionalmente custosas causando sobrecargas de processamento. Além disso, seu mecanismo interno também se baseia na busca sobre elementos passivos, favore-

cendo o surgimento de redundâncias temporais (RONSZCKA, 2019) (SIMÃO; STADZISZ, 2008) (GABBRIELLI; MARTINI, 2010) (RONSZCKA, 2014).

2.2.3 Paradigmas Emergentes

Outros paradigmas, chamados de emergentes, propõem melhorias, adições e/ou alterações nos paradigmas dominantes. Como exemplo, pode-se citar o Paradigma Orientado a Eventos (POE), o Paradigma Orientado a Agentes (POAg) e o Paradigma Orientado a Atores (POAt).

2.2.3.1 Paradigma Orientado a Eventos

O Paradigma Orientado a Eventos (POE) ou também chamada de Programação Dirigida a Eventos (do inglês *Event-Driven Programming*) é primordialmente o modelo de construção de *software* que trata eventos. Nesse contexto, um evento é definido como um acontecimento de interesse, ou seja, a mudança de estado de um determinado elemento (COULOURIS; DOLLIMORE; KINDBERG, 2011). Conforme apresentado na Figura 14, em contraste com o modelo de programação imperativa, os programas orientados a eventos não controlam a sequência na qual os eventos ocorrem, sendo escritos para reagir a eles, quando ocorrerem.

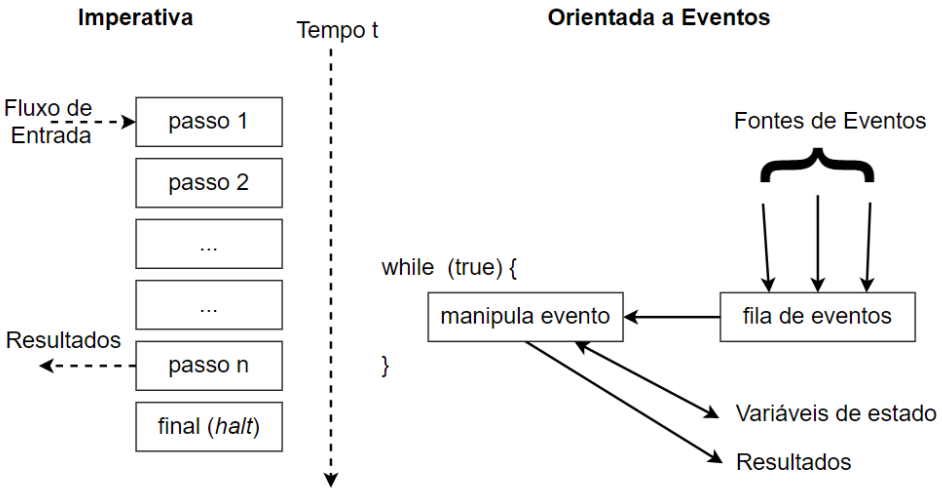
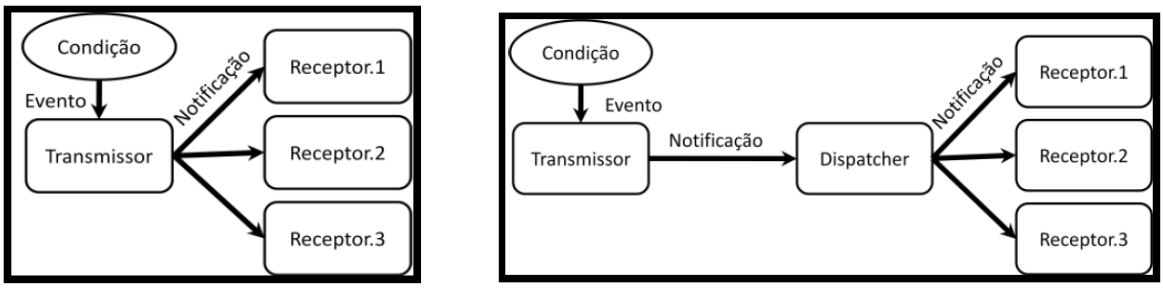


Figura 14 – Contraste entre programação imperativa e orientada a eventos.
Fonte: Adaptado de Tucker e Noonan (2007)

Como exemplo, um evento pode ser um botão pressionado, uma interrupção de hardware ou uma mensagem recebida, oriundo de um sistema ou componente. Esse evento pode instigar uma determinada ação (*i.e.*, função, método, processo ou comportamento) e essa ação comumente está contida em um tipo determinado de módulo, como um bloco, objeto, observador, consumidor ou até mesmo agente (XAVIER, 2014).

Conforme apresentado na Figura 15, a relação entre os objetos transmissores e receptores pode ocorrer de forma direta ou indireta. Na Figura 15a, um objeto transmissor detecta a ocorrência de um evento e notifica este evento diretamente a um ou mais objetos receptores interessados, ou seja, a comunicação ocorre segundo uma relação bilateral entre um remetente e um destinatário, com os remetentes direcionando explicitamente as mensagens/invocações para os destinatários associados. Geralmente, os destinatários conhecem a identidade dos remetentes e, na maioria dos casos, as duas partes devem existir ao mesmo tempo (COULOURIS; DOLLIMORE; KINDBERG, 2011). Em contraste, a Figura 15b apresenta a comunicação indireta na qual um objeto transmissor detecta a ocorrência de um evento e notifica este evento indiretamente a um ou vários objetos receptores por um intermediário, conhecido como *Dispatcher*. Dessa forma, possibilita-se um alto grau de desacoplamento entre remetentes e destinatários. Como exemplo de notificação indireta, pode-se citar os sistemas *Publish/Subscribe*, previamente apresentado na Seção 2.1.1.1. Destaca-se que os sistemas *Publish/Subscribe* são os mais amplamente utilizados dentre as técnicas de comunicação indireta (COULOURIS; DOLLIMORE; KINDBERG, 2011).



(a) Detecção direta. **(b) Detecção indireta.**

Figura 15 – Processo de detecção de eventos.

Fonte: Adaptado de Faison (2011)

Em termos práticos, não existe uma linguagem de programação que implemente especificamente o POE, sendo ele geralmente incorporado por *frameworks* e afins nas linguagens dos paradigmas vigentes, normalmente o POO (BANASZEWSKI, 2009) (NEVES, 2021).

2.2.3.2 Paradigma Orientado a Agentes/Atores

No paradigma emergente POAg, a construção dos programas é realizada por meio da abstração de agentes. Neste contexto, os agentes são geralmente definidos como entidades informáticas capazes de perceber (ainda que parcialmente) o ambiente em que estão situados e de executar ações de maneira flexível e autônoma para atingir objetivos definidos no momento de sua concepção (JENNINGS, 1999) (NEVES, 2021). Dessa forma, os agentes são capazes de apresentar comportamento proativo, guiado por seus objetivos, e flexibilidade para perceber as mudanças do ambiente e se adaptar a elas. Usualmente, os agentes são implementados sobre os conceitos do POO. Porém, um agente difere do conceito de objeto, pois conceitualmente

um agente apresenta propriedades mais complexas do que um objeto (BANASZEWSKI, 2009). Uma das diferenças fundamentais entre o POAg e o POO é que um agente atua de forma proativa, tendo objetivos individuais ou coletivos. Diferentemente de um agente, um objeto atua passivamente até que um de seus métodos seja invocado. Em outras palavras, um objeto aguarda que outro objeto lhe indique o que fazer enquanto o agente decide por si só quando e o que deve ser feito (BANASZEWSKI, 2009). Ainda, em POAg as relações entre os entes acontecem na dinâmica do sistema, de maneira fluida e não tudo pré-definido como normalmente ocorre em POO.

O POAt, por sua vez, é consideravelmente similar ao POAg, sendo o modelo de agentes uma extensão do modelo de atores. A diferença entre agentes e atores é que os agentes são tipicamente mais complexos e capazes de tomada de decisão lógica (AGHA, 1985) (NEVES, 2021) (NEGRINI, 2019). De maneira resumida, um ator é uma entidade computacional que, em resposta a uma mensagem que recebe, pode (de maneira concorrente aos demais atores) enviar mensagens para outros atores, criar outros atores ou designar o comportamento a ser usado para a próxima mensagem que receber (HEWITT; BISHOP; STEIGER, 1973) (SHALI, 2010) (NEGRINI, 2019). Conforme apresentado na Figura 16, um ator é normalmente executado em uma *Thread* exclusiva e normalmente é composto por (NEGRINI, 2019) (SHALI, 2010): (i) uma fila de mensagens na qual estão as mensagens a serem processadas pelo ator por ordem de chegada. Geralmente a comunicação entre os atores ocorre por meio de troca de mensagens assíncronas; (ii) os estados internos, nos quais são armazenadas as informações necessárias para controle do ator. Destaca-se que os estados são exclusivos, isto é, não são compartilhados por mais de um ator; (iii) os métodos ou a parte do código fonte, na qual há a lógica para processamento das mensagens.

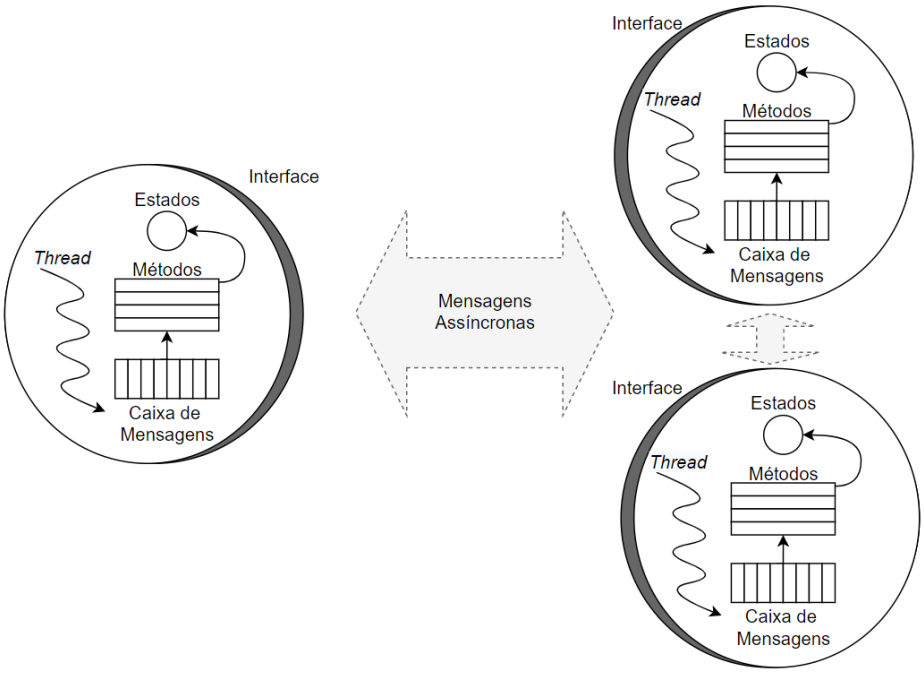


Figura 16 – Exemplo de atores e suas interações por meio de mensagens assíncronas.
Fonte: Adaptado de Shali (2010)

Ambos os modelos do POAt e POAg trazem a vantagem de facilitar o paralelismo devido ao alto desacoplamento entre as entidades, enquanto uma das principais desvantagens é o alto tempo de execução da comunicação entre os atores/agentes (NEVES, 2021). Observa-se também que, por serem construídos normalmente sobre os paradigmas dominantes, os paradigmas emergentes herdam os problemas destes paradigmas que foram previamente apresentados na Seção 2.2.1.

Com o objetivo de eliminar (ou ao menos reduzir) o acoplamento entre os elementos e as redundâncias apresentadas pelos paradigmas vigentes (PI e PD), bem como paradigmas emergentes deles derivados, apresenta-se o Paradigma Orientado a Notificações (PON) (SIMÃO, 2005). Além de tratar as deficiências apresentadas pelos paradigmas usuais, o PON unifica certas características desejáveis dos demais paradigmas como a flexibilidade de expressão e um nível apropriado de abstração de módulos (geralmente presentes no PI), além da representação do conhecimento em regras (geralmente presentes no PD) (RONSZCKA, 2019) (BANASZEWSKI, 2009).

2.3 Paradigma Orientado a Notificações

O Paradigma Orientado a Notificações (PON) define e apresenta novos conceitos (inclusive) para a concepção, programação e execução de *softwares* em sistemas computacionais. Esses conceitos visam melhorar o desempenho dos *softwares* e facilitar a sua concepção e programação. Esta seção apresenta, na Subseção 2.3.1, os detalhes dos elementos e do funcionamento do PON. Na sequência, na Subseção 2.3.2, são apresentadas considerações acerca

do PON. Por fim, na Subseção 2.3.3 são apresentadas as materializações do PON disponíveis em *software*.

2.3.1 Descrição detalhada do PON

No PON, as interações entre os elementos constituintes acontecem de forma pontual por meio de notificações somente quando um dos elementos de interesse apresentar alterações de valores ou de estado lógico. Na verdade, o PON propõe a divisão de uma aplicação computacional em dois grandes grupos funcionais: um facto-execucional e um lógico-causal. O grupo facto-executional é representado pelas *Fact Base Elements* (ou *FBEs*) enquanto o grupo lógico-causal é representado pelas *Rules*. Os *FBEs* e as *Rules* são expressas em termos de entidades menores, inteligentes, complementares e cooperativas. Esses dois grandes grupos funcionais interagem exclusivamente por meio de notificações (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009) (OSHIRO, 2021). Como exemplo, a Figura 17 apresenta as entidades constituintes do PON e a relação entre os dois conjuntos de entidades e seus construtos no contexto de um sistema de monitoramento de sensores e alarmes. Enquanto entidades, os *FBEs* e as *Rules* são compostos por entidades menores que possibilitam a interação por meio de notificações entre *FBEs* e *Rules* e vice-versa.

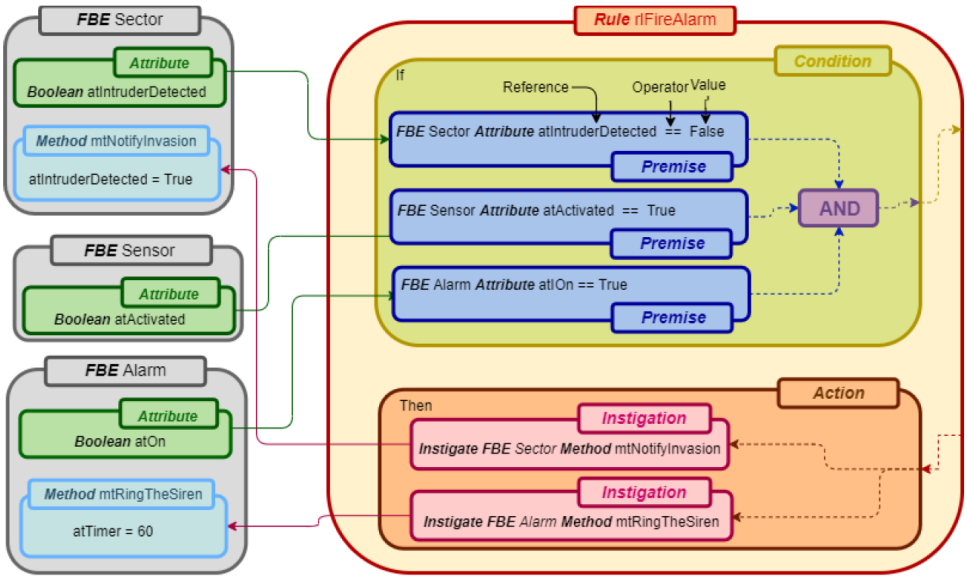


Figura 17 – Exemplo de interação entre as entidades do PON.

Fonte: Oshiro (2021)

O *FBE* é uma entidade responsável por descrever estados (*Attributes*) e serviços (*Methods*) de entidades reais ou imaginárias, as representando em um sistema computacional. Pode-se descrever um *FBE* de forma análoga, mas diferente, aos objetos do Paradigma Orientado a Objetos (POO). O *FBE* agrega os *Attributes* e os *Methods* que operam sobre os *Attributes*, com a particularidade de que cada *Attribute* notifica pontualmente entidades interes-

sadas no seu estado, enquanto cada *Method* é pontualmente instigável por entidades interessadas em seu serviço ou funcionalidade.

As *Rules*, por sua vez, apresentam o conhecimento lógico-causal e são descritas, normalmente, por meio de uma regra 'se-então', o que é uma maneira natural de expressão deste tipo de conhecimento (BANASZEWSKI, 2009). O conhecimento lógico da *Rule* é descrito pelas *Conditions* e pelas *Actions*. A Figura 18 apresenta um exemplo de uma *Rule* para acionamento de um alarme conforme as seguintes especificações: se o sensor de temperatura e de presença forem acionados (isso é, se o estado de ambos for verdadeiro) e se o alarme estiver ligado, então dispara o alarme por 60 segundos.

Nas *Rules*, as *Conditions* são responsáveis pelas decisões (*If statements*) e, de forma complementar, as *Actions* são responsáveis pelas ações (*Then statements*). O conhecimento lógico das *Conditions* é expresso por meio de operações lógicas (e.g., conjunção ou disjunção) sobre as *Premises* que realizam as avaliações baseadas em um comparador relacional (e.g., igual, diferente, maior e menor) sobre os *Attributes* de um *FBE*.

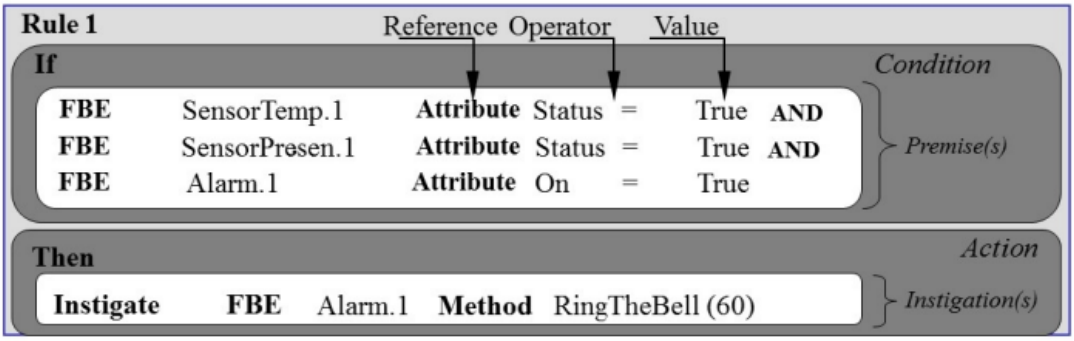


Figura 18 – Exemplo de uma *Rule*.

Fonte: Ronszcka *et al.* (2017).

No exemplo proposto, as *Premises* avaliam os *Attributes SensorTemp.1, SensorPresen.1 e Alarm.1*. As *Premises*, nesse caso, são avaliadas em conjunção como uma *Condition* e uma vez que todas as *Premises* assumam estado verdadeiro, a respectiva *Action* é acionada. A *Action*, por sua vez, é relacionada a uma ou mais *Instigations* que são ativadas quando a *Action* é acionada pela *Rule*. Quando ativadas, as *Instigations* são responsáveis pela execução dos *Methods* que operam, normalmente, sobre os *Attributes* dos *FBEs*. No exemplo proposto, a *Action* ativa a *Instigation* que, por sua vez, instiga a execução do *Method RingTheBell*(60) responsável pelo disparo do alarme.

A Figura 19 apresenta o esquema de colaboração entre as entidades do PON. Conforme previamente comentado, os *Attributes* representam os estados, propriedades ou valores de um determinado *FBE*. Neste esquema, tão logo ocorra a alteração (de valor ou de estado lógico) de um *Attribute*, uma notificação é enviada pelo próprio *Attribute* precisamente apenas para as *Premises* pertinentes. Uma vez notificada, a *Premise* é responsável pela avaliação de um ou dois *Attributes* segundo uma pré-determinada operação relacional (igual, diferente, maior ou igual, maior, menor ou igual, menor). Quando uma *Premise* é aprovada, isto é, quando a comparação

lógica da determinada *Premise* resulta em verdadeiro, é gerada uma notificação precisamente apenas para as *Conditions* pertinentes. Cada *Condition* agrupa uma ou mais *Premises* e é notificada por cada *Premise* quando o estado delas é alterado. Quando todas as *Premises* de uma determinada *Condition* forem aprovadas, isto é, tiverem seus estados booleanos verdadeiros, diz-se que a *Condition* foi aprovada. A aprovação de uma *Condition* gera uma notificação para a *Rule* pertinente (SIMÃO; STADZISZ, 2008) (SIMÃO; TACLA; STADZISZ, 2009) (NEVES, 2021).

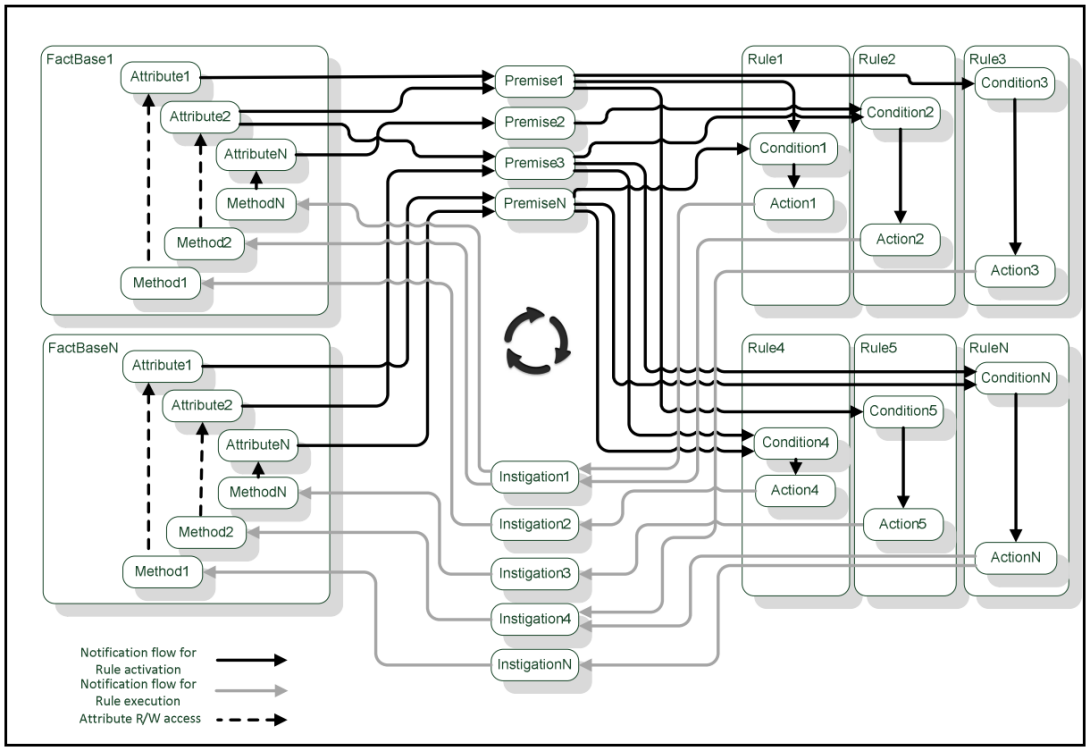


Figura 19 – Esquema de colaboração entre as entidades do PON.
Fonte: Linhares (2015).

A *Rule* relaciona uma *Condition* e uma *Action*. Uma vez que a *Condition* seja aprovada, a respectiva *Action* é notificada. Em tempo, a teoria do PON também admite *SubConditions* ou similares na *Condition*. Estas se relacionam logicamente agrupando uma ou mais *Premises*. Cada *Action* agrega uma ou mais *Instigations* as quais são executadas quando a *Action* for executada. Cada *Instigation* referencia um ou mais *Methods* os quais são executados quando notificados pela *Instigation*. Os *Methods* são análogos às funções membro ou métodos em POO, podendo eventualmente alterar o estado de um ou mais *Attributes* o que reiniciaria o ciclo de notificações.

A Figura 20 apresenta as relações entre os elementos do modelo de Notificações do PON na forma de um diagrama de classes UML (*Unified Modeling Language*). Mais precisamente, à luz do diagrama de classes, os elementos constituintes do PON são resumidos conforme segue (NEVES, 2021):

- *Fact Base Element* (*FBE*): classe que agrega *Attributes* e *Methods*, podendo ser considerada similar aos objetos do POO em termos simplistas. A diferença reside no fato de os *Attributes* serem notificadores precisos e *Methods* serem precisamente instigáveis.

- *Attributes*: classe que representa uma propriedade de um *FBE*, sendo responsável por armazenar um valor discreto-factual que representa estados. Cada *Attribute* (*i.e.*, cada instância de *Attribute*) difere de uma variável do PP ou atributo do POO tradicional no sentido de que possui a capacidade de notificar *Premises* relacionadas quando seu estado é alterado.

- *Premise*: classe que define as entidades responsáveis por realizar a avaliação lógica de estados de um ou dois *Attributes* por meio de um operador de comparação relacional (*e.g.*, igual, diferente, maior e menor) e notificar *Conditions* relacionadas quando muda de estado lógico.

- *Condition*: classe que define as entidades responsáveis por avaliar as *Premises* por meio de um operador lógico (*e.g.*, conjunção ou disjunção) e notificar as *Rules* relacionadas quando muda de estado, como de aprovada para reprovada e vice-versa.

- *Rule*: classe que define as entidades que se relacionam a uma *Condition*. Tipicamente, cada *Rule* executa sua *Action* quando tem sua *Condition* aprovada. Em tempo, pode haver mecanismo de resoluções de conflito para *Rules*, caso essas sejam elaboradas sem eliminar os conflitos.

- *Action*: cada *Rule* é relacionada a uma *Action* que se relaciona com uma ou mais *Instigations*. Quando notificada pela *Rule*, a *Action* executa todas as suas *Instigations*.

- *Instigation*: classe que define as entidades responsáveis por instigar os *Methods* relacionados quando é ativada pela *Action*.

- *Method*: classe que pode alterar o estado de um ou mais *Attributes* de um *FBE*. Os *Methods* são definidos, de forma análoga às funções membro (ou métodos) do POO, entretanto, são executados apenas quando notificado/instigado por alguma *Instigation* ou chamado por algum outro *Method*.

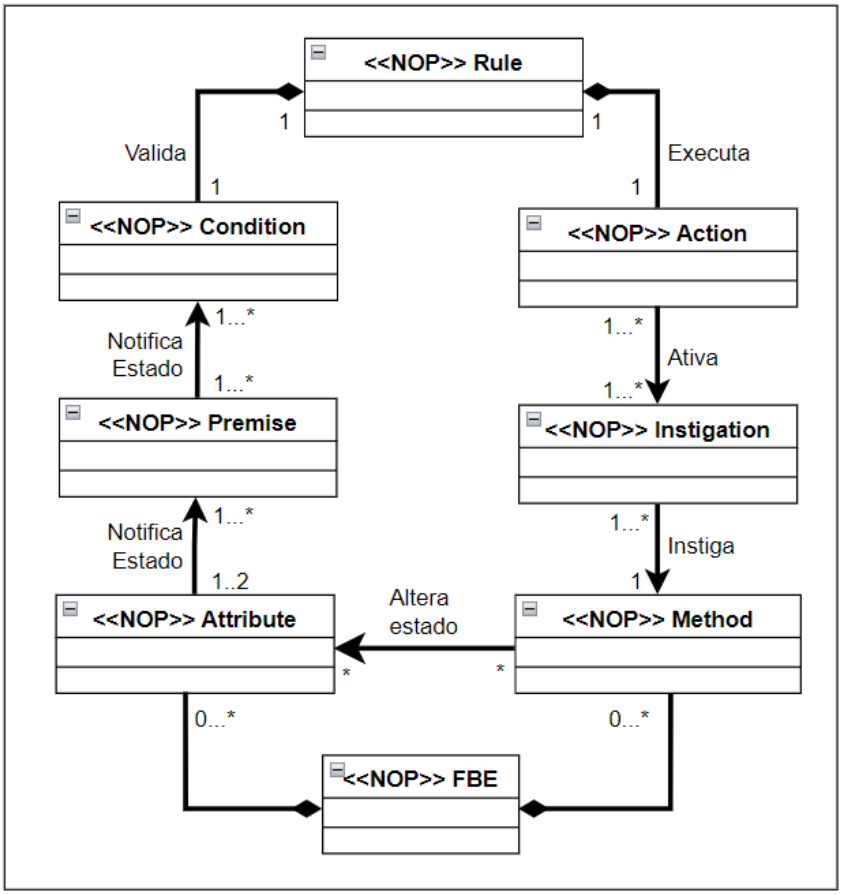


Figura 20 – Diagrama de Classes das entidades do PON.

Fonte: Adaptado de Ronszcka (2019)

2.3.2 Considerações sobre o PON

Em resumo, o PON proporciona uma nova visão de desenvolver, estruturar e executar *software* por meio de entidades facto-execucionais e lógico-causais que colaboram por notificações precisas e pontuais. Com isso, o PON apresenta três propriedades elementares que o diferencia e o destaca dos demais paradigmas (RONSZCKA, 2019):

- A facilidade de desenvolvimento de *software* em alto nível garantida pela expressividade declarativa e separação das regras lógico-causais e do código facto-execucional.
- A diminuição ou mesmo ausência de redundâncias estruturais e temporais devido ao fato de o PON apresentar entidades reativas (isto é, que reagem a mudanças de estados) e com notificações pontuais, sendo que assim são avaliadas apenas expressões lógico-causais afetadas pela mudança de estado.
- Desacoplamento estrutural ou também chamado de acoplamento mínimo entre os elementos. Essa propriedade é garantida pelo modelo de notificações pontuais entre os elementos do PON permitindo, dentre outros, o paralelismo e/ou distribuição da aplicação.

De modo geral, o PON resolve problemas existentes nos outros paradigmas de programação atuais (nomeadamente o PI e o PD com seus sub paradigmas dominantes ou emergentes) como, por exemplo, as redundâncias temporais e estruturais na análise lógico-causal e o acoplamento excessivo entre entidades computacionais que dificulta o reaproveitamento e paralelização/distribuição (SIMÃO; STADZISZ, 2008) (BANASZEWSKI, 2009) (LINHARES, 2015) (KERSCHBAUMER *et al.*, 2015) (RONSZCKA, 2019).

2.3.3 Materializações do PON

Com o objetivo de facilitar o desenvolvimento de aplicações segundo o PON surgiram as suas materializações, sendo que as mais estáveis são os arquétipos ou *frameworks* que constituem o estado da técnica em PON. Os *frameworks* atuam como uma camada de abstração para o programador disponibilizando uma interface de declaração dos elementos PON (*FBE*, *Attribute*, *Method*, *Rule*, *Premise*, *Condition*, *Instigation*, *Action*) de forma que o ciclo de notificações aconteça conforme o esperado sem a necessidade de implementação de código específico uma vez que já são cobertos no *framework*.

Desde as primeiras versões até o momento, os *frameworks* PON foram sendo gradualmente melhorados de forma a incrementar a gama de funcionalidades suportadas e diminuir o processamento adicional (*overhead*) causado pelo processamento das estruturas internas dos próprios *frameworks* (*e.g.*, cada estrutura de dados existentes dentro de cada *Attribute* para poder notificar as *Premises* pertinentes). Atualmente existem versões de *frameworks* desenvolvidos em diversas linguagens de programação e com diferentes características e funcionalidades. Destaca-se principalmente os *frameworks* desenvolvidos em linguagem de programação C++, adequando a linguagem para que esta trabalhe de forma orientada a notificações.

O *framework* inicial prototipal foi proposto por Simão em 2007 usando a linguagem de programação C++ em decorrência do prévio *framework* de Controle Orientado a Notificações (SIMÃO, 2001) (SIMÃO; STADZISZ, 2008). Subsequentemente, foi proposto por Banaszewski (2009), em 2009, o *Framework* PON C++ 1.0, o qual permite a criação de aplicações em um arquétipo mais estável, apresentando o uso de empacotamentos para as classes do *framework* e da aplicação (BANASZEWSKI, 2009). Em 2013, o *Framework* PON C++ 2.0 foi proposto por Ronszcka (2012) e Valença (2012) utilizando-se inclusive de padrões de projeto e estruturas de dados e códigos mais enxutos, visando melhor facilidade de programação e melhor desempenho de execução. A partir da extensão do *Framework* PON C++ 2.0, propôs-se o assaz instável *Framework* PON C++ 3.0, o qual integrou, dentre outras modificações, funcionalidades para execução *multicore* (BELMONTE; SIMÃO; STADZISZ, 2012) (SCHÜTZ, 2019).

Subsequentemente, este arquétipo do *Framework* PON C++ 3.0 também prototipalmente integrou a comunicação em rede utilizando a biblioteca PONIP (TALAU, 2016) (MENDES *et al.*, 2019). A biblioteca PONIP, desenvolvida por Talau (2016), permite o envio e recebimento de valores de estados de *Attributes* e *Premises* em uma interface de rede de forma transparente

ao programador por meio de um protocolo de comunicação próprio do PON sobre IP. Essa biblioteca é apresentada com detalhes na Seção 2.4.3.

Recentemente, Neves (2021) propôs o *Framework* PON C++ 4.0 com o objetivo de promover avanços no PON e obter um *framework* mais genérico e fácil de utilizar que os precedentes, bem como facilitar a manutenibilidade, melhorar o desempenho e atualizar as tecnologias da linguagem C++ utilizadas. Como o *Framework* PON C++ 4.0 é o estado da técnica em PON e um dos objetos desta dissertação, ele está detalhado na Seção 2.4.8.

Além dos *frameworks* em C++, foram desenvolvidas também materializações em outras linguagens de programação. Inicialmente, surgiram os *frameworks* em C# e Java, como uma adaptação do *Framework* C++ 1.0 para essas duas linguagens de programação (HENZEN, 2015). Pertinente destacar também o trabalho realizado por Oliveira (2019), no qual o *Framework* C# desenvolvido por Henzen (2015) foi adaptado/evoluído e nomeado de *Framework* PON C# IoT. Nesta nova versão foram adicionados, entre outras melhorias, recursos para paralelismo e distribuição de entidades do PON (principalmente *Attributes*), além da capacidade de reconfiguração das *Rules* em tempo de execução. O *Framework* PON C# IoT é apresentado com mais detalhes na Seção 2.4.5.

A saber, existem ainda dois *frameworks* implementados em sinergia com modelos de atores em Elixir/Erlang (NEGRINI *et al.*, 2019) (NEGRINI, 2019) e em Akka.Net (MARTINI; SIMÃO; LINHARES, 2018). Os detalhes de implementação de cada *framework*, os benefícios e as limitações podem ser encontrados na Seção 2.4.4 e nos trabalhos de Neves (2021), Negrini *et al.* (2019), (NEGRINI, 2019) e Martini, SIMÃO e LINHARES (2018).

É pertinente ressaltar ainda que nas materializações em *software*, além das implementações em *frameworks* apresentadas nesta seção, existe também desenvolvimento por meio de linguagem de programação própria do PON, através da Tecnologia LingPON (Linguagem de Programação do PON). Com a LingPON é possível desenvolver programas diretamente em PON que é compilado para código alvo por sistema de compilação da chamada Tecnologia LingPON. A saber, mais precisamente, o programa desenvolvido por meio da Tecnologia LingPON passa por um processo de compilação capaz de gerar código alvo para múltiplas plataformas, as quais incluem alguns dos *frameworks* disponíveis (RONSZCKA, 2019) (OSHIRO, 2021).

A Tecnologia LingPON possui diferentes versões: Tecnologia LingPON Prototipal, Tecnologia LingPON 1.X (1.0 e 1.2), Tecnologia LingPON HD 1.0, Tecnologia LingPON 2.0, cada qual com seu sistema de compilação. No caso da Tecnologia LingPON 2.0, a linguagem de programação LingPON 2.0 também é chamada de NOPL (*Notification Oriented Programing Language*) (RONSZCKA, 2019). Ainda, tal tecnologia tem a mesma proposta e protótipo de uma segunda linguagem de programação, a NOPLite (CHIERICI, 2020). Esta última seguiria um padrão mais direto e pontual especialmente voltado para especialistas do PON (RONSZCKA, 2019).

Porém, destaca-se que cada linguagem de programação via Tecnologia LingPON se constitui em estado da arte por ainda ser consideravelmente prototipal. Por sua vez, o conjunto

de *frameworks* se constituem no estado da técnica por parte deles se encontrar em estado estável de desenvolvimento (NEVES, 2021).

Como síntese do conteúdo apresentado nesta presente seção, o Quadro 3 relaciona as potencialidades das propriedades elementares do PON contempladas pelas materializações. Conforme apresentado na Seção 2.3, o PON apresenta a propriedade de programação orientada a regras em alto nível, que evita redundâncias de código lógico-causal viabilizando alto desempenho de execução e apresenta a propriedade de desacoplamento implícito de construtos que viabiliza paralelismo e distribuição (RONSZCKA, 2019).

Quadro 3 – Propriedades elementares contempladas nas materializações do PON.

Framework	**Programação em alto nível**	**Paralelismo via desacoplamento**	**Distribuição via desacoplamento**	**Código não redundante**
Framework PON C++ Prototipal	parcialmente	-	-	-
Framework PON C++ 1.0	parcialmente	-	-	-
Framework PON C++ 2.0	parcialmente via *wizard*	-	-	parcialmente
Framework PON C++ 3.0	parcialmente	parcialmente	-	-
***Framework* PON C++ 4.0**	**sim**	**sim**	-	**sim**
Framework PON Java/C#	parcialmente	-	-	parcialmente
Framework PON C# IoT	parcialmente	sim	sim	-
Framework PON Erlang/Elixir	parcialmente	sim	suportado, porém não validado	-
Framework PON Akka.Net	parcialmente	sim	suportado, porém não validado	-

Fonte: Adaptado de Neves (2021)

Dos *frameworks* existentes, o *Framework* PON C++ 4.0 destaca-se por atender consideravelmente bem, para *frameworks*, três das quatro propriedades elementares, carecendo apenas da propriedade de distribuição (NEVES, 2021): (i) a programação em alto nível é alcançada por permitir uma estrutura de declaração de suas entidades de forma assaz similar à linguagem de programação do PON (NOPL); (ii) o paralelismo é realizado nas políticas de execução de forma transparente por meio da aplicação dos recursos paralelizáveis da linguagem C++ nos componentes desacoplados do PON; (iii) o código redundante (temporal e estrutural) é evitado por apresentar estruturação e flexibilidade algorítmica que favorece o compartilhamento de entidades, reduzindo o número de notificações necessárias, sempre à luz da teoria do PON. Além disso, quando comparado com as outras materializações em C++, o *Framework* PON C++ 4.0 apresenta melhora no desempenho das aplicações, reduzindo tanto os tempos de execução como o consumo de memória (NEVES, 2021).

2.4 Trabalhos relacionados

Esta seção tem como objetivo apresentar uma revisão da literatura do PON aplicado a sistemas distribuídos sob o aspecto teórico e prático. Devido às potenciais similaridades e

sinergias entre as áreas, apresentam-se também trabalhos que exploram o PON no potencial uso de múltiplos núcleos em um mesmo processador (*Multicore*).

A metodologia utilizada foi a busca no repositório do PON [http://nop.dainf.ct.utfpr.edu.br/] partindo dos trabalhos mais recentes e seguindo manualmente pela análise das referências bibliográficas. Dessa forma, conseguiu-se cobrir, se não todos, pelo menos os artigos mais relevantes sobre os temas propostos. As perguntas de pesquisa que nortearam a revisão sistemática foram:

PP1: Quais vantagens e limitações do PON foram analisadas no trabalho?

PP2: Qual a contribuição do trabalho para o PON em sistemas distribuídos?

O resultado resumido da revisão é mostrado graficamente na Figura 21. Utilizou-se na figura uma escala de cores, na qual a cor azul significa que o trabalho teve como principal objetivo a análise teórica e amarelo significa que o trabalho abordou também implementações ou alterações dos *frameworks* existentes.

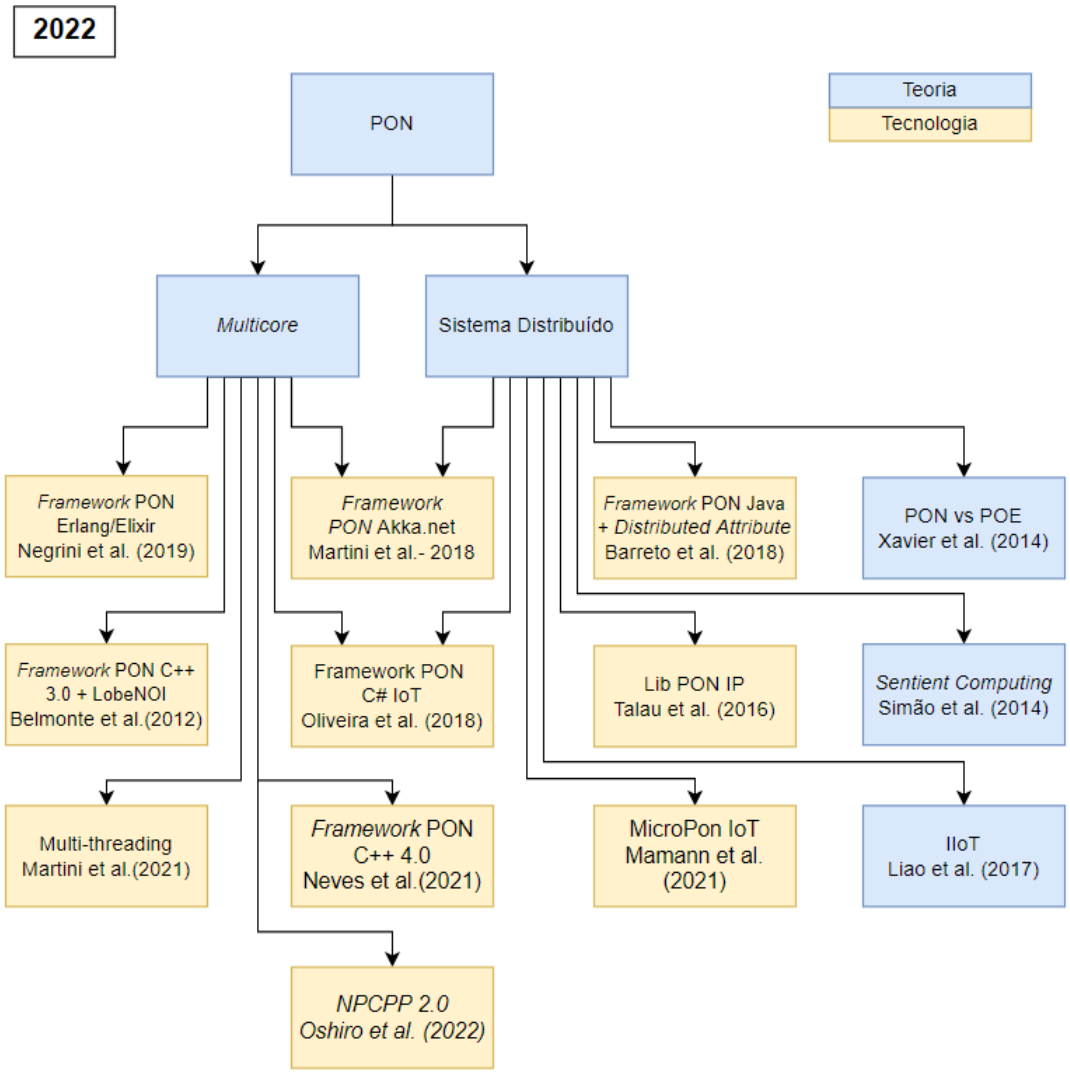


Figura 21 – Trabalhos de PON no contexto de sistemas distribuídos e/ou *multicore*.

Na Subseção 2.4.1 são apresentados os trabalhos teóricos, seguido pela Subseção 2.4.2 na qual são apresentados trabalhos técnicos no contexto de *multicore*. Nas seções subsequentes são detalhados cada um dos trabalhos técnicos envolvendo o PON em sistemas distribuídos. Ao final, na Subseção 2.4.9, são apresentadas as considerações sobre os trabalhos relacionados.

2.4.1 Trabalhos Teóricos abordando o PON em sistemas distribuídos

Em 2014, foi publicado por Xavier (2014) e Xavier *et al.* (2014) o primeiro trabalho analisando o PON em ambientes ou paradigmas para sistemas distribuídos. Neste trabalho, os autores apresentam uma comparação teórico-prática do PON e do POE. Os autores apresentam correlações das características estruturantes dos paradigmas conforme uma taxonomia comum e mensuram, por meio de casos de estudo, a complexidade de código fonte (número de linhas de código, escopos e *tokens*) e o desempenho em execução (tempo de resposta e tempo total de execução). Os autores destacam que o PON emprega o conceito de Orientação a Eventos em seu modelo de execução implícito (*i.e.*, ciclo de notificações). Dessa forma, as notificações ocorrem nas trocas de mensagem entre as unidades computacionais mínimas. No trabalho apresentado, o PON, apesar de ter inspiração em eventos, apresenta diferenças conceituais em relação ao POE, principalmente em relação às características estruturantes de um paradigma (Registro, Recipientes de Escopo Léxico, Independência, Estado Nomeado e Não determinismo Observável). Em relação aos experimentos, o uso do *Framework* PON apresentou tempos de resposta comparáveis ao POE (XAVIER, 2014).

No mesmo ano, em 2014, foi publicado por Simão *et al.* (2014) um estudo teórico sobre a aplicabilidade do PON frente às demandas da computação senciente. A Computação senciente ou computação ciente do contexto (*Context-Aware Computing*) é uma forma de computação ubíqua cuja função é monitorar ou perceber um ambiente e, eventualmente, agir sobre ele visando o benefício de seus usuários. Particularmente, a computação senciente se refere a sistemas computacionais, envolvendo IoT, capazes de responder às necessidades de cuidado humano como, por exemplo, identificação ou rastreabilidade de pessoas e identificação de movimentos e quedas para assistência médica. Nesse artigo, as demandas da computação senciente são analisadas nos paradigmas PD, PI e PON. Os autores destacam que o PON atende as demandas da computação senciente, principalmente em termos de expressividade, responsividade, distribuição e robustez. Destaca-se também que o PON pode ser especialmente interessante para a implementação de computação senciente baseada em eventos abstratos (SIMÃO *et al.*, 2014).

Ainda no escopo de trabalhos teóricos relacionando PON em ambientes distribuídos, tem-se, no ano de 2017, o trabalho de Liao *et al.* (2017) no qual é apresentada e descrita teoricamente uma aplicação do PON no contexto de IIoT (*Industrial Internet of Things*) e Indústria 4.0 através do desenvolvimento de um estudo de caso para uma fábrica inteligente. Nesse trabalho,

o PON se mostra como uma alternativa adequada, pois permite uma abordagem de controle e manufatura holônica de forma muito natural. O PON permite o desenvolvimento em alto nível (separação lógico-causal e facto-execucional) ao mesmo tempo em que se mostra naturalmente distribuído.

2.4.2 PON em *multicore* via implementações em C++

O primeiro trabalho a abordar o tema *Multicore* foi o de Belmonte, Simão e Stadzisz (2012), o qual foi publicado em 2016 (BELMONTE *et al.*, 2016). Os autores apresentam uma solução (conceito e uma implementação) para o balanceamento dinâmico de cargas de trabalho para a Inferência Orientada a Notificações (ION) do PON, a qual foi chamada LobeNOI (*Load Balance Engine for NOI*). São apresentados resultados do algoritmo utilizado em conjunto com o *Framework* C++ 3.0, demonstrando uma melhora na utilização do hardware disponível. O balanceamento da aplicação PON ocorre de forma transparente ao programador (BELMONTE; SIMÃO; STADZISZ, 2012) (BELMONTE *et al.*, 2016). Ademais, a tecnologia, ainda instável, foi aprimorada e utilizada por Schütz *et al.* (2018) e Schütz (2019).

Em 2021, foram apresentados três trabalhos considerando o PON aplicado em processamentos *muticore*. No primeiro, Neves (2021) e Neves, SIMÃO e Linhares (2021) apresentam uma nova versão do *framework* para o PON, nomeadamente *Framework* PON C++ 4.0. Conforme apresentado com detalhes na Seção 2.4.8, o *Framework* PON C++ 4.0 insere implementações recentes da linguagem C++ que, dentre outras melhorias, disponibiliza o suporte para programação concorrente possibilitando a execução de programas PON em ambientes *multithread*.

Em outro trabalho no mesmo ano de 2021, Martini *et al.* (2021) propôs uma implementação *multi-threading* gerada a partir de um código em Tecnologia LingPON. Os resultados mostraram a viabilidade da solução proposta além de evidenciar a melhora no desempenho, sendo que os códigos compilados em ambiente *multi-threading* apresentaram redução no tempo necessário para execução das aplicações propostas em comparação aos métodos originais, não *multi-threading*.

Por fim, no terceiro trabalho no ano de 2021, Oshiro *et al.* (2021) e Oshiro (2021) apresentam uma nova versão para a tecnologia LingPON (nomeadamente, LingPON 2.0), com a possibilidade de geração de código notificante modular em C++, inclusive com suporte para execução *multi-threading*. Oshiro (2021) apresenta implementações para os *Methods* do PON, utilizando as tecnologias *pThreads* e *Thread Pooling*.

2.4.3 *Framework* PON C++ 2.0 e 3.0 + PONIP

Com relação aos trabalhos técnicos abordando o PON em contexto de sistemas distribuídos, em 2016, Talau (2016) apresentou uma biblioteca para comunicação em rede nomeada de PONIP. Essa biblioteca permite o envio e o recebimento de *Attributes* e *Premises* via rede (sob TCP ou UDP (*User Datagram Protocol*)) por meio de um protocolo da camada de aplicação desenvolvido pelos autores. O protocolo, apresentado na Figura 22, possui uma estrutura de mensagem para o envio dos estados dos *Attributes* e uma estrutura para envio do estado das *Premises*.

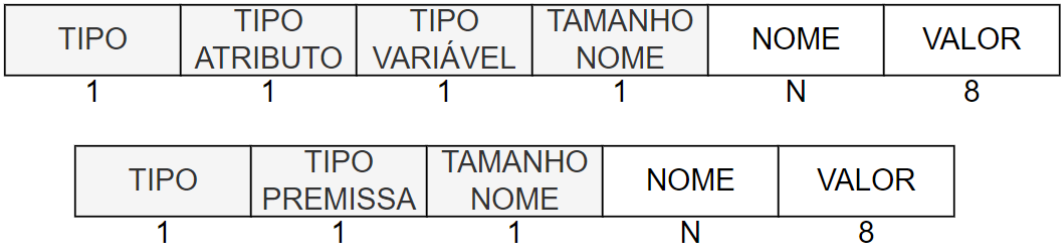


Figura 22 – Protocolo de comunicação utilizado pelas *Attributes* e *Premises* no PONIP
Fonte: Talau (2016)

A identificação do tipo de entidade é indicada no primeiro byte da mensagem. No caso de *Attributes*, o segundo campo indica se o atributo vai ser do tipo *RENOTIFY* ou não e o terceiro campo indica o tipo da variável (*int, float, string* ou *char*). Na sequência, o campo 'tamanho nome' indica o tamanho do nome do *Attribute*, em bytes, seguido pelo nome do *Attribute* e o valor, limitado em oito bytes. No caso das *Premises*, o protocolo é semelhante ao dos *Attributes* exceto pela ausência do campo 'tipo variável' e pela limitação do campo valor com apenas um byte, cobrindo os dois valores possíveis das *Premises*: verdadeiro ou falso. O desenvolvimento do PONIP como uma biblioteca dinâmica permitiu a sua importação e seu uso em múltiplas linguagens de programação. O artigo-relatório demostra ainda três casos de uso da biblioteca mostrando sua aplicabilidade em cenários com clonagem de regras (redundância), computação senciente e adequação ao *Framework* PON C++ 2.0.

Em complemento ao trabalho de Talau (2016), tem-se o artigo-relatório de Mendes *et al.* (2019) adaptando a linguagem LingPON para a geração de código em C++, via *Frameworks* PON C++ 2.0 e 3.0, com suporte para a comunicação em rede via PONIP e funcionalidades de *multicore*.

2.4.4 *Framework* PON Akka e Erlang/Elixir

Em 2018, foi proposto e implementado por Martini, SIMÃO e LINHARES (2018) um *framework* baseado nas bibliotecas Akka.net. O Akka.net se apresenta como uma versão do Akka (desenvolvido em Java ou Scala) para plataforma .net (C#). O Akka.net é um conjunto de bi-

bliotecas de código aberto criado para facilitar o desenvolvimento de sistemas distribuídos e escaláveis baseados no Modelo de Atores. Os autores apresentam resultados mostrando que em relação ao tempo de execução das aplicações o *Framework* PON Akka.net é bem semelhante aos *frameworks* existentes em C++ ou até melhor (em alguns casos). Duas vantagens do Akka.net são as ferramentas disponíveis e o suporte nativo a *multicore*/distribuição. Além disso, o Akka.net trabalha no modelo de atores que possui algumas similaridades muito úteis para o desenvolvimento em PON. Os autores destacam ainda o *Framework* PON Akka.net como um caminho produtivo para o desenvolvimento e avaliações do PON. Pertinente ressaltar, que apesar do *Framework* PON Akka.net permitir (via herança das propriedades da linguagem) a distribuição, esse *framework* não foi avaliado nesse contexto.

