

UNIOESTE UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
CENTRO DE ENGENHARIAS E CIÊNCIAS EXATAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Desenvolvimento de um sistema multiusuário para edifícios inteligentes

Victor Hugo de Almeida Alicino

FOZ DO IGUAÇU

2025

Sumário

1	Introdução	1
1.1	Objetivos	2
1.1.1	Objetivo Geral	2
1.1.2	Objetivos Específicos	2
1.2	Estrutura do Trabalho	2
2	Revisão Bibliográfica	3
2.1	Edifício Inteligente	3
2.1.1	Intelligent Building	3
2.1.2	Smart Building	4
2.1.3	Sistema de Gerenciamento de Edifício	5
2.1.4	Sistema de Controle e Automação de Edifício	6
2.2	Internet das Coisas	8
2.3	Tecnologias	11
2.3.1	Tecnologias utilizadas em automação de edifícios	11
2.3.1.1	BACnet	11
2.3.1.2	KNX	11
2.3.1.3	ZigBee	12
2.3.2	Tecnologias utilizadas no desenvolvimento deste trabalho	12
2.3.2.1	Tasmota	12
2.3.2.2	MQTT	12
3	Sistema Opus	14
3.1	Servidor Local	16
3.1.1	Arquivo de Configuração	16
3.1.2	Interfaces	17
3.1.3	Drivers	19
3.1.3.1	A Classe de Dispositivos	21
3.1.3.2	O Registrando de Dispositivos	21
3.1.4	Gerenciadores	22
3.1.4.1	CloudManager	22
3.1.4.2	UserManager	22
3.1.4.3	LocationManager	23
3.1.4.4	DeviceManager	24
3.1.5	Banco de Dados	25

3.1.5.1	Estrutura do Banco de Dados	25
3.1.6	Registro e Aquisição de Dispositivos	28
3.2	Servidor Remoto	31
3.2.1	API REST	32
Referências		33

1 Introdução

As cidades inteligentes visam melhorar a qualidade de vida dos cidadãos e promover um desenvolvimento econômico e social sustentável por meio do uso de tecnologias da informação e comunicação (TIC) [MACAYA et al., 2017]. Nesse contexto, os edifícios inteligentes desempenham um papel fundamental. Indivíduos passam grande parte da sua vida dentro de edifícios, crescem, estudam e se desenvolvem neles [PAIVA; JEDON, 2019] [MEYER, 2022]. Dada sua importância no cotidiano urbano, a modernização dos edifícios torna-se essencial para a implementação do conceito de cidades inteligentes. Assim, a integração de novas tecnologias pode otimizar a experiência dos ocupantes e contribuir para a sustentabilidade dessas construções.

Para que um edifício seja classificado como inteligente, é necessária a incorporação de dispositivos capazes de monitorar e controlar o ambiente. Esses dispositivos incluem sensores, responsáveis por captar os dados do ambiente em que se encontram; atuadores que realizam as ações físicas baseadas nas entradas que recebem; e controladores que coordenam a comunicação entre sensores e atuadores para garantir uma automação eficiente [MORVAJ; LUGARIC; KRAJCAR, 2011]. A implementação desses componentes possibilita desde o gerenciamento de iluminação e climatização até sistemas avançados de segurança e eficiência energética.

Nos últimos anos, dispositivos com essas características têm se tornado cada vez mais populares no meio da Internet das Coisas (do inglês, IoT). Com a significativa redução de custo dos sensores nas últimas duas décadas [MICROSOFT, 2019], dispositivos IoT destinados a Casas Inteligentes ganharam ampla aceitação no mercado, devido ao seu custo acessível e facilidade na instalação, com isso, estima-se que ao final da década de 2020, dos mais de 30 bilhões de dispositivos IoT conectados no mundo, 60% serão de consumidores finais [STATISTA, 2024].