Em 2019, tem-se o trabalho de Negrini (2019) e Negrini *et al.* (2019) apresentando a integração das tecnologias do PON com o modelo de atores do Erlang utilizando a linguagem Elixir. É apresentado também um novo *target* para os compiladores NOPL permitindo que os programas escritos em NOPL sejam compilados para Erlang/Elixir. A tecnologia Erlang/Elixir é conhecida pela facilidade de desenvolvimento de aplicações distribuídas robustas e tolerantes a falhas. Neste trabalho é analisado e explorado o aspecto do PON com Erlang/Elixir para o balanceamento e aproveitamento de múltiplos núcleos de processamento em um sistema. Os resultados mostram que o PON junto com Erlang/Elixir conseguiu aproveitar todas as unidades de processamento disponíveis. É pertinente ressaltar que, de forma similar ao *Framework* PON Akka.net, apesar do Erlang/Elixir permitir (via herança das propriedades da linguagem) a distribuição dos elementos, o *framework* PON Erlang não foi experimentado e/ou avaliado nesse contexto.

2.4.5 *Framework* PON C# IoT

Em 2018, são apresentados por Oliveira (2019) e Oliveira *et al.* (2018) o *Framework* PON C# IoT juntamente com seu uso em uma aplicação para computação senciente. O *Framework* PON C# IoT foi desenvolvido utilizando como referência a materialização *Framework* PON C++ 2.0 (RONSZCKA, 2012) e o *Framework* PON C# 1.0 (HENZEN, 2015). Os autores apresentam e implementam funcionalidades sofisticadas como: (i) a mudança dinâmica de *Rules* (em tempo de execução); (ii) a persistência de *Rules* em banco de dados; (iii) uma interface gráfica para a criação e edição de *Rules*; (iv) a capacidade de aninhamento infinito de *Subconditions*; (v) o paralelismo nativo em C# (via função *Parallel.ForEach*); (vi) a impertinência dinâmica de *Premises* e *Conditions* e; (vii) a possibilidade de distribuição de *Attributes* via rede.

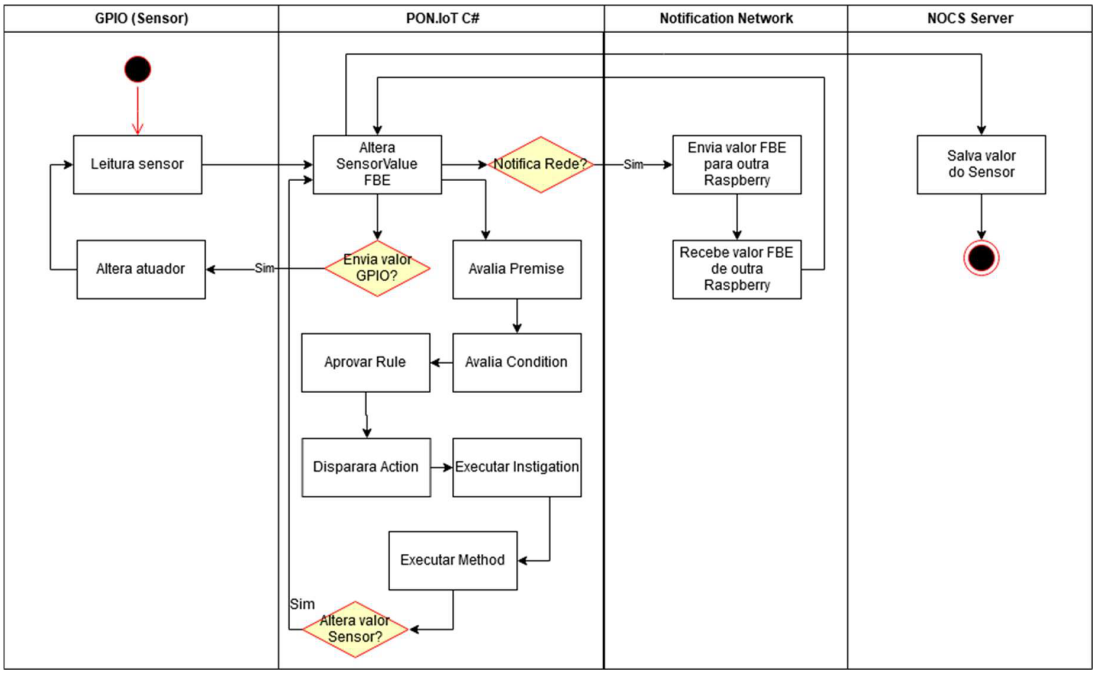


Figura 23 – Diagrama de atividades após a leitura de um sensor via *Framework* PON C# IoT

Fonte: Oliveira (2019)

Conforme exemplificado na Figura 23, a alteração de um *Attribute* (representado por *SensorValue*) pode provocar, além do fluxo natural de notificações locais das *Premises*, o envio da notificação via rede provocando a alteração de *Attributes* em aplicações remotas. Ao receber os valores via rede, os *Attributes* correspondentes são alterados podendo novamente reiniciar o ciclo de notificações. Conforme apresentado por Oliveira (2019), a distribuição dos *Attributes* ocorre via um protocolo definido pelo autor. O protocolo consiste no envio e recebimento de texto criptografado via HTTP e é composto por três valores, separados pelo caractere barra vertical ou *pipe* (|): *N*|*SensorKey*|*SensorValue*. Neste protocolo, o primeiro termo corresponde ao tipo da mensagem, podendo ser N para notificação de valores ou C para alterações (*Change*) em *Rules* ou Sensor *FBE*. O segundo termo da mensagem é a chave única do Sensor *FBE*, necessária para que o computador que receber a mensagem localize qual de seus sensores/atuadores deve ter o valor alterado. Por fim, o terceiro termo é o próprio valor do sensor.

O *Framework* PON C# IoT foi aplicado para o desenvolvimento de um experimento no contexto de computação senciente, conforme exemplificado na Figura 24. O diagrama ilustra os principais componentes da solução proposta e as interações entre seus respectivos dispositivos. A primeira parte é composta por uma estrutura eletrônica de hardware programável *Raspberry* Pi 3 e um conjunto de sensores e atuadores ligados a ele. A segunda parte é composta por quatro aplicativos: (i) o portal NOCS-*Server* (*Notification Oriented Care System*), responsável pela centralização, gerenciamento e armazenamento da solução; (ii) o controle do *Raspberry* Pi 3, com seus sensores, denominado NOCS *Control*; (iii) o NOCS Mobile, sendo um aplicativo para dispositivos móveis e; (iv) um simulador de ambientes.

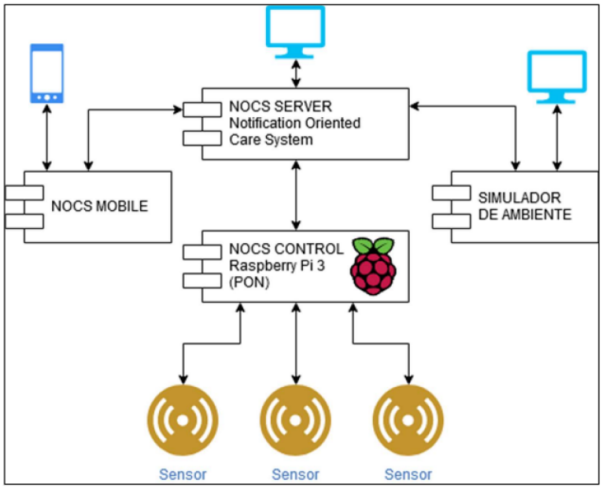


Figura 24 – Diagrama de componentes do experimento realizado com o *Framework* PON C# IoT
Fonte: Oliveira (2019)

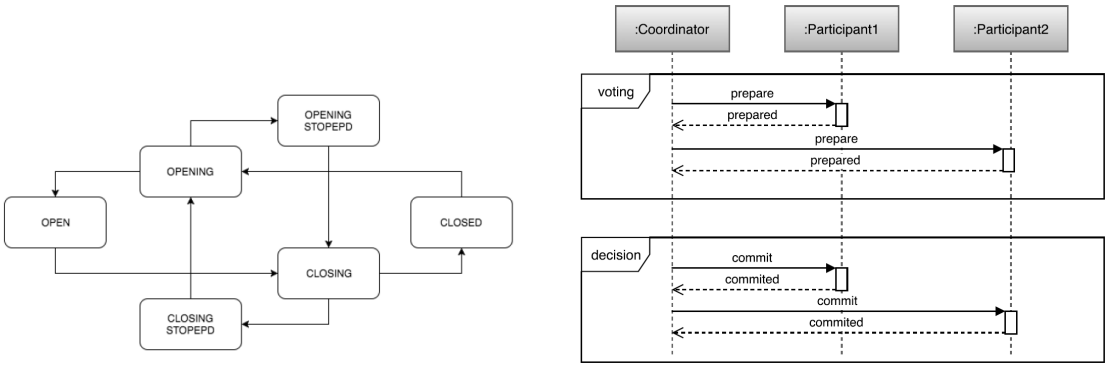
Conforme apresentado por Oliveira (2019), a Figura 24 também apresenta o fluxo de dados entre os componentes da solução NOCS. Primeiramente, há o portal NOCS *Server* que fornece as configurações de cada ambiente para os dispositivos *Raspberry* Pi com sistema NOCS *Control*, sendo que cada *Raspberry* Pi buscará as definições das Rules e sensores do seu respectivo ambiente. Para o monitoramento dos ambientes, há o NOCS *Mobile* que permite a visualização e alteração nos ambientes, a partir de consultas no NOCS *Server*. Por fim, existe um simulador que carrega as definições do ambiente para validação de *Rules* específicas, obtidas do NOCS *Server*. Os resultados do experimento mostraram que a aplicação desenvolvida é adequada ao contexto proposto, apresentando tempos de execução na ordem de milissegundos (OLIVEIRA *et al.*, 2018).

2.4.6 *Framework* PON Java + *Attributes* Distribuídos

Em 2018 tem-se o trabalho de Barreto, Vendramin e SIMÃO (2018), apresentando um método para escrita de aplicações utilizando PON no contexto de Sistemas Distribuídos. Os autores apresentam o conceito de *Attribute* Distribuído. Esse conceito é apresentado e testado no *framework* Java. O *Attribute* Distribuído é um elemento que interage com as outras entidades PON de forma semelhante aos *Attributes*. Ele pode ser utilizado por uma *Premise* ou ter seu estado alterado por um *Method*. Porém, quando o *Attribute* Distribuído tem seu estado alterado, uma mensagem de notificação com o novo estado é enviada via rede. Durante a inicialização da aplicação é realizada a configuração de um cliente de rede responsável por receber as mensagens. A distribuição dos *Attributes* foi implementada utilizando-se comunicação em grupo por *multicast*. O *multicast* é um método de transmissão de um único pacote de dados para um grupo de destinatários. Dessa forma, cada *Attribute* distribuído é identificado como um endereço *multicast* único para todas as aplicações, enviando e recebendo mensagens por meio desse endereço de grupo. Como cada *Attribute* está relacionado a um único grupo, utilizou-se como

conteúdo das mensagens trafegadas somente o novo valor do *Attribute* em questão. Dessa forma, o *Attribute* Distribuído pode coexistir e compartilhar seu estado com várias aplicações PON desde que estas estejam conectadas na mesma rede.

Os autores apresentam dois estudos de caso. Conforme exemplificado na Figura 25, o primeiro experimento corresponde a um simulador de controle para um portão eletrônico (Figura 25a) e o segundo um protocolo de transações distribuídas (*two phase commit* - Figura 25b).



(a) Portão Eletrônico. **(b) *Two Phase Commit*.**

Figura 25 – Experimentos realizados com o *Framework* PON Java + *Attributes* Distribuídos.
Fonte: Barreto, Vendramin e SIMÃO (2018)

As aplicações foram implementadas em PON (com o *Framework* PON em Java e a proposta de *Attributes* Distribuídos) e por meio de modelos e paradigmas tradicionais baseados em RPC (*Remote Procedure Calls*) (COULOURIS; DOLLIMORE; KINDBERG, 2011). A comparação ocorreu em termos da avaliação da exatidão do *software* (em relação aos requisitos) e da eficiência de rede, medido pelo número de mensagens trocadas entre as aplicações. Os resultados mostraram que as aplicações desenvolvidas em PON apresentaram a mesma exatidão que as aplicações baseadas em RPC. Com relação à métrica de eficiência (número de mensagens trocadas) os dois estudos de caso apresentaram resultados diferentes, sendo que no estudo de caso do portão eletrônico o PON precisou de mais mensagens e no estudo com o protocolo *Two Phase Commit* o número de mensagens foi menor. Com isso, os autores concluíram que o PON em conjunto com o *Attribute* Distribuído se mostrou aplicável no contexto de sistemas distribuídos e que a eficiência de rede não é afetada pelos aspectos declarativos do PON (BARRETO; VENDRAMIN; SIMÃO, 2018).

2.4.7 MicroPON em IoT

Em 2021, Mamann *et al.* (2021) apresentam o MicroPON em IoT. O MicroPON é uma materialização do PON no contexto de microcontroladores, os quais normalmente apresentam limitações de memória e processamento, quando comparados com os usuais computadores e *laptops*. Conforme apresentado na Figura 26, o MicroPON foi experimentado em uma aplicação envolvendo sensores e atuadores comunicando-se em rede. A comunicação entre os elementos

do sistema ocorre por meio do protocolo TCP para endereços definidos em tempo de codificação.

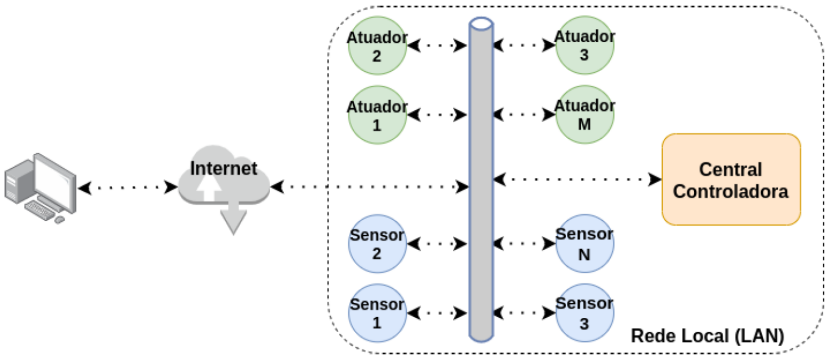


Figura 26 – Experimento realizado com o MicroPON em IoT.
Fonte: Mamann *et al.* (2021)

A aplicação foi desenvolvida em dois paradigmas de programação, a saber o PI e o PON. Mamann *et al.* (2021) propõe a experimentação utilizando duas principais plataformas: (i) um microcontrolador ATMEGA328P-PU, comparando o PI (via implementação em C++) com o PON (via *PON Namespaces AVR*); (ii) um servidor AAMD FX-4300, comparando o PI (via implementação em C++) com o PON (via *PON Namespaces* e *Framework* PON C++ 4.0).

Posteriormente, as implementações foram comparadas em termos do número de linhas de código, tempo médio de requisição sem ativação de *Rules*, tempo médio de requisição com ativação de *Rules* e tráfego de dados na rede. Em termos de linhas de código, as implementações em PON foram mais verbosas que em PI. O tempo de processamento das aplicações implementadas em PON foram, na grande maioria, melhores ou iguais ao PI. Em termos de tráfego de dados na rede, a implementação em PI apresentou valores entre 70 e 80 vezes maior que a implementação em PON.

A troca de mensagens entre a Central Controladora e os sensores/atuadores ocorreu utilizando-se uma implementação diretamente sobre o protocolo TCP. A troca de informações ocorreu por meio dos métodos GET e POST, similares porém mais simples em relação ao protocolo HTTP. Ademais, definiu-se um padrão para o conteúdo das mensagens pré-definido pelo autor. Definiu-se URLs específicas para comunicação dos sensores ("*/api/v1/sensor/state*") e dos atuadores ("*/api/v1/actuator/state*"). O conteúdo das mensagens trafegadas possuem basicamente dois campos: (i) o campo ‘id’, contendo a identificação do sensor ou do atuador em ambas as requisições (GET e POST); (ii) o campo ‘value’, contendo o valor correspondente ao estado do atuador ou do sensor nas respostas de requisições de GET, ou contendo o novo valor do sensor no caso de requisições POST.

2.4.8 *Framework* PON C++ 4.0

O *Framework* PON C++ 4.0 foi proposto e implementado por Neves (2021). Esse *framework* é atualmente o mais recente e o que apresenta os melhores resultados em comparação com os outros *frameworks* desenvolvidos em C++. Além de apresentar menor tempo de processamento e melhor uso dos recursos de memória, em comparação com os outros *frameworks* existentes, o *Framework* PON C++ 4.0 foi implementado utilizando programação genérica, o que permite maior flexibilidade nos tipos, comparações lógicas e ações realizadas pelos elementos do PON (*FBE*, *Attribute*, *Method*, *Rule*, *Premise*, *Condition*, *Instigation* e *Action*).

A Figura 27 apresenta o Diagrama de Classes UML do *Framework* PON C++ 4.0. De um modo geral, o *Framework* PON C++ 4.0 apresenta modelagem inspirada no modelo de estrutura de classes similar ao do *Framework* PON C++ 2.0, porém naturalmente faz algumas simplificações e melhorias, visando aumentar a facilidade de uso do código decorrente. Nesse contexto, podem ser destacadas algumas diferenças entre as duas versões sendo a principal diferença a eliminação do pacote *Application*, viabilizado ao utilizar elementos mais desacoplados que não precisam de estruturas gerenciadoras acoplantes (NEVES, 2021).

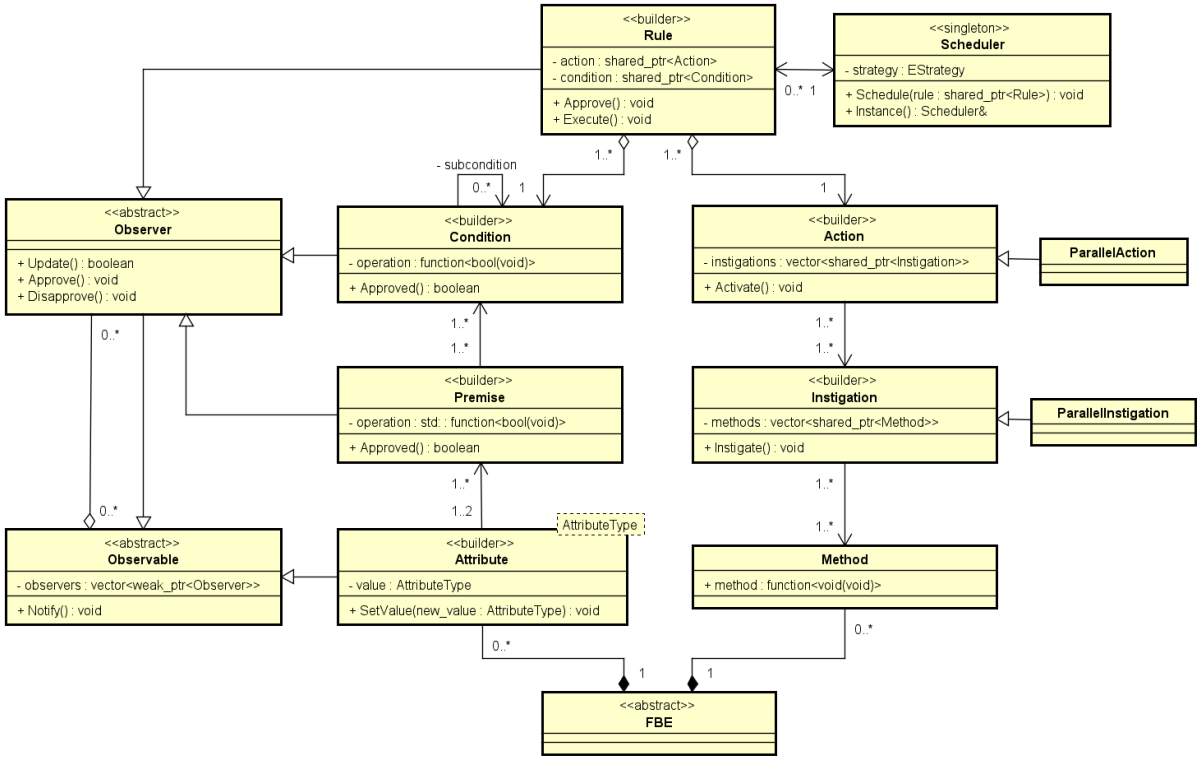


Figura 27 – Diagrama de classes do *Framework* PON C++ 4.0.

Fonte: Neves (2021)

Além das diferenças, algumas propriedades se assemelham ao *Framework* PON C++ 2.0. Um dos exemplos é o mecanismo de notificações implementado por meio do padrão de projeto *Observer* (GAMMA *et al.*, 1995). O padrão de projeto *Observer* define uma relação de um para muitos entre objetos. A implementação do padrão de projeto *Observer* no *Framework*

PON C++ 4.0 é dada por meio das classes *Observer* e *Observable*. A implementação dos *Attributes* é realizada por meio da classe *Attribute* que deriva apenas da classe *Observable*. Essa classe é implementada por meio do uso de *templates* para cobrir a flexibilidade de tipos padronizados (*int, bool, float, string* etc) ou até mesmo classes definidas pelo desenvolvedor. As classes *Premise*, *Condition* e *Rule* derivam da classe *Observer* e são implementadas de forma muito similar. É pertinente ressaltar que, apesar da classe *Attribute* ser implementada com *templates*, essa dependência não existe na classe *Premise*, pois os tipos foram abstraídos pelo uso de expressões *lambda*, de forma que a classe *Premise* não precisa armazenar a informação do tipo do *template* dos seus *Attributes* (NEVES, 2021).

Outros dois elementos do PON, as entidades *Action* e *Instigation* são implementadas de maneira bastante simples sendo que a *Action* guarda os ponteiros para as *Instigations*, utilizando a estrutura definida por *NOPContainer*, enquanto a *Instigation* guarda referências para os *Methods*. A *Action* é ativada quando a sua *Rule* é aprovada, instigando as *Instigations* em sua lista. A *Instigation*, por sua vez, quando instigada executa os *Methods* na sua lista. O *Method*, ao seu turno, é implementado por meio de *std::function*, apresentando grande flexibilidade pelo o uso de expressões *lambda*, sendo possível criar de forma fácil *Methods* que realizam operações complexas, como fazer múltiplas atribuições de valores a *Attributes* e variáveis ou chamada de outras funções e métodos (NEVES, 2021).

O Código 2.5 apresenta um exemplo de código, utilizando o *Framework* PON C++ 4.0, para uma aplicação baseada em uma rede de sensores na qual cada sensor possui um estado (ativado ou desativado) que pode ser observado. A representação desta aplicação sob a forma de um *FBE* e uma *Rule* é apresentada na Figura 28. Uma *Rule* determina o comportamento do sensor, sendo que quando o sensor é ativado esta *Rule* é aprovada, reiniciando os estados de ativação e leitura do sensor (NEVES, 2021).

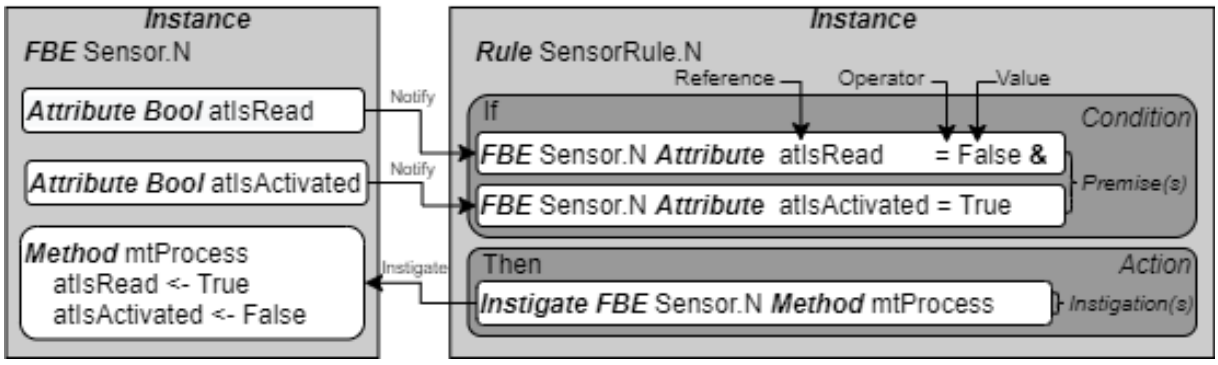


Figura 28 – Exemplo de uma *Rule* para avaliação de um sensor.

Fonte: Neves (2021)

Código 2.5 Exemplo de aplicação de sensores utilizando o *Framework* PON C++ 4.0.

```
struct NOPSensor{

    //Declaração dos Attributes
    NOP::SharedAttribute<bool> atIsRead{ NOP::BuildAttribute(false) };
    NOP::SharedAttribute<bool> atIsActivated{ NOP::BuildAttribute(false) };

    //Declaração das Premises
    NOP::SharedPremise prIsActivated{
        NOP::BuildPremise<bool>(atIsActivated, true, NOP::Equal()) };

    NOP::SharedPremise prIsNotRead{
        NOP::BuildPremise<bool>(atIsRead, false, NOP::Equal()) };

    //Declaração das Rule juntamente com conditions, actions
    //e intigations implicitamente
    NOP::SharedRule rlSensor{ NOP::BuildRule(
        NOP::BuildCondition<NOP::Conjunction>(prIsActivated, prIsNotRead),
        NOP::BuildAction(NOP::BuildInstigation([&]() {
            this->Read(); this->Deactivate();
        }))) };

    //Declaraçãod dos Methods
    void Read() const { atIsRead->SetValue(true); }

    void Activate() const {
        atIsActivated->SetValue(true);
        atIsRead->SetValue(false); }

    void Deactivate() const { atIsActivated->SetValue(false); }

};
```

Fonte: Neves (2021)

Além de evoluções no projeto em relação ao modo de implementação dos elementos do PON, o *Framework* PON C++ 4.0 utiliza-se também de evoluções provenientes de novos recursos adicionados nas revisões mais recentes da tecnologia C++, também chamado de C++ moderno (NEVES, 2021). Além disso, o *Framework* PON C++ 4.0 apresenta melhorias no processo de construção e manutenção por meio do desenvolvimento orientado a testes conforme apresentado na Seção 2.4.8.1.

2.4.8.1 Testes de *software*

Além das inovações de tecnologia, o *Framework* PON C++ 4.0 apresenta também inovações em sua forma de manutenção através de testes de *software* unitário, de integração e de desempenho. O conjunto de testes além de garantir o comportamento esperado dos elementos do PON implementados no *framework*, permite aos programadores adicionar e modificar códigos internos do *framework* com segurança, uma vez que é possível a reexecução do conjunto de testes para garantir que as alterações efetuadas não prejudicaram as funcionalidades básicas cobertas nos testes. Conforme apresenta Neves (2021), os testes unitários foram elaborados para as *Premises*, *Conditions*, *Subconditions*, *Rules* e *Methods* dos *FBEs*, enquanto os *Attributes*, *Actions* e *Instigations*, por sua vez, não precisam passar por testes unitários, devido ao fato dos *Attributes* representarem apenas uma declaração de estado, enquanto *Actions* e *Instigations* têm sempre o mesmo comportamento garantido por definição (KOSSOSKI, 2015).

Em conjunto com os testes unitários, os testes de integração verificam o comportamento do sistema na interação entre os componentes. Conforme proposto por Kossoski (2015), duas abordagens são possíveis para o teste de integração utilizando casos de uso: desenvolver testes que exercitem as informações da documentação e comportamento esperado dos casos de uso; e desenvolver testes que exercitem diretamente as entidades que implementam o caso de uso (*i.e.*, *Attributes*, *Premisses*, *Conditions* e *Rules*).

Adicionalmente aos testes unitários e de integração, foram desenvolvidos também testes de desempenho que permitem a comparação entre aplicações desenvolvidas no *Framework* PON C++ 4.0 com outros *frameworks* PON ou com código desenvolvido em linguagem C++. Essas comparações de desempenho utilizam como métricas o tempo de execução total e tempo de CPU (*Central Process Unit)* da aplicação durante múltiplos ciclos de execução.

Os testes foram desenvolvidos utilizando-se o *framework* de teste Google Test e os testes de desempenho foram desenvolvidos utilizando-se o *framework* Google Benchmark (NEVES, 2021).

O *Framework* PON C++ 4.0, proposto e implementado por Neves (2021) apresenta avanços no estado da técnica do PON. Conforme apresentado na Seção 2.3.3, dentre os *Frameworks* disponíveis atualmente, este é o que implementa, de forma satisfatória, a programação em alto nível, paralelismo e bom desempenho na execução de aplicações por evitar redundâncias (NEVES, 2021) (BABU, 2022). Em comparação com os *frameworks* em C++, apesar de se inspirar em partes dos demais *frameworks* (principalmente no *Framework* PON C++ 2.0), o *Framework* PON C++ 4.0 utiliza recursos inseridos nas últimas versões do C++ apresentando melhores resultados em testes de tempo de processamento, paralelismo e alocação e uso de memória RAM. Além disso, é possível observar o uso de menos linhas de código para composição do *framework* e das aplicações que o utilizam em virtude do uso de programação genérica (via *templates*) em sua composição (NEVES, 2021).

2.4.9 Considerações sobre os Trabalhos Relacionados

Em complemento à Figura 21 previamente apresentada, o Quadro 4 apresenta um resumo das contribuições para o desenvolvimento de tecnologia do PON. É importante ressaltar que as contribuições dos trabalhos não se limitam somente às apresentadas no quadro, sendo estas apresentadas de forma resumida e com foco no contexto de sistemas distribuídos e IoT. Destaca-se também que embora os trabalhos desenvolvidos com os *Framework* Akka.Net (MARTINI; SIMÃO; LINHARES, 2018) e Erlang/Elixir (NEGRINI *et al.*, 2019) suportem a distribuição dos elementos por herança das linguagens/bibliotecas utilizadas, essa característica não foi explorada nos trabalhos apresentados e, portanto, foram destacadas como Não Avaliadas (N/A).

Quadro 4 – Trabalhos realizados em PON no contexto de sistemas distribuídos.

Framework	**Distribuição**	**Paralelismo**	**Entidades Distribuídas**	**Protocolo de Transporte**	**Protocolo de Aplicação**
Framework PON C++ 2.0 + PONIP	Sim	Não	*Attributes* *Premises*	UDP/TCP	PONIP
Framework PON C++ 3.0 + (LobeNOI + PONIP)	Sim	Sim	*Attributes* *Premises*	UDP/TCP	PONIP
Framework PON Akka	Potencialmente	Sim	N/A	N/A	N/A
Framework PON Erlang/Elixir	Potencialmente	Sim	N/A	N/A	N/A
Framework PON C# IoT	Sim	Sim	*Attributes*	UDP/TCP	HTTP Próprio
Framework PON Java + *Attributes* Distribuídos	Sim	Não	*Attributes*	UDP (*Multicast*)	Próprio
MicroPON em IoT	Sim	Não	*Attributes*	TCP	Próprio

Pode-se observar que, apesar de alguns trabalhos já terem explorado o PON no contexto de sistemas distribuídos e em IoT, cada um segue uma abordagem distinta e não completa por não permitirem a distribuição de todas as estidades do PON. Os *frameworks* e bibliotecas existentes até o momento não permitem a distribuição de todos os elementos do PON. Além disso, as implementações atuais divergem em protocolos e modelos de distribuição. Esse último fato pode dificultar o uso e a adoção do PON em sistemas no contexto de IoT pois, conforme apresentado na Seção 2.1.1, geralmente são utilizados protocolos padronizados.

Em comparação, o trabalho de Oliveira (2019) apresenta uma versão distribuída para IoT porém utilizando o protocolo HTTP para comunicação e o *Framework* PON C# IoT o qual é relativamente menos maduro (encontra-se ainda nas primeiras versões) e apresenta uma abordagem não completa em relação à distribuição de todas as entidades do PON.

Nota-se também que os *frameworks* apresentados na Seção 2.3.3, principalmente em C++, não implementam ainda, de forma estável, o paralelismo e distribuição na mesma materialização. Por esses motivos, o presente trabalho propõe um estudo amplo com o objetivo de convergir os esforços e os protocolos de distribuição do PON em uma mesma direção além de se basear no *framework* C++ mais recente e estado da técnica, nomeadamente o *Framework*

PON C++ 4.0. Neste contexto, o próximo capítulo apresenta uma proposta de distribuição das entidades constituintes do PON por meio do *Framework PON C++ 4.0* e do protocolo MQTT.

3 *FRAMEWORK* PON C++ 4.0 IOT

Conforme apresentado na Seção 2.4.8, o *Framework* PON C++ 4.0 foi projetado com o objetivo de facilitar o desenvolvimento e a execução de aplicações em PON no estado da técnica. Por utilizar recursos recentes da linguagem C++ e fazer uso de programação genérica, o *Framework* PON C++ 4.0 se apresenta como uma solução versátil e expansível, permitindo quiçá uma ampla adoção pela comunidade acadêmica e de desenvolvedores de *software* em geral (NEVES, 2021).

Como visto na Seção 2.3.3, o *Framework* PON C++ 4.0 por si mesmo não atende a propriedade de distribuição. Neste contexto, a implementação de interfaces e módulos voltados para a distribuição desse *framework* permitiriam, além do seu uso para análises acadêmicas do PON em ambiente distribuído, a viabilidade do uso do PON em aplicações distribuídas em geral e, mais especificamente, em IoT que está na visada deste trabalho de pesquisa.

Neste contexto dado, este presente trabalho apresenta e implementa uma proposta de distribuição computacional do processamento das entidades do PON, por meio da conformação do *Framework* PON C++ 4.0 para tal. Além da distribuição das entidades PON, outro aspecto técnico contemplado para a aderência ao ambiente de IoT em geral é o uso de uma arquitetura e de um protocolo de comunicação padronizado, nesse caso, o MQTT.

Ademais, manteve-se o padrão de desenvolvimento adotado por Neves (2021), seguindo um modelo de desenvolvimento guiado por testes (TDD da sigla em inglês para *Test Driven Development*) no qual são criados testes unitários e de integração para cada nova funcionalidade adicionada. A disponibilidade de testes pode ser considerada um ponto muito relevante para a manutenibilidade e expansão de um *framework* dentro da comunidade acadêmica e de potenciais desenvolvedores subsequentes.

Este capítulo se inicia com a classificação dos modos de distribuição das entidades do PON, detalhado na Seção 3.1. Em seguida, são apresentados na Seção 3.2 os detalhes da integração do PON com o MQTT. Na Seção 3.3 são apresentadas as mudanças adicionadas ao *Framework* PON C++ 4.0 para integração com o MQTT. Em seguida, na Seção 3.3.3.2, é apresentada uma abordagem para testes de integração. Por último, são apresentadas as considerações do capítulo na Seção 3.4.

3.1 Compartilhamento de entidades do PON via Distribuição

De modo resumido, uma aplicação em PON é desenvolvida por meio da definição de um conjunto de entidades do modelo do paradigma, organizadas de maneira a criar uma cadeia de notificações coerente para a solução de um problema específico (RONSZCKA, 2019). Neste sentido, é necessária uma boa organização das entidades para a criação de uma aplicação coesa e bem estruturada. Nesse âmbito, é considerada uma boa prática compartilhar entidades no PON mesmo em ambiente distribuído, tanto para questões de facilidade de desenvolvimento,

quanto para questões de desempenho (BANASZEWSKI, 2009) (RONSZCKA, 2019) (NEVES, 2021).

Para exemplificar e relembrar o compartilhamento de entidades, considera-se um sistema de alarme composto por três *Attributes* representando o estado do Alarme (*Alarm.atON*) e os estados de dois sensores (*Sensor1.atStatus* e *Sensor2.atStatus*), conforme apresentado na Figura 29. São consideradas também, neste exemplo, três *Premises* (*prAlarmOn, prSensor1* e *prSensor2*), representando respectivamente a avaliação do estado dos três *Attributes*, sendo estas ativadas caso os *Attributes* assumam valor *true* (verdadeiro).

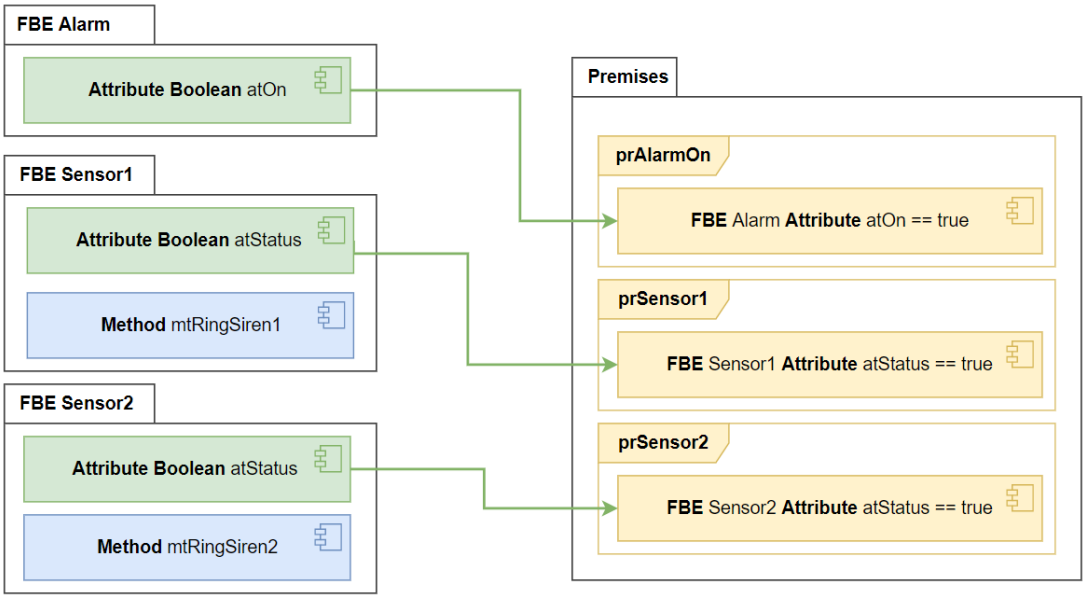


Figura 29 – Declaração de *Premises* referentes à um sistema de alarme.

Fonte: Adaptado de Neves (2021)

Com base nas *Premises* apresentadas na Figura 29, descrevem-se duas *Rules*, mostradas na Figura 30. A *Rule Alarm 1* é responsável por ativar uma sirene 1 (*mtRingSiren1*) caso o alarme esteja ligado (*prAlarmOn*) e o sensor 1 seja ativado (*prSensor1*). De forma similar, a *Rule Alarm 2* é responsável por ativar uma sirene 2 (*mtRingSiren2*) caso o alarme esteja ligado (*prAlarmOn*) e o sensor 2 seja ativado (*prSensor2*). Destaca-se que, neste caso, uma mesma avaliação (*prAlarmOn*) é realizada pelas *Conditions* de ambas as *Rules* em questão. Para evitar a redundância estrutural (e mesmo temporal) utiliza-se o compartilhamento de uma mesma *Premise* (*prAlarmOn*) nas duas *Conditions*, sendo que ambas serão notificadas por esta *Premise*, quando pertinente (NEVES, 2021).

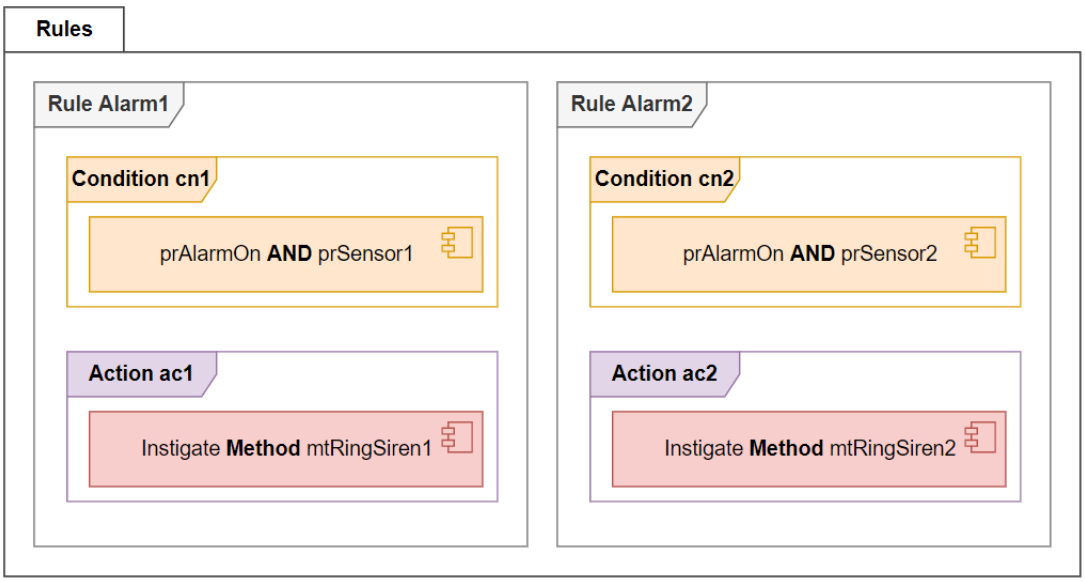


Figura 30 – Declaração de *Rules* utilizando *Premises* compartilhadas.

Fonte: Adaptado de Neves (2021)

Neste contexto, o correto uso do compartilhamento elimina a criação de entidades redundantes e, ao mesmo tempo, evita notificações desnecessárias o que é ainda mais pertinente no caso de sistemas distribuídos. Por esse motivo, o compartilhamento de entidades é um meio útil para a resolução do problema de redundância estrutural e temporal, presentes nos usuais paradigmas de programação (conforme apresentado na Seção 2.2) os quais afetam uso apropriado de processamento, uso apropriado de rede em caso de sistemas distribuídos, dentre outros fatores.

No caso dos *Attributes*, o compartilhamento destas entidades permite que diferentes *Premises* sejam notificadas quando um único *Attribute* tiver seu estado alterado. Dessa forma, ajuda-se a evitar a redundância temporal, disparando notificações precisas somente para as *Premises* pertinentes. Para o caso das *Premises*, das *Conditions* e das *Rules* (ou *Master Rules*), o compartilhamento destas entidades permite que uma única avaliação causal seja utilizada por diferentes entidades em diferentes partes do código. Dessa forma, esse mecanismo evita que uma mesma avaliação seja declarada múltiplas vezes (redundância estrutural) ao mesmo tempo que evita que essa mesma avaliação seja realizada múltiplas vezes desnecessariamente (redundância temporal).

No caso das *Actions* e *Instigations*, o compartilhamento destas entidades evita que uma mesma sequência de execuções (ou ações) seja declarada múltiplas vezes em diferentes partes do código, evitando-se a redundância estrutural. Utilizando-se o compartilhamento, diferentes *Rules* podem executar uma mesma *Action*, quando pertinente. De maneira similar, diferentes *Actions* podem ativar uma mesma *Instigation*, quando necessário.

Com tudo isso relembrado e considerado, o compartilhamento das entidades via rede (*i.e.*, entidades distribuídas) apresenta ainda mais benefícios do que compartilhamento de entidades localmente. Objetivamente, ao se utilizar entidades distribuídas objetiva-se evitar a re-

dundância estrutural e temporal também em aplicações distribuídas para evitar redundâncias desnecessárias de processamento e também de comunicação. Neste contexto dado, diferentes *Rules* poderiam, por exemplo, utilizar as mesmas *Conditions* ou *Actions* mesmo que estas estejam em diferentes computadores, desde que estes possuam um meio para comunicação de dados e um mecanismo de troca de mensagens.

Utilizando-se como exemplo o sistema apresentado na Figura 30, pode-se exemplificar ainda o compartilhamento das *Premises* via rede, conforme apresentado na Figura 31. Neste exemplo, a *Rule Alarm 1* pode estar em execução em um computador distinto da *Rule Alarm 2* e, mesmo assim, sua respectiva *Condition* poderia manter o compartilhamento via rede da *Premise prAlarmOn*.

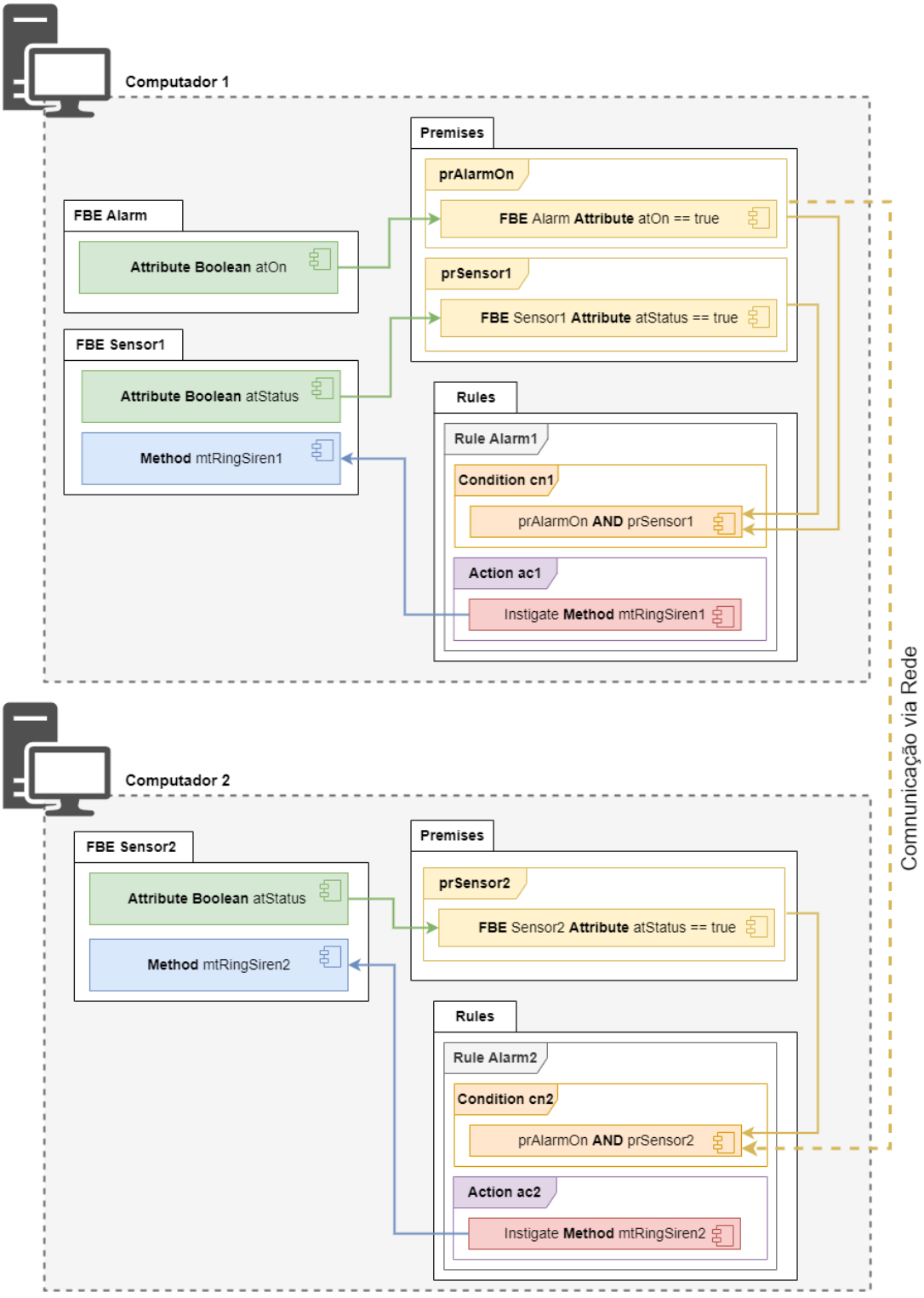


Figura 31 – Declaração de *Rules* utilizando *Premises* compartilhadas via rede.

No aspecto técnico da distribuição, observa-se que as entidades constituintes do PON quando distribuídas podem ser classificadas de acordo com quatro tipos básicos em relação ao envio e recebimento de notificações via rede. Estes quatro tipos básicos em si são propostos e apresentados no Quadro 5. De forma breve e sucinta, as Entidades Locais são as entidades não distribuídas (apresentadas na Subseção 3.1.1). Por sua vez, as Entidades Compartilhadas via Rede, apresentadas na Subseção 3.1.2, representam as entidades cujos estados são compar-

tilhados também com outros computadores por meio de notificações via rede (isto é, elas são "exportadas" da aplicação local para eventualmente serem utilizadas por outras aplicações). As entidades *Proxy* Remoto, apresentadas na Subseção 3.1.3, representam uma referência para uma Entidade Compartilhada via Rede de forma a permitir que computadores distintos utilizem ("importem") o estado dessa entidade por meio de notificações recebidas via rede. Por último, a Entidade Redundante via Rede, apresentada na Subseção 3.1.4, representa as entidades que enviam notificações e são notificadas de alterações que ocorram em outros computadores. Na Subseção 3.1.5 é apresentado um exemplo abordando a interação entre os diferentes tipos de entidades.

Quadro 5 – Tipos Básicos de Entidades quanto à distribuição.

	Alteração por processo local	Alteração por processo remoto	Envio de Notificações via Rede	Recebimento de Notificações via Rede
Entidade Local (*Local Entity*)	sim	não	não	não
Entidade Compartilhada via Rede (*Network shared entity*)	sim	não	sim	não
Entidade *Proxy* Remoto (*Remote proxy Entity*)	não	sim	não	sim
Entidade Redundante via Rede (*Network Redundant Entity*)	sim	sim	sim (exceto quando alterada por processo remoto)	sim

Com exceção dos *FBEs* e *Methods*, as demais entidades do PON (*Attributes, Premises, Conditions, Rules, Instigations* e *Actions*) podem ser compartilhadas nos vários tipos. Os *FBEs*, por consistirem em agregações de *Attributes* e *Methods* existem somente no escopo local e são classificadas, portanto, sempre como entidades locais. Eventualmente os *FBEs* podem agregar *Attributes* distribuídos (Compartilhado via Rede ou Redundante via Rede) e dessa forma os elementos agregados atuam em um ambiente distribuído, porém a agregação continuará ocorrendo no escopo local.

Sobre os *Methods*, por se tratar da execução de 'procedimentos' (de forma análoga, mas distinta, às funções do PI ou métodos do POO), sua execução e ação origina-se sempre na máquina local sendo, portanto, sempre entidades locais. Porém, caso necessário, os *Methods* podem invocar a alteração de *Attributes* distribuídos (nomeadamente, *Attributes* Compartilhados via Rede ou *Attributes* redundantes via Rede) ou até mesmo invocar diretamente a execução de procedimentos que enviem mensagens via rede. Dessa forma, os *Methods* podem atuar em um ambiente distribuído, porém sua execução continua ainda acontecendo no escopo local. Por definição, os *Methods* não podem alterar Entidades *Proxy* Remoto, pois estas somente podem ser alteradas via notificações recebidas via rede.

3.1.1 Entidade Local

A Entidade Local, ou não distribuída, representa uma entidade em PON que não possui referências ou dependências de aplicações em execução em outros computadores. Dessa forma, as alterações do estado da entidade não geram notificações via rede e notificações via rede não podem provocar alterações no estado da entidade. Como exemplo, considera-se um *Attribute* que é modificado exclusivamente por *Methods* locais e sua alteração é avaliada somente por *Premises* em execução na mesma aplicação.

3.1.2 Entidade Compartilhada via Rede

A Entidade Compartilhada via Rede representa uma entidade que, uma vez que seu estado seja alterado localmente, esta notificará seu novo valor para computadores remotos por meio de uma rede de comunicação. Em outras palavras, o estado desta entidade é “exportado” da aplicação local para ser eventualmente utilizado por aplicações remotas. Porém, destaca-se que notificações recebidas via rede não podem provocar alterações no estado desta entidade. Conforme apresentado na Figura 32, a alteração do estado de uma Entidade Compartilhada via Rede pode provocar a notificação de entidades locais e de uma ou mais Entidades *Proxy* Remoto, sendo que esta última ocorre por meio de uma rede de comunicação como, por exemplo, a Internet. O funcionamento das Entidades *Proxy* Remoto é posteriormente detalhado na Seção 3.1.3.

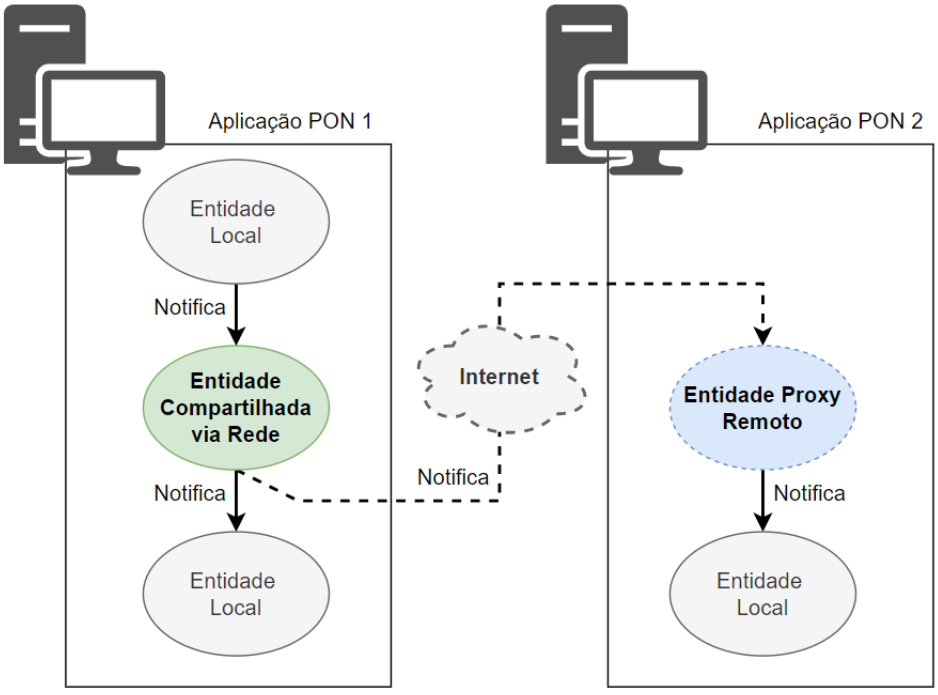


Figura 32 – Diagrama de interações entre entidades compartilhadas via rede e Entidades *Proxy* Remoto.

De um modo similar ao compartilhamento de entidades no escopo local, o objetivo ao se utilizar uma Entidade Compartilhada via Rede é evitar a redundância estrutural também em aplicações distribuídas. Por exemplo, conforme anteriormente apresentado na Figura 31, diferentes *Conditions* (*cn1* e *cn2*) poderiam utilizar a mesma *Premise* (*prAlarmOn*), mesmo que estas estejam em diferentes aplicações, desde que os computadores utilizados possuam um meio para comunicação de dados. Ainda nesse exemplo, a *Premise prAlarmOn* em execução no Computador 1 poderia ser classificada como *Premise* Compartilhada via Rede.

3.1.3 Entidade *Proxy* Remoto

A Entidade *Proxy* Remoto, por sua vez, permite a alteração do estado das entidades somente por uma aplicação em execução em um computador remoto, isto é, seu valor será alterado somente pelo recebimento de uma notificação via rede. Conforme apresentado previamente na Figura 32, uma Entidade *Proxy* Remoto terá seu estado modificado pelo recebimento de uma notificação via rede, a qual foi gerada pela alteração do estado da respectiva Entidade Compartilhada via Rede existente em um computador remoto. Em outras palavras, a Entidade *Proxy* Remoto é uma referência utilizada pela aplicação local para a “importação” do estado de uma Entidade Compartilhada via Rede existente em uma aplicação remota. Dessa forma, nenhuma, uma ou mais Entidades *Proxy* Remoto podem ser associadas a uma mesma Entidade Compartilhada via Rede.

Em termos conceituais, a implementação das Entidades *Proxy* Remoto consiste apenas em um *placeholder* para uso de valores remotos. Neste contexto, a Entidade *Proxy* Remoto é necessária apenas em aspectos técnicos representando uma conexão ou um mapeamento entre o identificador da entidade (comum entre todos os computadores) e sua localização lógica dentro do programa (por exemplo, um ponteiro em C/C++), sem acarretar redundância estrutural ou em processamento adicional.

Conforme exemplificado na Figura 33, por meio desse mecanismo de compartilhamento, uma *Premise* (*Pr_2*) em uma aplicação em determinado computador (Computador 2) poderá se basear em um *Attribute* (*Shared_Attr_1*) em execução em um computador distinto (Computador 1), tendo seu estado reavaliado a cada mudança nesse *Attribute*. Nomeadamente, no Computador 1 o *Attribute* em questão é declarado como *Attribute* compartilhado via Rede (*Network Shared Attribute*) e no Computador 2 esse mesmo *Attribute* é declarado como *Attribute Proxy* Remoto (*Remote Proxy Attribute*).

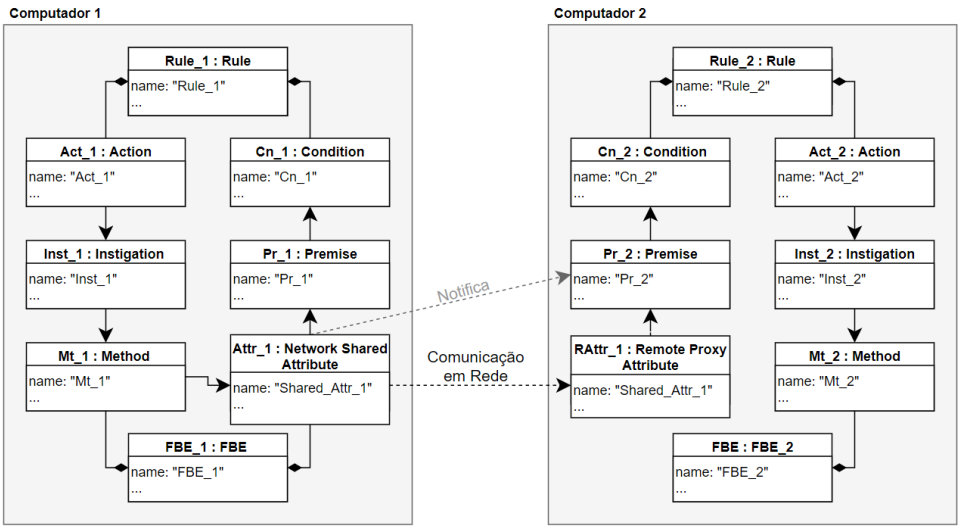


Figura 33 – Diagrama de instâncias de uma aplicação PON com *Attribute Proxy* Remoto e *Attribute* Compartilhado via Rede.

Ademais, em complemento ao exemplo apresentado anteriormente na Figura 31 no qual as diferentes *Conditions* (*cn1* e *cn2*) em diferentes computadores utilizam a mesma *Premise* (*prAlarmOn*), nomeadamente tem-se a *Premise prAlarmOn* em execução no Computador 2 como *Premise Proxy* Remoto.

3.1.4 Entidade Redundante via Rede

A Entidade Redundante via Rede consiste na sincronização de estados das entidades entre todos os computadores pertinentes, toda vez que uma entidade tiver seu estado alterado local ou remotamente. Conforme apresentado na Figura 34, uma Entidade Redundante via Rede pode ser alterada por entidades locais em execução na aplicação do computador local ou por uma aplicação em execução em um computador remoto.

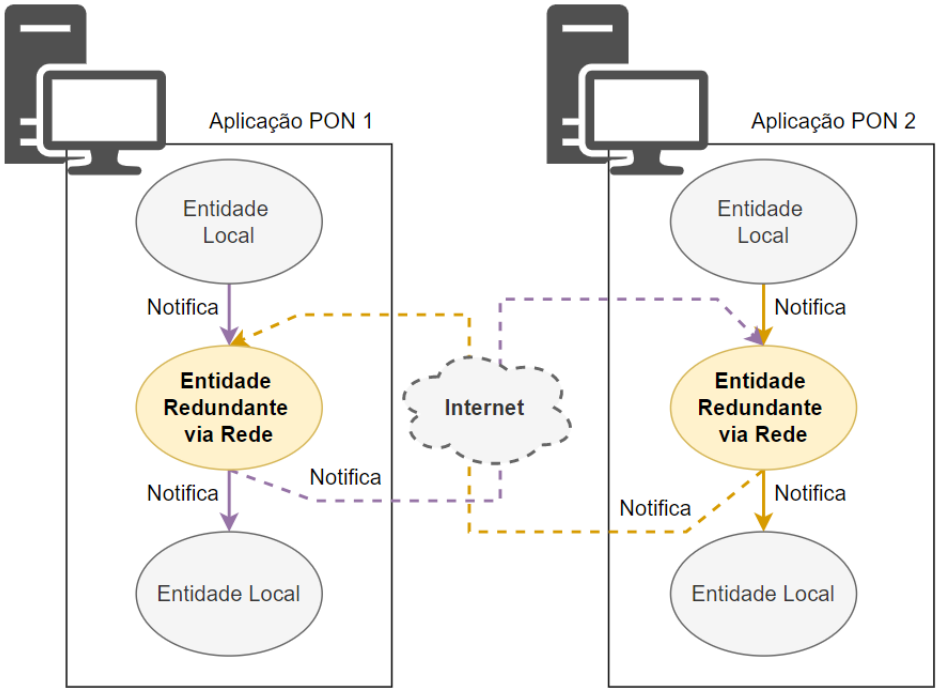


Figura 34 – Diagrama de interações para Entidades Redundantes via Rede.

Por exemplo, um *Attribute* redundante via rede terá seu valor enviado pela rede de comunicação sempre que seu valor for modificado e também terá seu valor alterado toda vez que receber uma notificação de alteração em computadores remotos.

Conforme apresentado na Figura 35, com exceção dos *FBEs* e *Methods*, as demais entidades do PON podem ser redundantes. É importante observar que quando alteradas via rede, as entidades redundantes não devem notificar a alteração de estado novamente via rede, pois isso geraria um ciclo desnecessário de notificações.

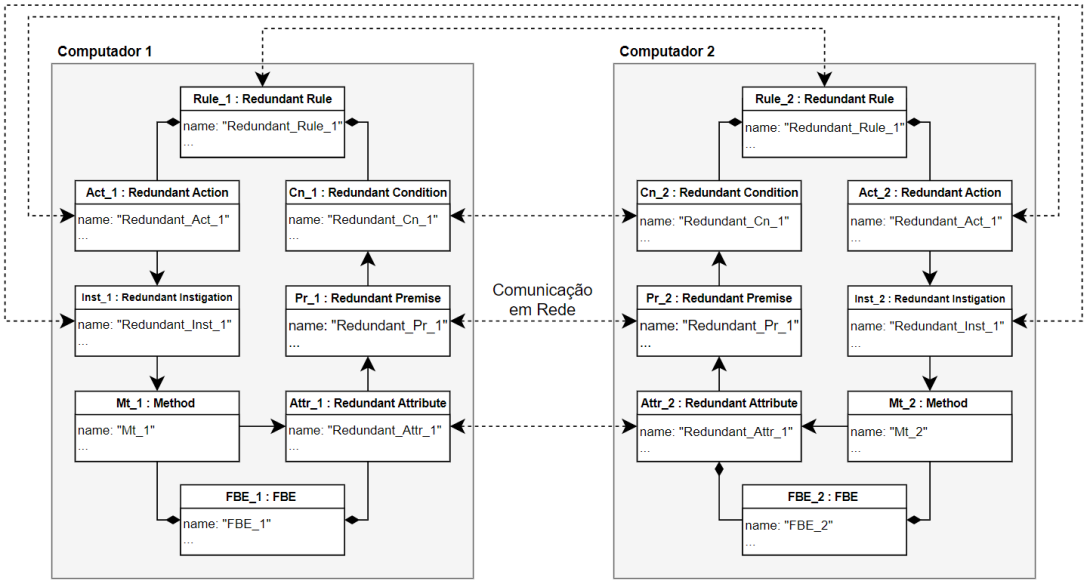


Figura 35 – Diagrama de instâncias de uma aplicação PON com entidades redundantes.

3.1.5 Exemplo de uso das entidades distribuídas do PON

Como exemplos dos tipos de entidades apresentados, considera-se um sistema de uma casa inteligente no qual múltiplos sistemas utilizam e eventualmente compartilham o estado ou avaliações lógicas sobre um conjunto de sensores e/ou atuadores. Algumas partes desse sistema são apresentadas aqui como um exemplo de compartilhamento. Um experimento neste mesmo contexto é apresentado com detalhes posteriormente na Seção 4.3.

Nesse contexto, o estado de um sensor de presença, representado por um *Attribute* Compartilhado via Rede pode ser avaliado, por exemplo, por um sistema responsável pela iluminação do ambiente e por um sistema responsável pela climatização. Dessa forma, ambos podem utilizar um *Attribute Proxy* Remoto representando o respectivo *Attribute* Compartilhado via Rede permitindo que suas respectivas *Premises* locais sejam notificadas quando o estado do sensor for alterado. Ou ainda, é possível que determinada avaliação relacional sobre o estado do sensor (*Premise*) seja compartilhada entre os sistemas, adicionando-se um nível de abstração sobre os *Attributes*. Por exemplo, uma *Premise* que indique a presença no ambiente pode ser avaliada por um único sistema (*Premise* Compartilhada via Rede) e utilizada pelos demais sistemas, utilizando-a como *Premise Proxy* Remoto. O mesmo conceito de abstrações pode ser estendido para as demais entidades do PON. Por exemplo, uma *Condition* que avalie, via *Premises* (locais ou distribuídas), a detecção de pessoas em múltiplos ambientes. Essa avaliação pode ser realizada uma única vez e compartilhada entre múltiplas partes.

De forma similar, múltiplos sistemas podem compartilhar o mesmo conjunto de *Instigations* ou a mesma *Action*. Por exemplo, um sistema de climatização que, em determinadas condições, executa uma *Action* referente ao fechamento de um ambiente, notificando uma *Instigation* responsável por alterar o estado de uma porta e uma *Instigation* responsável por alterar o estado de uma janela. Essa mesma *Action* pode ser utilizada, por exemplo, por um sistema de segurança para o fechamento completo do ambiente em determinado horário. Pode-se também compartilhar apenas uma determinada *Instigation* como, por exemplo, a responsável pela alteração do estado da porta, que pode ser compartilhada e utilizada por um sistema de proteção contra incêndios.

Para a efetiva distribuição das entidades constituintes do PON, faz-se necessário a troca de mensagens entre os computadores para o envio e o recebimento de notificações via rede. Para que isto ocorra, todas as aplicações envolvidas devem seguir um protocolo comum de comunicação com regras e padrões predefinidos. O protocolo de distribuição proposto no presente trabalho é discutido em detalhes na Seção 3.2.

3.2 Distribuição do PON via protocolo MQTT

Conforme apresentado no Capítulo 2, um dos pontos que deve ser levado em consideração para a construção de sistemas em IoT é a interoperabilidade entre sistemas heterogê-

neos os quais são geralmente construídos utilizando paradigmas de programação dominantes como, por exemplo, o POO e o POE. Para a interoperabilidade das aplicações desenvolvidas em PON com aplicações desenvolvidas em outros paradigmas ('não-PON'), como por exemplo uma Aplicação POO, propõe-se a distribuição do PON por meio de um protocolo padronizado pela OASIS, o MQTT (OASIS STANDARD, 2014). Conforme apresentado na Seção 2.1.1.1, o MQTT implementa o padrão *publish/subscribe*, desacoplando o cliente produtor do(s) cliente(s) consumidor(es) por meio de uma estrutura de tópicos inserida em um elemento intermediário conhecido como *broker*. Quando o cliente produtor publica uma mensagem em determinado tópico, esta mensagem é repassada para os clientes consumidores que previamente tenham registrado interesse no respectivo tópico. O *broker* é responsável pela criação e gerenciamento de tópicos os quais permitem a identificação e filtragem de mensagens de acordo com um determinado domínio (OASIS STANDARD, 2014).

3.2.1 Nomenclatura dos Tópicos

Para permitir a distribuição das entidades do PON via MQTT, optou-se por realizar a correspondência direta entre cada entidade do PON e os tópicos MQTT. Para isso, utiliza-se um padrão para a nomenclatura dos tópicos, nomeadamente ***<domínio>/<tipo de entidade PON>/<identificação única do elemento>/.*** Desta forma, o conteúdo da mensagem publicada representa o valor assumido pelo elemento em questão. Neste padrão são considerados três níveis de separação para a identificação do tópico:

- **Domínio**: identificação do contexto ou domínio da aplicação. Considerando que um mesmo *broker* pode ser utilizado para troca de mensagens entre múltiplas aplicações, a separação por domínio restringe o escopo de identificação única dos elementos para o domínio conhecido, evitando conflitos por duplicidades. Por exemplo, no caso do sistema de alarme apresentado anteriormente na Figura 31, o domínio utilizado neste contexto pode ser o mesmo para toda uma casa ou então um domínio por seção de uma casa (quarto, sala, cozinha etc.), dependendo da quantidade e do padrão de nomenclatura adotado para cada entidade desse domínio.

- **Tipo de entidade**: identifica o tipo de entidade em questão. Essa separação pode ser utilizada para a realização de inscrições com filtros por tipo de entidade ou para a otimização do escopo de busca dos elementos nas aplicações consumidoras (*subscribers*), restringindo-se o escopo de buscas ao domínio e tipo de elemento em questão. Ademais, para otimização do uso de rede, adotou-se a identificação do tipo de entidade por dois caracteres: at, pr, cn, rl, in e ac para *Attributes, Premises, Conditions, Rules, Instigations* e *Actions*, respectivamente. Como exemplo, no caso da *Premise prAlarmOn* apresentada anteriormente na Figura 31, o respectivo tipo de entidade seria identificado como "pr".

- **Identificação única do elemento**: é a identificação comum do elemento em todas as aplicações que estejam utilizando o mesmo domínio. Por exemplo, ainda considerando-se a aplicação da Figura 31, a identificação da *Premise prAlarmOn* no sistema poderia ser descrita como *prAlarmOn*.

As aplicações PON que utilizarem entidades que recebam notificações via rede (nomeadamente, Entidade *Proxy* Remoto ou Entidade Redundante via Rede) devem se inscrever nos respectivos tópicos do *broker* específicos para o elemento em questão (segundo o padrão de nomenclatura definido). Dessa forma, a aplicação será notificada somente pelas entidades de interesse, evitando processamento ou avaliações desnecessárias.

Como exemplo, considera-se o *Attribute* distribuído (*proxy* ou redundante) *atState* do *FBE Sensor* apresentado na Figura 36. Neste exemplo, a aplicação PON se inscreve no tópico pertinente ao *Attribute atState*, nomeadamente *Sensor/at/atState/*, recebendo uma mensagem de confirmação (*ack*) do *broker* caso a operação seja processada com sucesso. Após a inscrição, tão logo seja recebida uma publicação no respectivo tópico a aplicação PON é notificada e recebe uma nova mensagem para processamento. Essa nova mensagem recebida provoca a alteração do estado do respectivo *Attribute* provocando notificações para a(s) *Premise(s)* pertinente(s). Neste exemplo, a *Premise prSensor* é notificada e pode dar sequência ao fluxo de notificações, se aprovada.

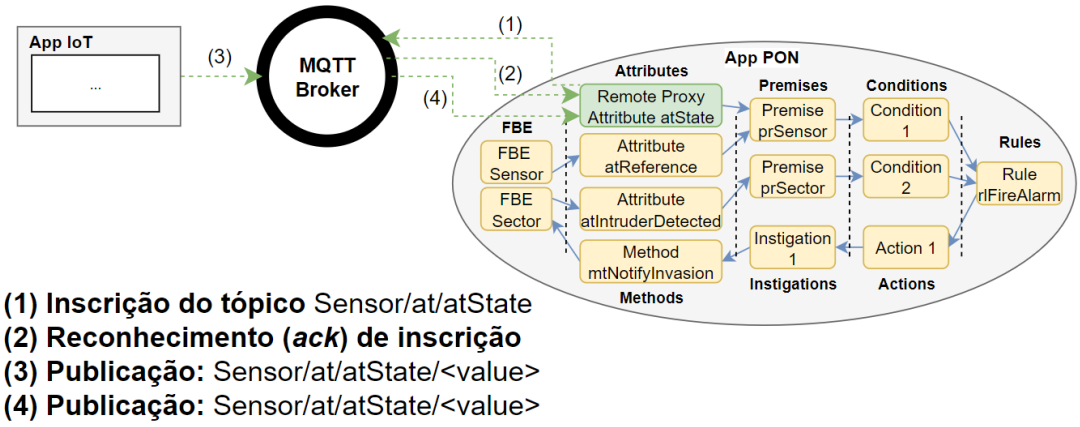


Figura 36 – *Attribute* distribuído por meio do protocolo MQTT.
Fonte: Adaptado de Figueiredo, SIMÃO e Vendramin (2022)

Para as Entidades Compartilhadas via Rede, não é necessária a inscrição em tópicos pois essas entidades não podem ter seu estado alterado por notificações recebidas via rede. Porém, ao publicar mensagens de notificações de alterações de seu estado, deve-se utilizar o mesmo padrão definido para a nomenclatura de tópicos. Ademais, recomenda-se que a conexão do cliente da aplicação com o *broker* seja feita uma única vez e seja mantida enquanto estiver em execução, evitando-se o *overhead* de uma nova conexão a cada publicação.

3.2.2 Reflexões sobre os níveis de QoS do MQTT

Conforme apresentado na Seção 2.1.1.1, o MQTT define três níveis para o QoS das mensagens. O QoS 0 não garante a entrega de notificações, o QoS 1 garante a entrega de notificações podendo haver duplicidades e o QoS 2 garante a entrega das notificações uma única vez aos consumidores. Destaca-se que o nível de QoS deve ser definido conforme as necessidades de cada aplicação e que cada nível utiliza uma quantidade diferente de mensagens trafegadas na rede (1, 2 e 4 para o QoS 0, 1 e 2, respectivamente).

De um modo geral, o uso do QoS 1 pode ser indicado para *Attributes*, *Premises* e *Conditions* pois, para esses elementos e considerando-se que o PON evita o envio de notificações redundantes, a perda de mensagens poderia levar ao funcionamento incorreto da aplicação, porém o recebimento de notificações em duplicidade não geraria impactos no seu estado lógico.

Por outro lado, para *Rules*, *Actions* e *Instigations*, o uso do QoS 2 é o mais recomendado pois, para esses elementos, a perda de mensagens poderia levar a um comportamento inesperado da aplicação e eventuais notificações recebidas em duplicidade provocariam a reexecução deles, podendo gerar inconsistências na aplicação.

3.2.3 Reflexões sobre o uso de *Wildcards*

Conforme apresentado na Seção 2.1.1.1, o MQTT apresenta também o conceito de *wildcards* os quais permitem a seleção de um conjunto de tópicos em uma única operação de inscrição no *broker*. No contexto de distribuição das entidades do PON, o uso dos *wildcards* pode ser benéfico somente em situações específicas. A vantagem na utilização de *wildcards* de inscrição multinível ou de nível único acontece no momento da inicialização da aplicação, reduzindo o número de operações de inscrições necessárias. Por exemplo, quando usado o *wildcard* multinível *#*, pode-se realizar a inscrição em todos os tópicos do *broker* com uma única operação. Pode-se também utilizar uma inscrição global para um determinado domínio desejado. No contexto do PON, utilizando-se como exemplo o sistema apresentado previamente na Figura 36, a inscrição da aplicação poderia ocorrer em *#* englobando todas as entidades presentes nos diversos domínios existentes no *broker* ou em *Sensor/#*, englobando todos as entidades existentes no domínio *Sensor*.

Porém, destaca-se que caso existam tópicos no *broker* referentes a entidades distribuídas que não são utilizadas no contexto da respectiva aplicação, as notificações referentes a essas entidades serão também enviadas para a aplicação, acarretando processamento desnecessário. O uso da inscrição por domínio pode reduzir esse problema, mas não o elimina.

Para exemplificar, apresenta-se o sistema da Figura 37 o qual complementa o sistema apresentado previamente na Figura 36. Neste exemplo, considera-se a existência de um segundo sensor referente a uma segunda aplicação (‘App2 IoT’ e ‘App2 PON’), que implementa um *Attribute* Compartilhado via Rede (por exemplo, *Sensor/at/atState2*) e uma *Premise* Com-

partilhada via Rede (*Sensor/pr/prAlarm*). Considera-se que ambas as aplicações PON (‘App PON’ e ‘App2 PON’) e POO (‘App IoT’ e ‘App2 IoT’) utilizem o mesmo *broker* para a troca de mensagens. Neste caso, ambas as mensagens referentes ao segundo sensor (*atState2* e *prAlarm*) seriam recebidas e processadas, sem efeito, pela aplicação ‘App PON’ caso esta utilizasse uma inscrição multinível # ou Sensor/#.

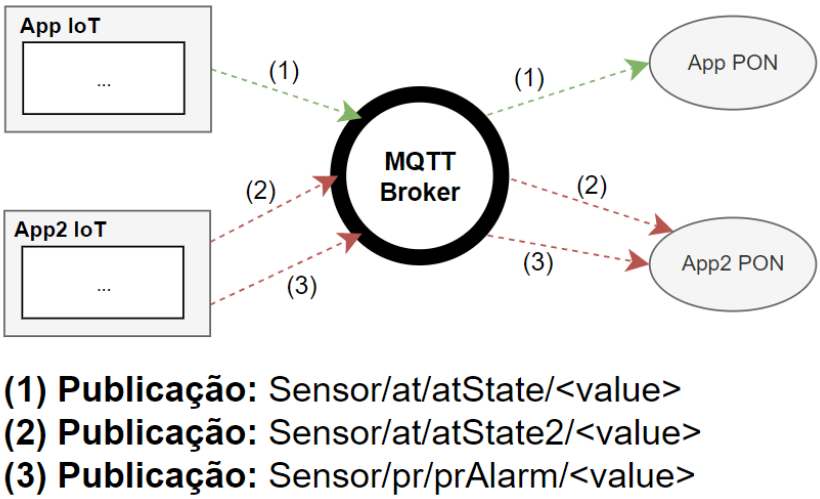


Figura 37 – Compartilhamento de um mesmo *broker* por diferentes aplicações.

No caso de *wildcards* de nível único, o número de operações mínimas dependerá do nível selecionado e da quantidade e/ou variedade dos tipos de entidades *Proxy* Remoto ou Entidades Redundantes via Rede presentes na aplicação. No exemplo apresentado na Figura 36, a inscrição poderia ocorrer em *Sensor/at/+*, englobando todos os *Attributes* existentes no domínio Sensor.

Observa-se também neste caso o mesmo problema apresentado para a inscrição multinível. Caso existam tópicos no *broker* referentes a entidades distribuídas que não são utilizadas no contexto de uma determinada aplicação, as notificações referentes a essas entidades serão também enviadas para esta aplicação, acarretando processamento desnecessário. Por exemplo, considerando-se o cenário apresentado na Figura 37, a inscrição da aplicação ‘App PON’ utilizando-se, por exemplo, do *wildcard* de nível único *Sensor/at/+*, englobaria também o *Attribute atState2* não pertinente ao seu contexto. Com isso, as notificações referentes ao *atState2* seriam recebidas e processadas, sem efeito, pela aplicação ‘App PON’.

Por outro lado, para as inscrições individuais (sem o uso de *wildcards*) é necessário, no mínimo, uma operação de inscrição no *broker* para cada entidade *Proxy* Remoto e uma para cada Entidade Redundante via Rede presentes na aplicação. Destaca-se que, dessa forma, todas as mensagens recebidas pela aplicação serão, de fato, pertinentes a ela. Neste caso, mesmo para o cenário previamente apresentado na Figura 37, a inscrição individual garante que somente as mensagens pertinentes à aplicação serão recebidas, isto é, ‘App PON’ receberia notificações somente de *atState*.

No caso dos *wildcards* de inscrições compartilhadas (*$share/Nome Compartilhado/*), o uso no contexto de aplicações do PON utilizando o padrão de tópicos previamente definido pode levar ao funcionamento incorreto, pois pode gerar estados inconsistentes em diferentes aplicações que deveriam se comportar de maneira similar. Destaca-se que somente nas situações em que a aplicação se comportar de modo *stateless*, isto é, nenhuma referência ou informação sobre eventos ou notificações antigas impactam o processamento atual, o uso de *wildcards* de inscrições compartilhadas pode ser adequado e trazer benefícios.

Como exemplo de possível uso dos *wildcards* de inscrições compartilhadas, pode-se citar uma aplicação na qual múltiplos sistemas são responsáveis exclusivamente por alterações de estados de Entidades Redundantes via Rede ou de chamadas para sistemas externos. Nesse caso, independente de qual aplicação seja acionada, o estado final será sempre o mesmo, pois a alteração de uma Entidade Redundante via Rede será sincronizada com as demais aplicações ou a ação sobre o sistema externo será executada de maneira similar. Nesse caso, seria possível ao sistema escalar horizontalmente, caso necessário, adicionando-se novas aplicações à inscrição compartilhada conforme o número de notificações aumente.

No caso geral, recomenda-se o uso de inscrições individuais pois, apesar de um uso maior de mensagens na inicialização, as notificações posteriormente recebidas serão sempre pertinentes ao contexto da aplicação. Ademais, utilizando-se o nível QoS adequado, o estado tende a ser sempre consistente em todas as aplicações.

3.2.4 Conjunto de Ferramentas para uso do MQTT

Conforme apresentado na Seção 2.1.1.1, o MQTT é um protocolo padronizado pela OASIS. Com isso, é possível encontrar implementações *Open Source* de *brokers* e bibliotecas de comunicação para uso em múltiplas linguagens (OASIS STANDARD, 2014). Dentre as implementações disponíveis, encontra-se o Mosquitto (LIGHT, 2017), parte do projeto *Eclipse* IoT mantido pela *Eclipse Foundation*. O Mosquitto se destaca por ser relativamente leve e adequado para uso em diversos tipos de dispositivos, desde dispositivos com poucos recursos até servidores completos.

No pacote do Mosquitto encontra-se, além de um *broker* de mensagens, também um conjunto de bibliotecas em C e C++ disponíveis para comunicação de aplicações com o *broker*. Essas bibliotecas foram utilizadas no contexto da conformação do *Framework* PON C++ 4.0 IoT, apresentado na Seção 3.3.

Além das próprias bibliotecas, o *broker* Mosquitto possui compatibilidade, dentre outras, com a biblioteca *Eclipse Paho* (ECLIPSE FOUNDATION, 2022) a qual também faz parte do projeto *Eclipse* IoT. A biblioteca *Eclipse Paho* disponibiliza uma interface para comunicação MQTT em diversas linguagens, incluindo C++, Java, Python (ECLIPSE FOUNDATION, 2022). Essa biblioteca em Python foi utilizada para execução de testes de integração de sistemas, conforme apresentado na Seção 3.3.3.2.

Além do conjunto de ferramentas para desenvolvimento das aplicações com MQTT, utilizou-se também softwares auxiliares para análise das mensagens e dos resultados produzidos durante o desenvolvimento e os experimentos. Nomeadamente, utilizou-se o software Wireshark e o MQTT Explorer (NORDQUIST, 2020).

3.3 Distribuição do PON via MQTT com o *Framework* PON C++ 4.0 IoT

Para conformar o *Framework* PON C++ 4.0 e permitir a implementação de aplicações distribuídas no PON, por meio do protocolo MQTT, adicionou-se três classes à sua estrutura original, constituindo-se assim o *Framework* PON C++ 4.0 IoT. Em complemento ao Diagrama de Classes UML apresentado previamente na Figura 27, as classes adicionadas para distribuição são apresentadas na Figura 38. Essas novas classes são responsáveis por tratar da inscrição dos tópicos pertinentes e por enviar e/ou receber notificações ao/do *broker* MQTT. [1]

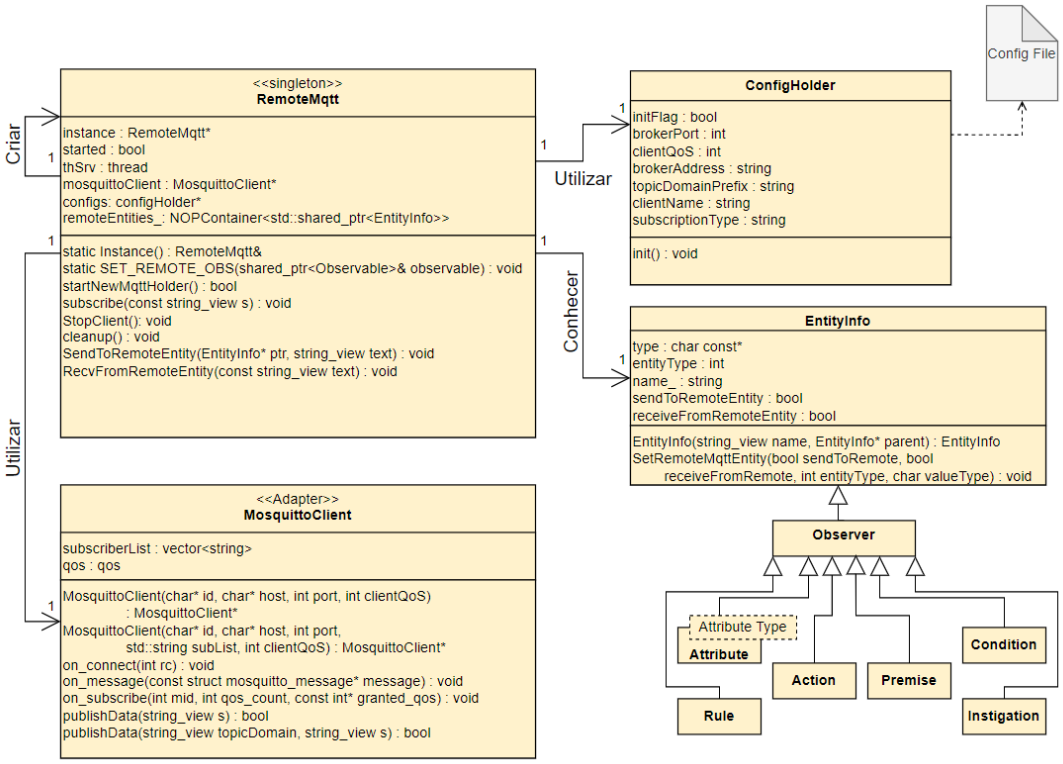


Figura 38 – Diagrama de classes em UML com detalhamento das classes adicionadas para distribuição.

Além das três classes adicionadas, alterou-se também a classe *EntityInfo*, previamente chamada por Neves (2021) de *DebugEntity* no contexto do *Logger*. Além do já existente identificador do elemento (*name*), adicionou-se nessa classe dois booleanos referentes à característica

[1] O código completo do *Framework* PON C++ 4.0 IoT e dos testes realizados encontra-se no servidor de artefatos do PON acessível através de login e senha no endereço <*https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4/nop-framework-cpp-4-iot*>. O acesso à quem de direito seria provido pelo Prof. Jean Marcelo Simão via contato por e-mail.

de envio (*sendToRemoteEntity*) e recebimento de notificações via rede (*receiveFromRemoteEntity*), além dos identificadores de tipo, representando o tipo de entidade PON (*entityType*) e o tipo de dados (*type*).

A classe *RemoteMQTT* é responsável pelo envio e recebimento das mensagens e foi implementada segundo o padrão *Singleton*. O padrão *Singleton* especifica que apenas uma instância da classe pode existir, e esta será utilizada por toda a aplicação. Dessa forma, na primeira referência, isto é, na primeira necessidade de objeto desta classe, um novo processo (*thread*) para comunicação MQTT é iniciado e uma instância é iniciada (*i.e.*, alocada dinamicamente) via ponteiro *RemoteMQTT*, a qual é retornada nas referências subsequentes.

Conforme apresentado na Figura 39, no contexto do *Framework* PON C++ 4.0 IoT, a referência para a classe *RemoteMQTT* acontece na declaração de uma Entidade Distribuída, dos tipos Redundante via Rede ou *Proxy* Remoto. Neste momento ocorre a inicialização da instância da classe *RemoteMQTT*, caso esta ainda não tenha sido inicializada. Durante a inicialização, são carregadas as configurações do respectivo arquivo (por meio de uma instância da classe *ConfigHolder*, conforme detalhado na Seção 3.3.1) e iniciada a conexão com o *broker* de mensagens. Caso seja especificado o tipo de inscrição global no arquivo de configuração, a inscrição no tópico global (#) é realizada no momento da inicialização. Após a inicialização, a respectiva entidade distribuída é adicionada na lista interna da classe *RemoteMQTT*. Caso não seja especificado o tipo de inscrição global no arquivo de configuração utiliza-se a opção padrão (ou seja, inscrição individual), é solicitada a inscrição da respectiva entidade no respectivo tópico do *broker*, conforme padrão de nomes apresentado na Seção 3.2.1.

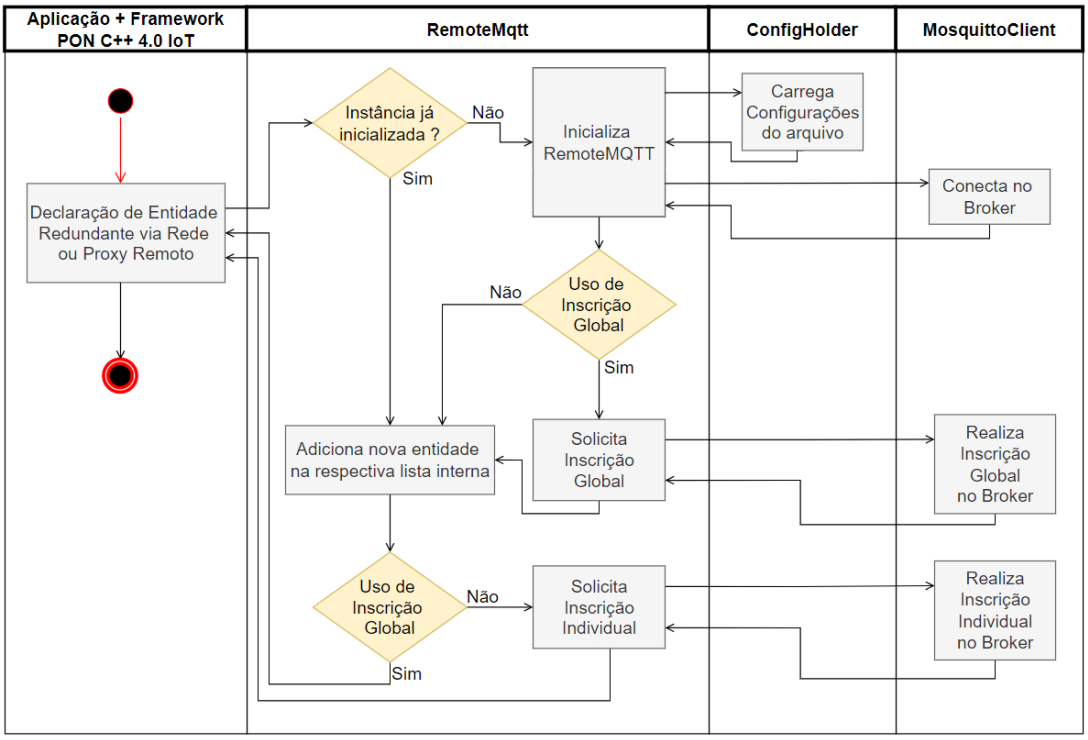


Figura 39 – Diagrama de atividades com detalhamento para a inicialização.

Observa-se que toda a comunicação da aplicação com o *broker* (por meio da instância da classe *RemoteMQTT*) é feita através de uma instância da classe *MosquittoClient*, que implementa um adaptador para as chamadas de métodos da biblioteca de comunicação *Mosquitto* (LIGHT, 2017), incluindo as rotinas de conexão, inscrição, publicação e recebimento de mensagens do *broker*.

O envio de notificações via rede nas entidades ou instância de base do PON é realizado pelo método *SendToRemoteEntity* oriundo da classe base *RemoteMqtt*. Conforme apresentado na Figura 40, o fluxo de notificações para a rede é invocado no fluxo de notificações do *Framework* PON C++ 4.0 IoT, quando ocorre uma mudança de estado de uma entidade distribuída, nomeadamente dos tipos Compartilhada ou Redundante via Rede. Durante o fluxo de envio da notificação, as informações referentes ao tipo e identificação da entidade (atributos da classe *EntityInfo*) são utilizados para a criação do nome do tópico, conforme padrão de nomes previamente apresentado na Seção 3.2.1, e da conversão do valor da entidade em uma mensagem.

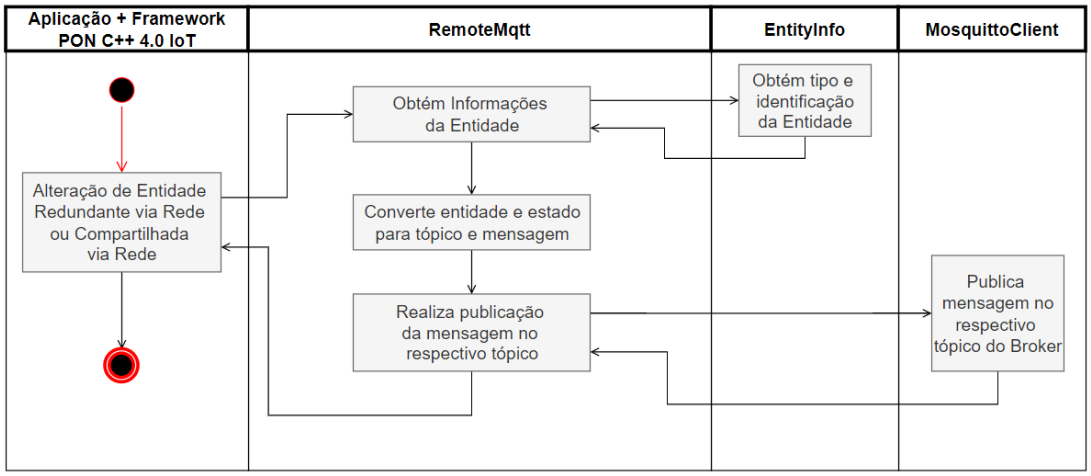


Figura 40 – Diagrama de atividades com detalhamento para a alteração das entidades e consequente envio de notificações.

Por sua vez, conforme apresentado na Figura 41, as notificações para a aplicação são recebidas pela instância de *MosquittoClient*, o qual invoca, quando notificado, o método *RecvFromRemoteEntity* da classe *RemoteMQTT*. Esse método é responsável por mapear a identificação da entidade recebida com o respectivo ponteiro na aplicação, se existente. Ao identificar a entidade, o seu valor é atualizado conforme a mensagem recebida e o fluxo de notificações do PON segue conforme previamente declarado.

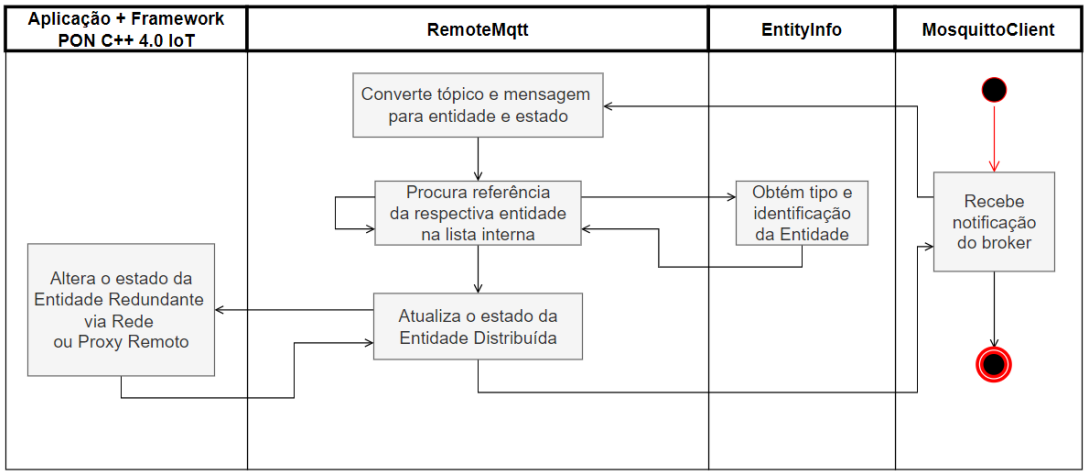


Figura 41 – Diagrama de atividades com detalhamento para o recebimento de notificações via rede e consequente alteração das respectivas entidades.

3.3.1 Interface de configuração do *Framework* PON C++ 4.0 IoT

A configuração do *Framework* PON C++ 4.0 IoT ocorre por meio de um arquivo de configuração, nomeado config.ini, localizado no diretório de execução da aplicação. Esse arquivo é avaliado no ciclo de inicialização (método *init*) de uma instância da classe *ConfigHolder* (apresentada na Figura 38). Durante a inicialização (previamente apresentado na Figura 39), os valores do arquivo de configuração config.ini são interpretados, e os valores são armazenados nos respectivos atributos da instância de *ConfigHolder* (*brokerPort*, *clientQoS*, *brokerAddress*, *topicDomainPrefix*, *clientName* e *SubscryptionType*). No arquivo config.ini, a seção *[MQTT Config]* descreve:

- *brokerAddress*: atributo obrigatório que descreve o **endereço IP** do *broker* MQTT.
- *brokerPort*: atributo obrigatório que descreve a **porta** na qual está o *broker* MQTT.
- *topicDomainPrefix*: atributo obrigatório que descreve o **domínio** da aplicação em questão. Relembra-se que esse valor será utilizado na construção dos nomes dos tópicos, conforme apresentado na Seção 3.2.1. Considera-se válido qualquer valor textual com exceção dos caracteres reservados para o MQTT e dos *wildcards* (/, # e +).
- *clientQoS*: atributo opcional que descreve o **nível de QoS** a ser utilizado durante a aplicação. Consideram-se válidos os valores 0, 1 ou 2. Caso não seja especificado, utiliza-se o QoS de nível 2.
- *clientName*: atributo opcional que representa a **identificação do cliente** nas operações no *broker*. Caso não seja especificado, utiliza-se um identificador aleatório.

- *SubscryptionType*: atributo opcional que descreve o **tipo de inscrição** a ser utilizado. Caso não seja especificado, utiliza-se a inscrição individual. Consideram-se válidos os valores GLOBAL ou INDIVIDUAL.

A Figura 42 exemplifica um arquivo de configuração utilizando os valores padrão, a saber: o *broker* em *localhost* (“127.0.0.1”), a porta 1833 (porta padrão da comunicação MQTT) e o prefixo dos tópicos “test”.

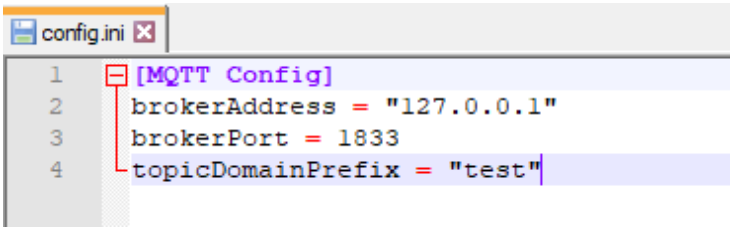


Figura 42 – Exemplo de arquivo de configuração do *Framework* PON C++ 4.0 IoT.

3.3.2 Interface de uso do *Framework* PON C++ 4.0 IoT

Para o usuário do *Framework* PON C++ 4.0 IoT a diferenciação entre uma entidade local e uma entidade que se comunique via rede ocorre somente no momento da declaração do elemento. Para a construção das entidades distribuídas foram adicionados novos construtores (*builders*), além dos previamente implementados por Neves (2021). As subseções subsequentes apresentam os construtores implementados para cada entidade do PON. No geral, para os novos construtores, além do valor da entidade (também presente na inicialização de uma entidade local), faz-se necessário também que seja informado um novo parâmetro obrigatório do tipo *string* que corresponde ao nome ou à identificação da entidade. Essa identificação deve ser única no contexto do domínio do sistema, pois será utilizada como referência de mapeamento para todas as aplicações em execução nos diferentes computadores (conforme previamente apresentado na Seção 3.2).

3.3.2.1 *Attributes*

O Código 3.1 mostra um exemplo de inicialização dos *Attributes* em seus diferentes modos de distribuição. Observa-se que o retorno da função se mantém sempre o mesmo, porém nas declarações dos *Attribute* distribuídos é necessário também informar um parâmetro adicional. Mais precisamente, além do valor inicial do *Attribute* em si (também presente em implementações não distribuídas), é necessário que seja informado o identificador referente ao *Attribute* em questão. Esse identificador deve ser único em todas as aplicações no domínio do sistema pois será usado para comunicação, conforme descrito na Seção 3.2.

Código 3.1 Declaração de *Attributes* e seus possíveis tipos

```
1   /* Declaração de um Attribute local
2   * (Local Attribute)
3   * com valor inicial "-1" e identificador "at1" */
4   NOP::SharedAttribute<int> at1 =
5       NOP::BuildAttribute(-1);
6
7   /* Declaração de um Attribute Compartilhado via Rede
8   * (Network Shared Attribute)
9   * com valor inicial "-1" e identificador "at1" */
10  NOP::SharedAttribute<int> at11 =
11      NOP::BuildNetworkSharedAttribute("at1", -1);
12
13  /* Declaração de um Attribute Proxy Remoto
14  * (Remote Proxy Attribute)
15  * com valor inicial "-1" e identificador "at1" */
16  NOP::SharedAttribute<int> at11 =
17      NOP::BuildRemoteProxyAttribute("at1", -1);
18
19  /* Declaração de um Attribute Redundante via Rede
20  * (Network Redundant Attribute)
21  * com valor inicial "-1" e identificador "at1" */
22  NOP::SharedAttribute<int> at11 =
23      NOP::BuildNetworkRedundantAttribute("at1", -1);
```

3.3.2.2 *Premises*

O Código 3.2 apresenta um exemplo de inicialização das *Premises* em seus diferentes tipos de distribuição. Nota-se que o retorno da função se mantém sempre o mesmo. Na declaração das *Premises* Compartilhadas via rede e Redundante via Rede, além da referência para o(s) *Attribute*(s) e da operação relacional (*i.e*, maior, menor, igual) também presentes na inicialização de uma *Premise* não distribuída, é necessário um parâmetro referente ao identificador único da *Premise*. Por sua vez, para a construção de uma *Premise Proxy* Remoto é necessário apenas o identificador único da *Premise*, visto que o processamento relacional é realizado em outra aplicação, por uma *Premise* Compartilhada via Rede.

Código 3.2 Declaração de *Premises* e seus possíveis tipos

```
1     /* Declaração de uma Premise local
2     * (Local Premise)
3     * at1 e at2 são referências para os Attributes
4     * NOP::Equal representa a operação relacional */
5     NOP::SharedPremise pr1 = NOP::BuildPremise(at1, at2, NOP::Equal());
6
7     /* Declaração de uma Premise Compartilhada via Rede
8     *  (Network Shared Premise)
9     * "pr1" representa o identificador única dessa Premise
10    * at1 e at2 são referências para os Attributes
11    * NOP::Equal representa a operação relacional */
12    NOP::SharedPremise pr1 =
13        NOP::BuildNetworkSharedPremise("pr1", at1, at2, NOP::Equal());
14
15    /* Declaração de uma Premise Proxy Remoto
16    * (Remote Proxy Premise)
17    * "pr1" representa o identificador única dessa Premise */
18    NOP::SharedPremise pr2 =
19        NOP::BuildRemoteProxyPremise("pr1");
20
21    /* Declaração de uma Premise redundante
22    * (Network Redundant Premise)
23    * "pr1" representa o identificador única dessa Premise
24    * at1 e at2 são referências para os Attributes
25    * NOP::Equal representa a operação relacional */
26    NOP::SharedPremise pr1 =
27        NOP::BuildNetworkRedundantPremise("pr1", at1, at2, NOP::Equal());
```

3.3.2.3 *Conditions*

O Código 3.3 apresenta um exemplo de construção das *Conditions* em seus diferentes tipos. De forma similar às *Premises*, a construção das *Conditions* Compartilhadas via Rede e das *Conditions* Redundantes via Rede descrevem, além dos parâmetros necessários para construção de *Conditions* Locais, um parâmetro adicional referente ao identificador único da *Condition* em questão em todo o domínio do sistema. Também de maneira similar às *Premises*, a construção de *Conditions Proxy* Remoto necessita apenas do identificador único da *Condition* em questão.

Código 3.3 Declaração de *Conditions* e seus possíveis tipos

```
1   /* Declaração de uma Condition local
2   * (Local Condition)
3   * pr1 e pr2 são referências para as Premises correspondentes
4   * seguida pela operação lógica conjunção (&&) */
5   NOP::SharedCondition cn1 =
6       NOP::BuildCondition(CONDITION(*pr1 && *pr2), pr1, pr2);
7
8   /* Declaração uma Condition Compartilhada via Rede
9   *  (Network Shared Condition)
10  * "cn1" representa o identificador da Condition
11  * pr1 e pr2 são referências para as Premises correspondentes
12  * seguida pela operação lógica conjunção (&&) */
13  NOP::SharedCondition cn1 =
14      NOP::BuildNetworkSharedCondition("cn1",
15          CONDITION(*pr1 && *pr2), pr1, pr2);
16
17  /* Declaração de uma Condition Proxy Remoto
18  * (Remote Proxy Condition)
19  * "cn1" representa o identificador da Condition */
20  NOP::SharedCondition cn1 =
21      NOP::BuildRemoteProxyCondition("cn1");
22
23  /* Declaração de uma Condition Redundante via Rede
24  * (Network Redundant Condition)
25  * "cn1" representa o identificador da Condition
26  * pr1 e pr2 são referências para as Premises correspondentes
27  * seguida pela operação lógica conjunção (&&) */
28  NOP::SharedCondition cn1 =
29      NOP::BuildNetworkRedundatCondition(
30          "cn1", CONDITION(*pr1 && *pr2), pr1, pr2);
```

3.3.2.4 *Rules*

O Código 3.4 apresenta um exemplo de construção das *Rules* em seus diferentes tipos de distribuição. No modo local, são necessárias duas referências, uma para a *Condition* e outra para a *Action* associada à respectiva *Rule*. Para o modo Compartilhado via Rede e Redundante via Rede, são necessários também, além da referência para *Condition* e *Action*, um identificador único para a *Rule*. Conforme apresentado na Seção 3.2, esse identificador deve ser único em todo o domínio do sistema. Para a construção de uma *Rule Proxy* Remota, é necessário somente o identificador único e uma referência para a *Action*, que será acionada quando a respectiva *Rule* for executada.