Diante da crescente adoção de dispositivos IoT em Casas Inteligentes, surge a questão de por que esses dispositivos ainda não são amplamente utilizados em edifícios coletivos, como escritórios e coworking, para torná-los inteligentes. Os sistemas de automação tradicionais, projetados para grandes edifícios, costumam ser caros e complexos, enquanto os dispositivos IoT voltados ao consumidor final são acessíveis e fáceis de instalar. No entanto, esses dispositivos foram concebidos para um uso individual ou residencial, assumindo que os ocupantes do espaço são previamente conhecidos. Em edifícios coletivos, essa abordagem se torna inviável, pois a rotatividade de usuários exige um modelo flexível e adaptável de controle e segurança. Assim, há um desafio técnico e econômico a ser superado para que dispositivos IoT de Casas Inteligentes possam ser utilizados nesses

ambientes de forma segura e eficiente.

Apesar dessa ampla adoção no mercado residencial, a aplicação desses dispositivos em edifícios coletivos ainda enfrenta desafios específicos. Considerando que dispositivos IoT de Casas Inteligentes são mais acessíveis e fáceis de implementar que sistemas tradicionais de automação predial, mas não são viáveis em edifícios coletivos devido à necessidade de múltiplos usuários compartilharem o mesmo ambiente, como é possível adaptá-los para o uso em edifícios coletivos, garantindo controle adequado para múltiplos ocupantes?

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho visa desenvolver um sistema básico que demonstre a viabilidade da operação simultânea de múltiplos usuários em um edifício inteligente. A proposta é criar uma solução funcional que atenda a múltiplos usuários em um ambiente compartilhado.

1.1.2 Objetivos Específicos

Para alcançar o objetivo do trabalho, é proposto uma aplicação simplificada que permita entender a viabilidade do uso de um sistema de edifício inteligente por diversos usuários, focando no controle das unidades de climatização.

Dentre os principais objetivos específicos destacam-se:

- Realizar uma pesquisa bibliográfica sobre edifícios inteligentes e suas demandas;
- Estudar sobre as aplicações da internet das coisas em um edifício inteligente;
- Definir requisitos para um sistema de gerenciamento de um edifício inteligente;
- Propor e implementar um sistema de gerenciamento para edifícios inteligentes;

1.2 Estrutura do Trabalho

Este trabalho está organizado com a seguinte estrutura:

Capítulo 2: Revisão Bibliográfica, apresenta o contexto histórico dos edifícios inteligentes e uma introdução as tecnologias utilizadas. . .

Capítulo 3: Sistema Opus, onde serão apresentados os componentes do sistema proposto, sua arquitetura e funcionalidades

Capítulo 4: Testes e Resultados, onde serão apresentados os testes realizados após a implementação do sistema proposto.

Capítulo 5: Conclusão e considerações finais.

2 Revisão Bibliográfica

2.1 Edifício Inteligente

O conceito de edifícios com algum tipo de autonomia humana é chamado de edifício inteligente, esse conceito surge nos Estados Unidos em meados da década de 80 junto com os sistemas de automação de segurança e iluminação para edifícios [NEVES; CAMARGO, 2002]. Esses edifícios chamados de edifícios inteligentes têm se tornado populares nos últimos anos, mas os limites e requisitos do que um edifício precisa ter para ser considerado inteligente é algo nebuloso. O conceito de edifício inteligente em português origina de dois conceitos pouco distintos na língua inglesa, *Intelligent Buildings* e *Smart Buildings*, “Intelligent” segundo o *Oxford Learner’s Dictionaries* é aquele que é bom de aprendizado [INTELLIGENT, 2023], enquanto “Smart” é aquele que é inteligente [SMART, 2023], o significado de ambas as palavras não diferem muito entre si, assim como os conceitos de *Intelligent Buildings* e *Smart Buildings*, ambos carregam as mesmas primícias, de um edifício com algum nível de automação, minimizando a interação humana [WONG; LI; WANG, 2005].