Código 3.4 Declaração de *Rules* e seus possíveis tipos

```
/* Declaração de uma Rule local
* (Local Rule)
* cn1 é a referência para a condition
* e ac1 é a referência para a action */
NOP::SharedRule rl1 =
    NOP::BuildRule(cn1, ac1);

/* Declaração de uma Rule Compartilhada via Rede
* (Network Shared Rule)
* "rl1" representa o identificador único
* cn1 é a referência para a condition
* acLocal1 é a referência para a action */
NOP::SharedRule rl1 =
    NOP::BuildNetworkSharedRule("rl1", cn1, acLocal1);

/* Declaração de uma Rule Proxy Remoto
* (Remote Proxy Rule)
* "rl1" representa o identificador único
* acRemote1 é a referência para a action */
NOP::SharedRule rl2 =
    NOP::BuildRemoteProxyRule("rl1", acRemote1);

/* Declaração de uma Rule redundante
* (Network Redundant Rule)
* "rl1" representa o identificador único
* cn1 é a referência para a condition
* acLocal1 é a referência para a action */
NOP::SharedRule rl1 =
    NOP::BuildNetworkRedundantRule("rl1", cn1, acLocal1);
```

3.3.2.5 *Actions*

O Código 3.5 apresenta um exemplo de inicialização das *Actions* em seus diferentes tipos de distribuição. De forma similar às demais entidades já apresentadas, além dos parâmetros em comum para as entidades não distribuídas (uma ou mais referências para *Instigations*), faz-se necessário um parâmetro adicional referente à identificação única da respectiva *Action* no domínio do sistema em questão.

Código 3.5 Declaração de *Actions* e seus possíveis tipos

```
1     /* Declaração de uma Action local
2     * (Local Action)
3     * inLocal1 é a referência para a Instigation */
4     NOP::SharedAction ac1 =
5         NOP::BuildAction(inLocal1);
6
7     /* Declaração de uma Action Compartilhada via Rede
8     *  (Network Shared Action)
9     * (Local Action)
10    * "ac1" representa o identificador único dessa Action
11    * inLocal1 é a referência para a Instigation */
12    NOP::SharedAction acLocal1 =
13        NOP::BuildNetworkSharedAction("ac1", inLocal1);
14
15    /* Declaração de uma Action Proxy Remoto
16    * (Remote Proxy Action)
17    * "ac1" representa o identificador único dessa Action
18    * inRemote1 é a referência para a Instigation */
19    NOP::SharedAction acLocal1 =
20        NOP::BuildRemoteProxyMqttAction("ac1", inRemote1);
21
22    /* Declaração de uma Action redundante
23    * (Network Redundant Action)
24    * "ac1" representa o identificador único dessa Action
25    * inLocal1 é a referência para a Instigation */
26    NOP::SharedAction acLocal1 =
27        NOP::BuildNetworkRedundantAction("ac1", inLocal1);
```

3.3.2.6 *Instigations*

O Código 3.6 apresenta um exemplo de construção das *Instigations* em seus diferentes tipos de distribuição. De forma muito similar às *Actions*, além dos parâmetros em comum para as entidades não distribuídas (a referência para o *Method*), faz-se necessário um parâmetro adicional referente à identificação única da respectiva *Instigation* no domínio do sistema em questão.

Código 3.6 Declaração de *Instigations* e seus possíveis tipos

```
1    /* Declaração de uma Instigation local
2    * (Local Instigation)
3    * mt é a referência para o Method */
4    NOP::SharedInstigation in1 =
5        NOP::BuildInstigation(mt);
6
7    /* Declaração de uma Instigation Compartilhada via Rede
8    *  (Network Shared Instigation)
9    * "in1" representa o identificador único dessa Instigation
10   * mtLocal é a referência para o Method */
11   NOP::SharedInstigation inLocal1 =
12       NOP::BuildNetworkSharedInstigation("in1", mtLocal);
13
14   /* Declaração de uma Instigation Proxy Remoto
15   * (Remote Proxy Instigation)
16   * "in1" representa o identificador único dessa Instigation
17   * mtRemote é a referência para o Method */
18   NOP::SharedInstigation inRemote1 =
19       NOP::BuildRemoteProxyInstigation("in1", mtRemote);
20
21   /* Declaração de uma Instigation redundante
22   * (Network Redundant Instigation)
23   * "in1" representa o identificador único dessa Instigation
24   * mtLocal é a referência para o Method */
25   NOP::SharedInstigation inLocal1 =
26       NOP::BuildNetworkRedundantInstigation("in1", mtLocal);
```

3.3.3 Testes

Com o objetivo de verificar o correto funcionamento do *Framework* PON C++ 4.0 IoT e facilitar também o seu uso e expansão por potenciais novos desenvolvedores, implementou-se novos testes unitários explorando as novas funcionalidades, conforme apresentado na Seção 3.3.3.1. Em complemento, apresenta-se, na Seção 3.3.3.2, um *framework* para testes de integração de sistemas, permitindo que sejam realizados testes de forma automatizada em aplicações (ou partes de aplicações) PON distribuídas, por meio do envio e recebimento de mensagens MQTT.

3.3.3.1 Testes Unitários

Mantendo-se o padrão de desenvolvimento adotado por Neves (2021) (apresentado na Seção 2.4.8.1), foram criados testes unitários para as novas funcionalidades adicionadas utilizando-se o *framework* Google Test (GOOGLE, 2022).

Na Figura 43 são apresentados os resultados da execução dos testes unitários, com a aprovação de todos os 51 testes unitários implementados no *Framework* PON C++ 4.0, incluindo os 9 novos testes implementados para as novas funcionalidades de distribuição via MQTT na Suíte de Testes *Distributed Entities*.

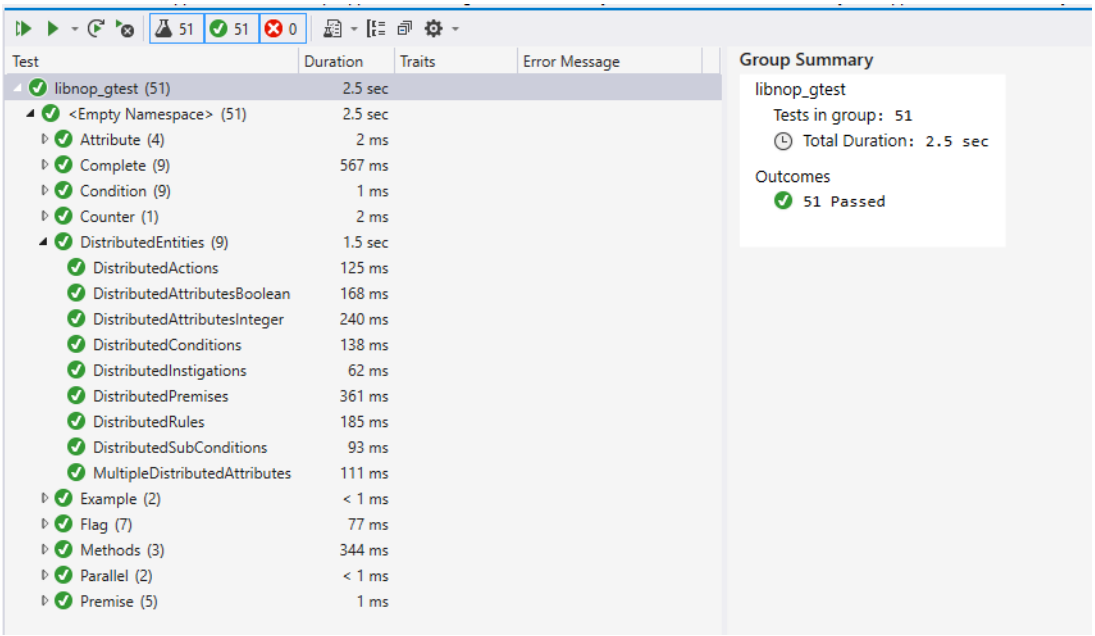


Figura 43 – Resultados dos testes unitários.

Dessa forma, os novos testes unitários implementados verificam o correto funcionamento das entidades distribuídas adicionadas enquanto os demais testes, implementados por Neves (2021), garantem que as funcionalidades existentes no *Framework* PON C++ 4.0 não foram afetadas pelas alterações.

Observa-se que foram criados testes unitários para todas as entidades distribuídas implementadas (*i.e.*, *Attributes, Premises, Conditions, Rules, Actions* e *Instigations*). Dentro de cada teste foram validados todos os possíveis tipos de distribuição (Compartilhado via Rede, *Proxy* Remoto e Redundante via Rede). É pertinente ressaltar que a execução dos testes depende que um *broker* MQTT esteja em execução, para que seja possível a efetiva troca de mensagens entre as entidades distribuídas. Para fins de facilidade e agilidade na execução, pode-se executar o *broker* Mosquitto na máquina local, utilizando-se a interface *localhost*. Conforme apresentado na Figura 44, o *broker* pode ser iniciado facilmente via linha de comando. Além disso, é possível a inicialização com o argumento '-v' que permite que seja observada a interação do *broker* com as aplicações durante o teste.

```
C:\Program Files\mosquitto>mosquitto.exe -v
1657936377: mosquitto version 2.0.14 starting
1657936377: Using default config.
1657936377: Starting in local only mode. Connections will only be possible from clients running on this machine.
1657936377: Create a configuration file which defines a listener to allow remote access.
1657936377: For more details see https://mosquitto.org/documentation/authentication-methods/
1657936377: Opening ipv4 listen socket on port 1883.
1657936377: Opening ipv6 listen socket on port 1883.
1657936377: mosquitto version 2.0.14 running
1657936404: New connection from ::1:57823 on port 1883.
1657936404: New client connected from ::1:57823 as auto-8069F7DA-20A2-B49D-4CB2-A6711D6B6876 (p2, c1, k60).
1657936404: No will message specified.
1657936404: Sending CONNACK to auto-8069F7DA-20A2-B49D-4CB2-A6711D6B6876 (0, 0)
1657936404: Received SUBSCRIBE from auto-8069F7DA-20A2-B49D-4CB2-A6711D6B6876
1657936404:     test/at/at1/ (QoS 2)
1657936404: auto-8069F7DA-20A2-B49D-4CB2-A6711D6B6876 2 test/at/at1/
```

Figura 44 – *Broker* Mosquitto em execução *localhost* com logs de comunicação habilitados.

Observa-se na Figura 45, a estrutura de tópicos e um exemplo de mensagens no *broker* MQTT após a execução dos testes unitários. Observa-se a organização dos tópicos por tipo de entidade do PON (at, pr, cn, in, ac e rl) e um exemplo de mensagens para os *Attributes*.

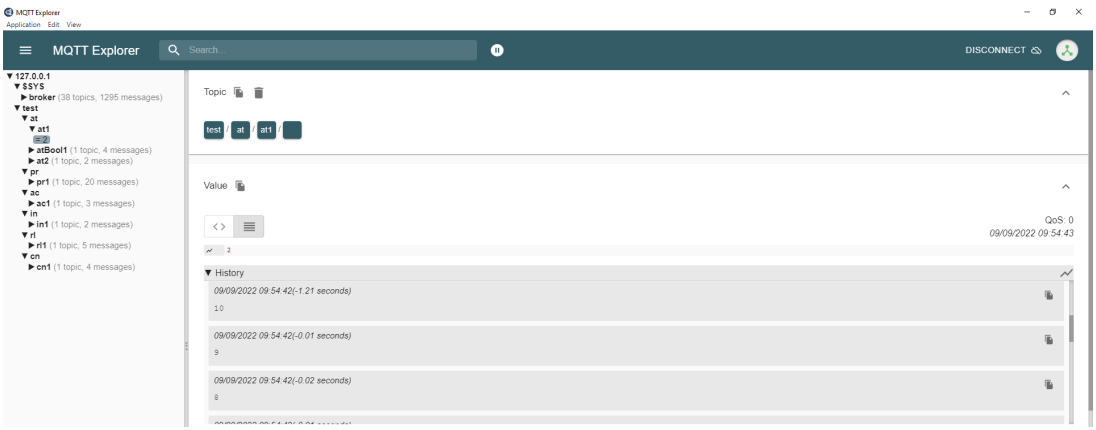


Figura 45 – Visualização da estrutura de tópicos e mensagens criadas durante a execução dos testes unitários.

Como exemplo, o Código 3.7 apresenta o teste unitário para *Attributes* Distribuídos do tipo inteiro (int). Neste exemplo, são instanciados um *Attribute* Compartilhado via Rede at1 com identificador “at1”, um *Attribute Proxy* Remoto at2 com o mesmo identificador “at1” e um *Attribute* Redundante via Rede at3 com o mesmo identificador “at1”. O teste consiste na alteração de at1 e verificação (após 10 milissegundos) do valor de at2 e de at3. Considerando-se que o *broker* esteja em execução, a alteração de at1 acarretará o envio de uma mensagem MQTT no tópico *test/at/at1/* com valor respectivo a sua alteração. Ao receber a notificação e por apresentar o mesmo identificador “at1” o *Attribute Proxy* Remoto at2 e o *Attribute* Redundante via Rede at3 terão também seus valores alterados. Essa verificação é realizada nas linhas 38 e 39, respectivamente.

Por outro lado, quando o *Attribute* Redundante via Rede at3 é alterado, na linha 43, espera-se que somente o *Attribute Proxy* Remoto at2 tenha seu valor alterado. O *Attribute* Compartilhado via Rede at1 não deve ter seu valor alterado, apresentando valores diferentes de at2 e at3. Essas verificações são realizadas nas linhas 46, 47 e 48, respectivamente.

Código 3.7 Teste unitário para *Attributes* Compartilhado via Rede, Redundante e *Proxy* Remoto.

```
TEST(RemoteMqtt, DistributedAttributesInteger)
{
    struct Test : NOP::FBE
    {
        explicit Test(const std::string_view name) : FBE(name) {}

        /* Declaração do Attribute Compartilhado via Rede at1 com
        * identificador "at1" e valor inicial "-1" */
        NOP::SharedAttribute<int> at1 =
            NOP::BuildNetworkSharedAttribute("at1", -1);

        /* Declaração do Attribute Proxy Remoto at2 com
        * identificador "at1" e valor inicial "-2" */
        NOP::SharedAttribute<int> at2 =
            NOP::BuildRemoteProxyAttribute("at1", -2);

        /* Declaração do Attribute Redundante via Rede at1 com
        * identificador "at1" e valor inicial "-3" */
        NOP::SharedAttribute<int> at3 =
            NOP::BuildNetworkRedundantAttribute("at1", -3);
    };

    Test test{"TestFBE"};

    /*Alteração do Attribute Compartilhado via Rede at1 por processo local.
    * Adicionou-se um atraso de 10 ms para comunicação com o broker */
    test.at1->SetValue(100);
    NOP::Scheduler::Instance().FinishAll();
    Sleep(10);

    /* Como os Attributes at1, at2 e at3 possuem o mesmo identificador
    * "at1" a alteração de at1 para "100" provocará uma notificação
    * via rede a qual provocará a alteração de at2 e at3 para "100" */
    EXPECT_EQ(test.at1->GetValue(), test.at2->GetValue());
    EXPECT_EQ(test.at1->GetValue(), test.at3->GetValue());

    /* Alteração do Attribute Redundante via Rede at3 por processo local */
    test.at3->SetValue(5);
    NOP::Scheduler::Instance().FinishAll();
    Sleep(10);

    /* Como at1 é um Attribute Compartilhado via Rede, seu valor não
    * pode ser alterado por notificações via rede. Dessa forma, o valor
    * de at1 deve ser diferente de at3. Por outro lado, como at1 é um
    * Attribute Proxy Remoto, seu valor deverá ser igual à at3 */
    EXPECT_NE(test.at1->GetValue(), test.at2->GetValue());
    EXPECT_NE(test.at1->GetValue(), test.at3->GetValue());
    EXPECT_EQ(test.at2->GetValue(), test.at3->GetValue());
}
```

3.3.3.2 Testes de Integração de Sistemas

Em complemento aos testes unitários, são realizados também testes funcionais de integração de sistemas. Para esses testes, definiu-se uma metodologia de testes que consiste em utilizar um conjunto de valores de entrada (ou pré-requisitos) e, em seguida, estimular o software com mensagens relacionadas ao caso de teste em análise e avaliar as respostas dos sistemas. Dessa forma, é possível a definição de um *framework* para testes de integração, no qual o conjunto de pré-requisitos, casos de teste e respostas é definido de acordo com os requisitos e casos de uso de cada aplicação em teste.

Devido à utilização de um protocolo de comunicação padronizado no contexto do *Framework* PON C++ 4.0 IoT, nomeadamente o MQTT, é possível a utilização de diferentes linguagens de programação e bibliotecas. Considerando-se a facilidade e agilidade de desenvolvimento no contexto de testes, optou-se por implementar a automação dos testes em Python, com o auxílio da biblioteca Eclipse Paho MQTT (ECLIPSE FOUNDATION, 2022) e unittest (PURCELL; DRAKE; HETTINGER, 2003). A biblioteca Eclipse Paho MQTT implementa, entre outras funcionalidades, um cliente *publish/subscribe* para uso em plataformas embarcadas ou servidores (ECLIPSE FOUNDATION, 2022). Por sua vez, a biblioteca *unittest* apresenta um *framework* para automação de testes, com suporte para inicialização, execução e geração de relatórios, além de facilitadores para organização e compartilhamento de código entre testes (PURCELL; DRAKE; HETTINGER, 2003).

De forma sucinta, os testes de integração publicam mensagens MQTT conforme a estrutura de tópicos e valores esperados e avaliam as consequentes ações da aplicação em teste. Como exemplo, considera-se o teste apresentado na Figura 46 referente a uma aplicação de avaliação de um sensor. Neste exemplo, a aplicação desenvolvida avalia o estado do sensor (*atSensor*) e dispara um alarme (*atAlarmedState*) caso o sensor seja habilitado. Neste exemplo, o *Attribute atSensor* é declarado como *Proxy* Remoto e o *Attribute atAlarmedState* é declarado como Compartilhado via Rede. Durante a etapa de início do teste (seção de pré-requisitos) é publicada uma mensagem alterando o estado do sensor (*atSensor*) garantindo-se que este esteja desabilitado (*false*) antes do início das validações. Após a execução dos passos referentes ao pré-requisito do cenário, é iniciado o cenário das validações. Para o exemplo em questão, é publicada uma mensagem alterando o estado do sensor (*atSensor*) para habilitado (*true*). Essa mensagem é entregue para a aplicação PON a qual deverá disparar um alarme, representado pelo *Attribute atAlarmedState* assumindo valor verdadeiro (*true*). A alteração do *Attribute* Compartilhado via Rede *atAlarmedState* provoca o envia de uma mensagem MQTT equivalente, a qual é verificada pelo teste. Caso a mensagem seja recebida e apresente os valores esperados, o teste é então aprovado.

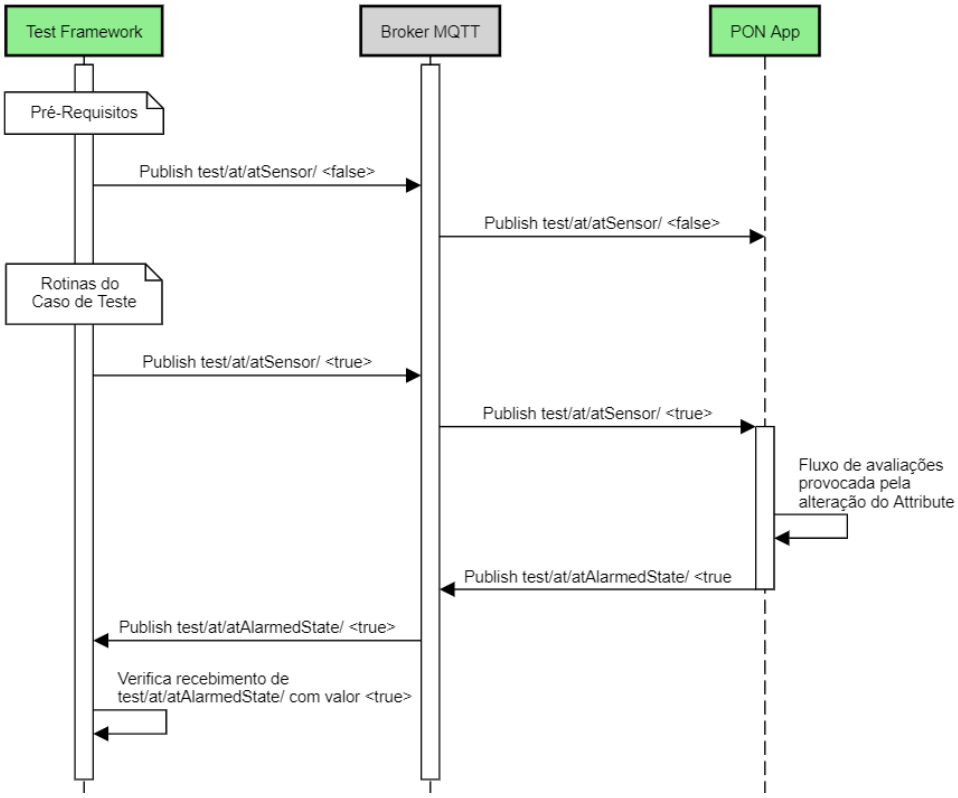


Figura 46 – Diagrama de Sequência para testes de integração.

Este conceito e metodologia de testes de integração foram utilizados para a avaliação dos experimentos apresentados posteriormente no Capítulo 4.

3.4 Considerações do Capítulo

Conforme apresentado nas Seções 2.3.3 e 2.4.8, o *Framework* PON C++ 4.0 destaca-se como estado da técnica por apresentar programação em alto nível, paralelismo e desempenho de execução melhor, quando comparado com os demais *frameworks* existentes. Neste contexto, o *Framework* PON C++ 4.0 IoT apresentado neste capítulo agrega as propriedades do *Framework* PON C++ 4.0 além de adicionar a propriedade de distribuição. Com isso, obtém-se uma abordagem que abrange, de forma inédita, as quatro potencialidades das propriedades do PON na mesma implementação, conforme apresentado no Quadro 6.

Quadro 6 – Propriedades elementares contempladas nas materializações do PON.

Framework	**Programação em alto nível**	**Paralelismo via desacoplamento**	**Distribuição via desacoplamento**	**Código não redundante**
Framework PON C++ Prototipal	parcialmente	-	-	-
Framework PON C++ 1.0	parcialmente	-	-	-
Framework PON C++ 2.0	parcialmente via *wizard*	-	-	parcialmente
Framework PON C++ 3.0	parcialmente	parcialmente	-	-
Framework PON C++ 4.0	sim	sim	-	sim
Framework PON Java/C#	parcialmente	-	-	parcialmente
Framework PON C# IoT	parcialmente	sim	sim	-
Framework PON Erlang/Elixir	parcialmente	sim	suportado, porém não validado	-
Framework PON Akka.Net	parcialmente	sim	suportado, porém não validado	-
***Framework* PON C++ 4.0 IoT**	**sim**	**sim**	**sim**	**sim**

Em complemento, conforme apresentado no Quadro 7, no aspecto de sistemas distribuídos e IoT, a implementação proposta destaca-se por permitir a distribuição de todas as entidades "distribuíveis" do PON. Além disso, o *Framework* PON C++ 4.0 IoT apresenta também, de forma inédita, uma proposta para a integração do PON com uma arquitetura *Publish/Subscribe* por meio de um protocolo padronizado comumente utilizado em ambientes IoT, o MQTT.

Quadro 7 – Trabalhos realizados em PON no contexto de sistemas distribuídos.

Framework	**Distribuição**	**Paralelismo**	**Entidades Distribuídas**	**Protocolo de Transporte**	**Protocolo de Aplicação**
Framework PON C++ 2.0 + PONIP	Sim	Não	*Attributes* *Premises*	UDP/TCP	PONIP
Framework PON C++ 3.0 + (LobeNOI + PONIP)	Sim	Sim	*Attributes* *Premises*	UDP/TCP	PONIP
Framework PON Akka	Potencialmente	Sim	N/A	N/A	N/A
Framework PON Erlang/Elixir	Potencialmente	Sim	N/A	N/A	N/A
Framework PON C# IoT	Sim	Sim	*Attributes*	UDP/TCP	HTTP Próprio
Framework PON Java + *Attributes* Distribuídos	Sim	Não	*Attributes*	UDP (*Multicast*)	Próprio
MicroPON em IoT	Sim	Não	*Attributes*	TCP	Próprio
***Framework* PON C++ 4.0 IoT**	Sim	Sim	*Attributes* *Premises* *Conditions* *Rules* *Actions* *Instigations*	TCP	MQTT

O próximo capítulo apresenta um conjunto de experimentos no contexto de IoT implementados com o *Framework* PON C++ 4.0 IoT, descrevendo as implementações e os resultados.

4 *FRAMEWORK* PON C++ 4.0 IOT - EXPERIMENTOS E RESULTADOS

Com o objetivo de verificar o correto funcionamento das novas funcionalidades de distribuição desenvolvidas, nesta seção são apresentados três experimentos utilizando o PON por meio do *Framework* PON C++ 4.0 IoT. Ademais, apresentam-se também comparações com o POE (Paradigma Orientado a Eventos) por meio de implementações em C++ utilizando o *Publish/Subscribe* (*Pub/Sub*) e o MQTT, os quais são comumente utilizados no contexto de sistemas distribuídos e IoT. Dessa forma, objetivam-se realizar comparações e analisar aspectos teóricos do PON e do POE nas implementações mencionadas, como a expressividade de *Rules* e os benefícios do compartilhamento via distribuição das entidades constituintes do PON.[1]

Os experimentos são apresentados por ordem de complexidade em relação às regras lógico-causais do sistema, iniciando-se com o mais simples. Nesse contexto, o primeiro experimento, detalhado na Seção 4.1, trata de um sistema simples e genérico de avaliação de sensores para IoT, desenvolvido em POE (por meio do *Pub/Sub* e MQTT em C++) e PON (por meio do *Framework* PON C++ 4.0 IoT) com objetivo de comparar o correto cumprimento dos requisitos funcionais além de aspectos não funcionais, principalmente relacionados com o uso dos recursos computacionais (processamento, memória e rede).

O segundo experimento, detalhado na Seção 4.2, apresenta uma implementação distribuída para o experimento do Portão Eletrônico utilizando-se as mesmas materializações do PON e POE do primeiro experimento. Com este experimento, objetivam-se comparações funcionais e não funcionais, por meio do cumprimento dos requisitos, da comparação de aspectos declarativos do paradigma e do uso de recursos de rede.

Por último, é apresentado na Seção 4.3 uma implementação para uma aplicação IoT no contexto de uma dada simulação de casa inteligente utilizando-se também POE (por meio do *Pub/Sub* e MQTT em C++) e PON (por meio do *Framework* PON C++ 4.0 IoT). Com este experimento, objetiva-se avaliar o *Framework* PON C++ 4.0 IoT em uma aplicação com múltiplos sensores e atuadores e avaliar o ganho de eficiência obtido com o compartilhamento de entidades, também de modo distribuído. Objetiva-se ainda comparar a expressividade e eficiência na distribuição das materializações do PON e do POE em uma aplicação com um número maior de elementos (sensores/atuadores) e regras lógico-causais.

4.1 Sistema de sensores para IoT

O primeiro experimento abordou um tipo de aplicação encontrada tipicamente no contexto de IoT. Mais precisamente, a aplicação desenvolvida no primeiro experimento consiste em uma central controladora que realiza a “avaliação de sensores e acionamento de atuadores”,

[1] O código completo dos experimentos e testes realizados encontra-se no servidor de artefatos do PON acessível através de login e senha no endereço <*https://nop.dainf.ct.utfpr.edu.br/nop-implementations/frameworks/nop-framework-cpp-4/nop-framework-cpp-4-iot-application*>.

isto é, a central avalia logicamente o estado de um conjunto de sensores e pode agir por meio de um conjunto de atuadores. O Código 4.1 apresenta um vislumbre da aplicação por meio de um pseudo código em formato livre, ressaltando a simplicidade do sistema em questão.

Código 4.1 Pseudo-Código do sistema de avaliação de sensores para IoT.

```
-------------------------------------------------------------
ELEMENTOS AVALIADOS
-------------------------------------------------------------
SENSOR
- Atributos:
    Valor
- Métodos:
    MudarValor
    ObterValor

-------------------------------------------------------------
ELEMENTOS AVALIADORES
-------------------------------------------------------------
CONTROLADOR
- Atributos:
    ValorReferência
- Métodos:
    Avaliar
    IncrementarValorReferência
    AcionarAtuador

-------------------------------------------------------------
REGRA DE DECISÃO //Implementada no método CONTROLADOR.Avaliar
-------------------------------------------------------------
SE
  SENSOR.ObterValor = CONTROLADOR.ValorReferência
ENTÃO
  CONTROLADOR.IncrementarValorReferência
  CONTROLADOR.AcionarAtuador
```

Como exemplo, é apresentado na Figura 47 um conjunto de possíveis sensores (detectores de chuva, detectores de movimento de automóveis e detectores de vagas de estacionamento), além de um conjunto de possíveis atuadores (semáforos, controle de iluminação pública e sinalizações de estacionamento).

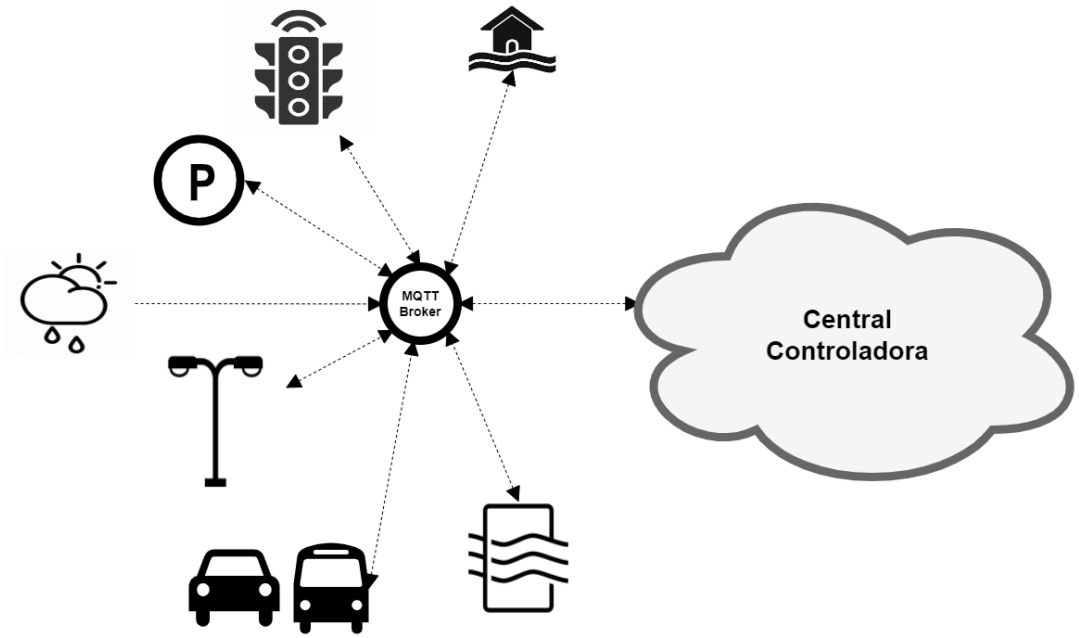


Figura 47 – Exemplo de sistema de sensores para IoT.

Nesse contexto, propõe-se um sistema capaz de realizar uma avaliação lógica simples: comparando-se o estado atual de cada sensor (ou valor do sensor) com um valor de referência para o respectivo sensor. Caso esses dois valores sejam iguais, é então acionado o atuador correspondente e incrementado o respectivo valor de referência. Dessa forma, os requisitos funcionais do sistema de sensores podem ser descritos como:

- Cada atuador está relacionado exclusivamente a um respectivo sensor.
- Cada sensor possui um contador de ativações que se inicia em zero e é incrementado a cada ativação.
- A ativação do sensor ocorre quando o valor lido pelo sensor é igual ao contador de ativações já ocorridas.
- A ativação do sensor provoca o acionamento do respectivo atuador.

Para o cenário de testes, considerou-se um sistema composto por *N* sensores e *M* atuadores os quais interagem com uma central controladora exclusivamente por meio de mensagens MQTT enviadas e recebidas ao/do *broker*, conforme ilustrado na Figura 48.

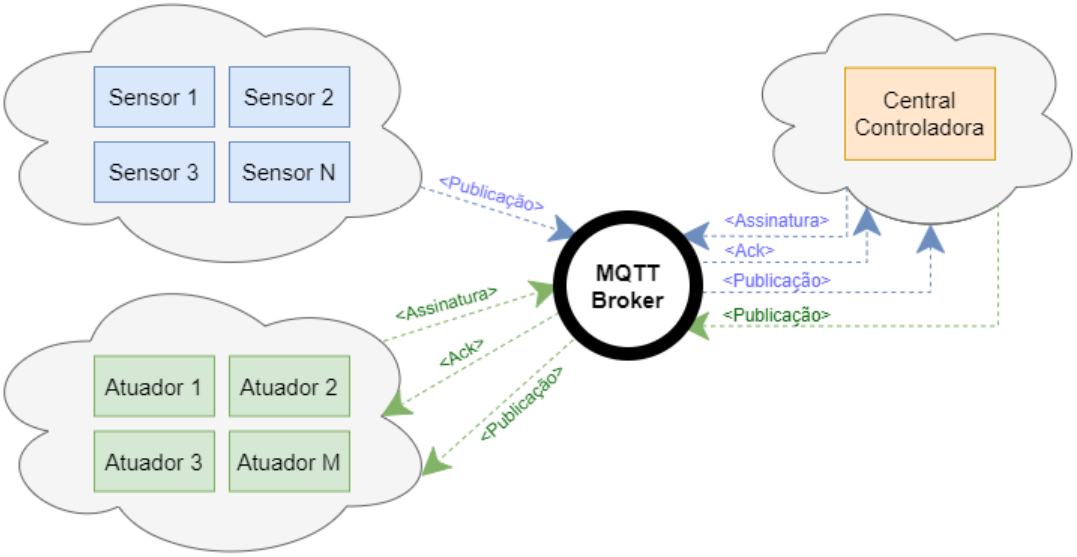


Figura 48 – Representação do sistema de sensores para IoT.

Fonte: Adaptado de Figueiredo, SIMÃO e Vendramin (2022)

Neste contexto e conforme exemplificado na Figura 49, ao detectar uma mudança de estado, os sensores podem publicar mensagens em tópicos específicos no *broker*, servindo como notificação à central controladora. Por sua vez, os elementos atuadores recebem comandos da central controladora, via *broker*, por meio de mensagens publicadas nos tópicos nos quais os atuadores se inscreveram previamente. Dessa forma, a central controladora implementa a lógica de interação entre os elementos do sistema, interagindo com os sensores e atuadores exclusivamente por meio de mensagens MQTT enviadas e recebidas ao/do *broker*.

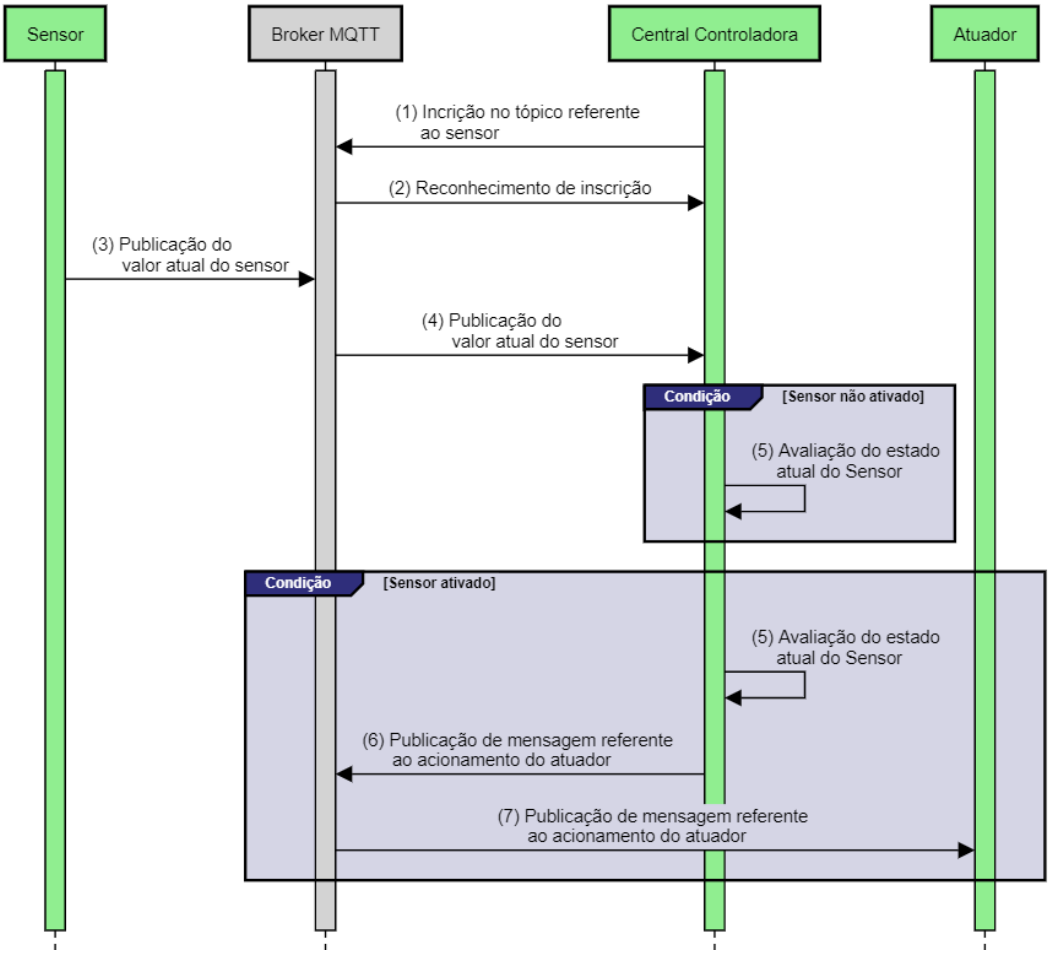


Figura 49 – Diagrama de Sequência com a interação entre os componentes do sistema de sensores.

Como os sensores e atuadores possuem uma lógica relativamente simples, necessitando apenas notificar uma mudança de estado ou receber comandos da central controladora, optou-se por não considerá-los nesta análise. Dessa forma, utilizou-se o *framework* de testes previamente apresentado na seção 3.3.3.2, sendo este *framework* responsável por duas ações básicas. A primeira delas consiste em um “gerador de eventos”, responsável por publicar mensagens de mudanças de estados dos sensores em seus respectivos tópicos no *broker*. A segunda é responsável por receber os comandos de controle enviados pela central controladora para os atuadores.

Por sua vez, a central controladora, na qual a lógica de processamento dos eventos dos sensores acontece, foi implementada utilizando-se dois paradigmas: (i) o PON via *Framework* PON C++ 4.0 IoT, apresentado na Seção 4.1.1; (ii) o Paradigma Orientado a Eventos (POE) em C++ utilizando o modelo *Publish/Subscribe* e o protocolo MQTT, apresentado na Seção 4.1.2.

Para uma comparação justa entre as implementações distribuídas, utilizou-se o mesmo *broker* de mensagens e bibliotecas de comunicação MQTT. O *broker* utilizado foi o *Mosquitto* e suas respectivas bibliotecas de comunicação em C++ (LIGHT, 2017).

Para avaliar o desempenho dos paradigmas em diferentes condições e configurações típicas em ambientes de IoT, foram criadas diferentes condições de testes. A primeira condição variável é relativa ao número de sensores presentes no sistema. Dessa forma, é possível avaliar características relativas à escalabilidade do sistema. A segunda condição variável é referente à ativação ou não do sensor (e o consequente acionamento ou não do atuador). Dessa forma, é possível avaliar o comportamento e eficiência em diferentes situações lógicas.

4.1.1 Detalhes da Implementação em *Framework* PON C++ 4.0 IoT

Para a implementação em PON por meio do *Framework* PON C++ 4.0 IoT, cada sensor foi descrito como um *FBE Sensor* agregando dois *Attributes* e dois *Methods*, conforme exemplificado na Figura 50. Ademais, utiliza-se também uma Rule com uma *Condition* relacionando uma *Premise*, além de uma *Action* com duas *Instigations*.

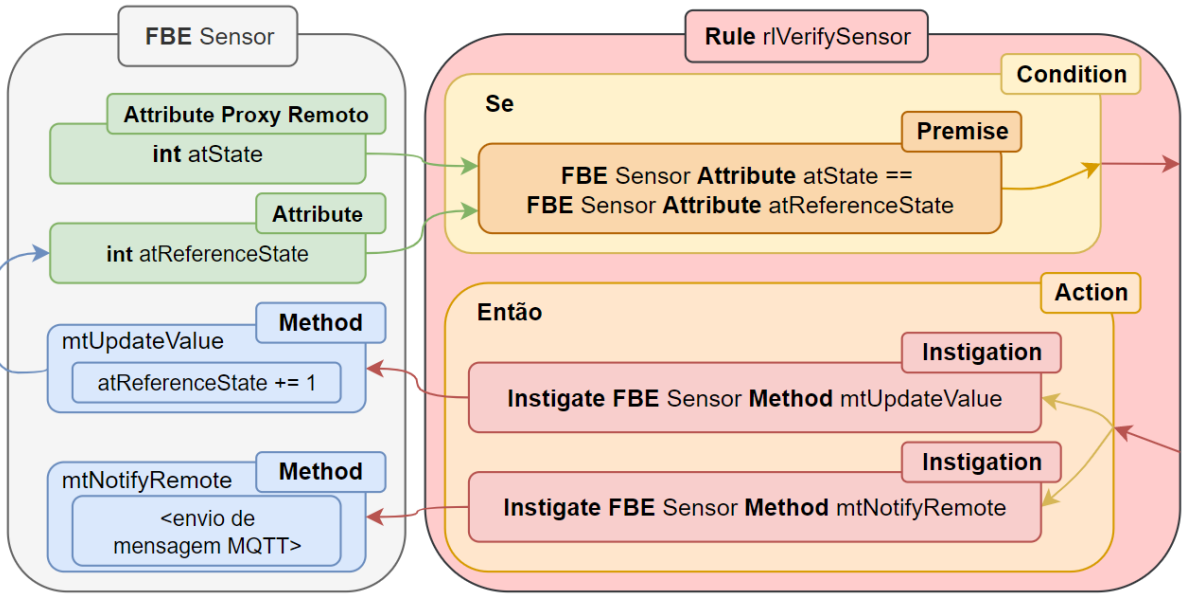


Figura 50 – *FBE* e *Rule* referente ao sistema de sensores no PON.

A implementação destas entidades utilizando-se o *Framework* PON C++ 4.0 IoT é apresentada no Código 4.2. O primeiro *Attribute* (*atState*) foi implementado como *Attribute Proxy* Remoto, representando o estado do sensor, recebido via Rede. O segundo *Attribute* (*atReferenceState*) é um *Attribute* local representando o valor atual de ativações do sensor em questão. Com relação aos dois *Methods*, o *Method mtNotifyRemote* realiza o envio de uma mensagem referente ao acionamento do atuador utilizando uma chamada direta para a classe de comunicação com o *broker* MQTT (nomeadamente, *MQTT_CALL_REMOTE*, implementada na Classe *RemoteMQTT* do *Framework* PON C++ 4.0 IoT) e o *Method mtUpdateValue* realiza a atualização do valor de referência do sensor.

Código 4.2 Código de um FBE sensor em PON referente ao sistema de sensores em *Framework* PON C++ 4.0 IoT.

```
struct Sensor : NOP::FBE
{
    //Definição da construtora da FBE sensor com string de identificação
    explicit Sensor(const string_view name) : FBE(name) , name(name){}
    string name;

    //Declaração de um Attribute com identificação incremental
    NOP::SharedAttribute<int> atState =
        NOP::BuildRemoteProxyAttribute("at" + name, -1);

    //Declaração de um Attribute para referência iniciado em zero
    NOP::SharedAttribute<int> atReferenceState = NOP::BuildAttribute(0);

    //Declaração de um Method referente ao envio
    //do valor do Attribute via rede
    NOP::Method mtNotifyRemote
    {
        METHOD(MQTT_CALL_REMOTE(nullptr, "{}\ {}", "mtNt" + name,
                                atReferenceState->GetValue());)
    };

    //Declaração de um Method referente à atualização
    //do valor de referência
    NOP::Method mtUpdateValue
    {
        METHOD(atReferenceState->SetValue(
            atReferenceState->GetValue() + 1);)
    };
};
```

Considerando a *FBE Sensor* apresentada previamente, implementaram-se os elementos referentes às *Premises*, *Conditions*, *Rules*, *Actions* e *Instigations*. Conforme apresentado no Código 4.3, cada *Premise* realiza a avaliação relacional dos dois *Attributes* do Sensor (nomeadamente *atState* e *atReferenceState*), notificando a respectiva *Condition* em caso de igualdade. A *Condition* implementada considera apenas uma *Premise* em sua composição. Quando notificada, a respectiva *Rule* é ativada, executando uma *Action* que, por sua vez, ativa uma *Instigation* responsável pela instigação do *Method* do *FBE* Sensor. A quantidade de *FBEs* é variável, conforme as condições do teste. Todas as entidades são adicionadas em listas respectivas a cada uma. Essas listas são utilizadas neste contexto apenas para organização e persistência das entidades utilizadas durante a execução da aplicação.

Código 4.3 Código lógico causal para a implementação dos sensores em *Framework* PON C++ 4.0 IoT.

```
...

//Ciclo de declaração de N sensores, com N assumindo valor 100000
for (int i = 0; i < 100000; i++)
    {
        //Declaração de um sensor com identificação incremental
        alarmList.emplace_back(new Sensor(to_string(i)));

        //Declaração de uma Premise avaliando a igualdade
        //de atState e atReferenceState
        premiseList.emplace_back(NOP::BuildPremise<int>(
            alarmList[i]->atState, alarmList[i]->atReferenceState,
            NOP::Equal()));

        //Declaração de uma Condition com uma única Premise
        conditionList.emplace_back(NOP::BuildCondition(
            CONDITION(*premiseList[i]), premiseList[i]));

        //Declaração de duas Instigations com um Method cada,
        //sendo um referente à notificação remota e um referente
        //à atualização do valor de referência
        instigationList.emplace_back(
            NOP::BuildInstigation((*alarmList[i]).mtNotifyRemote));
        instigationList.emplace_back(
            NOP::BuildInstigation((*alarmList[i]).mtUpdateValue));

        //Declaração de uma Action contendo duas Instigations
        actionList.emplace_back(
            NOP::BuildAction<NOP::Parallel>(instigationList[i*2],
                instigationList[(i*2)+1]));

        //Declaração de uma Rule relacionada à Condition e
        //a Action declaradas previamente
        ruleList.emplace_back(
            NOP::BuildRule(conditionList[i], actionList[i]));
    }

...
```

4.1.2 Detalhes da Implementação em POE via *Pub/Sub* em C++

Para a implementação em POE por meio do *Pub/Sub* e MQTT em C++ utilizaram-se estruturas decisórias usuais de “se-então” avaliando (por meio de chamadas de métodos) os estados dos objetos e, quando pertinente, atuando (também) por meio dos métodos dos respectivos objetos. Conforme apresentado no diagrama de classes apresentado na Figura 51, implementou-se uma classe referente à representação virtual dos sensores (*Sensor*) e uma classe referente à Central Controladora (*Controller*), além das classes auxiliares para comunicação com o *broker* MQTT. Para uma comparação justa entre os paradigmas, utilizou-se uma estrutura de classes para a comunicação com o *broker* MQTT semelhante (com as devidas adaptações) à implementada no *Framework* PON C++ 4.0 IoT (previamente apresentada na Seção 3.3). Implementou-se também duas classes auxiliares base para os *Publishers* e para os *Subscribers*, que encapsulam a execução das operações no *broker*: inscrição nos tópicos, envio e recebimento de mensagens. Quando um evento é recebido pelo *subscriber*, o método *update* do respectivo *subscriber* é invocado.

A classe Sensor, derivada da classe *Publisher*, representa um elemento sensor, o qual agrega dois atributos representando a identificação e o valor (*name* e *value*, respectivamente) e um método para alteração do valor (*changeValue*). Quando o valor do sensor é alterado, uma notificação é enviada para o *broker* por meio do método *publish*, descrito na classe *Publisher*.

A classe *SensorImage* representa um “espelho computacional” (*Computational Mirror*), ou seja, uma abstração virtual de um Sensor, utilizado pela central controladora para armazenamento dos atributos e métodos de interesse de cada sensor conhecido. Dessa forma, a classe *SensorImage* contempla três atributos (um inteiro e dois do tipo *string*) e quatro métodos. Os dois atributos do tipo *string* representam a identificação abreviada (*name*) e a identificação completa (*completeName*) do objeto. O atributo do tipo inteiro representa o contador atual de ativações (*referenceValue*). Com relação aos quatro métodos, além dos métodos de leitura e alteração de determinados atributos (*getReferenceValue, setValue* e *getName*) é descrito também um método para a ativação do sensor, o qual incrementa o contador de ativações (*referenceValue*) e dispara uma ação de acionamento do respectivo atuador.

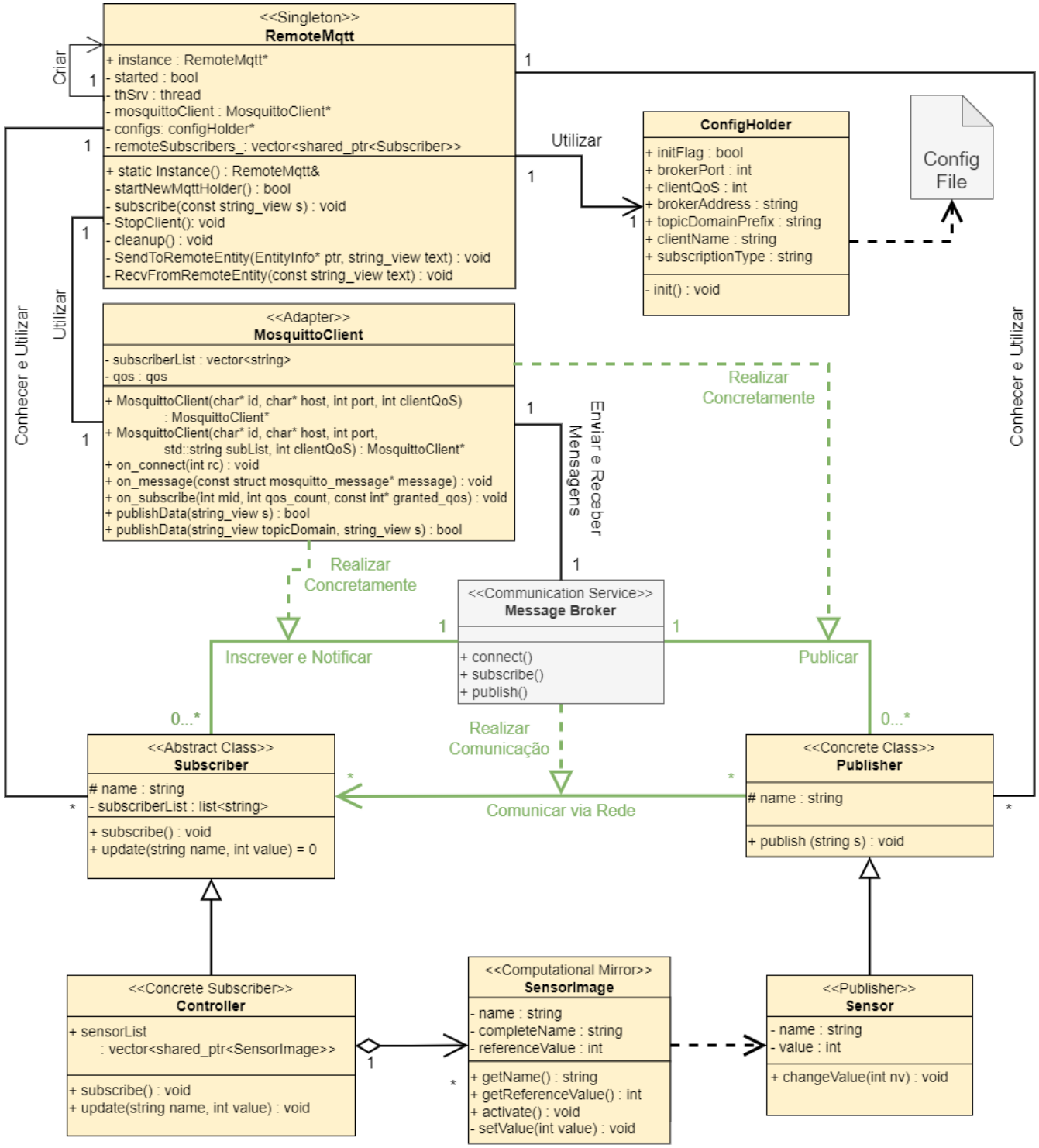


Figura 51 – Diagrama de classes em UML da implementação do sistema de sensores em POE via *Pub/Sub*.

Por sua vez, a classe *Controller* implementa a lógica de controle do sistema, verificando os valores dos sensores e, quando pertinente, ativando os sensores e, consequentemente, acionando os respectivos atuadores. A lógica de avaliação da classe *Controller* acontece dentro do método *Update*, detalhado no Código 4.4.

Código 4.4 Código em C++ do método Update da classe referente à Central Controladora em POE via *Pub/Sub* em C++.

```
...
void Controller::update(std::string nameSensor, int valueAsInt) {

  //Percorre a lista de sensores conhecidos pela central controladora
  for (int i = 0; i < sensorList.size(); i++)
  {
    SensorImage* sensor = (SensorImage*)sensorList[i].get();

    //compara a identificação de cada item sensor
    //com a identificação da notificação recebida
    if (sensor->getName() == nameSensor) {

      //verifica se o valor recebido é igual ao valor de
      //acionamentos já ocorridos
      if (sensor->getReferenceValue() == valueAsInt) {

        //ativa o sensor e, consequentemente, aciona o atuador
        sensor->activate();

      }
    }
  }
}
...
```

Conforme apresentado no diagrama de sequência apresentado na Figura 52, o método *update*, responsável pela verificação lógica do sensor, recebe como parâmetro um valor inteiro e uma *string*, representando o estado do sensor e sua identificação, respectivamente. Com a identificação do sensor, ele é encontrado dentro da lista de sensores da Central e seu valor de referência é consultado. Caso esse valor recebido seja igual ao valor de referência atual do sensor o sensor é ativado (Condição “Sensor ativado”). Quando ativado, o método *activate* é executado no sensor (*Sensor*), provocando o envio de uma notificação para os possíveis interessados. Além disso, o valor de referência do respectivo sensor é incrementado. Caso o sensor não seja ativado, nenhum acionamento é realizado.

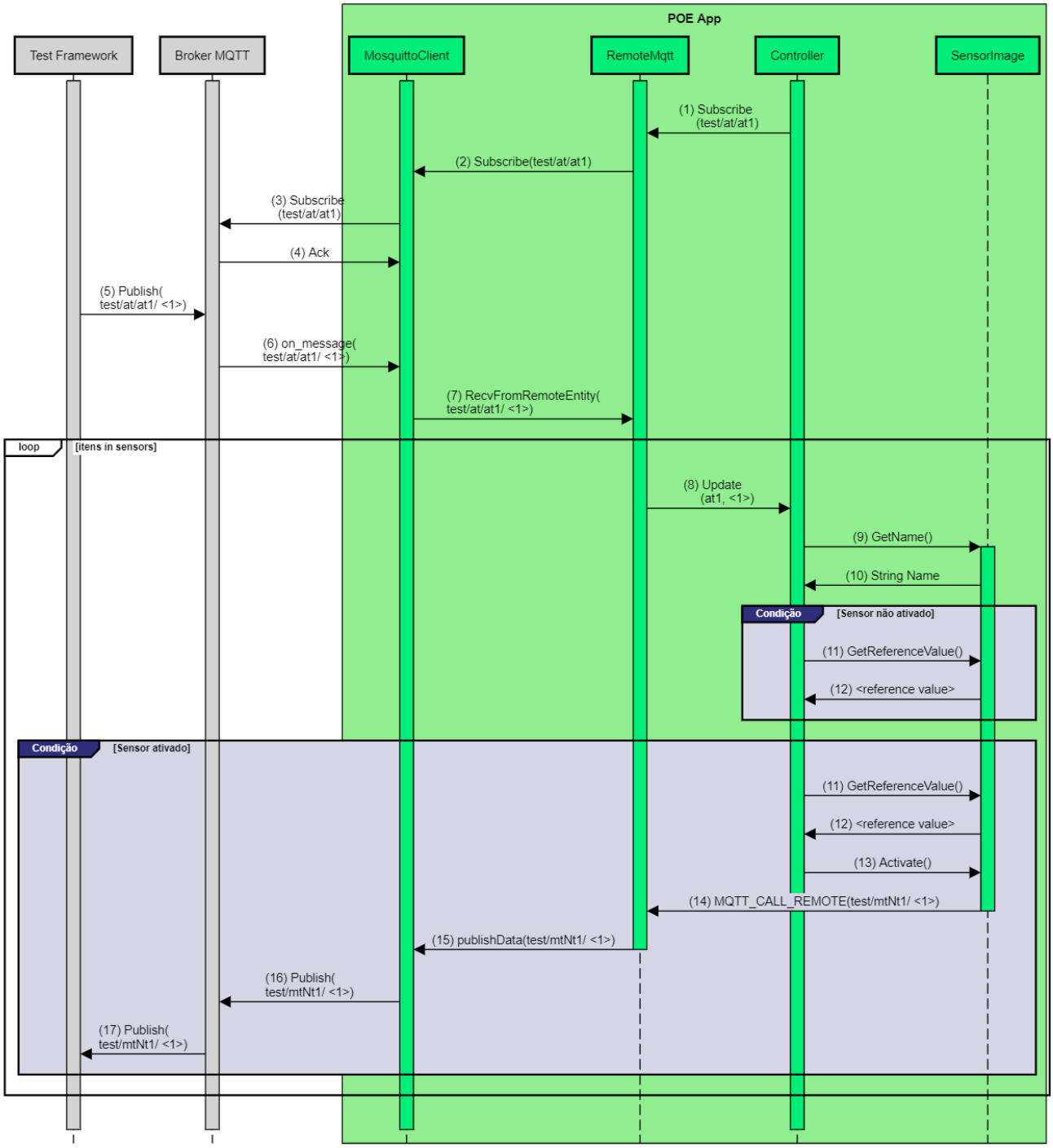


Figura 52 – Diagrama de sequência com a interação das classes do sistema de sensores em POE via *Pub/Sub* em C++.

Observa-se que os objetos da classe *SensorImage* são utilizados, neste contexto, como uma representação virtual e interna dos objetos sensores (classe *Sensor*). Dessa forma, é possível que uma aplicação externa simule as mudanças dos valores dos sensores e verifique o correto acionamento dos atuadores. Neste exemplo, as mensagens da rede são enviadas e recebidas pela aplicação auxiliar denominada *Test Framework*, responsável por publicar mensagens de mudanças de estados dos sensores e receber os comandos de controle enviados pela central controladora para os atuadores.

Considerando a classe *SensorImage*, implementou-se a inicialização de um número variável de objetos, conforme as condições do teste. Como exemplo, o Código 4.5 descreve um ciclo de inicialização de N sensores.

Código 4.5 Código em C++ implementando a lógica de inicialização dos sensores no POE via *Pub/Sub* em C++.

```
1 void Controller::init(int n) {
2   //Ciclo de declaração de N sensores e N atuadores
3   for (int i = 0; i < n; i++)
4   {
5     auto sensor = std::make_shared<SensorImage>(std::to_string(i));
6     sensorList.emplace_back(sensor);
7     subscribe("/at/" + sensor->getName());
8   }
9 }
```

4.1.3 Descrição das Métricas

Para as avaliações, utilizou-se um processador Intel(R) Core(TM) i7-8665U, *Quad Core*, 2.11 GHz, com 16GB de memória RAM DDR4 e sistema operacional *Windows 10 Enterprise*. Para evitar eventuais atrasos na rede de comunicação, avaliou-se todo o sistema na máquina local. As seguintes métricas de desempenho foram avaliadas:

- **Tempo total para inicialização da aplicação:** avalia o tempo total para inicialização da aplicação, considerando-se diferentes quantidades de sensores (100, 1000, 10000 e 100000) em duas situações: (i) quando a aplicação realiza a inscrição de cada elemento individualmente; (ii) quando a aplicação realiza a inscrição global em todos os tópicos do *broker*.
- **Uso máximo de recursos de memória:** avalia a alocação de memória RAM máxima necessária para a execução de uma aplicação, considerando-se diferentes quantidades de sensores (100, 1000, 10000 e 100000) e o uso de inscrições individuais.
- **Uso máximo de recursos de processamento:** avalia o uso máximo de recursos de processamento para a execução de uma aplicação, considerando-se diferentes quantidades de sensores (100, 1000, 10000 e 100000) e o uso de inscrições individuais.
- **Tempo médio de processamento das mensagens:** avalia o tempo médio de processamento após a alteração de estado de um sensor, sendo este a diferença de tempo entre o recebimento de uma mensagem e a finalização da avaliação dessa mensagem.

Optou-se por avaliar o tempo médio de processamento das mensagens considerando-se o uso de inscrições individuais, com diferentes quantidades de sensores (100, 1000, 10000 e 100000) e em duas situações: (i) quando a mensagem recebida provoca acionamento de um atuador e; (ii) quando a mensagem recebida não provoca acionamento de um atuador.

Para a avaliação do **tempo total para inicialização da aplicação** utilizou-se a biblioteca *chrono* para a medição do tempo total para criação dos elementos do sistema. Para referência, no caso da implementação em POE, esse tempo corresponde ao tempo para execução do *loop* descrito no Código 4.5. No caso da implementação em PON, é avaliado o tempo total para execução do *loop* descrito no Código 4.3. Optou-se por avaliar a implementação utilizando inscrições individuais e globais. Relembra-se que inscrições globais representam uma única inscrição por cliente, utilizando-se *wildcards* que englobam um conjunto de tópicos. No caso das inscrições individuais são utilizados uma operação por tópico. Para a implementação em POE, no caso das inscrições globais removeu-se a linha 13 do Código 4.5 e adicionou-se a inscrição na inicialização do cliente da instância da classe *RemoteMQTT*. No caso da implementação em PON via *Framework* PON C++ 4.0 IoT, alterou-se o arquivo de configuração adequadamente (conforme apresentado na Seção 3.3.1).

Para avaliação do **tempo médio de processamento das mensagens** utilizou-se a metodologia descrita na Seção 3.3.3.2 para a geração das mensagens que serão processadas pela aplicação. De forma resumida, o fluxo de mensagens é apresentado na Figura 53. O *framework* de testes é responsável por publicar a mensagem referente a um respectivo sensor na etapa (1) como, por exemplo, uma mensagem com conteúdo 1 no tópico *test/at/at1* referente ao sensor *at1*. Esta mensagem será enviada pelo *broker* para a aplicação na etapa (2). Essa mensagem será, então, processada pela aplicação PON entre as etapas (3) e (6). Quando a mensagem recebida provocar a ativação de um sensor, a aplicação enviará uma mensagem de acionamento do respectivo atuador, descrito na etapa (5). Por exemplo, envia-se a mensagem com conteúdo 1 no tópico respectivo ao atuador correspondente ao sensor em avaliação (*at1*), nomeadamente o tópico *test/mtNT1/*.

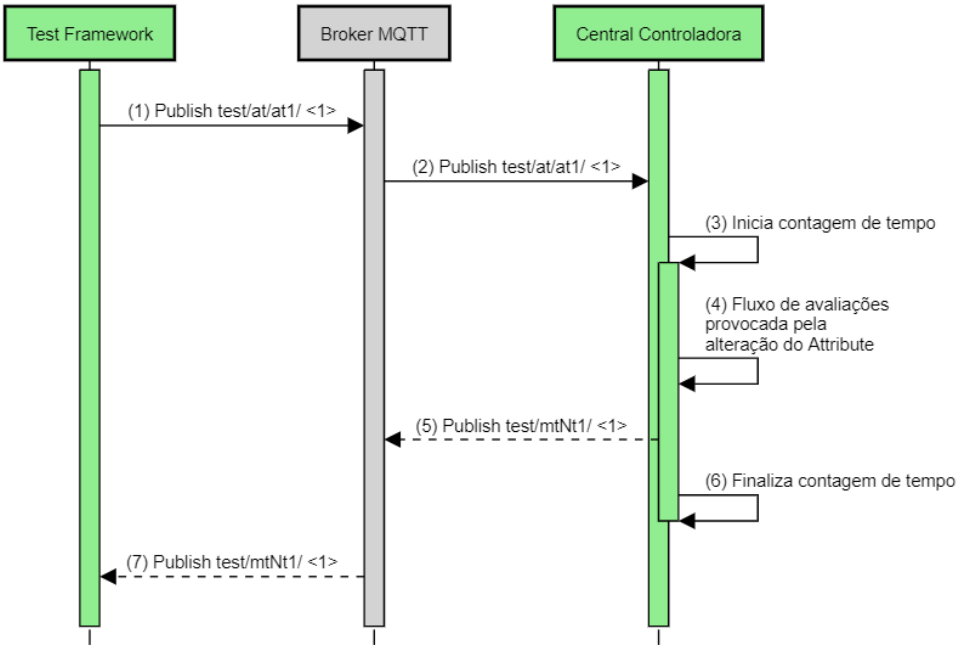


Figura 53 – Diagrama de sequência para testes de avaliação do Tempo de Processamento das mensagens no experimento de sensores para IoT.

Neste experimento, utilizou-se a biblioteca *chrono* para a medição da diferença de tempo entre o recebimento de uma mensagem até a finalização desse processamento e liberação para o processamento de uma nova mensagem. Em referência à Figura 53, o tempo de processamento é iniciado na etapa (3) e finalizado na etapa (6). As mensagens de alteração dos sensores foram produzidas a cada 0,01 segundo. Ademais, cada avaliação foi realizada utilizando-se 5000 ciclos de processamento de mensagens.

Para avaliar o **uso de processamento e memória** utilizou-se a ferramenta de diagnóstico disponibilizada no ambiente de desenvolvimento do *Microsoft Visual Studio 2019*.

4.1.4 Resultados das Implementações em *Framework* PON C++ 4.0 IoT e POE via *Pub/Sub* em C++

O resultado da avaliação para o **tempo total para inicialização da aplicação** é apresentado na Figura 54 e detalhado no Quadro 8. Observa-se que, em geral, a implementação em *Framework* PON C++ 4.0 IoT demorou até 5,3 vezes mais tempo na rotina de inicialização da aplicação quando comparado com a implementação em POE via *Pub/Sub* em C++. Esse resultado pode ser atribuído ao tempo de inicialização das estruturas adicionais implementadas no *framework*, as quais nesse caso são responsáveis pelo mecanismo de notificações do PON. Em comparação, no caso do Sensor em POE (apresentado na Seção 4.1.2) é instanciado um único objeto com três atributos por sensor presente no experimento. No caso das implementações em *Framework* PON C++ 4.0 IoT, são necessários, no mínimo, seis objetos para cada sensor (2 *Attributes*, 1 *Premise*, 1 *Condition*, 1 *Action* e 1 *Rule*), sendo estes objetos compostos

por, no mínimo, um atributo cada. Além disso, no caso das aplicações em PON, as 'conexões' entre as entidades notificantes do PON é também realizada durante a inicialização.

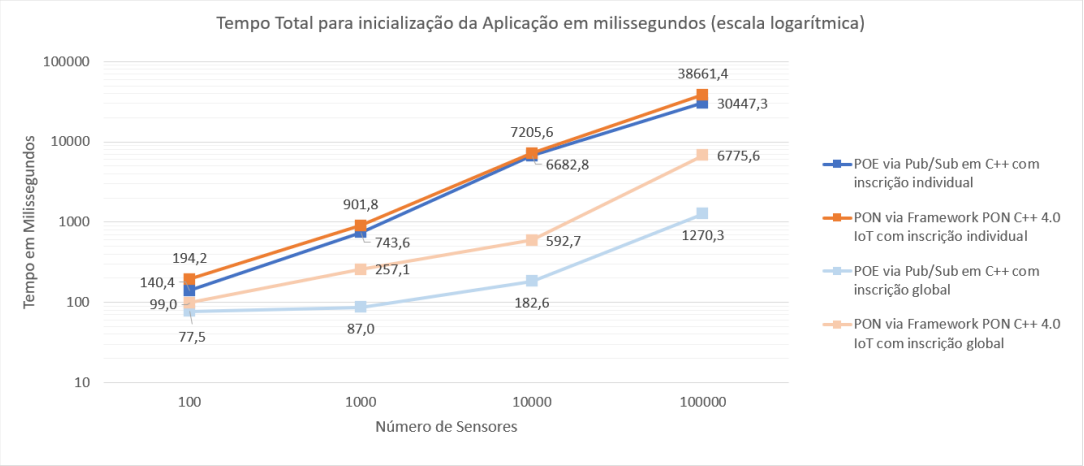


Figura 54 – Tempo total para inicialização da aplicação no Experimento de Sensores IoT.

Quadro 8 – Comparativo do tempo total para inicialização da aplicação no Experimento de Sensores IoT.

	Quantidade de Sensores	POE via *Pub/Sub* em C++ (em ms)	PON via *Framework* PON C++ 4.0 IoT (em ms)	**Desempenho da implementação PON em relação ao POE em %**
Inscrição Individual	100	140,4	194,2	**138,3%**
	1000	743,6	901,8	**121,3%**
	10000	6682,8	7205,6	**107,8%**
	100000	30447,3	38661,4	**127,0%**
Inscrição Global	100	77,5	99,0	**127,7%**
	1000	87,0	257,1	**295,4%**
	10000	182,6	592,7	**324,5%**
	100000	1270,3	6775,6	**533,4%**

Observou-se também que o tempo total para inicialização da aplicação é impactado pelo tipo de inscrição utilizada. Nesse contexto, o uso de uma inscrição global é mais eficiente para inicialização, para ambos os paradigmas. Esse resultado é esperado considerando-se que para a inscrição global é necessário somente uma mensagem de inscrição, independentemente da quantidade de sensores presentes no sistema. Para as inscrições individuais são necessárias uma mensagem de inscrição por sensor. Porém, destaca-se que em alguns casos o uso de uma inscrição global pode acarretar processamento desnecessário, decorrente do recebimento de mensagens que não são pertinentes à aplicação (conforme previamente apresentado na Seção 3.2.3).

Os resultados referentes ao **tempo médio para processamento de uma mensagem** para múltiplas condições são apresentados na Figura 55 e detalhados no Quadro 9. Os resultados mostram que o comportamento das aplicações é semelhante para os dois cenários observados (com acionamento de atuadores e sem acionamento dos atuadores). Para todos os cenários, o POE via *Pub/Sub* em C++ apresentou tempos até 11,2 vezes maiores que a

implementação em *Framework* PON C++ 4.0 IoT. Esses resultados podem ser atribuídos ao mecanismo de notificações precisas do PON além do uso de estruturas eficientes na implementação do *Framework* PON C++ 4.0 IoT. Observa-se também que os tempos medidos quando existe a ativação de atuadores são, no geral, um pouco maiores do que sem a ativação de atuadores. Esse resultado está dentro do esperado considerando-se a presença de uma notificação para a rede no caso da ativação.

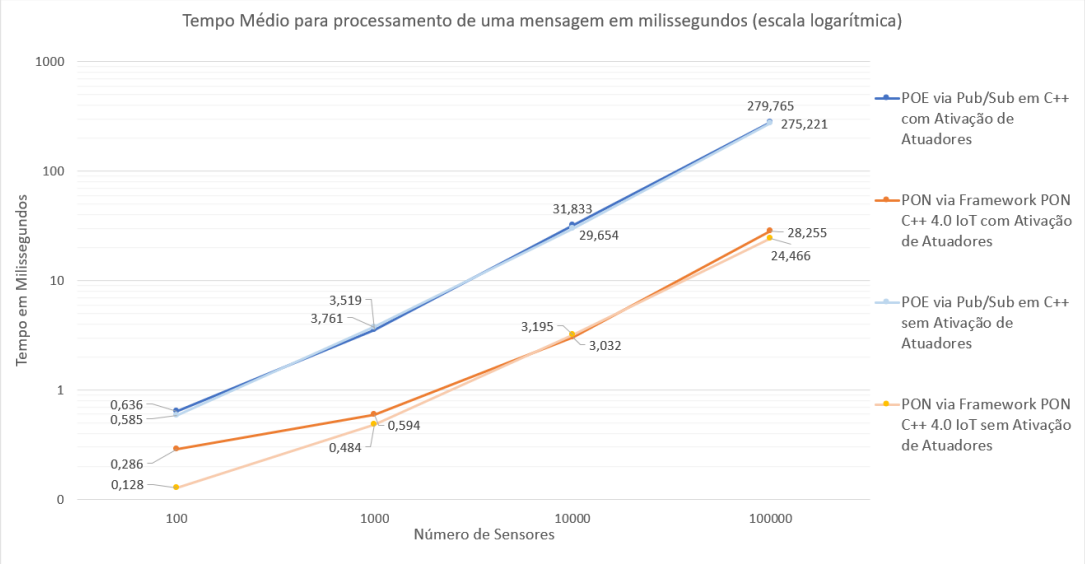


Figura 55 – Tempo médio para processamento de uma mensagem no Experimento de Sensores IoT.

Quadro 9 – Comparativo do tempo médio para processamento de uma mensagem no Experimento de Sensores IoT.

	Quantidade de Sensores	POE via *Pub/Sub* em C++ (em ms)	PON via *Framework* PON C++ 4.0 IoT (em ms)	**Desempenho da implementação PON em relação ao POE em %**
Com ativação de Atuadores	100	0,636	0,286	**45,0%**
	1000	3,519	0,594	**16,9%**
	10000	31,833	3,032	**9,5%**
	100000	279,765	28,255	**10,1%**
Sem ativação de Atuadores	100	0,585	0,128	**21,8%**
	1000	3,761	0,484	**12,9%**
	10000	29,654	3,195	**10,8%**
	100000	275,221	24,466	**8,9%**

Os resultados da avaliação do **uso de memória RAM** para a execução da aplicação de sensores são apresentados na Figura 56 e detalhados no Quadro 10. Observa-se que as implementações em PON via *Framework* PON C++ 4.0 IoT utilizaram até 6,7 vezes mais memória RAM do que as aplicações em POE via *Pub/Sub* em C++. Nesse contexto, o maior uso de memória RAM pelas aplicações em PON via *frameworks* pode ser considerado dentro do esperado por conta da estrutura adicional do próprio *framework* (conforme mencionado anteriormente), além das estruturas adicionais necessárias para distribuição das entidades.

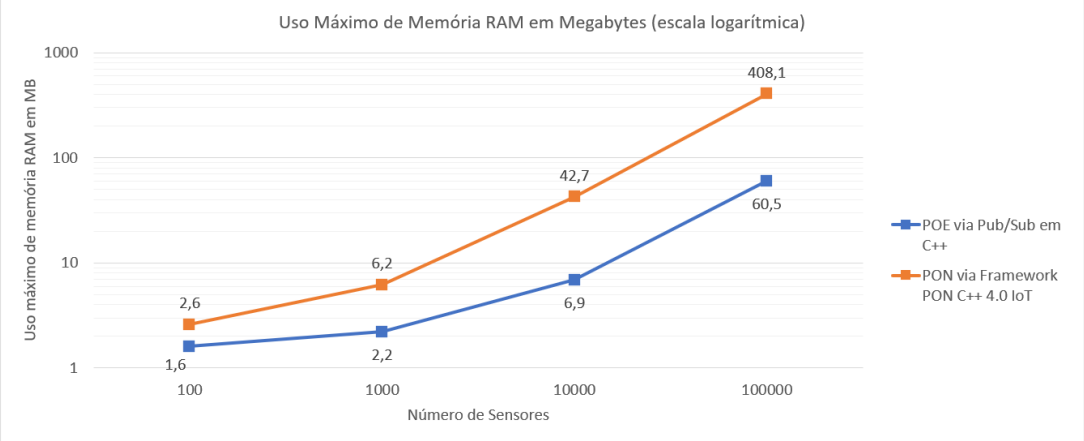


Figura 56 – Uso máximo de memória RAM no Experimento de Sensores IoT.

Quadro 10 – Comparativo do uso máximo de memória RAM no Experimento de Sensores IoT.

Quantidade de Sensores	POE via *Pub/Sub* em C++ (em MB)	PON via *Framework* PON C++ 4.0 IoT (em MB)	**Desempenho da implementação PON em relação ao POE em %**
100	1,6	2,6	**162,5%**
1000	2,2	6,2	**281,8%**
10000	6,9	42,7	**618,8%**
100000	60,5	408,1	**674,5%**

Com relação ao **uso de recursos de processamento**, observou-se que nas quatro situações consideradas no experimento (100, 1000, 10000 e 100000 sensores) e para ambos os paradigmas, o uso de processamento se manteve abaixo de 13%. Dessa forma, o uso de processamento das CPUs foi praticamente o mesmo para todas as implementações e em todos os cenários.

4.1.5 Considerações do Experimento

Os resultados do experimento de Sensores para IoT mostraram que em todos os cenários (100, 1000, 10000 e 100000) a implementação em PON via *Framework* PON C++ 4.0 IoT obteve melhor desempenho que a aplicação em POE via *Pub/Sub* em C++, apresentando um tempo até 11,2 vezes menor para o processamento de uma mensagem nas duas diferentes condições observadas (com acionamento de atuadores e sem acionamento de atuadores). De forma geral, esses resultados corroboram com a teoria do PON, evidenciando a redução das redundâncias estrutural e temporal, mesmo para aplicações distribuídas.

Em relação ao tempo para a inicialização da aplicação, observa-se que as aplicações em *Framework* PON C++ 4.0 IoT apresentaram, no geral, um tempo maior ou igual que as aplicações em POE via *Pub/Sub* em C++. Para a condição de uso de *wildcards* para inscrição global, a inicialização da aplicação do POE ocorreu mais rapidamente que a aplicação do *Framework* PON C++ 4.0 IoT para todas as quantidades de sensores avaliados. Neste caso, considera-se que o tempo para a realização da única operação no *broker* seja significativamente menor que

o tempo para inicialização das demais estruturas das aplicações (por exemplo, instâncias de objetos e suas interconexões). Dessa forma, ficaria evidente nos resultados apenas o tempo para inicialização das estruturas e dos objetos instanciados pelas aplicações. Neste contexto, observa-se que a implementação em *Framework* PON C++ 4.0 IoT apresenta um *overhead* maior ocasionado pelas estruturas necessárias do próprio *framework*. Para o caso da inicialização com o uso de inscrições individuais, o tempo de inicialização da aplicação em *Framework* PON C++ 4.0 IoT e POE via *Pub/Sub* em C++ foram semelhantes. Observa-se que, neste caso, cada inicialização de um elemento do sistema acarreta uma operação de inscrição no *broker*. Considera-se que o tempo desta operação seja significativamente maior que o tempo de inicialização dos objetos internos das aplicações e por esse motivo os tempos para inicialização das aplicações são semelhantes.

Observa-se também que as implementações em PON via *Framework* PON C++ 4.0 IoT apresentaram um uso de memória RAM maior que as aplicações em POE via *Pub/Sub* em C++. Esse maior consumo de memória RAM já havia sido identificado também por Neves (2021) durante a avaliação do *Framework* PON C++ 4.0, o qual observou um consumo até 12 vezes maior em relação ao POO. Observa-se que a implementação em POE via *Pub/Sub* em C++ apresentada neste experimento de sensores em IoT se baseia também no POO. Neste contexto, atribui-se o maior uso de memória RAM à necessidade da estrutura do próprio *framework*, visto que não existe essa sobrecarga para as implementações em POE via *Pub/Sub* em C++. Nesse aspecto, a implementação e execução da aplicação em sua forma distribuída não parece afetar significativamente o consumo de memória RAM. Destaca-se, porém, que o maior uso de memória RAM pode dificultar o uso do *Framework* PON C++ 4.0 IoT em dispositivos com poucos recursos de memória disponíveis.

4.2 Portão Eletrônico

Como segundo experimento, propôs-se a implementação da aplicação de controle de um portão eletrônico. Essa aplicação é um exemplo já abordado por alguns trabalhos relacionados ao PON (WIECHETECK, 2012) (BATISTA, 2013) (XAVIER, 2014) (BARRETO; VENDRAMIN; SIMÃO, 2018). Destacam-se o trabalho de Barreto, Vendramin e SIMÃO (2018) o qual avaliou o PON no contexto de sistemas distribuídos por meio desse exemplo e o trabalho de Xavier (2014) o qual apresenta uma comparação entre PON e POE também por meio deste exemplo. Conforme apresentado na Figura 57, o sistema alvo desse experimento é composto por quatro partes: um portão automatizado, uma lâmpada, um controle remoto e uma central controladora.

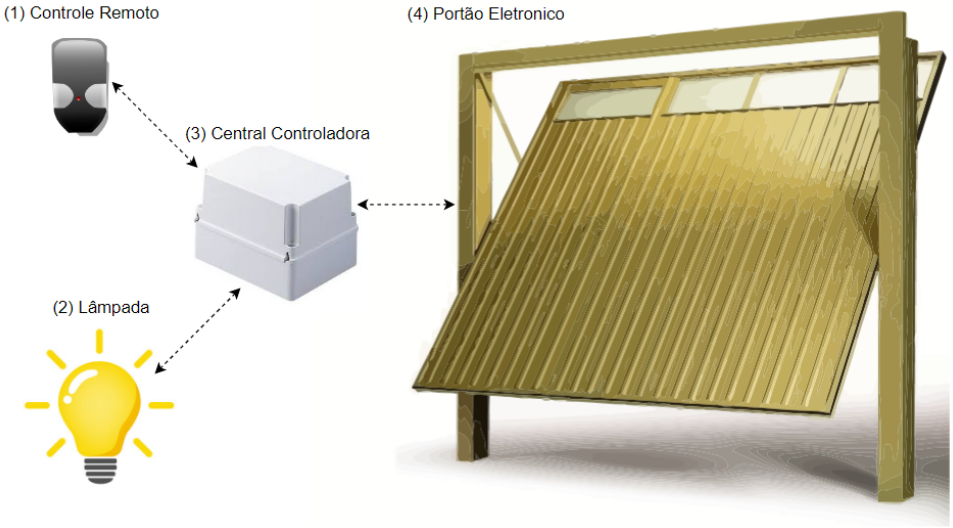


Figura 57 – Desenho conceitual do experimento do Portão Eletrônico.
Fonte: Adaptado de Xavier (2014)

O objetivo geral do sistema é permitir ao usuário executar ações de abertura e fechamento do portão por meio de um controle remoto. Além da movimentação do portão, as ações do usuário geram também efeitos na lâmpada a qual deve acender ou apagar em casos previamente definidos. Além disso, a abertura ou o fechamento do portão podem ser interrompidos durante sua execução, caso o controle remoto seja acionado. Os requisitos funcionais do sistema podem ser descritos como (XAVIER, 2014) (BATISTA, 2013):

- A central controladora deverá receber eventos de um controle remoto acionado conforme escolha do usuário (botão pressionado).
- A central controladora deverá controlar o processo de abertura e fechamento de um portão automatizado, sendo que o portão deve iniciar fechado e possuir um contador (*timer*) que se inicia zerado e parado.
- O processo de abertura do portão dura 30 segundos.
- O processo de fechamento do portão dura 30 segundos.
- Estando o portão fechado, a central controladora deverá iniciar a abertura do mesmo quando o controle remoto for pressionado e, após 30 segundos, o portão estará aberto. O contador deverá ser zerado após a abertura completa do portão.
- Durante a execução da abertura do portão, a lâmpada do ambiente deverá ser acesa.
- Uma vez aberto o portão, e caso o controle remoto seja pressionado novamente, a central controladora deverá iniciar o processo de fechamento do portão, que estará completamente fechado após 30 segundos.

- Durante a execução do fechamento do portão, a lâmpada do ambiente deverá ser apagada.
- Durante a execução de abertura do portão, em que não se passaram ainda os 30 segundos, caso haja um acionamento do controle remoto, o portão deverá parar de abrir, ficando semiaberto. Nesta situação, acionando-se o controle novamente o portão volta a fechar (*i.e.*, cancelamento da abertura).
- Durante a execução do fechamento do portão, em que não se passaram ainda os 30 segundos, caso haja um acionamento do controle remoto, o portão para de fechar, ficando semiaberto. Nesta situação, acionando-se o controle novamente o portão volta a abrir (*i.e.*, cancelamento do fechamento).

Os casos de uso do sistema, relacionando os componentes e os requisitos funcionais, são apresentados na Figura 58.

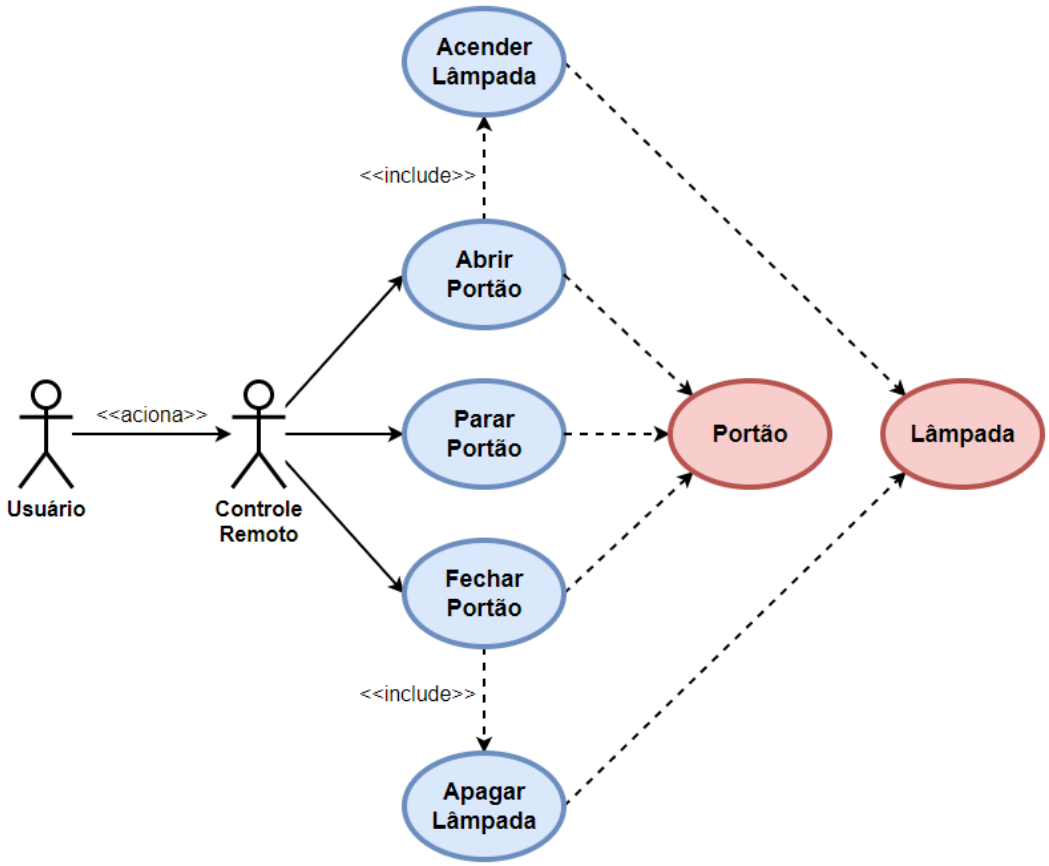


Figura 58 – Casos de uso do experimento do Portão Eletrônico.
Fonte: Adaptado de Xavier (2014)

Com base nos requisitos funcionais, os estados possíveis para o portão eletrônico podem ser descritos como uma máquina de estados, conforme apresentado na Figura 59. Em linhas gerais, o portão apresenta seis estados possíveis: Aberto, Fechando, Fechado, Abrindo,

Parou de Abrir e Parou de Fechar. A transição entre os estados ocorre por eventos relacionados à ação do usuário (acionamento do controle remoto) e/ou por eventos gerados por um *timer*.

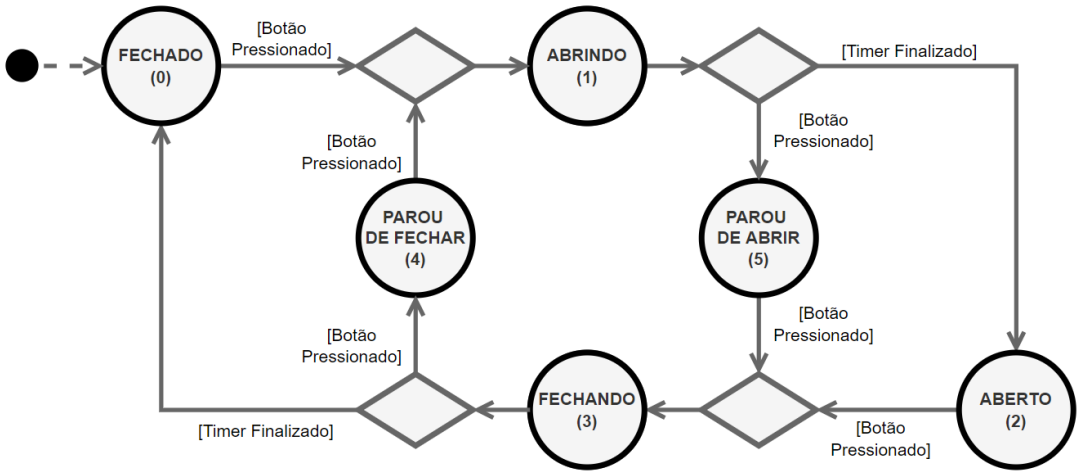


Figura 59 – Diagrama de estados do experimento do Portão Eletrônico.

Fonte: Adaptado de Xavier (2014)

4.2.1 Detalhes da Implementação em *Framework* PON C++ 4.0 IoT

A implementação do sistema segundo o PON foi realizada utilizando-se como base as *Rules* descritas por Xavier (2014), com uma adaptação: a adição de um novo *Method*, responsável por zerar (*resetar*) o registro do evento referente ao botão pressionado no controle remoto (*Control->mtResetEvent*) após esse evento ser processado pela central. Como os requisitos funcionais do sistema não diferenciam o tempo de pressionamento do botão, considerou-se desnecessário o processamento do evento de finalização, sendo este substituído por este novo *Method*.

As regras funcionais do sistema foram interpretadas em elementos do PON e são apresentadas no Quadro 11.

Quadro 11 – Descrição do sistema em PON para o Experimento Portão Eletrônico.

Rules	***Condition* e suas *Premises***	***Action* composta por suas *Instigations* e *Methods***	
rlOpeningGate	atGateState == CLOSED &&	in1	Gate→mtOpening
	atTimerState == ZERO &&	in2	Timer→mtStart
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlTurnOnLight	atGateState == CLOSED &&	in3	Light→mtOn
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlOpenedGate	atGateState == OPENING &&	in4	Gate→mtOpened
	atTimerState == FINISHED	in5	Timer→mtReset
rlOpeningStoppedGate	atGateState == STOP_CLOSING &&	in6	Gate→mtOpening
	atTimerState == ZERO &&	in2	Timer→mtStart
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlClosingGate	atGateState == OPENED &&	in8	Gate→mtClosing
	atTimerState == ZERO &&	in2	Timer→mtStart
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlTurnOffLight	atGateState == OPENED &&	in10	Light→mtOff
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlClosedGate	atGateState == CLOSING &&	in11	Gate→mtClosed
	atTimerState == FINISHED	in5	Timer→mtReset
rlClosingStoppedGate	atGateState == STOP_OPENING &&	in12	Gate→mtClosing
	atTimerState == ZERO &&	in2	Timer→mtStart
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlStopOpeningGate	atGateState == OPENING &&	in13	Gate→mtStopOpening
	atTimerState == RUNNING &&	in14	Timer→mtCancel
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent
rlStopClosingGate	atGateState == CLOSING &&	in15	Gate→mtStopClosing
	atTimerState == RUNNING &&	in14	Timer→mtCancel
	atEventState == REMOTE_CONTROL_ON	in16	Control→mtResetEvent

Fonte: Adaptado de Xavier (2014)

Para a implementação do sistema de forma distribuída, adotou-se a separação funcional dos elementos do sistema, por meio de quatro aplicações interagindo por meio de mensagens MQTT em um *broker*. Essa separação é exemplificada na Figura 60. Para a implementação via *Framework* PON C++ 4.0 IoT, optou-se por distribuir os *attributes atEventState*, *atLightState* e *atGateState*, simulando um controle remoto responsável pelo acionamento do sistema, uma lâmpada inteligente e o estado do portão, respectivamente. Especificamente, na Central controladora o *Attribute atEventState* foi implementado como *Attribute Proxy* Remoto representando uma referência para o evento de acionamento do botão do controle remoto, alterado em uma aplicação remota. Ainda na Central Controladora, os *Attributes atLightState* e *atGateState* foram implementados como *Attribute* Compartilhado via Rede pois, dessa forma, quando seus estados forem alterados as aplicações remotas interessadas são notificadas via rede.

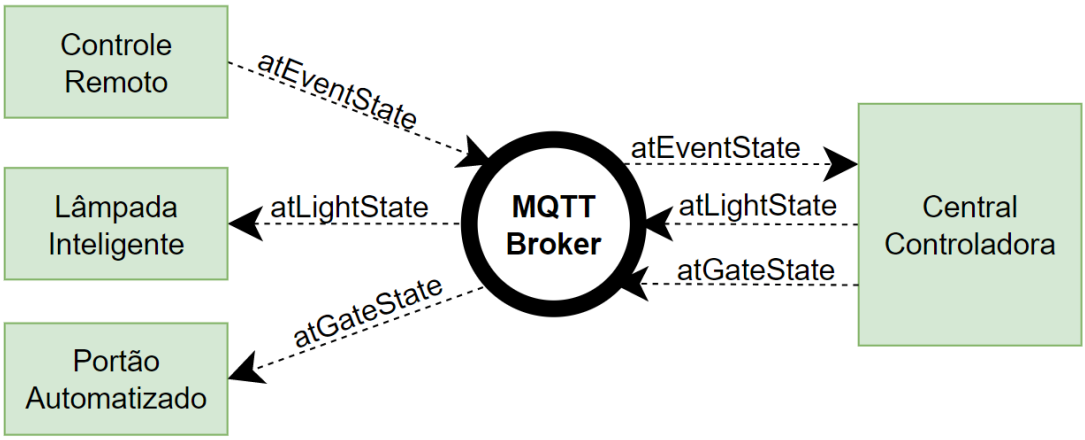


Figura 60 – Arquitetura do Experimento do Portão Eletrônico Distribuído.

Como exemplo, o Código 4.6 apresenta a declaração do *FBE GateStruct*, representando o portão automatizado. Esse *FBE* é composto por um *Attribute* Compartilhado via Rede do tipo Inteiro *atGateState* e por seis *Methods*. O *Attribute atGateState* indica o estado do portão conforme máquina de estados apresentada na Figura 59 e os *Methods* representam as ações sobre esse *Attribute*. Como *atGateState* é um *Attribute* Compartilhado via rede, quando os *Methods* alteram seu estado uma notificação é enviada para aplicações na forma de uma mensagem MQTT no tópico pertinente. Por exemplo, quando o *Method mtOpened* (linha 21) é executado altera-se o estado de *atGateState*, o qual passa a assumir o valor *GATE_STATE::OPENED* (linha 22), identificado pelo valor inteiro 2. Dessa forma, devido à declaração de *atGateState* como *Attribute* Compartilhado via rede, a alteração de seu estado provoca o envio de uma notificação via rede, disparada pelo *Framework* PON C++ 4.0 IoT. Neste caso, a notificação é representada pelo envio de uma mensagem no tópico *test/at/atGateState* com o conteúdo da mensagem 2.

Código 4.6 Exemplo de Código do FBE *GateStruct* do Experimento Portão Eletrônico

```
enum GATE_STATE
{
    CLOSED = 0,
    OPENING = 1,
    OPENED = 2,
    CLOSING = 3,
    STOP_OPENING = 4,
    STOP_CLOSING = 5
};

struct GateStruct : NOP::FBE
{
    explicit GateStruct(const std::string_view name) : FBE(name) {}

    //Declaração do Attribute compartilhado via Rede
    NOP::SharedAttribute<int> atGateState =
        NOP::BuildNetworkSharedAttribute("atGateState", 0);

    //Declaração dos seis Methods referentes à mudança
    //no estado do portão.
    NOP::Method mtOpened =
        METHOD(atGateState->SetValue(GATE_STATE::OPENED););
    NOP::Method mtClosed =
        METHOD(atGateState->SetValue(GATE_STATE::CLOSED););
    NOP::Method mtStopOpening =
        METHOD(atGateState->SetValue(GATE_STATE::STOP_OPENING););
    NOP::Method mtStopClosing =
        METHOD(atGateState->SetValue(GATE_STATE::STOP_CLOSING););
    NOP::Method mtOpening =
        METHOD(atGateState->SetValue(GATE_STATE::OPENING););
    NOP::Method mtClosing =
        METHOD(atGateState->SetValue(GATE_STATE::CLOSING););
};
```

Em complemento ao FBE *GateStruct*, é apresentado no fragmento de Código 4.7 um exemplo de declaração da *Rule rlOpeningGate*, previamente apresentada no Quadro 11. Neste código são apresentados também as *Premises*, *Conditions*, *Instigations* e *Actions* relacionadas a esta *Rule*.

Código 4.7 Exemplo de Código de uma *Rule* e suas referências para o Experimento do Portão Eletrônico

```
...
//----------------Premises
prGateClosed =
  BuildPremise<int>(gate.atGateState, CLOSED, Equal());

prRemoteControlOn =
  BuildPremise<int>(control.atEventState, REMOTE_CONTROL_ON, Equal());

prTimerZeroed =
  BuildPremise<int>(timer.atTimerState, ZERO, Equal());

//----------------Conditions
cnGateClosedControlOn =
  BuildCondition(CONDITION(*prGateClosed && *prRemoteControlOn),
    prGateClosed, prRemoteControlOn);

cnGateClosedControlOnTimerZeroed = BuildCondition(
  CONDITION(*cnGateClosedControlOn && *prTimerZeroed),
    cnGateClosedControlOn, prTimerZeroed);

//----------------Instigations
inOpenGate = BuildInstigation(gate.mtOpening);

inStartTimer = BuildInstigation(timer.mtStart);

inCleanRemoteControlEvent = BuildInstigation(control.mtCleanEvent);

//----------------Actions
acOpenAndStartTimer = BuildAction<Parallel>(
  inCleanRemoteControlEvent, inOpenGate, inStartTimer);

//----------------Rules
rlOpeningGate = BuildRule(cnGateClosedControlOnTimerZeroed,
  acOpenAndStartTimer);
...
```

4.2.2 Detalhes da Implementação em POE via *Pub/Sub* em C++

A implementação do sistema segundo o POE, tal qual no estudo anterior, também foi realizada utilizando-se o padrão *Publish/Subscribe* com comunicação indireta, por meio da utilização de um *broker*. A Figura 61 apresenta o diagrama de classes da solução implementada.

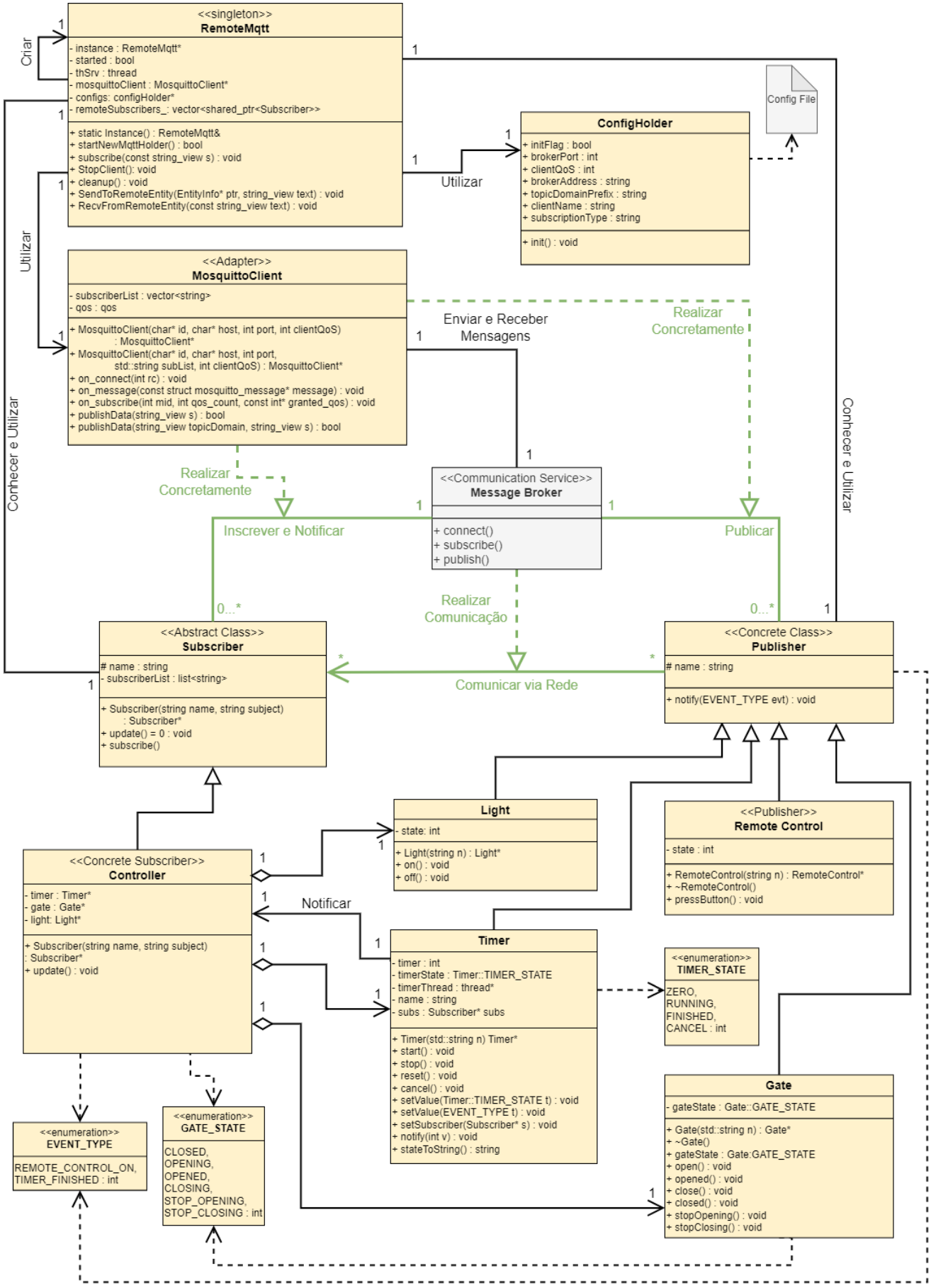


Figura 61 – Diagrama de classes da implementação do Portão Eletrônico no POE.

Observa-se, inicialmente, as classes responsáveis pela comunicação com o *broker* MQTT com base (com as devidas adaptações) nas estruturas adicionadas ao *Framework* PON C++ 4.0 IoT e previamente descritas na Seção 3.3. Essas classes (nomeadamente, *RemoteMQTT*, *MosquittoClient* e *ConfigHolder*) auxiliam na comunicação com o *broker* MQTT, isto é, por meio delas são enviadas e recebidas as mensagens MQTT.

Na sequência, observam-se os componentes do sistema representados por classes, nomeadamente: a Central Controladora (*Controller*), o controle remoto (*Remote Control*), o portão (*Gate*) e a lâmpada (*Light*). Com exceção da classe *Controller*, as demais classes são derivadas da classe *Publisher*. Dessa forma, os objetos dessas classes podem enviar uma notificação aos *subscribers* (por meio do *broker*) quando o estado de seus atributos internos for alterado. Por exemplo, ao se alterar o estado do portão (atributo *gateState*, da classe Gate), uma notificação contendo o estado do portão (representada pelo enum GATE_STATE) é enviada para os respectivos *subscribers*.

Por sua vez, a classe *Controller* deriva da interface *Subscriber* e implementa a lógica de controle dos eventos dentro do método *Update*. O método *Update* é invocado quando uma notificação é recebida em decorrência de um evento ocorrido em um dos objetos de interesse. Para a central controladora, são considerados eventos de interesse: (i) os eventos referentes ao botão pressionado, representado pelo enum *EVENT_TYPE REMOTE_CONTROL_ON* e gerado por objetos da classe *Remote Control*); (ii) os eventos referentes à finalização do time *timer*, representados pelo enum *TIMER_FINISHED* e gerado por objetos da classe *Timer*.

Destaca-se que, por ser um objeto interno da central controladora, optou-se por implementar um *Timer* com notificação direta para a classe *Controller*.

O Código 4.8 apresenta o método *Update*, o qual implementa a rotina de tratamento dos eventos recebidos na classe *Controller*. Observa-se que as possíveis condições são implementadas por meio de estruturas de *switch-case*, cada uma representando uma situação possível do portão. As alterações do sistema ocorrem por meio da chamada de métodos dos objetos *gate*, *timer* e *light* (instâncias das classes *Gate*, *Timer* e *Light*, respectivamente). Esses métodos alteram o estado dos atributos internos dos objetos, disparando notificações (publicações) que podem ser processadas pelos *subscribers* pertinentes.

Código 4.8 Código do método *Update* da classe *Controller* do Experimento Portão Eletrônico

```
...
void Controller::update(EVENT_TYPE _eventType) {
  switch (getGateState()) {
    case (Gate::CLOSED): {
      switch (_eventType) {
      case (REMOTE_CONTROL_ON): {
        gate->open(); light->on(); timer->start(); break;}
      default: { break; }
    }} break;

    case (Gate::OPENING): {
      switch (_eventType) {
      case (REMOTE_CONTROL_ON): {
        gate->stopOpening(); timer->cancel(); break; }
      case (TIMER_FINISHED): {
        gate->opened(); timer->reset(); break; }
      default: { break; }
    }} break;

    case (Gate::OPENED): {
      switch (_eventType) {
      case (REMOTE_CONTROL_ON): {
        gate->close(); light->off(); timer->start(); break;}
      default: { break; }
    }} break;

    case (Gate::CLOSING): {
     switch (_eventType) {
     case (REMOTE_CONTROL_ON): {
        gate->stopClosing(); timer->cancel(); break; }
      case (TIMER_FINISHED): {
        gate->closed(); timer->reset(); break; }
      default: { break; }
    }} break;

    case (Gate::STOP_OPENING): {
      switch (_eventType) {
      case (REMOTE_CONTROL_ON): {
        gate->close(); timer->start(); break; }
      default: { break; }
    }} break;

    case (Gate::STOP_CLOSING): {
      switch (_eventType) {
      case (REMOTE_CONTROL_ON): {
        gate->open(); timer->start(); break;}
      default: { break; }
    }} break;
  }
}
...
```

4.2.3 Descrição das Métricas

Conforme também observado por Xavier (2014), a avaliação de desempenho do experimento do portão eletrônico para tratativas de eventos pode não ser coerente, pois a simples geração de eventos continuamente e repetidamente em sequência e sem intervalos significativos entre os eventos não testaria todos os estados possíveis. Por exemplo, para eventos sendo gerados continuamente e sem um intervalo entre eles o ciclo de estados do Portão Eletrônico ficaria alterando entre <ABRINDO, PAROU DE ABRIR, FECHANDO, PAROU DE FECHAR>. Por esse motivo, optou-se pela avaliação funcional do cumprimento dos requisitos funcionais por meio da execução de casos de teste.