2.1.1 Intelligent Building

Intelligent Buildings vem sendo pesquisados e desenvolvidos há pelo menos três décadas [BUCKMAN; MAYFIELD; BECK, 2014] e existem no mínimo 30 definições diferentes para esse conceito [WIGGINTON; HARRIS, 2002], em 1990, Powell traz uma definição de edifício inteligente (Intelligent Building) comentando o que Stubbings diz em 1988,

At present the term ‘intelligent building’ is normally taken to mean ‘a building which totally controls its own environment’ [STUBBINGS, 1986]. This seems to imply that it is the technical control of heating and air conditioning, lighting, security, fire protection, telecommunication and data services, lifts and other similar building operations that is important a control typically given over to a management computer system. Such a definition for a conventionally intelligent building does not suggest user interaction at all. [POWELL, 1990]

Stubbings diz que um edifício inteligente é aquele edifício capaz de controlar seu próprio ambiente e Powell expande essa ideia, definindo que um edifício inteligente é aquele capaz de controlar seus sistemas de Aquecimento, Ventilação e Ar-Condicionado (AVAC), iluminação, segurança, combate a incêndio etc. através de um sistema de gerenciamento

de edifício (Building Management System). Clements-Croome em 2009 define um edifício inteligente como aquele que é responsivo aos requisitos dos seus ocupantes, organizações e sociedade, sendo sustentável no consumo de água e energia, gerando pouca poluição e sendo funcional de acordo com as necessidades do usuário [CLEMENTS-CROOME, 2011] e Brooks sugere que *Intelligent Building* e seu sistema de gestão (Building Management System) são essencialmente a mesma coisa [BUCKMAN; MAYFIELD; BECK, 2014], um sistema de controle que abrange todo o edifício, que conecta, controla e monitora a planta fixa e os equipamentos da instalação [BROOKS, 2012].