Para a verificação do funcionamento do sistema, optou-se por implementar testes funcionais de integração com foco nos principais casos de uso do sistema. A implementação e execução dos testes é realizada por meio da publicação e verificação de mensagens MQTT, conforme a metodologia de testes de integração de sistemas previamente apresentada na Seção 3.3.3.2. Para este experimento, o *framework* de testes é responsável por publicar mensagens referentes ao acionamento do controle remoto (*atEventState*) e verificar o correto recebimento de mensagens referentes ao acionamento da lâmpada (*atLightState*) e do portão (*atGateState*).

Com base nos requisitos funcionais e principais casos de uso para o sistema do portão eletrônico, são descritos quatro casos de teste:

- **Caso de teste 1**: dado que o portão se encontra fechado e com as luzes apagadas, quando o usuário pressionar o botão do controle remoto, então o portão deverá iniciar a abertura, a lâmpada deverá ser acesa e, após 30 segundos, o portão deverá estar aberto.

- **Caso de teste 2**: dado que o portão se encontra aberto e com as luzes acesas, quando o usuário pressionar o botão do controle remoto, então o portão deverá iniciar o fechamento, a lâmpada deverá ser apagada e, após 30 segundos, o portão deverá estar fechado.

- **Caso de teste 3**: dado que o portão se encontra fechado, quando o usuário pressionar o botão do controle remoto, então o portão deverá começar a abrir. Durante a abertura, quando o usuário pressionar o botão do controle remoto, então o portão deverá parar a abertura. Quando o usuário pressionar novamente o botão do controle remoto, então o portão deverá realizar o fechamento e, após 30 segundos, o portão deverá estar fechado.

- **Caso de teste 4**: dado que o portão se encontra aberto, quando o usuário pressionar o botão do controle remoto, então o portão deverá começar a fechar. Durante o fechamento, quando o usuário pressionar o botão do controle remoto, então o portão deverá

parar o fechamento. Quando o usuário pressionar novamente o botão do controle remoto, então o portão deverá realizar a abertura e, após 30 segundos, o portão deverá estar aberto.

Como exemplo de teste, apresenta-se o Código 4.9 que implementa, em linguagem Python, o Caso de Teste 1, conforme *framework* de testes de integração de sistemas e metodologia previamente apresentada na Seção 3.3.3.2.

Código 4.9 Exemplo de código implementando o Caso de Teste 1 para o Experimento do Portão Eletrônico

```
def test_open_gate(self):
    # Armazenando Valores de Referência
    initValues = copy.deepcopy(topicValues)

    # Publicação de mensagem referente ao acionamento do controle remoto
    client.publish("test/at/atEventState/", "1", qos=2)
    time.sleep(1)

    # Verificação da alteração do estado do portão
    # e do estado da lâmpada
    self.assertEqual(topicValues["test/at/atGateState/"][1],
        initValues["test/at/atGateState/"][1] + 1) # Abrindo
    self.assertEqual(topicValues["test/at/atLightState/"][1],
        initValues["test/at/atLightState/"][1] + 1)  # Light On

    time.sleep(30)

    # Verificação, após 30 segundos, do estado do portão
    self.assertEqual(topicValues["test/at/atGateState/"][2],
        initValues["test/at/atGateState/"][2] + 1) # Aberto

    # Publicação de mensagem referente ao acionamento do controle remoto
    client.publish("test/at/atEventState/", "1", qos=2)
    time.sleep(1)

    # Verificação da alteração do estado do portão
    # e do estado da lâmpada
    self.assertEqual(topicValues["test/at/atGateState/"][3],
        initValues["test/at/atGateState/"][3] + 1) # Fechando
    self.assertEqual(topicValues["test/at/atLightState/"][0],
        initValues["test/at/atLightState/"][0] + 1)  # Light Off

    time.sleep(30)

    # Verificação, após 30 segundos, do estado do portão
    self.assertEqual(topicValues["test/at/atGateState/"][0],
        initValues["test/at/atGateState/"][0] + 1) # Fechado
```

Além dos requisitos funcionais, observaram-se também aspectos não funcionais relacionados ao uso da rede de comunicação durante os testes. Para essa métrica, considerou-se o pior caso com relação à rede, utilizando-se QoS 2 no MQTT durante a troca de mensagens.

Relembra-se que o QoS 2 garante a entrega de somente uma mensagem utilizando-se de um *handshake* de quatro etapas (previamente apresentado na Seção 2.1.1.1). Ademais, observou-se o tráfego de mensagens em duas abordagens: (i) considerando apenas as mensagens da camada de aplicação (MQTT) e (ii) considerando as mensagens das camadas de aplicação (MQTT) e de transporte (TCP). Para a coleta e análise das mensagens utilizou-se o software *Wireshark*.

4.2.4 Resultados da implementação em *Framework* PON C++ 4.0 IoT

O resultado geral da execução dos casos de teste para a implementação em *Framework* PON C++ 4.0 IoT é apresentado na Figura 62. Observa-se que os quatro testes propostos foram executados com sucesso em 253,17 segundos.

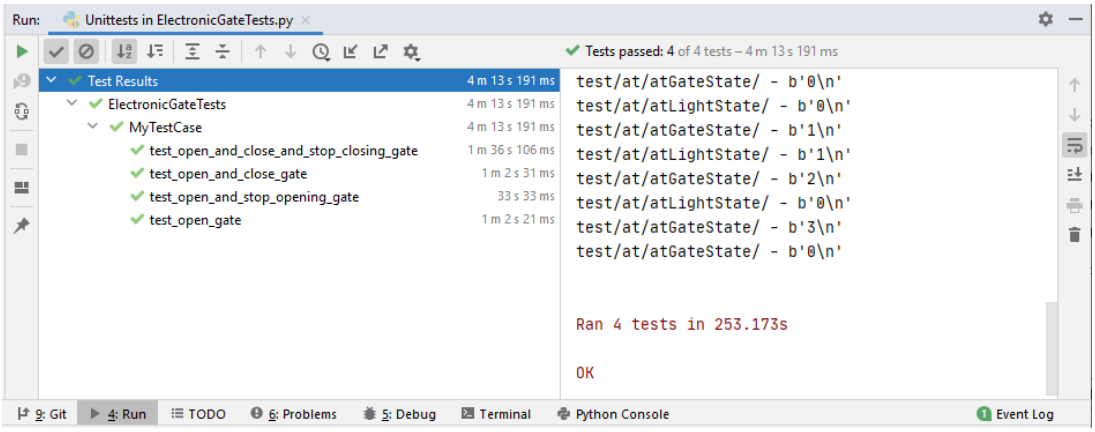


Figura 62 – Resultados dos testes de integração do Experimento do Portão Eletrônico em *Framework* PON C++ 4.0 IoT.

Como exemplo de evidência de teste, as mensagens MQTT capturadas para o Caso de Teste 1 são apresentadas nas Figuras 63 e 64. Na Figura 63 são apresentadas as mensagens capturadas via software *wireshark*, na qual é possível observar inicialmente as mensagens de inscrição nos respectivos tópicos. Em seguida, na mensagem destacada como número 211, é possível observar a publicação da mensagem respectiva ao acionamento do botão no controle remoto pelo *framework* de testes. Na sequência, nas mensagens de número 239 e 241, observa-se a ação relativa ao acionamento da lâmpada e da mudança do estado do portão, respectivamente. Após 30 segundos, o estado do portão é novamente alterado, conforme recebimento da mensagem número 423. Na sequência, um novo acionamento do botão do controle remoto é realizado (mensagem 443), iniciando o ciclo de fechamento do portão (mensagens 471 e 697) e o desligamento da lâmpada (mensagem 473).

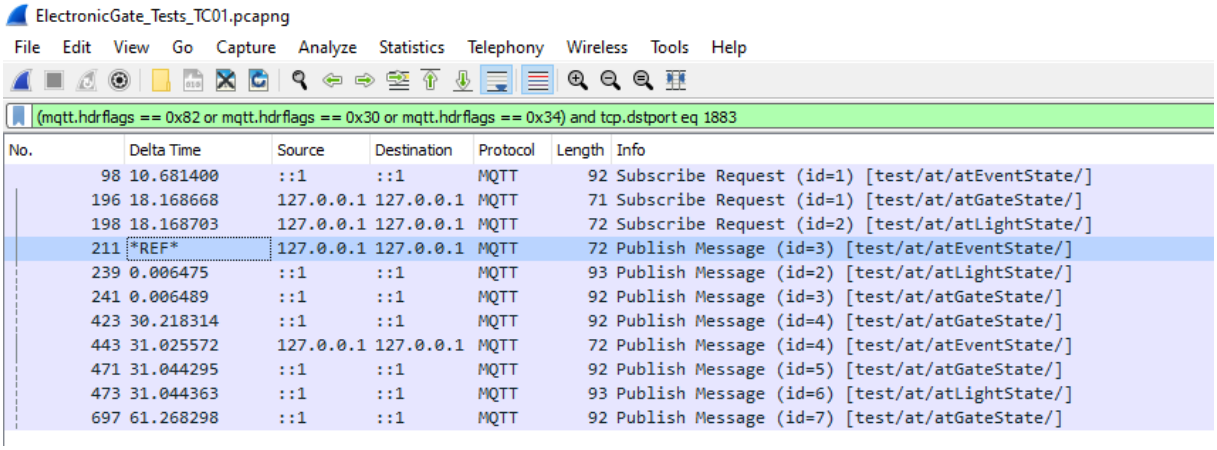


Figura 63 – Mensagens de aplicação capturadas com o Wireshark para o Caso de Teste 1 do Experimento Portão Eletrônico em *Framework* PON C++ 4.0 IoT.

Em complemento, na Figura 64 é apresentado o conteúdo das mensagens impressas no console do *framework* de testes, na mesma ordem apresentada previamente. Destacam-se as mensagens referentes ao estado do portão (*atGateState*) alterado no ciclo de abertura e fechamento, conforme a máquina de estados descrita previamente (Figura 59).

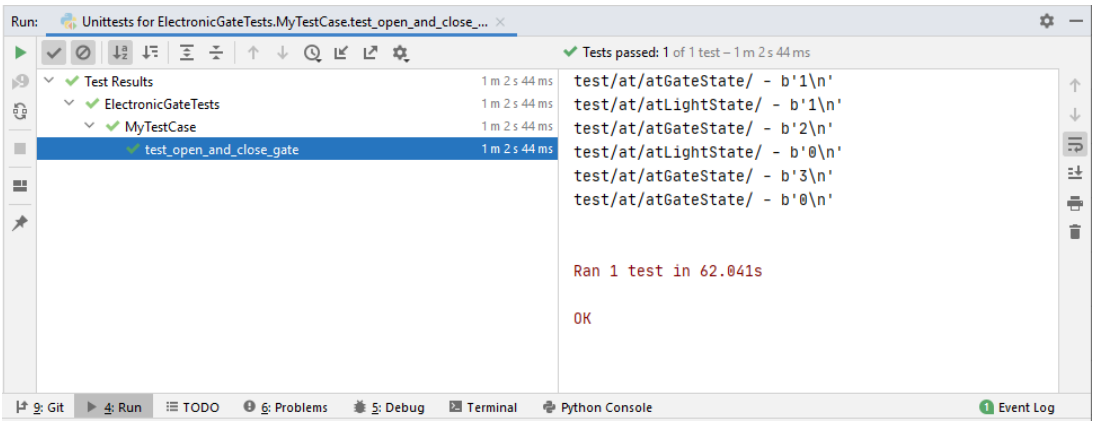


Figura 64 – Mensagens impressas no Console para o Caso de Teste 1 do Experimento Portão Eletrônico em *Framework* PON C++ 4.0 IoT.

Com relação ao uso da rede de comunicação durante os testes, utilizou-se como filtro a porta utilizada pela aplicação PON para comunicação com o *broker*. As Figuras 65 e 66 apresentam as mensagens originadas pela aplicação em PON. A Figura 65 apresenta, de forma gráfica, os pacotes recebidos e enviados pela aplicação durante a execução do Caso de Teste 1, utilizando-se a porta 52223. Observa-se que as mensagens são trocadas somente nos momentos de transição do estado do portão, disparada pelo evento relacionado ao acionamento do botão no controle remoto (*atEventState*). Observa-se também que além das mensagens da aplicação, existem também mensagens adicionais provocadas pelo protocolo de transporte utilizado, nomeadamente o TCP.

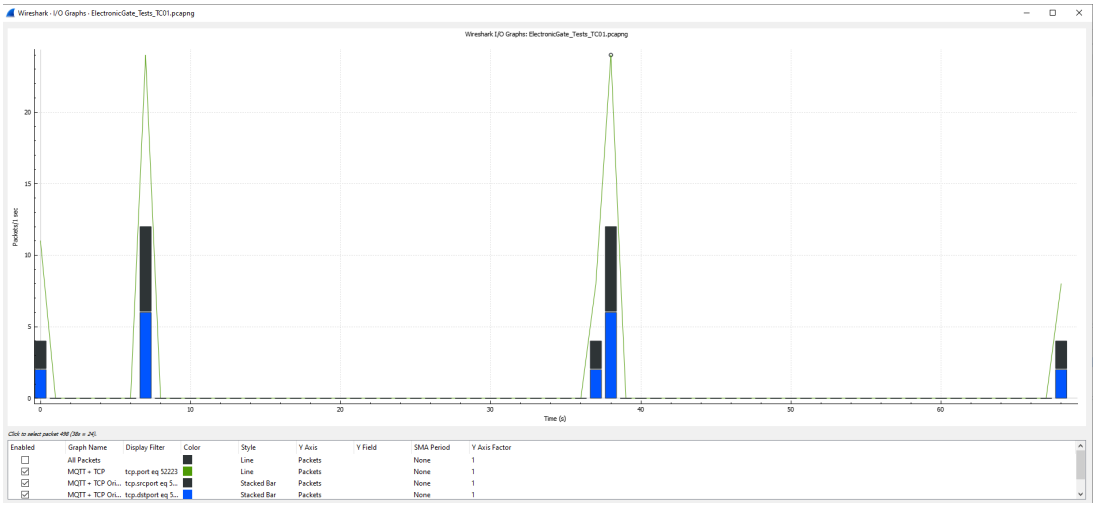


Figura 65 – Mensagens trafegadas pela rede no decorrer do tempo para o Caso de Teste 1 do Experimento Portão Eletrônico em *Framework* PON C++ 4.0 IoT.

Em virtude da presença da lâmpada inteligente e dos eventos relativos ao estado do portão no sistema, o número de mensagens será variável de acordo com a sequência dos eventos gerados pelo controle remoto. Adotando-se a mesma métrica utilizada por Barreto, Vendramin e SIMÃO (2018), a qual leva em consideração somente as mensagens de aplicação geradas pelo controle remoto, observa-se que são necessárias **4N** mensagens para cada publicação de alteração do estado do controle remoto, onde **N** representa a quantidade de eventos de acionamentos do botão. Como exemplo, consideram-se as mensagens referentes ao Caso de Teste 1, apresentado na Figura 66.

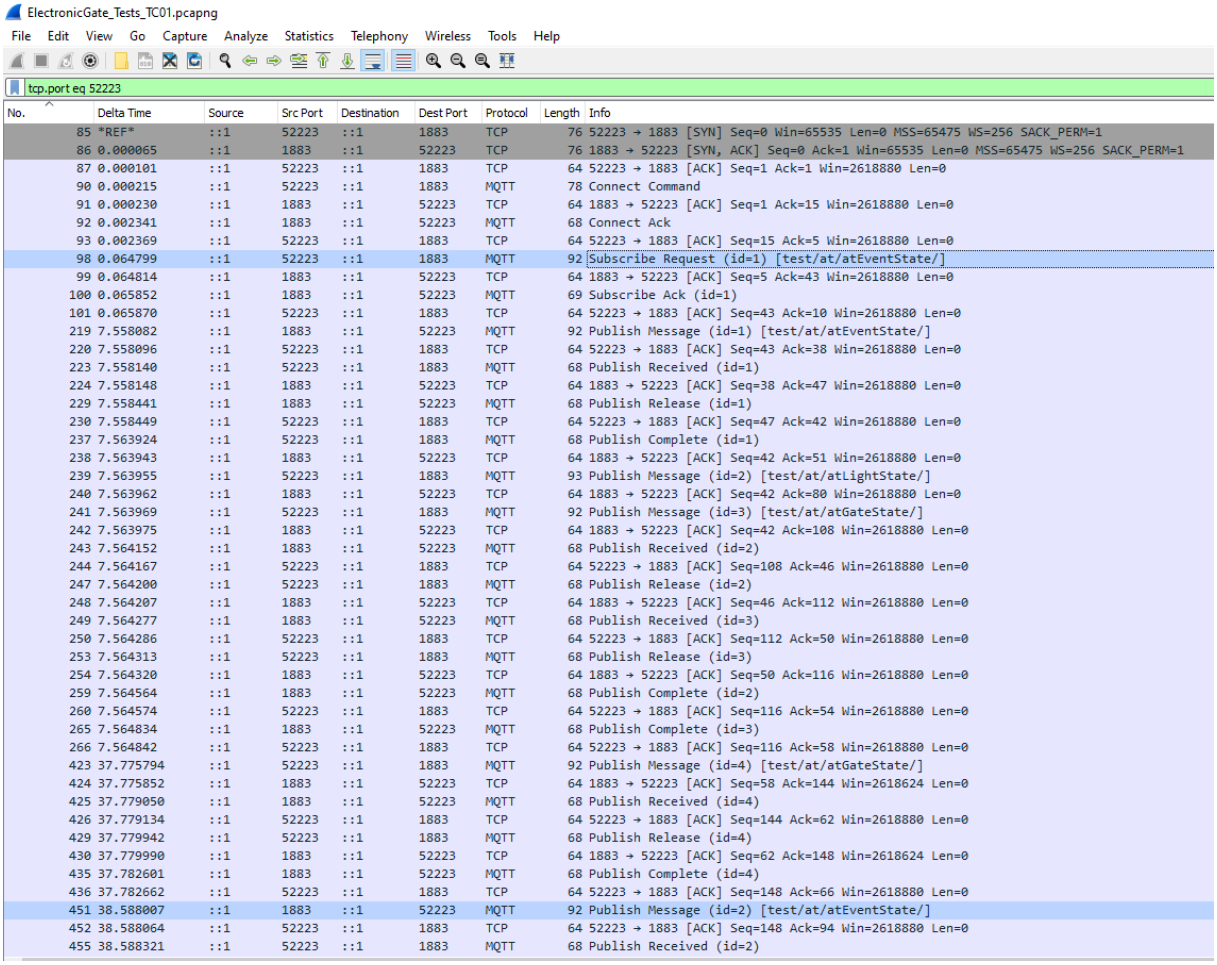


Figura 66 – Mensagens de transporte e aplicação capturadas com o Wireshark para o Caso de Teste 1 do Experimento Portão Eletrônico em PON.

Neste exemplo, são necessárias duas mensagens para conexão do cliente no *broker* (*Connect Command* e *Connect Ack*, representadas pelos pacotes de número 90 e 92, respectivamente) além de duas mensagens para inscrição no respectivo tópico (*Subscribe Request* e *Subscribe Ack), pacotes 98 e 100, respectivamente*. Em seguida, são apresentadas as mensagens referentes à primeira publicação de alteração do estado do sensor, representadas pelos pacotes de número 219, 223, 229 e 237. Dessa forma, totalizam-se **4N+4** mensagens considerando apenas as mensagens da camada de aplicação (MQTT).

Com relação ao número de mensagens necessárias considerando-se as camadas de aplicação (MQTT) e de transporte (TCP), observa-se que durante a inicialização da conexão são necessárias três mensagens representando o *handshake* do protocolo TCP (*SYN*, *SYN Ack* e *Ack* referentes aos pacotes 85, 86 e 87, respectivamente). Em seguida, para cada mensagem de aplicação é gerada uma nova mensagem TCP, referente à confirmação de recebimento da mensagem *TCP Ack* do protocolo de transporte. Como exemplo, o pacote 91 que representa o *TCP Ack* para a operação de conexão do protocolo MQTT (*Connect Command*, pacote 90). Dessa forma, considerando-se ambas as camadas, são necessárias **8N+7** mensagens.

4.2.5 Resultados da implementação em POE via *Pub/Sub* em C++

O resultado geral da execução dos casos de teste para a implementação em POE *Pub/-Sub* em C++ são apresentados na Figura 67. Destaca-se que os casos de teste são os mesmos previamente verificados na implementação em PON. Por facilidade, mantiveram-se inclusive os prefixos "test/at"na nomenclatura dos tópicos e também os mesmos nomes dos elementos correspondentes da implementação em PON (por exemplo, *atGateState*). Observa-se que os quatro testes propostos foram executados com sucesso e em 253,15 segundos.

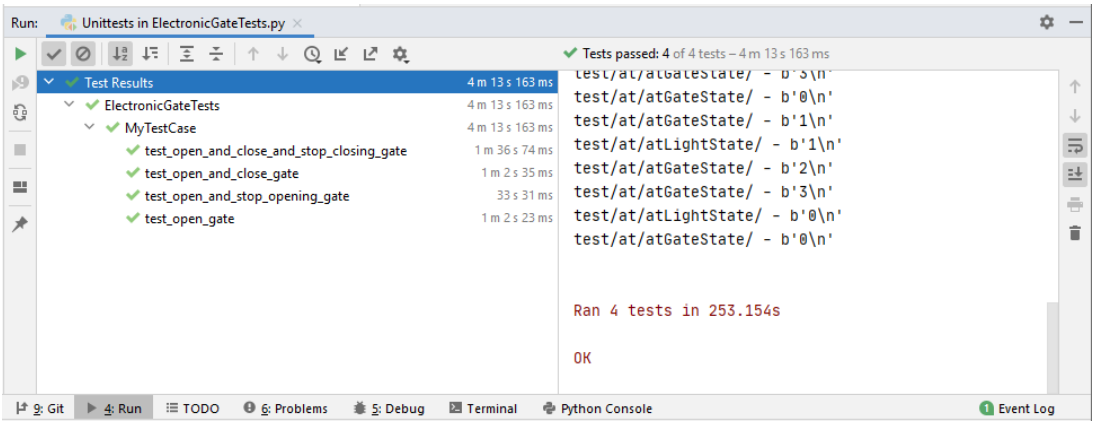


Figura 67 – Resultados dos testes de integração do Experimento do Portão Eletrônico no POE *Pub/Sub* em C++.

Como exemplo de evidência de teste, as mensagens MQTT capturadas para o Caso de Teste 1 são apresentadas nas Figuras 68 e 69. Na Figura 68 são apresentadas as mensagens capturadas via software *wireshark*. É possível observar inicialmente as mensagens de conexão (pacotes identificados pelos números 173 e 175) e de inscrição no respectivo tópico (pacotes 183 e 185). Em seguida, na mensagem destacada como número 398, é possível observar a publicação da mensagem respectiva ao acionamento do botão no controle remoto pelo *framework* de testes. Na sequência, nas mensagens de número 416 e 418, observa-se a ação relativa ao acionamento da lâmpada e da mudança do estado do portão, respectivamente. Após 30 segundos, o estado do portão é novamente alterado, conforme recebimento da mensagem número 878. Na sequência, um novo acionamento do botão do controle remoto é realizado (mensagem 904), iniciando o ciclo de fechamento do portão (mensagens 922 e 1367) e a lâmpada é desligada (mensagem 924).

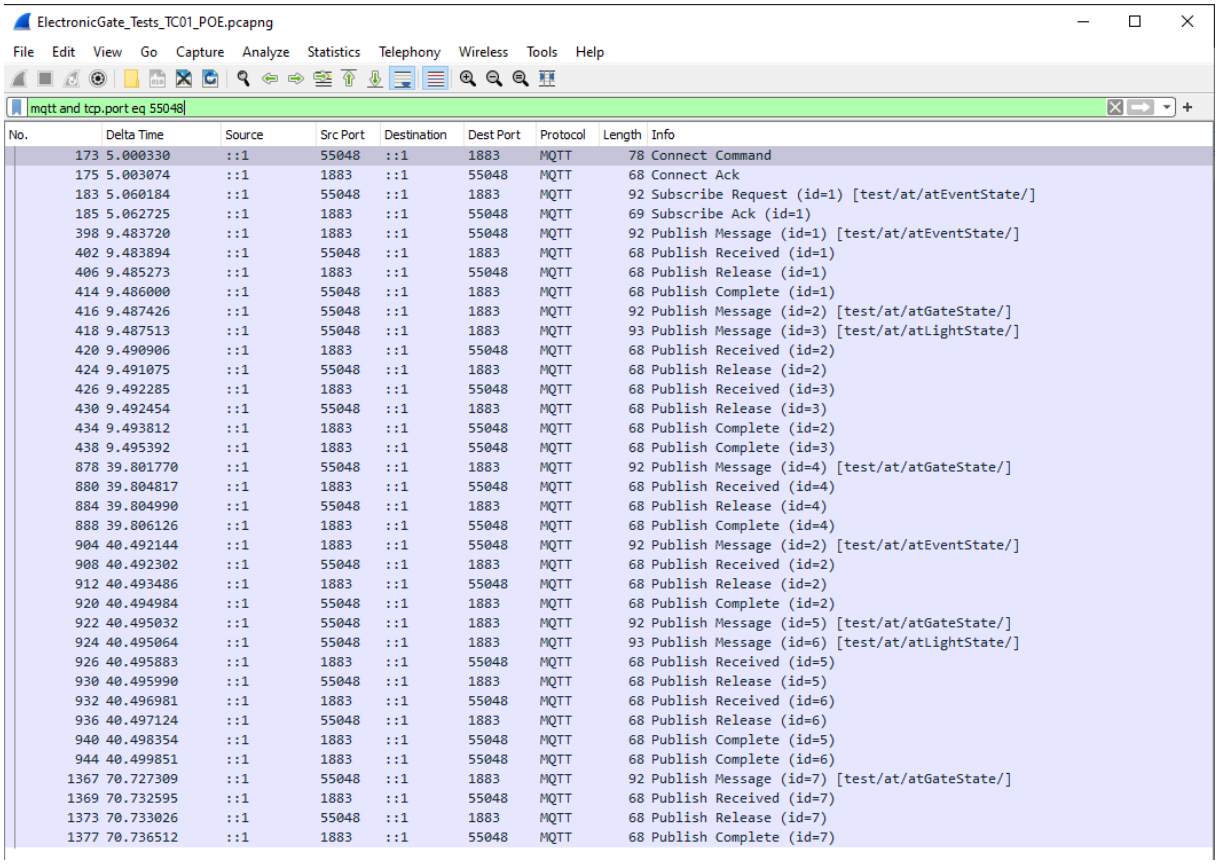


Figura 68 – Mensagens de aplicação capturadas com o Wireshark para o Caso de Teste 1 do Portão Eletrônico em POE *Pub/Sub* em C++.

Em complemento, na Figura 69 é apresentado o conteúdo das mensagens impressas no console do *framework* de testes, na mesma ordem apresentada previamente. Destacam-se as mensagens referentes ao estado do portão (*atGateState*) alterado no ciclo de abertura e fechamento, conforme a máquina de estados descrita previamente (Figura 59).

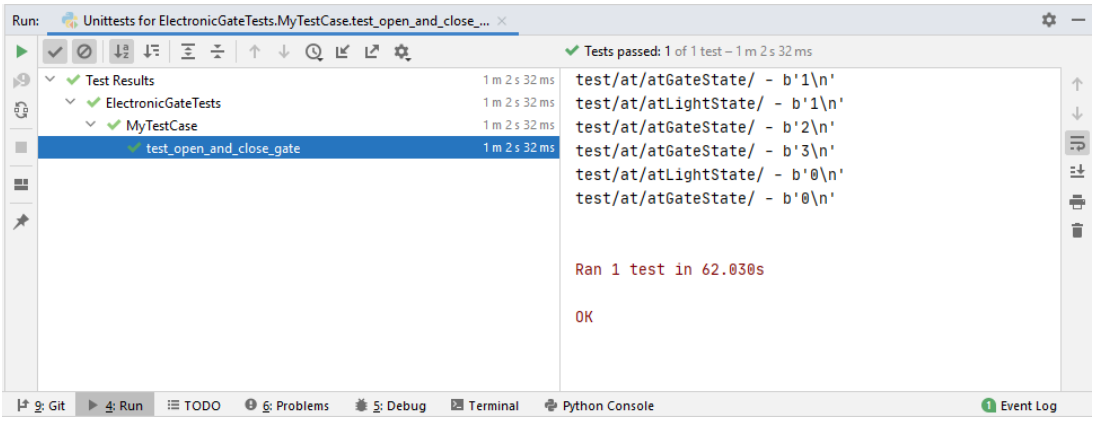


Figura 69 – Mensagens impressas no Console para o Caso de Teste 1 do Portão Eletrônico em POE *Pub/Sub* em C++.

Com relação ao uso da rede de comunicação durante os testes, utilizou-se como filtro a porta utilizada pela aplicação POE para comunicação com o *broker*. A Figura 70 apresenta de forma gráfica os pacotes recebidos e enviados pela aplicação durante a execução do Caso de

Teste 1, utilizando-se a porta 55048. Conforme também observado para a implementação em PON, observa-se que as mensagens são trocadas somente nos momentos de transição do estado do portão, disparada pelo evento relacionado ao acionamento do botão no controle remoto (*atEventState*). Observa-se também que além das mensagens da aplicação, existem também mensagens adicionais provocadas pelo protocolo de transporte utilizado, nomeadamente o TCP.

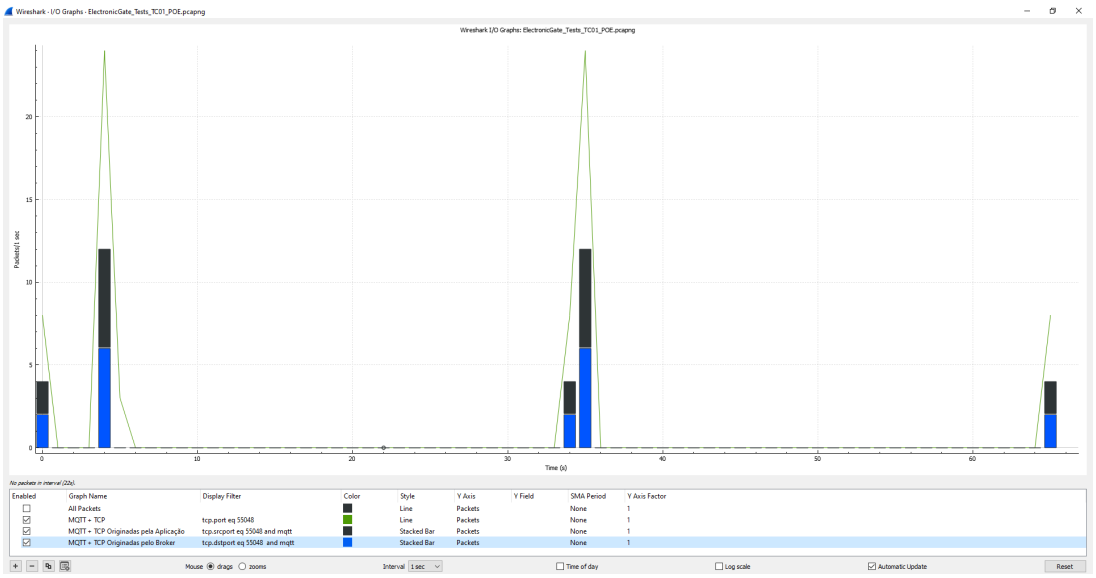


Figura 70 – Mensagens trafegadas pela rede no decorrer do tempo para o Caso de Teste 1 do Portão Eletrônico em POE *Pub/Sub* em C++.

Em virtude da presença da lâmpada inteligente e dos eventos relativos ao estado do portão no sistema, o número de mensagens será variável de acordo com a sequência dos eventos gerados pelo controle remoto. Adotando-se a mesma métrica previamente utilizada para a aplicação em PON, (considerando somente as mensagens de aplicação geradas pelo controle remoto), observa-se que são necessárias também **4N** mensagens para cada publicação de alteração do estado do controle remoto, onde **N** representa a quantidade de eventos de acionamentos do botão. Como exemplo, consideram-se as mensagens referentes ao Caso de Teste 1, apresentado na Figura 71.

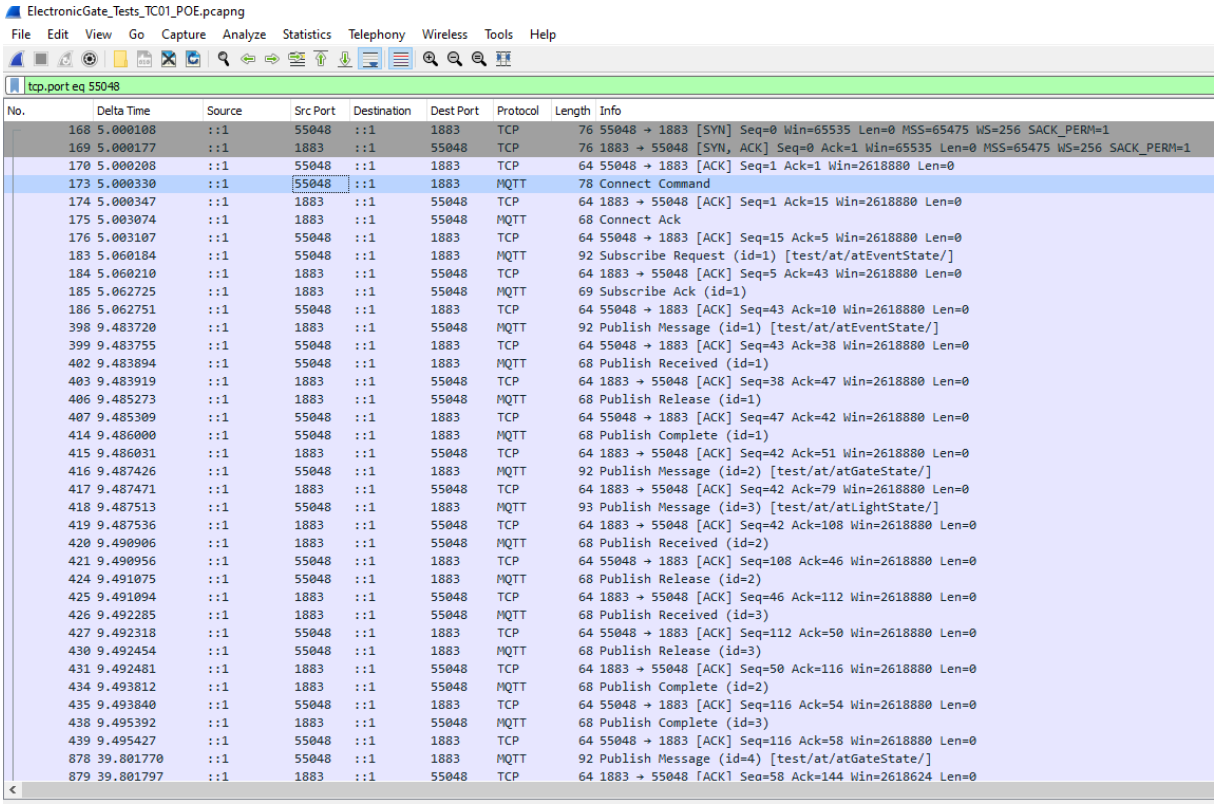


Figura 71 – Mensagens de transporte e aplicação capturadas com o Wireshark para o Caso de Teste 1 do Portão Eletrônico em POE *Pub/Sub* em C++.

Neste exemplo, são necessárias duas mensagens para conexão do cliente no *broker* (*Connect Command* e *Connect Ack*, representadas pelos pacotes de número 173 e 175, respectivamente) além de duas mensagens para inscrição no respectivo tópico (*Subscribe Request* e *Subscribe Ack), pacotes 183 e 185, respectivamente*. Em seguida, são apresentadas as mensagens referentes à primeira publicação de alteração do estado do controle, representadas pelos pacotes de número 398, 402, 406 e 414. Dessa forma, totalizam-se **4N+4** mensagens considerando apenas as mensagens da camada de aplicação (MQTT).

Com relação ao número de mensagens necessárias considerando-se as camadas de aplicação (MQTT) e de transporte (TCP), observa-se que durante a inicialização da conexão são necessárias três mensagens representando o *handshake* do protocolo TCP (*SYN*, *SYN Ack* e *Ack* referentes aos pacotes 168, 169 e 170, respectivamente). Em seguida, para cada mensagem de aplicação é gerada uma nova mensagem TCP, referente à confirmação de recebimento da mensagem *TCP Ack* do protocolo de transporte. Como exemplo, o pacote 174 representa o *TCP Ack* para a operação de conexão do protocolo MQTT (*Connect Command* do pacote 173). Dessa forma, considerando-se ambas as camadas, são necessárias **8N+7** mensagens.

4.2.6 Considerações do Experimento

Com relação à implementação em PON, observa-se, a partir do experimento do portão eletrônico, que o *Framework* PON C++ 4.0 IoT se mostrou também adequado a uma aplicação

já previamente implementada no contexto de sistemas distribuídos em PON ((BARRETO; VENDRAMIN; SIMÃO, 2018)). Os resultados apresentados pelos casos de teste mostram que os principais casos de uso foram implementados com sucesso. Observa-se também que a implementação das *Rules* em PON é muito próxima à descrição das regras em alto nível do sistema, o que tende a facilitar o seu desenvolvimento.

Com relação à implementação em POE *Pub/Sub* em C++, observa-se que o mecanismo reativo do POE (por meio do *publish/subscribe*) elimina certas redundâncias estruturais e temporais presentes naturalmente no POO. Por meio desse paradigma, as avaliações são realizadas somente quando os estados de interesse são alterados. Porém, as avaliações realizadas quando um estado é alterado podem não ser precisas, isto é, elas avaliam um conjunto de variáveis a cada ciclo, mesmo que algumas destas não tenham sido alteradas pelo evento recebido, podendo permanecer ainda redundâncias temporais. Neste exemplo, é possível observar, em certa medida, a expressividade das regras funcionais no código do sistema descritas de forma objetiva por estruturas de *switch-case*. Observa-se, porém, que caso novas regras precisem ser adicionadas ao sistema, a combinação de condições cobertas pelas estruturas de *switch-case* (eventos e estados atuais) pode aumentar de forma exponencial, tornando-se inviável em alguns casos. Ademais, destaca-se que é necessário uma estrutura de classes relativamente complexa para um problema relativamente simples, o que em partes pode ser entendido como *over-engineering*, ou seja, o uso de técnicas mais complexas que o necessário para resolver problemas simples (XAVIER, 2014).

Sobre os aspectos não funcionais, observa-se que ambas as implementações (*Framework* PON C++ 4.0 IoT e POE *Pub/Sub* em C++) necessitaram, no pior caso, de **8N+7** mensagens referentes aos eventos do portão onde **N** representa a quantidade de acionamentos do botão do controle remoto. Neste contexto, destaca-se que o presente trabalho utilizou o protocolo TCP em conjunto com o MQTT com QoS 2, o qual garante a entrega de exatamente uma mensagem (sem duplicações) para a aplicação. Caso seja utilizado o QoS 0 (sem garantia de entrega), a quantidade de mensagens diminui consideravelmente, passando para **N+4** mensagens de aplicação e **2N+7** mensagens de aplicação e transporte. Em comparação, o trabalho de Barreto, Vendramin e SIMÃO (2018) apresenta duas implementações das quais a implementação em PON (via *Attributes* Distribuídos) utilizou **2N** mensagens enquanto a implementação em POO (baseado em RPC) utilizou **N** mensagens, considerando-se o protocolo de transporte UDP. Destaca-se que o protocolo UDP não possui garantia de entrega das mensagens e, com isso, não apresenta também *overhead* de mensagens na comunicação. Destaca-se também, que a implementação utilizada neste presente trabalho difere da implementação utilizada por Barreto, Vendramin e SIMÃO (2018). Em referência ao evento do portão, considerando que uma única ação é possível (botão pressionado) e esta independe do tempo de duração (tempo de pressionamento do botão) adicionou-se um novo *Method* responsável por zerar o estado do evento toda vez que este for recebido (*mtResetEvent*). Dessa forma, não é necessário que seja

enviada a detecção e a não detecção do botão pressionado, como era o caso do experimento apresentado por Barreto, Vendramin e SIMÃO (2018).

4.3 Aplicação IoT para Casas Inteligentes

Como terceiro experimento, apresenta-se uma aplicação de IoT em um contexto simulado inspirado no domínio de casas inteligentes ou, do termo em inglês, *Smart homes*. Alguns trabalhos referem-se também ao termo como casa automatizada (DEORE; SONAWANE; SATPUTE, 2015) (WALEED *et al.*, 2018). Uma Casa Inteligente pode ser descrita como um ambiente residencial provido de tecnologia para prover, por exemplo, conveniência, conforto, segurança e/ou eficiência energética para seus moradores (DEORE; SONAWANE; SATPUTE, 2015). É pertinente ressaltar que, atualmente, um dos principais pilares das casas inteligentes está na eficiência energética. Esta eficiência se dá por meio do controle de iluminação, refrigeração, aquecimento, entre outros. O trabalho de Moser, Harder e Koo (2014), por exemplo, aponta ganhos significativos na redução de consumo de energia elétrica e de água após a implantação e uso de tecnologias para Casas Inteligentes.

A arquitetura de uma Casa Inteligente típica pode ser dividida em três grandes partes: Ambiente Doméstico (*Home Environment*), *Gateway* Doméstico (*Home Gateway*) e Ambiente Remoto (*Remote Environment*) (DEORE; SONAWANE; SATPUTE, 2015). O Ambiente Doméstico compreende os sensores e atuadores instalados no ambiente. Por sua vez, o *gateway* doméstico compreende a conexão do ambiente doméstico com o ambiente remoto. Por fim, o ambiente remoto pode ser descrito como o acesso, monitoramento e controle do ambiente doméstico remotamente.

O presente trabalho propõe quatro sistemas principais para implementação de um ambiente simulado inspirado em uma casa inteligente, descritos na sequência. É pertinente ressaltar que os sistemas e também as descrições apresentadas não são exaustivas, podendo ser incrementados dependendo dos requisitos desejados e/ou recursos disponíveis.

- **Sistema de detecção de invasores**: responsável pela detecção de pessoas com comportamento fora do padrão dos moradores da residência. Faz uso de sensores de presença, temperatura, pressão, luminosidade, câmeras e microfones para detecção. Utiliza alto falantes e *displays* para emissão de alertas e controladores de carga para acionamento de luzes remotamente;

- **Sistema de eficiência energética**: responsável por promover o uso consciente dos recursos energéticos por meio do correto dimensionamento de energia de acordo com as condições do ambiente;

- **Sistema de bem-estar**: responsável por promover o conforto dos moradores por meio da correta configuração de ambientes em termos de luminosidade, temperatura e umidade;
- **Sistema de proteção individual e coletiva**: responsável por detectar anomalias prejudiciais aos moradores do imóvel como vazamento de gás, fogo ou inundações.

Em termos práticos, os sistemas dos ambientes remotos utilizam os dispositivos IoT que são equipados com sensores e/ou atuadores para interagir com o ambiente doméstico. Conforme o exemplo apresentado na Figura 72, os sistemas podem fazer uso de um ou mais sensores, inclusive compartilhando sensores eventualmente. Por exemplo, sistemas de detecção de invasores podem utilizar sensores de presença nos ambientes, os quais também são utilizados por sistemas de eficiência energética, bem-estar ou proteção individual e coletiva.

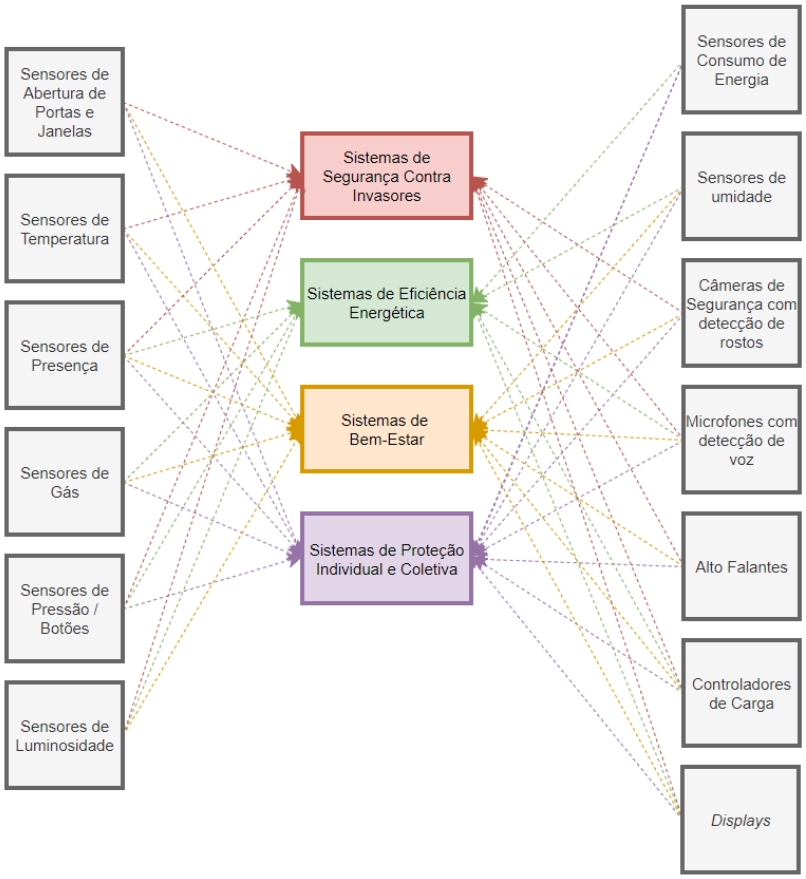


Figura 72 – Exemplo de sistemas IoT para automação residencial e sensores associados.

Além de compartilhar sensores e/ou atuadores para interagir com o ambiente doméstico, os sistemas podem eventualmente realizar as mesmas avaliações e/ou correlações lógicas entre um ou mais sensores. Por exemplo, todos os sistemas podem, eventualmente, avaliar se todas as janelas de um ambiente estão abertas ou podem avaliar se o ambiente está ocupado ou não. De maneira similar, os sistemas podem eventualmente realizar o mesmo conjunto de

ações sobre os atuadores. Por exemplo, abrir todas as janelas ou acender todas as luzes de um ou mais ambientes.

Com o objetivo de otimizar as notificações e o processamento, podem ser inseridos sistemas intermediários auxiliares no sistema, representados pelos Controladores de Seção apresentados na Figura 73. Esses sistemas auxiliares podem realizar avaliações ou correlações lógicas entre um ou mais sensores que estejam próximos ou em uma mesma seção, além de possibilitar uma interface para execução de um conjunto pré-determinado de ações sobre um ou mais atuadores. Com isso, esses controladores de seção podem implementar um nível de abstração, passando a gerar notificações baseada na ocorrência de um ou mais eventos ou abstraindo detalhes específicos envolvidos em determinadas ações. Por exemplo, uma casa inteligente pode ter um controlador para cada ambiente (quarto, sala, cozinha, jardim etc.) ou para cada conjunto de ambientes (1º, 2º andar etc.).

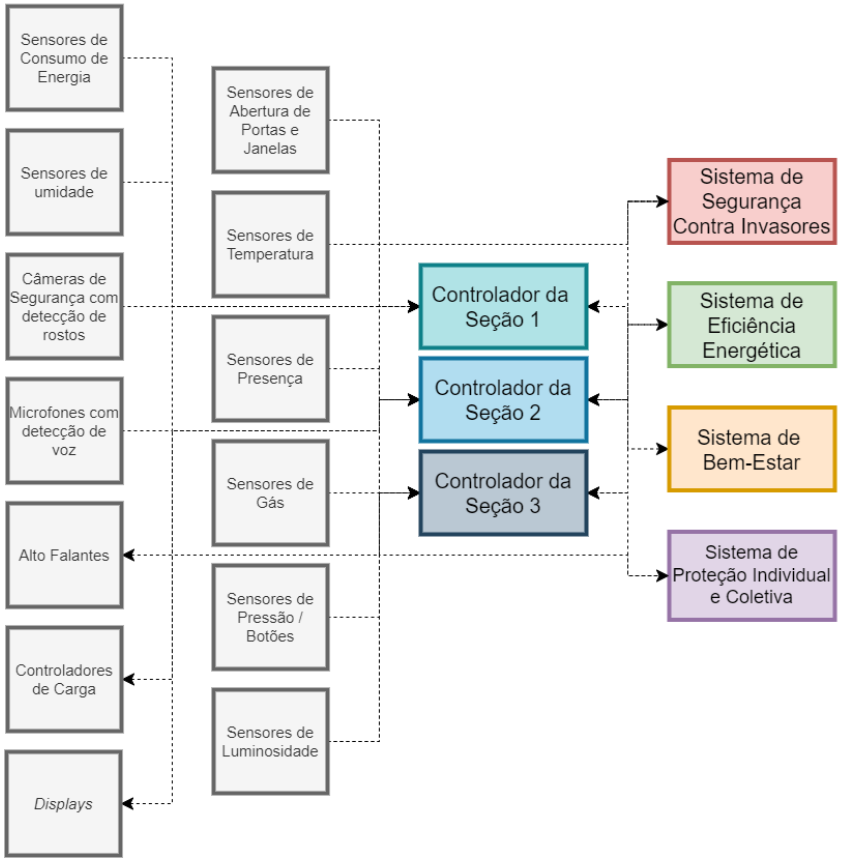


Figura 73 – Exemplo de automação residencial com elementos de processamento intermediários.

Dessa forma, o controlador pode também atuar como um *gateway* doméstico, conectando o ambiente doméstico com o ambiente remoto. Além disso, com o uso de controladores seria possível 'filtrar' as variações dos sensores ou correlacionar diferentes sensores, fazendo com que notificações sejam enviadas para um ambiente remoto somente quando pertinentes. Por exemplo, em uma seção com múltiplos sensores de presença, o controlador poderia implementar uma lógica simples para tratamento das alterações, notificando o ambiente remoto

somente quando for detectado presença de pessoas em qualquer parte da seção considerada, evitando notificações desnecessárias para todas as alterações do ambiente que ocorrerem.

Além disso, dependendo das dimensões e características do ambiente e/ou das características dos dispositivos utilizados, torna-se necessário o uso de sistemas intermediários para que os sensores tenham acesso à rede e se comuniquem com os demais sistemas.

Para implementação da Aplicação IoT para a Casa Inteligente proposta, propôs-se utilizar uma arquitetura orientada a eventos adequado ao protocolo MQTT, conforme apresentado na Figura 74. Tipicamente os dispositivos IoT providos de sensores possuem acesso à Internet e publicam, via endereço IP e porta do *broker*, as mensagens nos tópicos correspondentes. De maneira similar, os dispositivos IoT providos de atuadores se inscrevem nos tópicos correspondentes. Dessa forma, os sistemas de controle/monitoramento podem receber as leituras pertinentes ao se inscrever nos tópicos de interesse e, de maneira análoga, controlar os atuadores via publicação de mensagens no *broker*. Essa arquitetura facilita a integração dos sistemas e o compartilhamento de um mesmo sensor por dois ou mais sistemas diferentes.

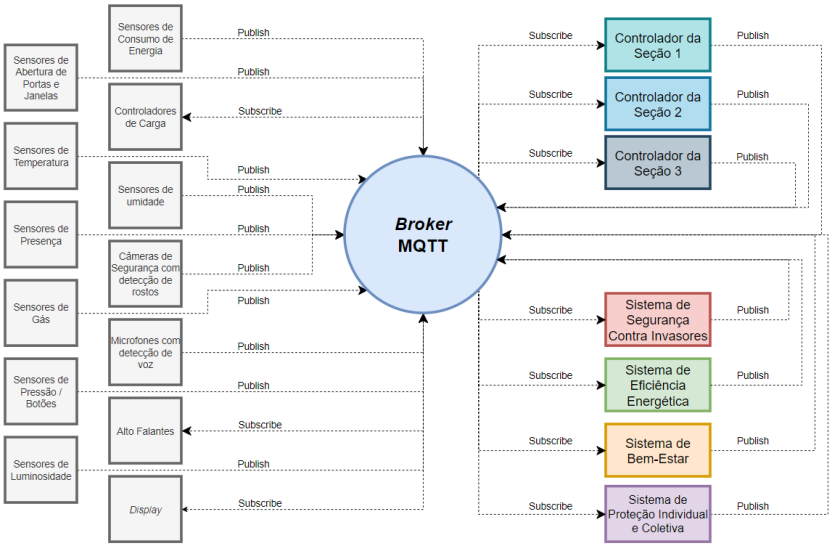


Figura 74 – Arquitetura de sistema IoT para automação residencial utilizando MQTT.

Como experimento de uma Casa Inteligente, propôs-se um ambiente de uma casa com arquitetura típica, conforme apresentado na Figura 75. Cada ambiente possui um conjunto de sensores e atuadores, conforme descrito no Quadro 12. Além dos sensores, os sistemas possuem estados virtuais internos (ligado/desligado) e acesso ao horário e fuso horário do local.

Quadro 12 – Descrição do ambiente de simulação do Experimento da Casa Inteligente.

Localização	Sensores Disponíveis	Atuadores Disponíveis
Portão Principal	Sensor de Presença Sensor de Estado do Portão	Abertura do Portão Fechamento do Portão Acionamento da Iluminação Desligamento da Iluminação
Portão Social	Sensor de Presença Sensor de Estado do Portão	Abertura do Portão Fechamento do Portão Acionamento da Iluminação Desligamento da Iluminação
Jardim Frontal \ Jardim Principal	Sensor de Presença Sensor de Luminosidade Sensor de Umidade Sensor de Temperatura Sensor de Estado da Irrigação Sensor de Chuva	Acionamento da Irrigação Desligamento da Irrigação Acionamento da Iluminação Desligamento da Iluminação
Sala	Sensor de Presença Sensor de Luminosidade Sensor de Estado da Janela Sensor de Estado da Portas Sensor de Umidade Sensor de Temperatura Sensor de Fumaça Sensor de Gás Sensor de Estado do Ar-Condicionado	Acionamento da Iluminação Desligamento da Iluminação Acionamento do Ar-Condicionado Desligamento do Ar-Condicionado Abertura da Porta Fechamento da Porta Abertura da Janela Fechamento da Janela Acionamento do Alarme de Incêndio Acionamento do Alarme de Gás
Cozinha / Lavanderia	Sensor de Presença Sensor de Luminosidade Sensor de Estado da Janela Sensor de Estado da Portas Sensor de Umidade Sensor de Temperatura Sensor de Fumaça Sensor de Gás Sensor de Estado do Ar-Condicionado	Acionamento da Iluminação Desligamento da Iluminação Acionamento do Ar-Condicionado Desligamento do Ar-Condicionado Abertura da Porta Fechamento da Porta Abertura da Janela Fechamento da Janela Acionamento do Alarme de Incêndio Acionamento do Alarme de Gás
Quarto 1 / Quarto 2 / Corredor	Sensor de Presença Sensor de Luminosidade Sensor de Estado da Janela Sensor de Estado da Portas Sensor de Umidade Sensor de Temperatura Sensor de Fumaça Sensor de Gás Sensor de Estado do Ar-Condicionado	Acionamento da Iluminação Desligamento da Iluminação Acionamento do Ar-Condicionado Desligamento do Ar-Condicionado Abertura da Porta Fechamento da Porta Abertura da Janela Fechamento da Janela Acionamento do Alarme de Incêndio Acionamento do Alarme de Gás
Banheiro 1 / Banheiro 2	Sensor de Presença Sensor de Luminosidade Sensor de Estado da Janela Sensor de Estado da Portas Sensor de Umidade Sensor de Temperatura Sensor de Fumaça Sensor de Gás Sensor de Estado do Ar-Condicionado	Acionamento da Iluminação Desligamento da Iluminação Abertura da Janela Fechamento da Janela
Cercas e Muros	Sensor de Presença	Acionamento do Alarme Desligamento do Alarme

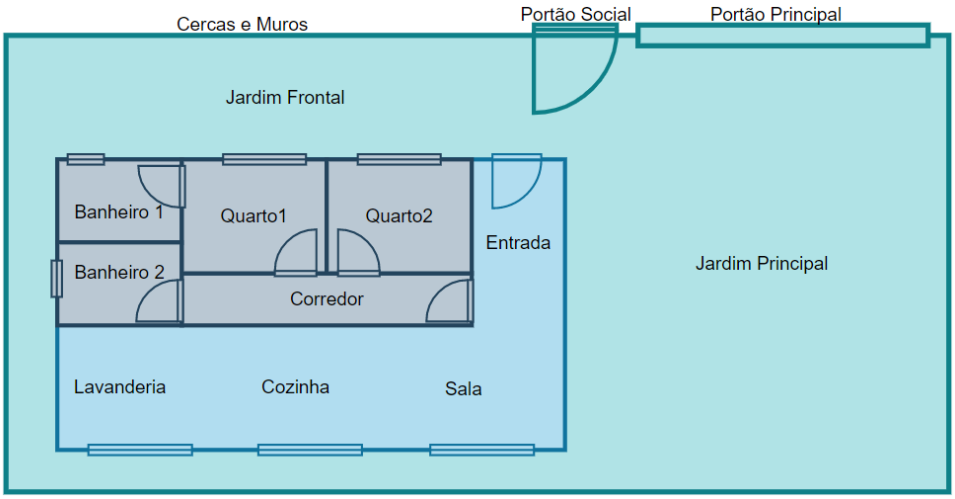


Figura 75 – Planta baixa simplificada do ambiente residencial simulado.

Para cada sistema apresentado, descreve-se um conjunto de regras em alto nível. Destaca-se que as regras apresentadas não são exaustivas, isto é, é apresentado e implementado apenas um conjunto de regras possíveis utilizando-se de um conjunto de sensores e atuadores com o objetivo de avaliar o contexto do experimento. É possível que sejam adicionadas novas regras ou novos sensores e atuadores. Ademais, as regras apresentadas podem ser modificadas dependendo das necessidades de cada aplicação.

Para o Sistema de Detecção de Invasores ou IDS da sigla em inglês para *Intrusion Detection System*, responsável por proteger a residência contra invasões ou intrusões não autorizadas, descrevem-se como regras:

- **Regra 1:** se IDS ligado, se alarmes não acionados, se alarmes da cerca não acionados, se sistema de monitoramento habilitado e se intruso detectado nas cercas e muros, **então** disparar alarmes e notificar usuários remotos;
- **Regra 2:** se IDS ligado, se alarmes não acionados, se monitoramento noturno habilitado, se alarme de intrusos no portão não acionado e se janela do quarto aberta ou se janela da cozinha aberta ou se janela da sala aberta ou se janela do banheiro aberta ou se porta da cozinha aberta ou se portão social aberto ou se portão principal aberto ou se presença no jardim detectada, **então** disparar alarmes e notificar usuários remotos.

Para o Sistema de Eficiência Energética ou EES da sigla em inglês para *Energy Efficiency System*, responsável por ajudar no uso eficiente dos recursos energéticos como, por exemplo energia elétrica e água, descrevem-se como regras:

- **Regra 1:** se EES ligado, se umidade do jardim estiver baixa, se pessoas não presentes no jardim, se irrigação do jardim desligada e se chuva não detectada, **então** ligar sistema de Irrigação;

- **Regra 2:** se EES ligado, se irrigação do jardim ligada e se umidade do jardim estiver alta ou se presença no jardim detectada, **então** desligar irrigação do jardim;
- **Regra 3:** se EES ligado, se horário não noturno, se iluminação do jardim ligada e se sistema de intrusão não alarmado, **então** desligar luzes externas.

Para o Sistema de Bem-Estar ou WBS da sigla em inglês para *Well Being System*, responsável por promover o conforto dos moradores por meio da correta configuração de ambientes em termos de luminosidade, temperatura e umidade, descrevem-se como regras:

- **Regra 1:** se WBS ligado, se janela da sala fechada, se porta da sala fechada, se fumaça na sala não detectada, se vazamento de gás na sala não detectado, se presença na sala detectada, se temperatura na sala estiver alta e se ar-condicionado da sala estiver desligado, **então** ligar sistema de ar-condicionado na sala;
- **Regra 2:** se WBS ligado, se janela do quarto fechada, se porta do quarto fechada, se fumaça no quarto não detectada, se vazamento de gás no quarto não detectado, se presença no quarto detectada, se temperatura no quarto estiver alta e se ar-condicionado no quarto estiver desligado, **então** ligar sistema de ar-condicionado no quarto;
- **Regra 3:** se WBS ligado, se janela da sala aberta, se porta da sala aberta, se fumaça na sala não detectada, se vazamento de gás na sala não detectado, se presença na sala detectada, se temperatura na sala estiver alta e se ar-condicionado da sala desligado, **então** fechar portas e janelas da sala;
- **Regra 4:** se WBS ligado, se janela do quarto aberta, se porta do quarto aberta, se fumaça no quarto não detectada, se vazamento de gás no quarto não detectado, se presença no quarto detectada, se temperatura no quarto estiver alta e se ar-condicionado no quarto estiver desligado, **então** fechar portas e janelas do quarto.

Para o Sistema de Proteção Individual e Coletiva ou ICPS da sigla em inglês para *Individual and Collective Protection System*, responsável por detectar anomalias prejudiciais aos moradores do imóvel, descrevem-se como regras:

- **Regra 1:** se ICPS ligado, se vazamento de gás detectado na sala, se alarme de ICPS para gás não acionado, se porta da sala fechada e se janela da sala fechada, **então** disparar alarmes e abrir ambiente (portas e janelas);
- **Regra 2:** se ICPS ligado, se vazamento de gás detectado na cozinha, se alarme de ICPS para gás não acionado, se porta da cozinha fechada e se janela da cozinha fechada, **então** disparar alarme de incêndio;

- **Regra 3:** se ICPS ligado, se alarme de ICPS para fogo não acionado e se fumaça detectada no quarto ou se fumaça detectada na sala ou se fumaça detectada no banheiro ou se fumaça detectada na cozinha, **então** disparar alarme de incêndio;
- **Regra 4:** se ICPS ligado, se alarme de ICPS para fogo na área externa não acionado e se fumaça detectada no jardim, **então** acionar alarme para fogo na área externa e disparar alarme de incêndio;
- **Regra 5:** se alarme de ICPS para fogo acionado e se porta do quarto fechada, **então** abrir porta do quarto;
- **Regra 6:** se alarme de ICPS para fogo acionado e se janela do quarto fechada, **então** abrir janela do quarto;
- **Regra 7:** se alarme de ICPS para fogo acionado e se porta da sala fechada, **então** abrir porta da sala;
- **Regra 8:** se alarme de ICPS para fogo acionado e se janela da sala fechada, **então** abrir janela da sala;
- **Regra 9:** se alarme de ICPS para fogo acionado e se porta da cozinha fechada, **então** abrir porta da cozinha;
- **Regra 10:** se alarme de ICPS para fogo acionado e se janela da cozinha fechada, **então** abrir janela da cozinha;
- **Regra 11:** se alarme de ICPS para fogo acionado e se porta do banheiro fechada, **então** abrir porta do banheiro;
- **Regra 12:** se alarme de ICPS para fogo acionado e se janela do banheiro fechada, **então** abrir janela do banheiro.
- **Regra 13:** se alarme de ICPS para fogo acionado e se iluminação de emergência não acionada, **então** ligar iluminação de emergência;
- **Regra 14:** se alarme de ICPS para fogo na área externa acionado e se irrigação do jardim desligada, **então** ligar sistema de irrigação.

4.3.1 Detalhes da Implementação em PON

A partir do escopo e arquitetura apresentados, propôs-se a implementação dos sistemas de controle segundo o PON. Para a implementação da casa inteligente, seguiu-se a implementação de cada subsistema individualmente e de forma não distribuída, conforme detalhados

nas seções subsequentes. Após a descrição das entidades PON, implementou-se o sistema de forma distribuída por meio do *Framework* PON C++ 4.0 IoT.

No total, foram implementados 72 *Attributes*, 59 *Premises*, 39 *Conditions*, 23 *Rules*, 22 *Instigations*, 21 *Actions* e 22 *Methods*.

4.3.1.1 Sistema de detecção de invasores

Para o sistema de detecção de invasores e com base nas regras previamente apresentadas, tem-se duas *Rules* conforme apresentado no Quadro 13. Cada *Rule* está relacionada com uma *Condition* que se divide em múltiplas *SubConditions*. Destaca-se nesse caso o compartilhamento da *Condition* 1 e das *Premises* 1 e 2 por diferentes entidades.

Quadro 13 – *Rules*, *Conditions* e *Premises* do Sistema de Segurança Contra Invasores.

Rule ID	Condition ID	Sub Condition ID	Logical Operator	Premise Id	Premise
1	33	1	AND	1	atIDSSystemState == Enabled
				2	atIDSAlarmedState == NotAlarmed
		2	AND	3	atElectricFenceInstrusionAlarm == Enabled
			AND	4	atElectricFenceState == Enabled
			AND	5	atElectricFenceIntrusion == Disabled
2	34	1	AND	1	atIDSSystemState == Enabled
			AND	2	atIDSAlarmedState == NotAlarmed
		3	AND	8	atIDSNightlySurveillance == Enabled
		4	OR	9	atBedroomWindowState == Open
			OR	10	atKitchenWindowState == Open
			OR	11	atRoomWindowState == Open
			OR	12	atBathroomWindowState == Open
			OR	13	atKitchenDoorState == Open
		5	OR	14	atMainGateState == Open
			OR	15	atSocialDoorState == Open
		6	OR	16	atGardenPresence == Detected
			OR	17	atGateInstrusionAlarm == NotAlarmed

Em complemento às Rules descritas no Quadro 13, o Quadro 14 apresenta as *Actions* com suas correspondentes *Instigations* e *Methods*. Destaca-se em especial o compartilhamento da mesma *Action* para ambas as *Rules* apresentadas.

Quadro 14 – *Actions*, *Instigations* e *Methods* do Sistema de Segurança Contra Invasores.

Rule Id	Action Id	Instigation Id	Method Id	Attribute	Novo Valor
1	1	1	1	atIDSAlarmedState	Alarmed
2	1	1	1	atIDSAlarmedState	Alarmed

4.3.1.2 Sistema de Eficiência Energética

Para o sistema de eficiência energética, tem-se três *Rules* conforme apresentado no Quadro 15. Cada *Rule* está relacionada com uma *Condition* composta por duas ou mais *Premises*. Destacam-se as *Premises* 2 e 16 compartilhadas com o sistema de detecção de invasores.

Quadro 15 – *Rules*, *Conditions* e *Premises* do Sistema de Eficiência Energética.

Rule Id	Condition Id	Logic Operator	Premise Id	Premise
3	7	AND	18	atEESSystemState == Enabled
		AND	19	atGardenSoilMoistureLevel <atEESSoilMoistureLowLevel
		AND	20	atGardenPresence == Disabled
		AND	21	atGardenIrrigationState == Disabled
		AND	22	atGardenRainDetectionState == Disabled
4	8	AND	18	atEESSystemState == Enabled
		OR	24	atGardenSoilMoistureLevel >atEESSoilMoistureHighLevel
			16	atGardenPresence == NotDetected
		AND	25	atGardenIrrigationState == Enabled
5	9	AND	18	atEESSystemState == Enabled
		AND	27	atEESNightTime == Disabled
		AND	28	atGardenLightingOn == Enabled
		AND	2	atIDSAlarmedState == NotAlarmed

Em complemento às *Rules* descritas no Quadro 15, o Quadro 16 apresenta as *Actions* com as suas correspondentes *Instigations* e *Methods*.

Quadro 16 – *Actions*, *Instigations* e *Methods* do Sistema de Eficiência Energética.

Rule Id	Action Id	Instigation Id	Method Id	Attribute	Novo Valor
3	2	2	2	atGardenIrrigationState	Enabled
4	3	3	3	atGardenIrrigationState	Disabled
5	4	4	4	atGardenLightingOn	Disabled

4.3.1.3 Sistema de Bem-Estar

Para o sistema de Bem-Estar, tem-se quatro *Rules* conforme apresentado no Quadro 17. Cada *Rule* está relacionada com uma *Condition* composta por duas ou mais *subconditions*. Destaca-se em especial o compartilhamento das *Conditions* 11 e 13 entre duas *Rules* do próprio Sistema. Além disso, observa-se o compartilhamento das *Premises* 30, 33, 34, 35, 36, 37, 41, 42, 43, 44, 45, 47, 48, 55 e 56 com outros sistemas.

Quadro 17 – *Rules*, *Conditions* e *Premises* do Sistema de Bem-Estar.

Rule Id	Condition Id	Sub Condition Id	Logic Operator	Premise Id	Premise
6	35	10	AND	30	atWBSSystemState == Enabled
			AND	31	atRoomWindowState == Closed
			AND	32	atRoomDoorState == Closed
		11	AND	33	atRoomSmokeDetection == NotDetected
			AND	34	atRoomGasLeakDetection == NotDetected
			AND	35	atRoomPresence == Detected
			AND	36	atRoomTemperature >atWBSTempHighSetting
			AND	37	atRoomAirConditioningState == Disabled
7	36	12	AND	30	atWBSSystemState == Enabled
			AND	39	atBedroomWindowState == Closed
			AND	40	atBedroomDoorState == Closed
		13	AND	41	atBedroomSmokeDetection == NotDetected
			AND	42	atBedroomGasLeakDetection == NotDetected
			AND	43	atBedroomTemperature >atWBSTempHighSetting
			AND	44	atBedroomPresence == Detected
			AND	45	atBedroomAirConditioningState == Disabled
8	37	14	AND	30	atWBSSystemState == Enabled
		15	OR	47	atRoomWindowState == Open
				48	atRoomDoorState == Open
		11	AND	33	atRoomSmokeDetection == NotDetected
			AND	34	atRoomGasLeakDetection == NotDetected
			AND	36	atRoomTemperature >atWBSTempHighSetting
			AND	35	atRoomPresence == Detected
			AND	37	atRoomAirConditioningState == Disabled
9	38	16	AND	30	atWBSSystemState == Enabled
		17	OR	55	atBedroomWindowState == Open
				56	atBedroomDoorState == Open
		13	AND	41	atBedroomSmokeDetection == NotDetected
			AND	42	atBedroomGasLeakDetection == NotDetected
			AND	43	atBedroomTemperature >atWBSTempHighSetting
			AND	44	atBedroomPresence == Detected
			AND	45	atBedroomAirConditioningState == Disabled

Em complemento às *Rules* descritas no Quadro 17, o Quadro 18 apresenta as *Actions* com suas correspondentes *Instigations* e *Methods*.

Quadro 18 – *Actions*, *Instigations* e *Methods* do Sistema de Bem-Estar.

Rule Id	Action Id	Instigation Id	Method Id	Attribute	Novo Valor
6	5	5	5	atRoomAirConditioningState	Enabled
7	6	6	6	atBedroomAirConditioningState	Enabled
8	7	7	7	atRoomWindowState	Closed
		8	8	atRoomDoorState	Closed
9	8	9	9	atBedroomDoorState	Closed
		10	10	atBedroomWindowState	Closed

4.3.1.4 Sistema de Proteção Individual e Coletiva

Para o Sistema de Proteção Individual e Coletiva, tem-se catorze *Rules* conforme apresentado no Quadro 19. Cada *Rule* está relacionada com uma *Condition* e suas respectivas *Premises*, exceto a *Rule* 12 que se relaciona com uma *Condition* composta por uma *Subcondition*. Destaca-se o compartilhamento das *Premises* 21, 31, 32, 39 e 40 entre os diferentes sistemas e o compartilhamento das *Premises* 62, 64, 70, 71 e 81 entre as *Conditions* do próprio sistema.

Quadro 19 – *Rules*, *Conditions* e *Premises* do Sistema de Proteção Individual e Coletiva.

Rule Id	Condition Id	Sub Condition Id	Logic Operator	Premise Id	Premise
10	18		AND	62	atICPSSystemState == Enabled
			AND	63	atRoomGasLeakDetection == Detected
			AND	64	atICPSGasAlarmedState == NotAlarmed
			AND	32	atRoomDoorState == Closed
			AND	31	atRoomWindowState == Closed
11	19		AND	62	atICPSSystemState == Enabled
			AND	68	atKitchenGasLeakDetection == Detected
			AND	64	atICPSGasAlarmedState == NotAlarmed
			AND	70	atKitchenDoorState == Closed
			AND	71	atKitchenWindowState == Closed
12	39	20	AND	62	atICPSSystemState == Enabled
			AND	73	atICPSFireAlarmedState == NotAlarmed
		21	OR	74	atBedroomSmokeDetection == Detected
				75	atRoomSmokeDetection == Detected
				76	atBathroomSmokeDetection == Detected
				77	atKitchenSmokeDetection == Detected
13	22		AND	62	atICPSSystemState == Enabled
			AND	79	atICPSExternalFireAlarmedState == NotAlarmed
			AND	80	atGardenSmokeDetection == Detected
14	23		AND	81	atICPSFireAlarmedState == Alarmed
			AND	40	atBedroomDoorState == Closed
15	24		AND	81	atICPSFireAlarmedState == Alarmed
			AND	39	atBedroomWindowState == Closed
16	25		AND	81	atICPSFireAlarmedState == Alarmed
			AND	32	atRoomDoorState == Closed
17	26		AND	81	atICPSFireAlarmedState == Alarmed
			AND	31	atRoomWindowState == Closed
18	27		AND	81	atICPSFireAlarmedState == Alarmed
			AND	70	atKitchenDoorState == Closed
19	28		AND	81	atICPSFireAlarmedState == Alarmed
			AND	71	atKitchenWindowState == Closed
20	29		AND	81	atICPSFireAlarmedState == Alarmed
			AND	94	atBathroomDoorState == Closed
21	30		AND	81	atICPSFireAlarmedState == Alarmed
			AND	96	atBathroomWindowState == Closed
22	31		AND	81	atICPSFireAlarmedState == Alarmed
			AND	98	atICPSEmergencyLights == Disabled
23	32		AND	99	atICPSExternalFireAlarmedState == Alarmed
			AND	21	atGardenIrrigationState == Disabled

Em complemento às *Rules* descritas no Quadro 17, o Quadro 20 apresenta as *Actions* com suas correspondentes *Instigations* e *Methods*. Destaca-se o compartilhamento da *Action* 2 com outros sistemas e o compartilhamento das *Instigations* 11, 12, 13, 14 e 15 entre as *Actions* do próprio sistema.

Quadro 20 – *Actions*, *Instigations* e *Methods* do Sistema de Proteção Individual e Coletiva.

Rule Id	Action Id	Instigation Id	Method Id	Attribute	Novo Valor
10	9	11	11	atICPSGasAlarmedState	Alarmed
		12	12	atRoomDoorState	Open
		13	13	atRoomWindowState	Open
11	10	11	11	atICPSGasAlarmedState	Alarmed
		14	14	atKitchenDoorState	Open
		15	15	atKitchenWindowState	Open
12	11	16	16	atICPSFireAlarmedState	Alarmed
13	12	17	17	atICPSExternalFireAlarmedState	Alarmed
14	13	18	18	atBedroomDoorState	Open
15	14	19	19	atBedroomWindowState	Open
16	15	12	12	atRoomDoorState	Open
17	16	13	13	atRoomWindowState	Open
18	17	14	14	atKitchenDoorState	Open
19	18	15	15	atKitchenWindowState	Open
20	19	20	20	atBathroomDoorState	Open
21	20	21	21	atBathroomWindowState	Open
22	21	22	22	atICPSEmergencyLights	Enabled
23	2	2	2	atGardenIrrigationState	Enabled

4.3.2 Detalhes da Implementação Distribuída em *Framework* PON C++ 4.0 IoT

Segundo os requisitos do experimento e com o objetivo de melhorar a eficiência do sistema, evitando a redundância de processamento, propôs-se uma solução considerando três sistemas de processamento, conforme apresentado na Figura 76.

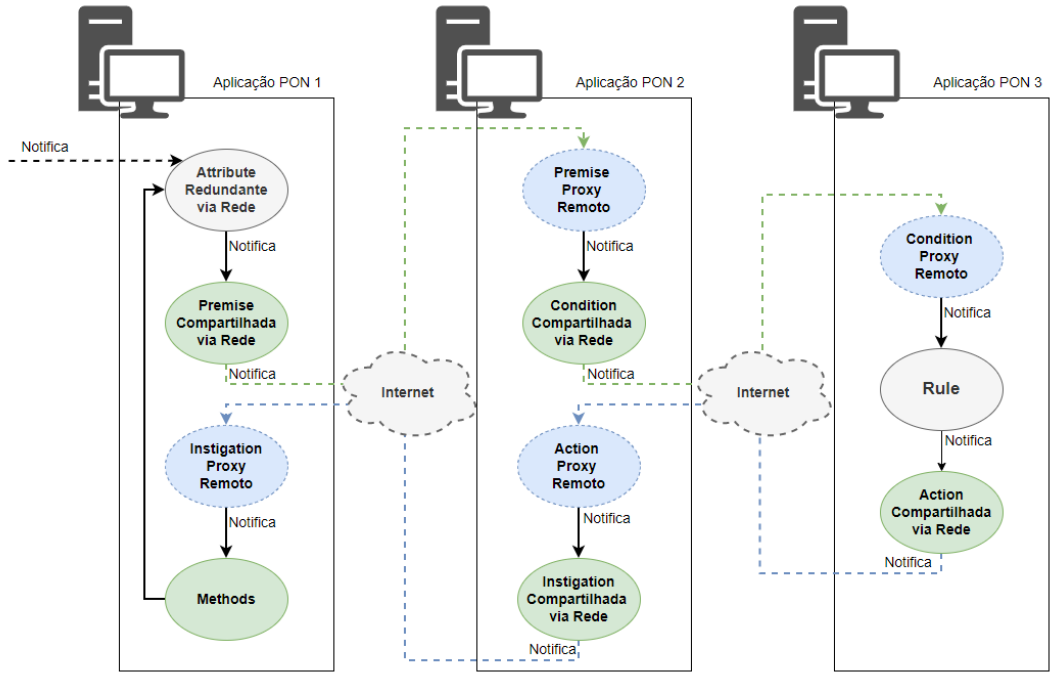


Figura 76 – Distribuição do processamento no experimento da Casa Inteligente no PON.

A primeira aplicação representa uma abstração do ambiente e é composto por *Attributes* Locais interagindo com *Premises* Compartilhadas via Rede e *Instigations Proxy* Remoto interagindo com *Methods*. Quando executados, os *Methods* podem interagir também com os *Attributes*, alterando o ambiente. Como exemplo, é apresentado no Código 4.10 um trecho com a declaração de parte das entidades PON do ambiente doméstico utilizadas na *Rule* 7 do WBS (previamente apresentada no Quadro 17).

Código 4.10 Exemplo de Código para a Aplicação PON 1 (ambiente doméstico) para o sistema WBS do Experimento da Casa Inteligente.

```
...

// Declaração dos Attributes:
SharedAttribute<bool> atBedroomPresence =
    NOP::BuildRemoteProxyAttribute("atBedroomPresence", false);

NOP::SharedAttribute<bool> atBedroomAirConditioningState =
    NOP::BuildRemoteProxyAttribute("atBedroomAirConditioningState", false);

NOP::SharedAttribute<int> atBedroomTemperature =
    NOP::BuildRemoteProxyAttribute("atBedroomTemperature", 20);

NOP::SharedAttribute<int> atWBSTempHighSetting =
    NOP::BuildRemoteProxyAttribute("atWBSTempHighSetting", 30);

// Declaração das Premises Compartilhadas via Rede:

prBedroomPresenceEqDetected =
    NOP::BuildNetworkSharedPremise<bool>("prBedroomPresenceEqDetected",
        smartHome.atBedroomPresence, (bool) Detected, NOP::Equal());

prBedroomTemperatureGtatWBSTempHighSetting =
    NOP::BuildNetworkSharedPremise(
        "prBedroomTemperatureGtatWBSTempHighSetting",
        smartHome.atBedroomTemperature, smartHome.atWBSTempHighSetting,
        NOP::GreaterEqual());

prBedroomAirConditioningStateEqDisabled =
    NOP::BuildNetworkSharedPremise<bool>(
        "prBedroomAirConditioningStateEqDisabled",
        smartHome.atBedroomAirConditioningState,
        (bool) Disabled, NOP::Equal());

// Declaração das Instigations Proxy Remoto:
in6 = NOP::BuildRemoteProxyInstigation("in6",
    smartHome.mtSetatBedroomAirConditioningStateEnabled);

// Declaração dos Methods
mtSetatBedroomAirConditioningStateEnabled =
    atBedroomAirConditioningState->SetValue(Enabled););

...
```

A segunda aplicação representa os controladores apresentados previamente na Figura 74. Os controladores apresentam uma abstração das condições do ambiente, sendo construídos por meio de *Premises Proxy* Remoto interagindo com *Conditions* Compartilhadas via Rede e por meio de *Actions Proxy* Remoto interagindo com *Instigations* Compartilhadas via Rede. Como exemplo, é apresentado no Código 4.11 um trecho com a declaração de parte das entidades PON dos controladores utilizadas no contexto da *Rule* 7 do WBS (previamente apresentada no Quadro 17).

Código 4.11 Exemplo de Código para a Aplicação PON 2 (controladores) para o sistema WBS do Experimento da Casa Inteligente.

```
...

// Declaração das Premises Proxy Remoto:
prBedroomAirConditioningStateEqDisabled  =
    NOP::BuildRemoteProxyPremise(
    "prBedroomAirConditioningStateEqDisabled");

prBedroomTemperatureGtatWBSTempHighSetting  =
    NOP::BuildRemoteProxyPremise(
    "prBedroomTemperatureGtatWBSTempHighSetting");

prBedroomPresenceEqDetected =
    NOP::BuildRemoteProxyPremise("prBedroomPresenceEqDetected");

// Declaração das Conditions Compartilhadas via Rede:
cn13 = NOP::BuildNetworkSharedCondition("cn13",
    CONDITION(*prBedroomSmokeDetectionEqNotDetected &&
    *prBedroomGasLeakDetectionEqNotDetected &&
    *prBedroomTemperatureGtatWBSTempHighSetting &&
    *prBedroomPresenceEqDetected &&
    *prBedroomAirConditioningStateEqDisabled),
    prBedroomSmokeDetectionEqNotDetected ,
    prBedroomGasLeakDetectionEqNotDetected ,
    prBedroomTemperatureGtatWBSTempHighSetting ,
    prBedroomPresenceEqDetected ,
    prBedroomAirConditioningStateEqDisabled);

cn12 = NOP::BuildNetworkSharedCondition("cn12",
    CONDITION(*prWBSSystemStateEqEnabled &&
    *prBedroomWindowStateEqClosed &&
    *prBedroomDoorStateEqClosed), prWBSSystemStateEqEnabled,
    prBedroomWindowStateEqClosed , prBedroomDoorStateEqClosed);

cn36 = NOP::BuildNetworkSharedCondition("cn36",
    CONDITION(*cn12 && *cn13), cn12 , cn13);

// Declaração da Instigation Compartilhada via Rede:
in6 = NOP::BuildRemoteOnlyInstigation("in6");

// Declaração da Action Proxy Remoto:
ac6 = NOP::BuildRemoteProxyAction("ac6",in6);

...
```

A terceira aplicação representa os sistemas de controle do ambiente, descrevendo as regras funcionais. Essas aplicações são implementadas por *Rules*, interagindo com *Conditions Proxy* Remoto e com *Actions* Compartilhadas via Rede. Como exemplo, é apresentado no Código 4.12 um trecho com a declaração de parte das entidades PON dos sistemas de controle utilizadas no contexto da *Rule* 7 do WBS (previamente apresentada no Quadro 17).

Código 4.12 Exemplo de Código para a Aplicação PON 3 (sistemas de controle) para o sistema WBS do Experimento Casa Inteligente.

```
...
// Declaração das Conditions Proxy Remoto:
cn36 = NOP::BuildProxyRemoteCondition("cn36");

// Declaração da Action Compartilhada via Rede:
ac6 = NOP::BuildRemoteOnlyAction<NOP::Parallel>("ac6");

// Declaração da Rule:
rl7 = NOP::BuildRule(cn36, ac6);

...
```

Com essa estrutura proposta, é possível que os sistemas compartilhem entidades de forma a evitar avaliações ou código redundante. Ademais, utiliza-se todo o conjunto de entidades apresentados previamente para uma implementação não distribuída, alterando-se apenas o seu tipo e o local de execução (podendo agora ocorrer em computadores distintos).

4.3.3 Detalhes da Implementação POE via *Pub/Sub* em C++

A implementação do sistema segundo o POE foi realizada utilizando-se também o padrão *Publish/Subscribe* com comunicação indireta, por meio da utilização de um *broker*. A Figura 77 apresenta o diagrama de classes da solução implementada. Utilizou-se uma estrutura auxiliar para comunicação com o *broker* MQTT as quais são baseadas (com as devidas adaptações) nas estruturas adicionadas ao *Framework* PON C++ 4.0 IoT e previamente descritas na Seção 3.3. Essas classes (nomeadamente, *RemoteMQTT*, *MosquittoClient* e *ConfigHolder*) auxiliam na comunicação com o *broker* MQTT, isto é, por meio delas são enviadas e recebidas as mensagens MQTT. Os atributos e métodos dessas classes são semelhante ao diagrama previamente apresentado na Figura 61.

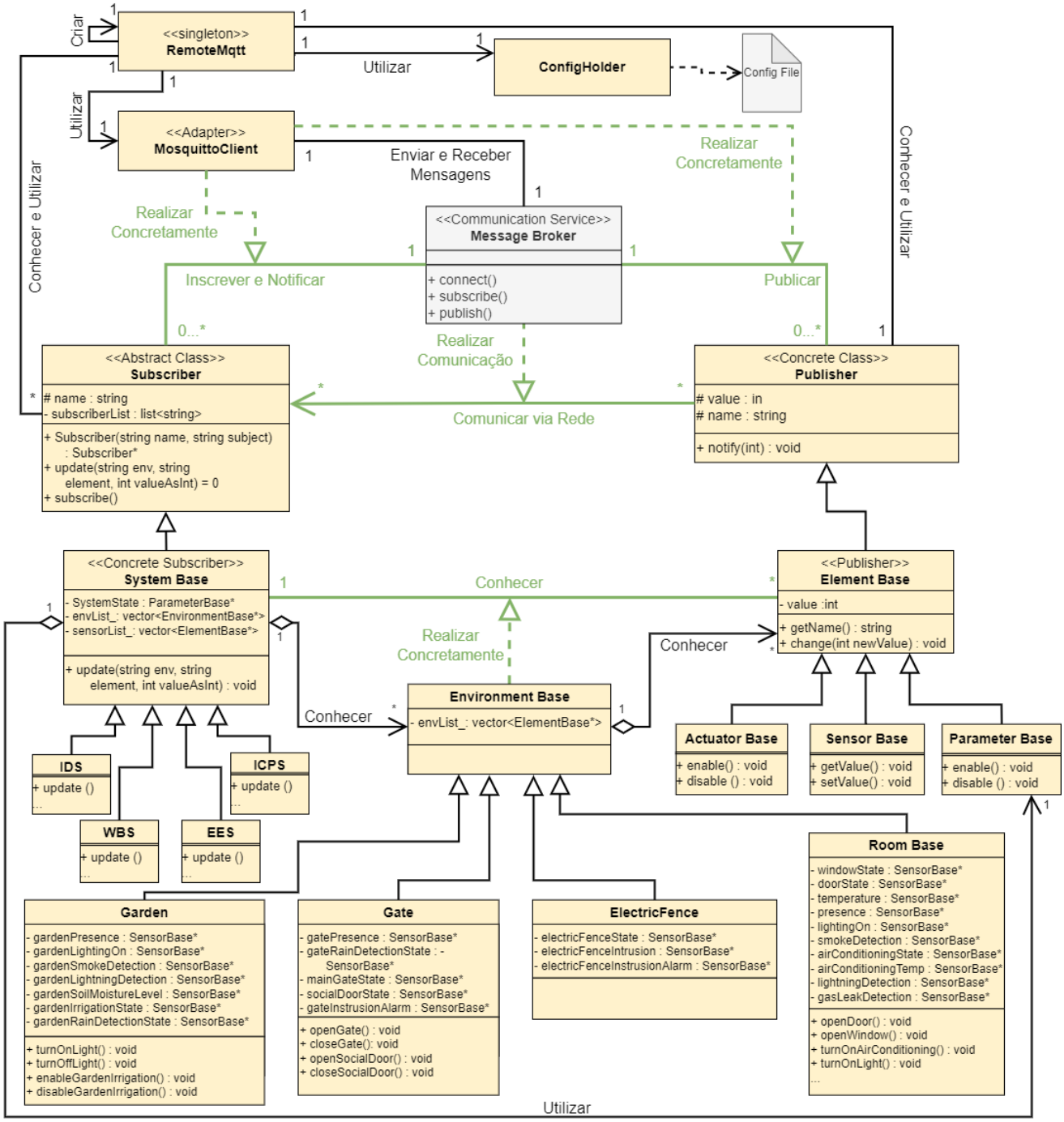


Figura 77 – Diagrama de classes da implementação da Casa Inteligente no POE.

Optou-se por descrever os elementos sensores, atuadores e alguns parâmetros do sistema por meio de uma classe base, chamada *Element Base*. A classe *Element Base* deriva da classe *Publisher*, permitindo assim o envio de notificações quando um evento acontecer (por exemplo, uma mudança de estado de um atributo). Dessa classe, derivam-se os elementos sensores (*Sensor Base*), atuadores (*Actuator Base*) e parâmetros (*Parameter Base*). Nesse contexto, os elementos são agregados por meio das classes específicas de cada ambiente, derivadas de uma classe base em comum, nomeadamente *Environment Base*. A classe *Environment Base* possui uma lista para os elementos (*Element Base*) do respectivo ambiente. Dessa

forma, cada classe derivada de *Environment Base* (*Garden*, *Gate*, *Electric Fence* e *Room Base*) é responsável por inicializar e conhecer os sensores e atuadores disponíveis no respectivo ambiente. Observa-se também que alguns ambientes (*Bedroom*, *Room*, *Bathroom* e *Kitchen*) não possuem uma classe específica. Por similaridade, esses ambientes são descritos como instâncias da classe *Room Base*, a qual descreve um conjunto de sensores e atuadores em comum.

Os sistemas (IDS, WBS, EES e ICPS) de controle da casa inteligente são descritos com base na classe *System Base*, que deriva e implementa um componente *Subscriber*. Dessa forma, os sistemas podem ser notificados quando um evento acontece em um dos objetos de seu interesse (sensores, atuadores e/ou parâmetros). As lógicas de controle dos sistemas são implementadas por meio do método *Update*, invocado no respectivo sistema (derivado de *Subscriber*). Esse método realiza o tratamento dos eventos e é invocado quando um novo evento é recebido. Como exemplo, apresenta-se no Código 4.13 o método *update* referente ao sistema de Eficiência Energética (EES).

Código 4.13 Exemplo de Código *Update* para o sistema EES do Experimento Casa Inteligente em POE

```
...
void update(string nameEnv, string nameSensor, int valueAsInt) {

  // Atualização do estado dos elementos conforme evento recebido
  for (int i = 0; i < environmentList_.size(); i++)
  {
    EnvironmentBase* entityEnv = (EnvironmentBase*)environmentList_[i];
    if (entityEnv->getName() == nameEnv)
    {
      for (int i = 0; i < entityEnv->sensorList_.size(); i++)
      {
        SensorBase* entitySensor = (SensorBase*)entityEnv->sensorList_[i];
        if (entitySensor->getName() == nameSensor){
          entitySensor->setValue(valueAsInt);
        }
      }
    }
  }

  if ((systemState->getValue() == 1) && (alarmedState->getValue() == 0))
  {
    // Código em POE que implementa a avaliação
    // lógica análoga à descrita pela Rule 3 do PON
    if ((garden->gardenSoilMoistureLevel->getValue()
           < gardenSoilMoistureLevel)
      && (garden->gardenPresence->getValue() == 0)
      && (garden->gardenIrrigationState->getValue() == 0)
      && (garden->gardenRainDetectionState->getValue() == 0)) {
        garden->gardenIrrigation->enable();
        garden->gardenIrrigationState->setValue(1);
      }

    // Código em POE que implementa a avaliação
    // lógica análoga à descrita pela Rule 4 do PON
    if (((garden->gardenSoilMoistureLevel->getValue()
           > gardenSoilMoistureHighLevel)
      || (garden->gardenPresence->getValue() == 1))
      && (garden->gardenIrrigationState->getValue() == 1)) {
        garden->enableGardenIrrigation->disable();
        garden->gardenIrrigationState->setValue(0);
      }

    // Código em POE que implementa a avaliação
    // lógica análoga à descrita pela Rule 5 do PON
    if (((this->nigthTime->getValue() == 0)
      && (garden->gardenLightingOn->getValue() == 1))) {
        garden->gardenLightsState->disable();
        garden->gardenLightingOn->setValue(0);
     }
  }
}
...
```

Observa-se, nas linhas iniciais, a avaliação das informações do evento recebido, atualizando-se os respectivos sensores em todos os ambientes conhecidos por esse sistema. Em seguida, são realizadas as avaliações das condições do sistema por meio de estruturas decisórias '*if*'. Organizou-se os sistemas de forma que os nomes dos sensores e atuadores fiquem expressivos como, por exemplo, *garden->gardenPresence->getValue()* e *garden->gardenIrrigation->enable()*. Dessa forma, é possível, em certa medida, relacionar cada estrutura '*if*' com uma regra funcional do sistema (e, consequentemente, com as *Rules* do PON). Observa-se também que é realizado, dentro de um mesmo contexto, o compartilhamento de algumas avaliações como, por exemplo, na linha 19 do Código 4.13, no qual a avaliação do estado do sistema e do alarme (presente em todas as regras do sistema) é realizado apenas uma vez.

Por utilizar o padrão *publish/subscribe*, a implementação apresentada pode ser utilizada em ambiente distribuído conforme apresentado na Figura 78.

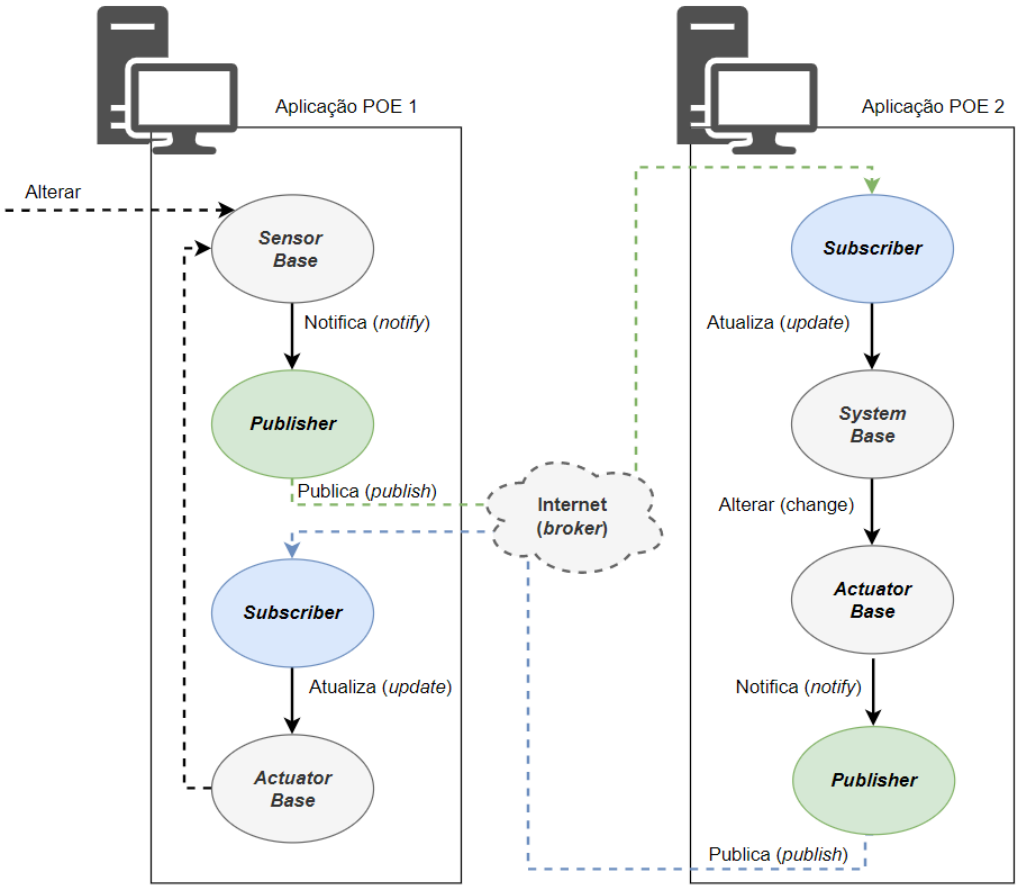


Figura 78 – Distribuição do processamento no experimento da Casa Inteligente no POE.

Porém, ao contrário da implementação em PON, o uso em diferentes partes (ambiente doméstico, controladores e sistemas de controle) não é facilmente relacionada de forma direta com os elementos do POE. Neste exemplo, todo o cálculo lógico é realizado dento do método *update* dos objetos das classes derivadas da classe base *subscriber*. Com isso, são necessários esforços adicionais de engenharia de software para separar de forma eficiente o

processamento. Pode-se, por exemplo, separar e/ou agrupar avaliações lógicas, adicionando-se variáveis intermediárias que representem essas avaliações. Nesse caso, essas avaliações intermediárias poderiam ser executadas uma única vez nos controladores e compartilhadas entre sistemas interessados. Observa-se, nesse caso, que a implementação em POE caminharia em direção a conceitos apresentados naturalmente pelo PON.

4.3.4 Descrição das Métricas

Para a validação e avaliação da implementação utilizou-se a metodologia de testes de integração de sistemas previamente apresentada na Seção 3.3.3.2. Para este experimento, o *framework* de testes é responsável por publicar mensagens referentes ao acionamento dos sensores e verificar o correto recebimento das mensagens referentes ao acionamento dos atuadores.

De forma similar ao experimento do Portão Eletrônico (apresentado na Subseção 4.2.3), para a verificação do correto funcionamento do sistema implementado adotou-se o uso de testes funcionais de integração com foco nos principais cenários de uso do sistema. Com base nos requisitos funcionais são descritos vinte e um casos de teste. Cada teste valida uma ou mais regras (*Rules*).

Como exemplo, apresenta-se no Código 4.14 o caso de teste implementado para a validação do acionamento do sistema de ar-condicionado no quarto quando a temperatura do quarto se eleva além de outras premissas (sistema de Bem-Estar ligado, janela do quarto fechada, porta do quarto fechada, fumaça no quarto não detectada, vazamento de gás no quarto não detectado e se presença no quarto detectada). Como referência, a regra funcional apresentada é implementada pela *Rule* 7, apresentada no Quadro 18.

Código 4.14 Exemplo de código implementando o Caso de Teste 1 para o Experimento da Casa Inteligente.

```
def test_bedroom_closed_air_conditioning_cooling(self):
    # Pré-Requisitos
    client.publish("test/at/atWBSSystemState/", "true", qos=2)
    client.publish("test/at/atBedroomWindowState/", "false", qos=2)
    client.publish("test/at/atBedroomDoorState/", "false", qos=2)
    client.publish("test/at/atBedroomSmokeDetection/", "false", qos=2)
    client.publish("test/at/atBedroomGasLeakDetection/", "false", qos=2)
    client.publish("test/at/atBedroomPresence/", "true", qos=2)
    client.publish("test/at/atWBSTempHighSetting/", "28", qos=2)
    client.publish("test/at/atBedroomAirConditioningState/",
                        "false", qos=2)
    time.sleep(1)

    # Valores de Referência
    initValues = copy.deepcopy(topicValues)

    # Ação referente a um sensor notificando a temperatura do ambiente
    client.publish("test/at/atBedroomTemperature/", "30", qos=2)

    time.sleep(1)

    # Verificação da consequente ação do sistema
    self.assertEqual(
        topicValues["test/at/atBedroomAirConditioningState/"][0],
        initValues["test/at/atBedroomAirConditioningState/"][0] + 1)

    # Finalização do teste, retornando ao estado padrão do sistema
    client.publish("test/at/atBedroomTemperature/", "25", qos = 2)
```

4.3.5 Resultados da implementação em *Framework* PON C++ 4.0 IoT

Os resultados gerais da execução dos casos de teste utilizando o *Framework* PON C++ 4.0 IoT são apresentados na Figura 79. Observa-se que todos os vinte e um testes propostos foram executados com sucesso em aproximadamente 54 segundos.

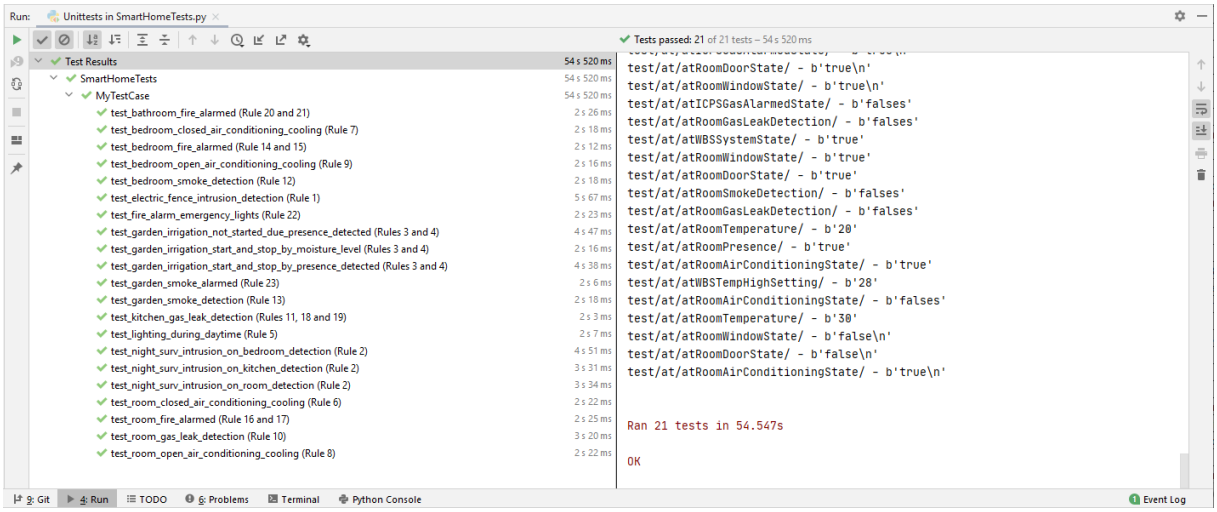


Figura 79 – Resultados dos testes de integração do Experimento da Casa Inteligente em *Framework* PON C++ 4.0 IoT.

Como exemplo de evidência de teste, são apresentadas na Figura 80 as mensagens MQTT capturadas para o Caso de Teste 1 (previamente apresentado na Seção 4.3.4) via software *Wireshark*. Observa-se, na mensagem 2243, o pacote referente à alteração do valor do sensor *atBedroomTemperature* para 30. Destaca-se que, neste caso, o valor do sensor acima do limite previamente configurado (28) deve provocar o acionamento do ar-condicionado deste respectivo ambiente. Com isso, observa-se, na sequência, as mensagens referentes às *Premises*, *Conditions* e *Rules* (nomeadamente, mensagens com número de identificação 2281, 2321, 2399) alteradas em decorrência da alteração do *Attribute*. Observa-se também as mensagens referentes ao acionamento das *Actions* e *Instigations* (mensagens 2397 e 2459) alteradas pela aprovação da respectiva *Rule* (*Rule* 7). Destaca-se também a mensagem de número 2499 correspondente ao acionamento do ar-condicionado, representado pela alteração do estado do *Attribute atBedroomAirConditioningState*, assumindo valor '*true*' (verdadeiro).

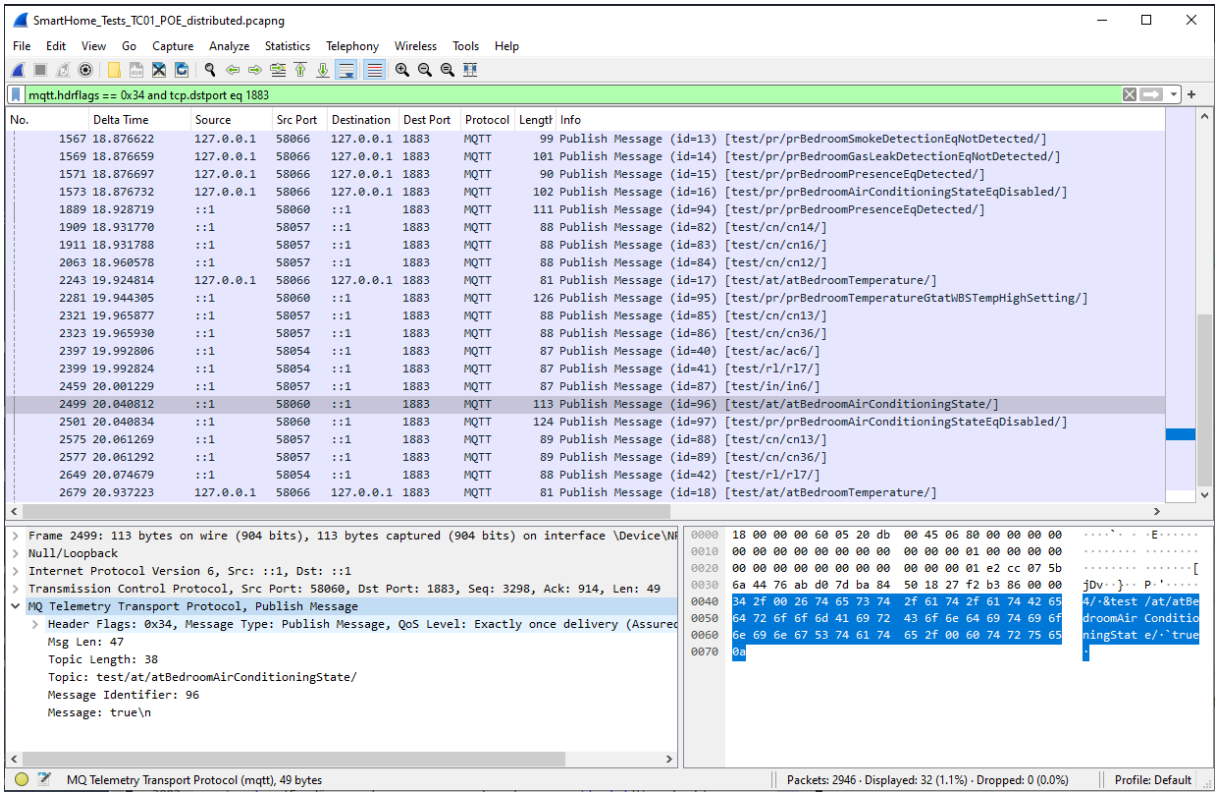


Figura 80 – Evidências de mensagens capturadas com o *Wireshark* para o Caso de Teste 1 em *Framework* PON C++ 4.0 IoT.

4.3.6 Resultados da implementação em POE via *Pub/Sub* em C++

Os resultados gerais da execução dos casos de teste para a implementação utilizando o POE via *Pub/Sub* em C++ são apresentados na Figura 81. Destaca-se que os casos de teste são os mesmos previamente verificados na implementação em PON, adaptando-se apenas o padrão de nomenclatura dos tópicos. Por facilidade na reutilização dos testes, utilizou-se os mesmos padrões para nomenclaturas para os sensores e atuadores do sistema (atributos das classes) como, por exemplo *"atBedroomWindowState"*. No caso da implementação em POE, alterou-se também o campo referente ao domínio (no padrão previamente apresentado na Seção 3.2.1), utilizando-o para representar o ambiente em que o sensor está localizado (conforme os nomes definidos no diagrama de classes, por exemplo, *Garden, Bedroom, Bathroom*, etc). Observa-se que todos os vinte e um testes propostos foram executados com sucesso em aproximadamente 56 segundos.

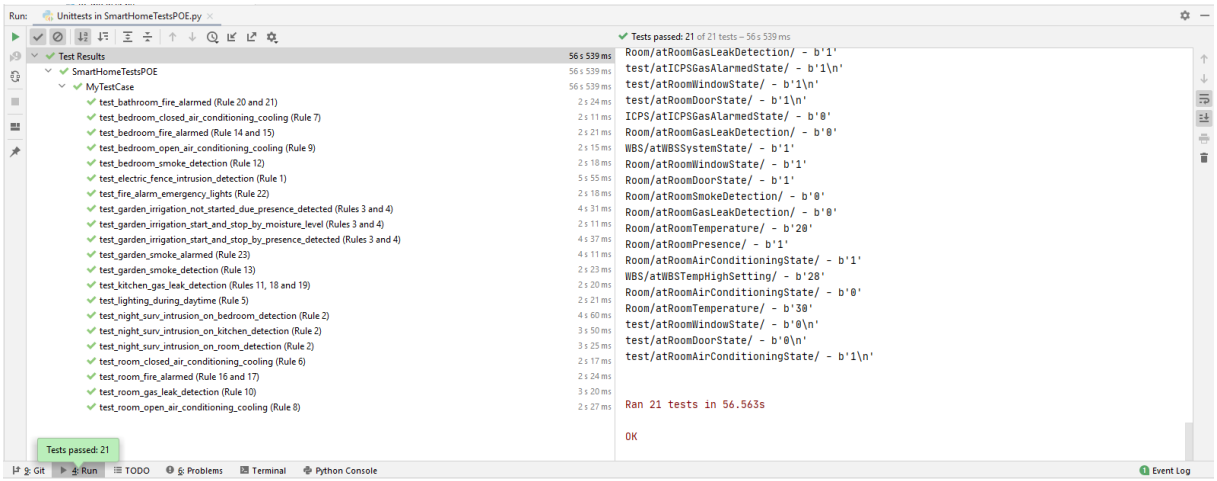


Figura 81 – Resultados dos testes de integração do Experimento da Casa Inteligente em POE via *Pub/Sub* em C++.

Como exemplo de evidência de teste, são apresentadas na Figura 82 as mensagens MQTT capturadas para o Caso de Teste 1 (previamente apresentado na Seção 4.3.4) via software *Wireshark*. Observam-se, inicialmente, as mensagens configurando os pré-requisitos (condições iniciais) para a execução do teste. Na sequência, na mensagem 349, observa-se o pacote referente à alteração do valor do sensor *atBedroomTemperature* para 30. Destaca-se que, neste caso, o valor acima do sensor acima do limite previamente configurado (28) deve provocar o acionamento do ar-condicionado deste respectivo ambiente. Com isso, observa-se, na sequência, a mensagem de número 375 correspondente ao acionamento do ar-condicionado, representado neste exemplo pela mensagem com conteúdo 1, enviada no tópico *Bedroom/atBedroomAirConditioningState*.

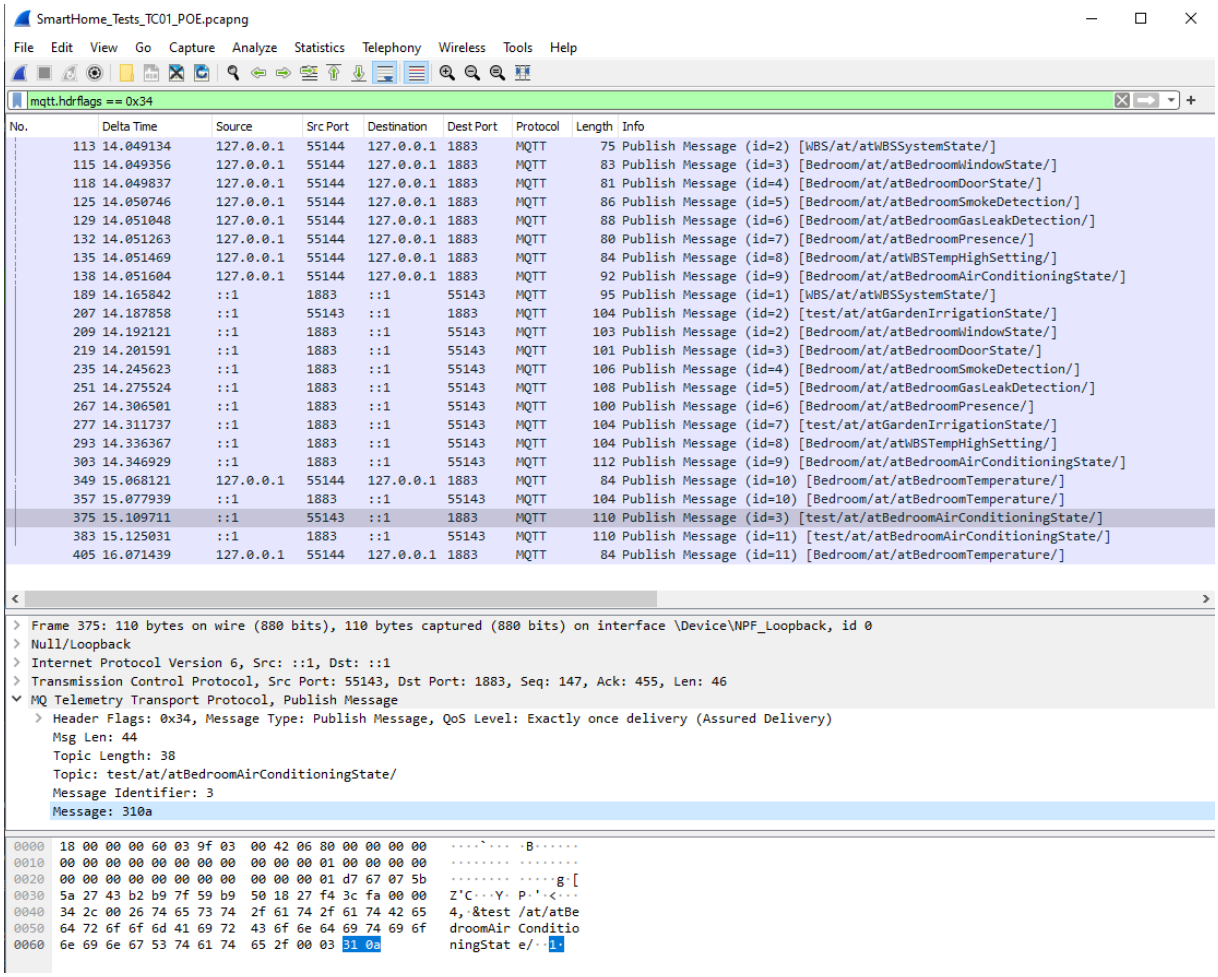


Figura 82 – Evidências de mensagens capturadas com o *Wireshark* para o Caso de Teste 1 em POE via *Pub/Sub* em C++.

4.3.7 Considerações do Experimento

Sobre a implementação em PON, conforme apresentado na Seção 4.3.2, no total foram implementados 72 *Attributes*, 61 *Premises*, 39 *Conditions*, 23 *Rules*, 22 *Instigations*, 21 *Actions* e 22 *Methods*. Destaca-se, porém, que estes valores se referem ao sistema considerando-se o compartilhamento das entidades. O Quadro 21 apresenta os valores comparativos das entidades implementadas em relação às entidades únicas, referente ao mesmo sistema caso o compartilhamento de entidades não estivesse disponível. Relembra-se que, conforme previamente comentado, o mecanismo de compartilhamento de entidades é um meio útil para a resolução do problema de redundância estrutural e temporal.

Quadro 21 – Número de entidades descritas e implementadas após o compartilhamento.

	Entidades Únicas	Entidades após Compartilhamento	Diferença em Porcentagem
Attributes	102	72	-29.4%
Premises	101	61	-39.6%
Conditions	42	39	-7.1%
Rules	23	23	0.0%
Actions	23	21	-8.7%
Instigations	29	22	-24.1%

Observa-se que para o sistema avaliado, o maior ganho em função do compartilhamento das entidades ocorre para as *Premises*, representando um valor de 39.6% de redução de código tecnicamente redundante. Destaca-se também a redução de código para *Attributes* e *Instigations*, representando 29.4% e 24.1%, respectivamente. Para os *Attributes*, avaliou-se quantas entidades seriam necessárias caso um mesmo *Attribute* não pudesse ser utilizado por mais de uma *Premise*. Observa-se que, em virtude das *Rules* representarem avaliações únicas e geralmente “complexas” em relação às demais entidades, o seu compartilhamento pode ser menos frequente que as demais entidades. Para o exemplo apresentado, nenhuma *Rule* foi compartilhada e, por isso, não houve redução em relação ao número de *Rules* únicas do sistema. Uma análise semelhante pode ser estendida para *Actions* e *Conditions*, as quais apresentaram valores mais baixos de compartilhamento (-8.7% e -7.1%, respectivamente).

Outra consideração acerca do experimento da Casa Inteligente implementado em PON é em relação à similaridade da descrição das entidades do PON com as regras funcionais do sistema e também entre as versões distribuídas e não distribuídas. No aspecto teórico, a estruturação do programa em PON por meio da definição de suas entidades para aplicações distribuídas ocorre de forma muito semelhante (senão igual) às aplicações não distribuídas. Dessa forma, mantém-se a mesma expressividade de regras, característica do PON. No aspecto técnico, a diferença entre uma aplicação distribuída e uma não distribuída ocorre somente na declaração das entidades no momento da inicialização do sistema, além também do local no qual a aplicação será executada, podendo ser em computadores distintos, desde que conectados via rede.

Sobre a implementação em POE, observa-se que a redundância temporal é, em partes, reduzida, pois o processamento das avaliações (execução do método *update*) é realizado somente caso alguns dos elementos de interesse do sistema tenha seu estado alterado. Porém, a redundância ainda continua presente, pois durante o ciclo de avaliações não é possível saber precisamente quais variáveis e/ou expressões lógicas tiveram de fato seu estado alterado, sendo necessário, portanto, reavaliar todas as condições novamente. No exemplo apresentado no Código 4.13, observa-se que caso o sensor de chuva se altere (*gardenRainDetectionState*), todas as condições são reavaliadas, mesmo as referentes às regras 4 e 5 as quais não possuem referência explicita à esse sensor.

Com relação à expressividade, o exemplo em POE via *Pub/Sub* em C++ apresenta também um nível apropriado de abstração, obtido com a utilização de uma estrutura de classes e de nomes que identificam adequadamente os objetos/classes.

Destaca-se também que a implementação em POE apresentada neste experimento baseou-se no padrão *publish/subscribe*, porém foi modelada e implementada conforme conhecimentos prévios do autor, os quais incluem também o conhecimento do funcionamento do PON. Portanto, observa-se que ela, em alguns pontos, apresenta influências do mecanismo de notificações e da expressividade do PON.

Na estrutura apresentada, observa-se que alguns mecanismos podem diminuir, ao menos em partes, a redundância estrutural. Por exemplo, o compartilhamento da avaliação referente ao estado do sistema e do estado do alarme (linha 19 no Código 4.13) são compartilhados entre as três regras do sistema. Porém, neste mesmo exemplo não existe um mecanismo para o compartilhamento caso essa mesma avaliação seja executada em outros métodos ou em outros computadores, mesmo que interligados via rede.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta na Seção 5.1 uma breve revisão do contexto desta dissertação, seguida pelas contribuições e conclusões observadas com os desenvolvimentos realizados. Por fim, apresenta-se na Seção 5.2 um conjunto de trabalhos futuros vislumbrados durante o desenvolvimento desta dissertação, mas que fogem ao seu escopo e seus objetivos.

5.1 Conclusões

A Internet das Coisas promove inúmeros benefícios para a sociedade em geral, adicionando novas possibilidades de uso de recursos e/ou o aprimoramento de funcionalidades já existentes, sendo o *software* um elemento que serve de base neste contexto. Com isso, observa-se o aumento da necessidade de desenvolvimento de softwares cada vez mais complexos, especialistas e heterogêneos que se adapte a este mundo de objetos interconectados e colaborativos. Para isso, é necessário que as técnicas, ferramentas de projeto, análise e implementação de *softwares*, em especial de sistemas distribuídos para IoT, tornem-se também cada vez mais fáceis e acessíveis aos desenvolvedores de sistemas e conduzam a implementações mais eficientes e eficazes (SAMUEL; SIPES, 2019) (AL-MASRI *et al.*, 2020). Entretanto, observa-se que os paradigmas de programação atualmente dominantes e usuais, derivados dos Paradigmas Imperativos e dos Paradigmas Declarativos, apresentam ineficiências para o desenvolvimento e execução de programas, principalmente em ambientes concorrentes e/ou distribuídos, o que prejudicariam os avanços acima elencados (RONSZCKA, 2019) (BAUTSCH, 2007).

Nesse contexto, o emergente Paradigma Orientado a Notificações (PON) se apresenta como uma alternativa, apresentando características que facilitam o desenvolvimento de aplicações por meio de programação organizada e em alto nível, ao mesmo tempo que promove o melhor uso dos recursos disponíveis por evitar certas redundâncias e facilita o desenvolvimento de aplicações paralelizadas e distribuídas pelo desacoplamento implícito de seus constituintes (RONSZCKA, 2019) (SIMÃO; STADZISZ, 2009a) (SCHÜTZ *et al.*, 2018). Essas características colocam o PON como uma alternativa efetiva frente à crescente demanda por desenvolvimento de software distribuído, agravada pelo advento e a expansão da Internet das Coisas (RONSZCKA, 2019) (SIMÃO; STADZISZ, 2009a) (SCHÜTZ *et al.*, 2018).

No tocante ao estado da técnica do PON, observou-se inicialmente que, apesar de haver algumas materializações disponíveis e de técnicas e tecnologias para distribuição do PON, nenhuma delas contemplava totalmente a potencialidade das propriedades elementares do PON em uma mesma implementação (Seção 2.3.3). Essas propriedades podem ser descritas como: (i) desenvolvimento em alto nível permitindo alguma facilidade de programação; (ii) desacoplamento permitindo paralelismo; (iii) desacoplamento permitindo distribuição; (iv) código não redundante (estruturalmente e temporalmente). Dentre os *frameworks* existentes antes do iní-

cio desta dissertação, o estado da técnica consistia no *Framework* PON C++ 4.0 que atendia três das quatro propriedades elementares, carecendo, entretanto, da propriedade de distribuição (NEVES, 2021).

Observou-se também, a partir da revisão apresentada na Seção 2.4, que apesar de alguns trabalhos já terem previamente explorado o PON no contexto de sistemas distribuídos e em ensaios em IoT, cada um seguiu uma abordagem distinta com uso não sistematizado (em si e para com o estado da técnica) e não completa por não permitirem a distribuição de todas as entidades do PON. As implementações divergiam em protocolos e arquiteturas de distribuição o que dificultava em partes o uso e a adoção do PON em sistemas no contexto de IoT. Além disso, cada trabalho utilizou uma terminologia para classificação dos tipos de entidades distribuídas (ainda que não explicitamente), não existindo um consenso na classificação e padronização de nomes.

Neste contexto, com o objetivo de promover avanços para o PON no contexto de sistemas distribuídos para IoT, o presente trabalho apresentou um conjunto de desenvolvimentos que permitem a implementação de aplicações distribuídas com o PON, convergindo também para nomenclaturas em comum e na utilização de um protocolo e arquitetura comumente utilizados em IoT.

Com relação às nomenclaturas das entidades do PON no contexto de sistemas distribuídos, foram apresentadas, na Seção 3.1, uma definição e classificação para os tipos de entidades distribuídas segundo as suas características quanto ao envio e recebimento de notificações via rede. Os tipos de entidades distribuídas são nomeados como Entidade Local, Entidade Compartilhada via Rede, Entidade *Proxy* Remoto e Entidade Redundante via Rede. Relembra-se que a Entidade Compartilhada via Rede significa que o estado dessa entidade poderá ser utilizado em computadores remotos e, por isso, é necessário um envio de notificação também via rede quando seu estado for alterado. Neste contexto, a Entidade *Proxy* Remoto pode ser entendida como uma referência para essa Entidade Compartilhada via Rede, no computador remoto. Por sua vez, a entidade Redundante via Rede representa entidades que estão sempre em sincronia, tanto pela alteração por um processo local (disparando nesse caso uma notificação via rede), quanto por uma alteração na respectiva Entidade Redundante via Rede por um processo remoto. Essas nomenclaturas favorecem uma melhor comunicação e integração entre diferentes pesquisas que abordem o PON nesse contexto e permite, por exemplo, que sejam feitas abstrações durante as implementações de *frameworks* ou bibliotecas, o que potencialmente facilita o desenvolvimento de aplicações no aspecto tecnológico.

Uma vez vencida a organização de tipos de entidades distribuídas, focou-se no tipo específico de sistema distribuído em voga na dissertação, nomeadamente IoT. No contexto de IoT, observando-se os protocolos existentes e os usos típicos encontrados para cada um deles, conforme revisão apresentada na Seção 2.1.1, destacou-se o protocolo MQTT por ser um protocolo leve e apresentar funcionalidades que permitem uma arquitetura desacoplante *Publish/Subscribe* escalável para comunicação entre vários clientes (1 para N ou N para N) de

forma assíncrona (AL-MASRI *et al.*, 2020). Neste sentido, o presente trabalho apresentou, de forma inédita, o PON em sinergia com um protocolo e arquitetura distribuída comumente utilizados no contexto de IoT, nomeadamente o protocolo MQTT e a arquitetura *publish/subscribe*. A proposta apresentada para integração do PON com o MQTT por meio da padronização da nomenclatura dos tópicos de acordo com o contexto, o tipo e a identificação única de cada entidade distribuída do PON presente no sistema se mostrou adequada e relativamente simples, bem como natural para o ambiente proposto.

Avaliou-se também a interação do PON e do MQTT quando são utilizados *wildcards* nas inscrições e conforme os diferentes níveis de QoS disponíveis (Seção 3.2.3 e Seção 3.2.2, respectivamente). Relembra-se que os *wildcards* permitem a inscrição em mais de um tópico em uma mesma operação. Observou-se que, dependendo do caso de utilização e das características do sistema, os *wildcards* podem ser úteis principalmente para redução do tempo de inicialização das aplicações. Porém, caso o *broker* seja utilizado por mais de um sistema, o uso de *wildcards* pode provocar o recebimento e consequente processamento de mensagens desnecessárias. Com relação ao QoS, relembra-se que estes definem o nível de garantia de entrega das mensagens (‘no máximo uma vez’, ‘pelo menos uma vez’ e ‘exatamente uma vez’), podendo, em consequência, utilizar um número diferente de mensagens. Neste contexto, observou-se que o QoS 1 pode ser adequado para *Attributes*, *Premises* e *Conditions*, pois o recebimento em duplicidade, em teoria, não afeta o seu comportamento. Por outro lado, para *Rules*, *Actions* e *Instigations*, recomenda-se o QoS 2, evitando-se duplicidades. Destaca-se que esses fatores podem ser alterados de acordo com os requisitos e características de cada aplicação.

Com relação à implementação que permite a execução do PON distribuído utilizou-se como base o *Framework* PON C++ 4.0 adicionando-se agora estruturas que permitem a comunicação via rede por meio do protocolo MQTT (conforme apresentado na Seção 3.3), constituindo assim o *Framework* PON C++ 4.0 IoT. Neste contexto, as estruturas auxiliares para distribuição das entidades do PON adicionadas ao *framework* são transparentes ao desenvolvedor que utilizar o *Framework* PON C++ 4.0 IoT. Esse fato ocorre devido à existência de abstrações, principalmente relacionadas às interfaces de construção de entidades (por meio do uso dos *builders* e do uso dos nomes para cada tipo de entidade distribuída do PON) e da interface de configuração (por meio do arquivo config.ini), conforme apresentado na Seção 3.3.2. Desta forma, o desenvolvedor do PON que utilizar o *Framework* PON C++ 4.0 IoT deve conhecer apenas os tipos de entidades distribuídas e as interfaces de construção das entidades.

Ademais, o *Framework* PON C++ 4.0 IoT manteve também a metodologia e padrão de desenvolvimento voltado a testes, utilizado previamente no *Framework* PON C++ 4.0, adicionando-se agora um conjunto de testes para cobrir as funcionalidades implementadas. Dessa forma, os testes desenvolvidos garantem que o *Framework* PON C++ 4.0 IoT possua o comportamento correto nos casos de uso previstos para cada uma das entidades distribuídas do PON adicionadas e também das entidades locais previamente existentes. Com isso, objetiva-se facilitar o uso, manutenção e eventual incremento ou alterações do *framework*, utilizando-se

como base o conjunto de testes que garantem o funcionamento das funcionalidades implementadas.

Em complemento aos testes unitários, apresentou-se também, na Seção 3.3.3.2, um *framework* para testes de integração de sistemas, permitindo que sejam realizados testes de forma automatizada em aplicações (ou partes de aplicações) PON distribuídas, por meio do envio e recebimento de mensagens MQTT. Esse *framework* de testes evidencia também a flexibilidade herdada pelo uso de protocolos padronizados para a distribuição do PON como é o caso do MQTT, permitindo a interação de diferentes linguagens de programação como, por exemplo, Python (*framework* de testes) e C++ (*Framework* PON C++ 4.0 IoT).

Além dos testes previamente comentados, propôs-se também a realização de três experimentos utilizando o *Framework* PON C++ 4.0 IoT com o objetivo de verificar a exatidão na execução das aplicações diante de um ambiente de IoT, além de aspectos não funcionais (por exemplo, tempo de execução das aplicações, uso de recursos de rede etc.). Compararam-se também, por meio dos experimentos, alguns aspectos das implementações em PON com um outro paradigma comumente utilizado no contexto de sistemas distribuídos e IoT, nomeadamente o Paradigma Orientado a Eventos (POE).

O primeiro experimento realizado no contexto de um sistema de sensores para IoT, avaliou aspectos funcionais e também métricas relacionadas ao uso de recursos computacionais (processamento e memória) e de tempo de inicialização da aplicação em PON (via *Framework* PON C++ 4.0 IoT) em comparação com o POE (via implementação distribuída com *Publish/-Subscribe* e MQTT). Os resultados mostraram que a implementação com o PON apresentou um tempo menor para o processamento das mensagens em comparação com a implementação em POE. Em contrapartida, observou-se um maior tempo para inicialização das aplicações em *Framework* PON C++ 4.0 IoT e um maior uso de memória RAM, quando comparadas ao POE via *Pub/Sub* em C++.[1]

O segundo experimento realizado abordou um sistema de controle para um portão eletrônico, também implementado em *Framework* PON C++ 4.0 IoT e POE via *Pub/Sub* em C++. Neste experimento, além dos aspectos funcionais apresentados pelo sistema, foram observadas e comparadas as implementações em ambos os paradigmas e o número de mensagens trafegadas pela rede, comparando-se também com trabalhos similares disponíveis na literatura. No geral, a implementação em PON se aproximou das regras apresentadas para o sistema. Para o POE, a falta de expressividade e as redundâncias (geralmente presentes neste paradigma) não ficaram evidentes neste exemplo, considerando-se a eficiência adicionada pelo uso do *Publish/Subscribe* e também da pequena quantidade de combinações possíveis para os eventos e estados. Ainda para o POE, observou-se possivelmente um *over-engineering* na resolução do problema. Com relação ao uso de recursos de rede, observou-se que o número de mensagens

[1] Destaca-se que o desenvolvimento do *Framework* PON C++ 4.0 IoT e uma versão deste experimento de sensores para IoT foram relatados também sob a forma de um artigo científico publicado na trilha de temas emergentes Cidades Inteligentes, a qual faz parte do Simpósio Brasileiro de Sistemas de Informação (SBSI) (FIGUEIREDO; SIMÃO; VENDRAMIN, 2022).

necessárias para as implementações em PON e POE foram semelhantes entre si e também com trabalhos relacionados como, por exemplo, ao resultado observado por Barreto, Vendramin e SIMÃO (2018), considerando-se requisitos de confiabilidade similares (sem garantia de entrega das mensagens). Ainda no mesmo experimento, foi possível observar também a similaridade de uma implementação da aplicação em PON totalmente local e uma implementação com partes distribuídas, evidenciando que as características de expressividade da declaração das regras do paradigma são mantidas, mesmo no cenário distribuído.

No terceiro experimento, abordou-se um exemplo simulado no contexto de uma casa inteligente a qual contempla um conjunto de sensores e atuadores sendo compartilhados para a realização de múltiplas funções. Neste contexto, através do compartilhamento de entidades na implementação em PON, observou-se, por exemplo, uma redução de 39.6% na quantidade de *Premises* do sistema como um todo, isto é, evitou-se que uma mesma avaliação relacional de uma *Premise* fosse realizada em mais de uma parte do código. Com o compartilhamento, a avaliação relacional passou a ser realizada apenas uma vez e utilizada em todos os contextos pertinentes. Observou-se também, de forma semelhante ao experimento do portão eletrônico, a proximidade das implementações da aplicação em PON em ambiente local e ambiente distribuído e a expressividade em alto nível das regras, reforçando a potencial facilidade no desenvolvimento de aplicações (mesmo distribuídas) utilizando-se PON. Em relação ao POE, observa-se neste experimento que a implementação utilizando-se o padrão *publish/subscribe* reduz, em certa medida, as redundâncias geralmente presentes nos paradigmas imperativos, mas não as elimina. A expressividade do conhecimento lógico causal no POE se mostrou adequada para o experimento proposto, porém tende a ficar confusa e pouco eficiente em cenários nos quais é necessário o compartilhamento de entidades e/ou com grande número de combinações das avaliações considerando-se os estados e os eventos possíveis. Ademais, a distribuição do processamento para a aplicação em POE não é observada com a mesma naturalidade e granularidade da implementação em PON, sendo necessário um esforço adicional para a sua realização.

Em suma, com os avanços incorporados no *Framework* PON C++ 4.0 IoT e os resultados apresentados com os testes e experimentos realizados, considera-se que os objetivos, previamente apresentados na Seção 1.4 e 1.5, foram satisfeitos. Com o novo *framework* apresentado, é possível a distribuição das entidades constituintes do PON por meio do uso de arquitetura e protocolo de comunicação padronizado para IoT. Assim sendo, o *Framework* PON C++ 4.0 IoT apresenta, enquanto arquétipo de paradigma de programação/desenvolvimento emergente, as quatro potencialidades das propriedades do PON, sendo o desenvolvimento em alto nível, com alto desempenho, paralelismo (sendo estas três herdadas e mantidas do *Framework* PON C++ 4.0) e agora também com a possibilidade de distribuição de cada elemento do PON.

5.2 Trabalhos Futuros

Esta seção apresenta um conjunto de trabalhos futuros vislumbrados durante os desenvolvimentos apresentados, mas que estão fora do escopo dos objetivos dessa presente dissertação.

5.2.1 Entidades distribuídas em LingPON

Atualmente, a tecnologia LingPON (na versão 2.0 e mesmo na versão 3.0, ainda em desenvolvimento ((CHIERICI; SKORA; SIMÃO, 2022) (SKORA; CHIERICI; SIMÃO, 2022)) não contempla a definição de entidades para execução distribuída. Como trabalhos futuros, sugere-se um aprimoramento da tecnologia LingPON utilizando-se os tipos de entidades distribuídas apresentadas na Seção 3.1. Inicialmente, seria possível a geração de código em *Framework* PON C++ 4.0 IoT, utilizando-se as interfaces apresentadas na Seção 3.3.2. Adicionalmente, pode-se estender também a adição do suporte nativamente em código C++ específico-notificante (como os definidos por Oshiro (2021)), utilizando as estruturas e o mecanismo de distribuição similares aos adicionados ao *Framework* PON C++ 4.0 IoT.

5.2.2 Ferramentas para depuração de aplicações PON utilizando o protocolo MQTT

Conforme previamente observado no âmbito das pesquisas do PON, uma das maiores dificuldades apresentadas no desenvolvimento de aplicações em PON é justamente a depuração de código, devido ao fluxo de execução do PON que difere daquele de uma aplicação tradicional em C++, que acontece de forma sequencial. Em função disso, as ferramentas de depuração existentes nas IDEs de C++ não atendem às necessidades da depuração de um programa em PON (RONSZCKA, 2012) (NEVES, 2021).

Com o uso do protocolo MQTT em sinergia com PON é possível também que seja utilizado no desenvolvimento de aplicações PON o conjunto de ferramentas relacionadas ao protocolo MQTT, disponibilizado de forma comercial, científica e/ou em comunidades de desenvolvedores. Como exemplo, utilizou-se durante a dissertação (Seções 4.2.4 e 4.3.5) o software Wireshark, o qual possui nativamente um interpretador (*dissector*) para a visualização das mensagens MQTT. Essas mensagens, no contexto do *Framework* PON C++ 4.0 IoT, são relacionadas às notificações. Como outro exemplo, pode-se utilizar a aplicação *MQTT Explorer* (NORDQUIST, 2020), a qual permite a visualização gráfica e hierárquica dos tópicos existentes em um *broker* MQTT e suas respectivas mensagens trafegadas (conforme apresentado na Seção 3.3.3.1). No contexto do *Framework* PON C++ 4.0 IoT, a hierarquia dos tópicos e as mensagens trafegadas estão relacionadas com as entidades distribuídas do sistema e com as suas interações via notificações, respectivamente.

Dessa forma, vislumbra-se a possibilidade do uso e/ou da adaptação de ferramentas existentes para o MQTT como mais uma possibilidade para depuração de código em PON, por meio da observação e/ou tratamento do tráfego (dinâmico ou estático) das mensagens que passam pelo *broker* MQTT.

5.2.3 Realização de experimentos para avaliar características do PON

A disponibilização do *Framework* PON C++ 4.0 IoT pode permitir também a realização de experimentos abordando conceitos teóricos do PON nunca ou pouco explorados em aspectos técnicos, particularmente em sistemas distribuídos.

Pode-se avaliar, por exemplo, características referentes à redundância e à robustez das aplicações desenvolvidas em PON. Neste contexto, o *Framework* PON C++ 4.0 IoT apresenta as bases para as implementações, particularmente utilizando-se dos tipos de Entidades Redundantes via Rede.

Outra característica, o *Framework* PON C++ 4.0 IoT pode ser utilizado para a realização de experimentos que avaliem conceitos referentes a potenciais mecanismos de resolução de conflitos e também com a garantia de determinismo em aplicações PON, particularmente em sistemas distribuídos.

REFERÊNCIAS

AGHA, G. A. **ACTORS: A Model of Concurrent Computation in Distributed Systems.** 1985. Tese (Doutorado) — University of Michigan, 1985. Disponível em: http://hdl.handle.net/1721.1/6952.

AL-MASRI, E. *et al.* Investigating messaging protocols for the internet of things (iot). **IEEE Access**, v. 8, p. 94880–94911, 2020.

BABU, A. A. **Notification Oriented Paradigm as a Green Technology. Development of a Simulated Sensor Correlation Application with NOP C++ Framework 4.0 and Comparing Green Aspects with usual OOP Languages**. 2022. Dissertação (Mestrado) — Université de Lorraine, France, 2022.

BALAJI, A. Dissecting mqtt using wireshark. 2017. Disponível em: https://dzone.com/articles/dissecting-mqtt-using-wireshark. Acesso em: 17 de outubro de 2021.

BANASZEWSKI, R. F. **Paradigma orientado a notificações: avanços e comparações**. 2009. Dissertação (Mestrado), 2009. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Brazil. Disponível em: http://files.dirppg.ct.utfpr.edu.br/cpgei/Ano_2009/dissertacoes/Dissertacao_500_2009.pdf. Acesso em: 17 de outubro de 2021.

BARRETO, W. R. M.; VENDRAMIN, A. C. B. K.; SIMÃO, J. M. Notification oriented paradigm for distributed systems. *In*: **Computer on the Beach 2018**. [*S.l.*: *s.n.*], 2018.

BATISTA, M. V. Proposta de um método de aplicação da teoria de projeto axiomático ao desenvolvimento de software pon-por. Curitiba, 2013. Dissertação (Mestrado em Engenharia Elétrica e Informática Industrial) - Universidade Tecnológica Federal do Paraná. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/613. Acesso em: 17 de outubro de 2021.

BAUTSCH, M. Cycles of software crises. 2007. ENISA Quarterly on Secure Software, vol. 3, no. 4, p. 3-5.

BELMONTE, D.; SIMÃO, J. M.; STADZISZ, P. C. Proposta de um método para distribuição de carga de trabalho usando o paradigma orientado a notificações (pon). **Revista SODEBRAS**, v. 8, n. 84, 2012.

BELMONTE, D. L. *et al.* A new method for dynamic balancing of workload and scalability in multicore systems. **IEEE Latin America Transactions**, v. 14, n. 7, p. 3335–3344, 2016.

BROOKSHEAR, G. **Computer Science: An Overview**. [*S.l.*]: Addison Wesley, 2012.

CHIERICI, G. B. Junoc++ e nopl lite: uma nova forma de compor aplicções do paradigma orientado a notificações em alto nível por meio de um novo framework em c++ e um dialeto de nopl. 2020. Trabalho realizado na disciplina Tópicos Especiais em EC: Paradigma Orientado A Notificações (TEC0301).

CHIERICI, G. B.; SKORA, L. E. B.; SIMÃO, J. M. Proposta e implementação da tecnologia lingpon 3.0 para o paradigma orientado a notificações. *In*: **XXVII Seminário de Iniciação Científica e Tecnológica da UTFPR**. [*S.l.*: *s.n.*], 2022.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems**: Concepts and design. [*S.l.*]: Addison Wesley Longman, 2011. 1047 p.

DEORE, R. K.; SONAWANE, V. R.; SATPUTE, P. H. Internet of thing based home appliances control. *In*: **2015 International Conference on Computational Intelligence and Communication Networks (CICN)**. [*S.l.*: *s.n.*], 2015. p. 898–902.

DIERKS, T.; RESCORLA, E. **The Transport Layer Security (TLS) Protocol Version 1.2**. 2008. Disponível em: https://www.rfc-editor.org/info/rfc5246. Acesso em: 02 de junho de 2022.

DORSEMAINE, B. *et al.* Internet of things: A definition amp; taxonomy. *In*: **2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies**. [*S.l.*: *s.n.*], 2015. p. 72–77.

ECLIPSE FOUNDATION. **Python Client**. [*S.l.*], 2022. Disponível em: https://www.eclipse.org/paho/index.php?page=clients/python/index.php. Acesso em: 24 de junho de 2022.

FAISON, T. **Event-Based Programming: Taking Events to the Limit**. 1st. ed. USA: Apress, 2011. ISBN 1430243260.

FIELDING, R. *et al.* **Hypertext Transfer Protocol – HTTP/1.1**. 1999. Disponível em: https://www.rfc-editor.org/info/rfc2616. Acesso em: 02 de junho de 2022.

FIGUEIREDO, L.; SIMÃO, J. M.; VENDRAMIN, A. Paradigma orientado a notificações para aplicações de internet das coisas em cidades inteligentes. *In*: **Anais Estendidos do XVIII Simpósio Brasileiro de Sistemas de Informação - Curitiba/PR**. SBC, 2022. p. 326–333. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/sbsi_estendido/article/view/21608.

FORGY, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. **Artificial Intelligence**, v. 19, n. 1, p. 17–37, 1982. ISSN 0004-3702. Disponível em: https://www.sciencedirect.com/science/article/pii/0004370282900200.

GABBRIELLI, M.; MARTINI, S. **Programming languages: principles and paradigms**. [*S.l.*]: Springer Science & Business Media, 2010.

GAMMA, E. *et al.* **Design Patterns: Elements of Reusable Object-Oriented Software.** [*S.l.*: *s.n.*], 1995.

GOOGLE. **Googletest Primer**. [*S.l.*], 2022. Disponível em: https://google.github.io/googletest/primer.html. Acesso em: 17 de julho de 2022.

HANDOSA, M.; GRAčANIN, D.; ELMONGUI, H. G. Performance evaluation of mqtt-based internet of things systems. *In*: **2017 Winter Simulation Conference (WSC)**. [*S.l.*: *s.n.*], 2017. p. 4544–4545.

HASAN, M. **State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally**. 2022. Disponível em: https://iot-analytics.com/number-connected-iot-devices/. Acesso em: 07 de junho de 2022.

HENZEN, A. F. Portabilidade do framework pon de c++ standard para c e java. 2015. Relatório técnico PPGCA/UTFPR.

HEWITT, C.; BISHOP, P.; STEIGER, R. A universal modular actor formalism for artificial intelligence. *In*: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. (IJCAI'73), p. 235–245.

ITU-T STANDARD. **Overview of IoT**. [*S.l.*], 2012. Disponível em: https://www.itu.int/rec/T-REC-Y.2060-201206-I. Acesso em: 17 de setembro de 2022.

JENNINGS, N. R. Agent-oriented software engineering. *In*: **Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: MultiAgent System Engineering**. Berlin, Heidelberg: Springer-Verlag, 1999. (MAAMAW '99), p. 1–7. ISBN 3540662812.

KAISLER, S. H. **Software paradigms**. [*S.l.*]: John Wiley Sons, 2005.

KERSCHBAUMER, R. *et al.* Paradigma orientado a notificações para a síntese de lógica reconfigurável. 2015. LA-CCI/CBIC. ISBN: 9788569972006. Disponível em: https://www.researchgate.net/publication/283018125_Paradigma_Orientado_a_Notificaes_para_a_Sntese_de_Lgica_Reconfigurvel. Acesso em: 17 de outubro de 2021.

KOSSOSKI, C. **Proposta de um método de teste para processos de desenvolvimento de software usando o paradigma orientado a notificações**. dez. 2015. 268 p. Dissertação (Mestrado) — Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2015.

LEE, P.-Y.; CHENG, A. Hal: a faster match algorithm. **IEEE Transactions on Knowledge and Data Engineering**, v. 14, n. 5, p. 1047–1058, 2002.

LIAO, Y. *et al.* Ontology-based model-driven patterns for notification-oriented data-intensive enterprise information systems. *In*: **7th International Conference on Information Society and Technology (ICIST 2017)**. Kopaonik, Serbia: ICIST, 2017. p. v. 1. p. 148–153. ISSN 0000-0000.

LIGHT, R. A. Mosquitto: server and client implementation of the mqtt protocol. **Journal of Open Source Software**, The Open Journal, v. 2, n. 13, p. 265, 2017. Disponível em: https://doi.org/10.21105/joss.00265. Acesso em: 17 de outubro de 2021.

LINHARES, R. R. **Contribuição para o desenvolvimento de uma arquitetura de computação própria ao paradigma orientado a notificações**. 2015. Tese (Doutorado) — Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology - Paraná (UTFPR). Curitiba, Brazil., 2015. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/1324. Acesso em: 17 de outubro de 2021.

MAMANN, L. *et al.* Paradigma orientado a notificações aplicado à programação de microcontroladores. *In*: **Anais Estendidos do XI Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2021. p. 134–139. ISSN 2763-9002. Disponível em: https://sol.sbc.org.br/index.php/sbesc_estendido/article/view/18505.

MANRIQUE, J. A.; RUEDA-RUEDA, J. S.; PORTOCARRERO, J. M. Contrasting internet of things and wireless sensor network from a conceptual overview. *In*: **2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)**. [*S.l.*: *s.n.*], 2016. p. 252–257.

MARTINI, G. H. K. *et al.* Multi-threading capability evaluation of the notification oriented programming language for the x86 architecture. **ICIST**, p. 44–49, 2021.

MARTINI, G. H. K.; SIMÃO, J.; LINHARES, R. A new method for dynamic balancing of workload and scalability in multicore systems. 2018. Trabalho realizado na disciplina Tópicos Avançados Em Sistemas Embarcado (CAES102 – PPGCA).

MENDES, C. C. S. *et al.* Lingpon 2.0 e compilador para framework pon c++ 3.0 e pon-ip. 2019. Relatório da disciplina 'Estudos Especiais Em Paradigmas de Programação', Programa de Pós-Graduação em Engenharia Elétrica e Informática Industria da Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. isponível em: https://pessoal.dainf.ct.utfpr.edu.br/jeansimao/PON/2019_Paradigmas_LingComp_Mendes_Ferreira_ArtigoRelatorio.pdf. Acesso em: 09 de junho de 2022.

MISHRA, B.; KERTESZ, A. The use of mqtt in m2m and iot systems: A survey. **IEEE Access**, v. 8, p. 201071–201086, 2020.

MOSER, K.; HARDER, J.; KOO, S. G. M. Internet of things in home automation and energy efficient smart home technologies. *In*: **2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)**. [*S.l.*: *s.n.*], 2014. p. 1260–1265.

NAIK, N. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. *In*: **2017 IEEE International Systems Engineering Symposium (ISSE)**. [*S.l.*: *s.n.*], 2017. p. 1–7.

NEGRINI, F. **Tecnologia NOPL Erlang-Elixir – paradigma orientado a notificações via uma abordagem orientada a microatores assíncronos.** dez. 2019. Dissertação (Mestrado) — Programa de Pós-Graduação em Engenharia Elétrica e Informatica Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2019.

NEGRINI, F. *et al.* Nopl-erlang: Programação multicore transparente em linguagem de alto nível. *In*: **Anais da V Escola Regional de Alto Desempenho do Rio de Janeiro**. Porto Alegre, RS, Brasil: SBC, 2019. p. 16–20. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/eradrj/article/view/9535. Acesso em: 17 de outubro de 2021.

NEVES, F. d. S. **Contribuição para a concepção de aplicações no Paradigma Orientado a Notificações por meio de programação genérica**. dez. 2021. 322 p. Dissertação (Mestrado) — Programa de Pós Graduação em Computação Aplicada, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2021.

NEVES, F. S.; SIMÃO, J. M.; LINHARES, R. R. Application of generic programming for the development of a c++ framework for the notification oriented paradigm. architecture. **ICIST**, p. 56–61, 2021.

NIKOUKAR, A. *et al.* Low-power wireless for the internet of things: Standards and applications. **IEEE Access**, v. 6, p. 67893–67926, 2018.

NORDQUIST, T. **MQTT Explorer, An all-round MQTT client that provides a structured topic overview**. [*S.l.*], 2020. Disponível em: http://mqtt-explorer.com/. Acesso em: 17 de setembro de 2022.

OASIS STANDARD. **MQTT Version 3.1.1**. [*S.l.*], 2014. Disponível em: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. Acesso em: 17 de outubro de 2021.

OLIVEIRA, R. N. **Paradigma Orientado a Notificações Aplicado em Sistemas de Assistência à Autonomia no Domicílio.** dez. 2019. Dissertação (Mestrado) — Programa de Pós-Graduação em Engenharia Elétrica e Informatica Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2019.

OLIVEIRA, R. N. *et al.* Notification oriented paradigm applied to ambient assisted living tool. **IEEE Latin America Transactions**, v. 16, n. 2, p. 647–653, 2018.

OSHIRO, L. *et al.* Linguagem e compilador para o paradigma orientado a notificações: Uma solução performante orientada a regras. *In*: **Anais da XII Escola Regional de Alto**

Desempenho de São Paulo. Porto Alegre, RS, Brasil: SBC, 2021. p. 61–64. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/eradsp/article/view/16706.

OSHIRO, L. K. **Contribuição em paradigma orientado a notificações: evolução da tecnologia LingPON 2.0 via aprimoramento da linguagem e compilador para código notificante modular em C++**. 2021. Dissertação (Mestrado) — Mestrado em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2021.

PAN, J.; DESOUZA, G.; KAK, A. Fuzzyshell: a large-scale expert system shell using fuzzy logic for uncertainty reasoning. **IEEE Transactions on Fuzzy Systems**, v. 6, n. 4, p. 563–581, 1998.

PURCELL, S.; DRAKE, F. L. J.; HETTINGER, R. **unittest — Unit testing framework**. [*S.l.*], 2003. Disponível em: https://docs.python.org/3/library/unittest.html. Acesso em: 24 de junho de 2022.

RICH, E.; KNIGHT, K. **Artificial Intelligence**. [*S.l.*]: McGraw-Hill, 1991.

RONSZCKA, A. F. Contribuição para a concepção de aplicações no paradigma orientado a notificações (pon) sob o viés de padrões. 2012. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/327. Acesso em: 17 de outubro de 2021.

RONSZCKA, A. F. Materialização efetiva do paradigma orientado a notificações e demonstração de suas propriedades elementares e determinísticas. 2014. Proposta de Doutorado, Programa de Pós-Graduação em Engenharia Elétrica e Informática Industria da Universidade Tecnológica Federal do Paraná (UTFPR). Curitiba – Paraná (PR), Brazil.

RONSZCKA, A. F. **Método para a criação de linguagens de programação e compiladores para o paradigma orientado a notificações em plataformas distintas.** dez. 2019. 375 p. Tese (Doutorado) — Programa de Pós Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2019.

RONSZCKA, A. F. *et al.* Notification-oriented programming language and compiler. *In*: **2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)**. [*S.l.*: *s.n.*], 2017. p. 125–131.

ROY, P. V. Programming paradigms for dummies: What every programmer should know. 2012.

SAMUEL, A.; SIPES, C. Making internet of things real. **IEEE Internet of Things Magazine**, v. 2, n. 1, p. 10–12, 2019.

SCHÜTZ, F. **NeuroPON: uma abordagem para o desenvolvimento de redes neurais artificiais utilizando o paradigma orientado a notificações**. 2019. Tese (Doutorado) — Doutorado em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná, Curitiba, 2019. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/4487. Acesso em: 14 de maio de 2022.

SCHÜTZ, F. *et al.* Proposal of a declarative and parallelizable artificial neural network using the notification-oriented paradigm. **Neural Computing and Applications**, v. 30, p. 1715–1731, 2018.

SCOTT, M. L. **Programming Language Pragmatics**. [*S.l.*]: Morgan Kaufmann Publishers Inc, 2000.

SHALI, A. Actor oriented programming in chapel. **Spring**, 2010. Disponível em: https://chapel-lang.org/education/cs380p-actors.pdf. Acesso em: 17 de setembro de 2022.

SIMÃO, J. M. **Proposta de uma arquitetura de controle para sistemas flexíveis de manufatura baseada em regras e agentes**. 2001. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology Paraná (UTFPR). Curitiba – Paraná (PR), Brazil.

SIMÃO, J. M. **A Contribution to the Development of a HMS simulation tool and Proposition of a Meta-Model for Holonic Control**. 2005. Tese (Doutorado) — School in Electrical Engineering and Industrial Computer Science (CPGEI) at Federal University of Technology - Paraná (UTFPR, Brazil) and Research Center For Automatic Control of Nancy (CRAN) - Henry Poincaré University (UHP, France), 2005. Disponível em: http://arquivos.cpgei.ct.utfpr.edu.br/Ano_2005/teses/Tese_012_2005.pdf. Acesso em: 17 de outubro de 2021.

SIMÃO, J. M. *et al.* Evaluation of the notification oriented paradigm applied to sentient computing. *In*: **2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing**. [*S.l.*: *s.n.*], 2014. p. 253–260.

SIMÃO, J. M.; STADZISZ, P. C. Paradigma orientado a notificações (pon)–uma técnica de composição e execução de software orientado a notificações. 2008. PEDIDO DE PATENTE: Privilégio de Inovação. Número do registro: PI08055181, data de depósito: 26/11/2008, INPI - Instituto Nacional da Propriedade Industrial. Universidade Tecnológica Federal do Paraná - UTFPR (Demanda Agência de Inovação, 2007). Disponível em: http://bit.ly/1SAQod3. Acesso em: 17 de outubro de 2021.

SIMÃO, J. M.; STADZISZ, P. C. Inference based on notifications: a holonic metamodel applied to control issues. **Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on**, IEEE, v. 39, n. 1, p. 238–250, 2009.

SIMÃO, J. M.; STADZISZ, P. C. Inference based on notifications: A holonic metamodel applied to control issues. **IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans**, v. 39, n. 1, p. 238–250, 2009.

SIMÃO, J. M.; TACLA, C. A.; STADZISZ, P. C. Holonic control metamodel. **Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on**, IEEE, v. 39, n. 5, p. 1126–1139, 2009.

SIMÃO, J. M. *et al.* Notification oriented paradigm (nop) and imperative paradigm: A comparative study. 2012. Journal of Software Engineering and Applications (JSEA), p.402-416, v.5, n.6, 2012. ISSN: 1945-3116. DOI 10.4236/jsea.2012.59083. Disponível em: https://www.researchgate.net/publication/272666606_Notification_Oriented_Paradigm_%28NOP%29_and_Imperative_Paradigm_A_Comparative_Study. Acesso em: 17 de outubro de 2021.

SKORA, L. E. B.; CHIERICI, G. B.; SIMÃO, J. M. Avanços nos alvos de compilação da linguagem do paradigma orientado a notificações. *In*: **XXVII Seminário de Iniciação Científica e Tecnológica da UTFPR**. [*S.l.*: *s.n.*], 2022.

SUEDA, Y.; SATO, M.; HASUIKE, K. Evaluation of message protocols for iot. *In*: **2019 IEEE International Conference on Big Data, Cloud Computing, Data Science Engineering (BCD)**. [*S.l.*: *s.n.*], 2019. p. 172–175.

TAIVALSAARI, A.; MIKKONEN, T. A taxonomy of iot client architectures. **IEEE Software**, v. 35, n. 3, p. 83–88, 2018.

TALAU, M. Ponip: Uso do paradigma orientado a notificações em redes ip. 2016. Trabalho realizado na disciplina Paradigmas Orientado a Notificações - CPGEI/UTFPR.

TUCKER, A. B.; NOONAN, R. E. **Programming Languages: Principles and Paradigms**. N/A: McGraw-Hill Internation Edition, 2007. 600 p.

VALENÇA, G. Z. Contribuição para materialização do paradigma orientado a notificações (pon) via framework e wizard. 2012. Dissertação (Mestrado em Computação Aplicada - PPGCA) – Universidade Tecnológica Federal do Paraná (UTFPR), Curitiba. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/393. Acesso em: 17 de outubro de 2021.

WALEED, J. *et al.* Smart home as a new trend, a simplicity led to revolution. *In*: **2018 1st International Scientific Conference of Engineering Sciences - 3rd Scientific Conference of Engineering Science (ISCES)**. [*S.l.*: *s.n.*], 2018. p. 30–33.

WATT, D. **Programming Language Design Concepts.** [*S.l.*]: J. Willey Sons, 2004.

WIECHETECK, L. V. B. Método para projeto de software usando o paradigma orientado a notificações–pon. 2012. Master in Science Thesis, Graduate School in Electrical Engineering and Industrial Computer Science (CPGEI) at the Federal University of Technology – Paraná (UTFPR). Curitiba – Paraná (PR), Brazil. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/212. Acesso em: 17 de outubro de 2021.

WUKKADADA, B. *et al.* Comparison with http and mqtt in internet of things (iot). *In*: **2018 International Conference on Inventive Research in Computing Applications (ICIRCA)**. [*S.l.*: *s.n.*], 2018. p. 249–253.

XAVIER, R. D. **Paradigmas de desenvolvimeto de software: Comparação entre abordagens orientada a eventos e orientada a notificações**. dez. 2014. 240 p. Dissertação (Mestrado) — Programa de Pós Graduação em Computação Aplicada, Universidade Tecnológica Federal do Paraná, Curitiba, dez. 2014.

XAVIER, R. D. *et al.* Paradigmas de desenvolvimeto de software: Comparação entre abordagens orientada a eventos e orientada a notificações. **Revista SODEBRAS**, SODEBRAS, v. 9, n. 101, 2014.

YOKOTANI, T.; SASAKI, Y. Comparison with http and mqtt on required network resources for iot. *In*: **2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)**. [*S.l.*: *s.n.*], 2016. p. 1–6.

YOKOTANI, T.; SASAKI, Y. Transfer protocols of tiny data blocks in iot and their performance evaluation. *In*: **2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)**. [*S.l.*: *s.n.*], 2016. p. 54–57.