2.1.2 Smart Building

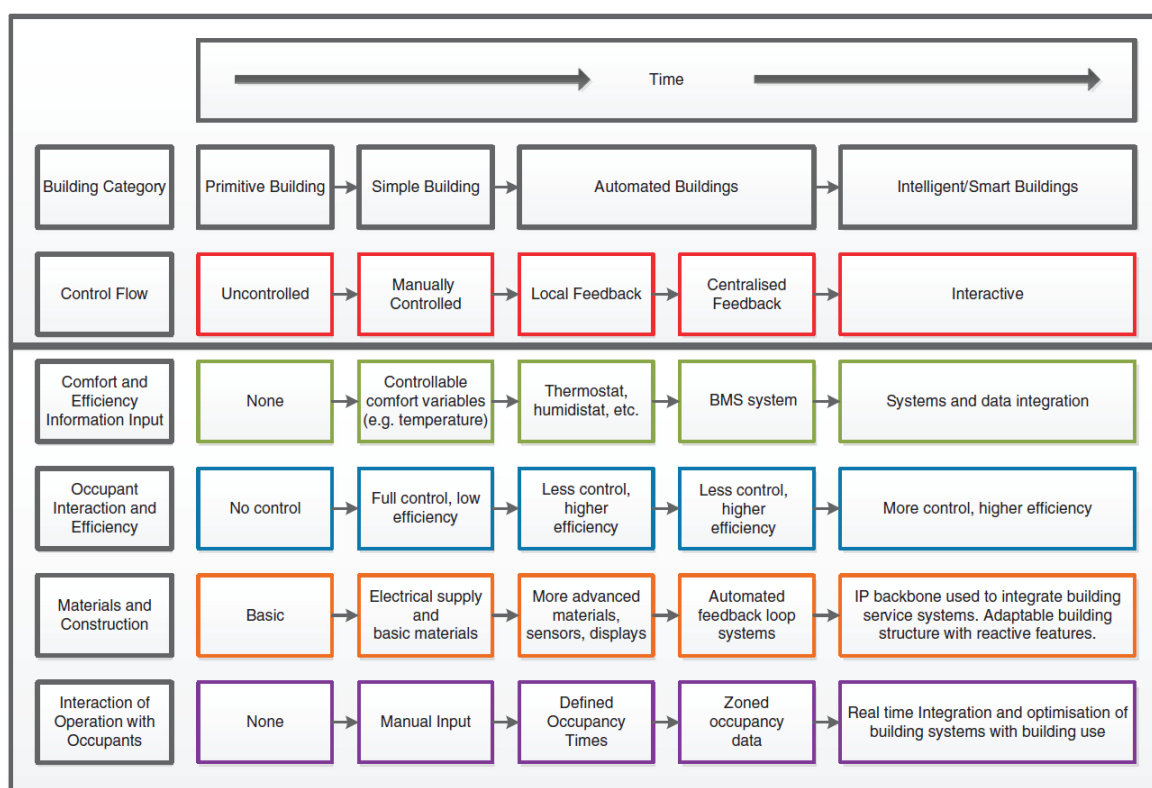


Figura 1 – Progressão da automação em edifícios

Fonte: Buckman, Mayfield e Beck [2014]

Na figura 1, Buckman, Mayfield, Beck mostram uma linha do tempo, partindo de um edifício primitivo com nenhum controle do seu ambiente até um edifício inteligente. Segundo sua pesquisa, sugerem que existem três principais pontos que desenvolvem um edifício até o estado de edifício inteligente, são eles:

1. Longevidade;
2. Energia e Eficiência; e
3. Conforto e Satisfação.

Para atingir o estado de edifício inteligente nos três pontos, existem quatro aspectos que variam o nível do edifício de primitivo a inteligente, são eles:

1. Inteligência, a forma como são coletadas informações das operações do edifício e sua resposta;
2. Controle, a interação entre ocupantes e o edifício;
3. Materiais e Construção, a forma física do edifício; e
4. Empresa, a forma como são coletadas informações e usadas para melhorar a performance dos ocupantes.

A evolução desses métodos em um edifício partindo do primitivo ao inteligente é mostrada na figura 1. De acordo com a pesquisa, Buckman, Mayfield, Beck sugerem que em um *Smart Building*, os quatro aspectos que variam o nível do edifício são desenvolvidos lado a lado, usando a informação de um na operação do outro, diferindo de um *Intelligent Building* que desenvolve a “Inteligência” citada acima de forma independente dos outros três aspectos. *Smart Buildings* usam tanto o controle humano, quanto a automação para atingir os quatro aspectos apresentados acima. O aspecto de “Controle”, o mais importante para esse trabalho, deve apresentar informações do aspecto de “Inteligência” para os ocupantes do edifício, para que os mesmos possam se adaptar ao edifício assim como o edifício se adapta a eles [BUCKMAN; MAYFIELD; BECK, 2014].

Sensores inteligentes, que podem ser adicionados a qualquer momento da vida de um edifício [KAMAL et al., 2021], frutos da internet das coisas (IoT, do inglês Internet of Things) permitem uma rápida implementação de instalações para edifícios inteligentes, como gerenciamento do sistema de AVAC, sistemas de segurança, monitoramento por câmeras, alertas para eventos como incêndio, vazamento de gás e monitoramento da integridade estrutural do edifício [BELLINI; NESI; PANTALEO, 2022]. A adoção de IoT promove a conectividade entre sensores, dispositivos e sistemas do edifício a nuvem, tal conexão promove o uso de aplicações que usam os dados coletados [BERKOBEN; KAED; SODORFF, 2020].

2.1.3 Sistema de Gerenciamento de Edifício

Sistemas de Gerenciamento de Edifício, ou Building Management Systems (BMS) em inglês são controladores inteligentes baseados em microprocessadores instalados na rede para monitorar e controlar os aspectos técnicos de um edifício e seus serviços [APPLIED RISK, 2019]. Um BMS pode controlar componentes com protocolos de mais baixo nível, como BACnet, Modbus e etc. [BERKOBEN; KAED; SODORFF, 2020] os subsis-

temas do BMS ligam as funcionalidades individuais desses equipamentos para que eles possam operar como um único sistema [APPLIED RISK, 2019].

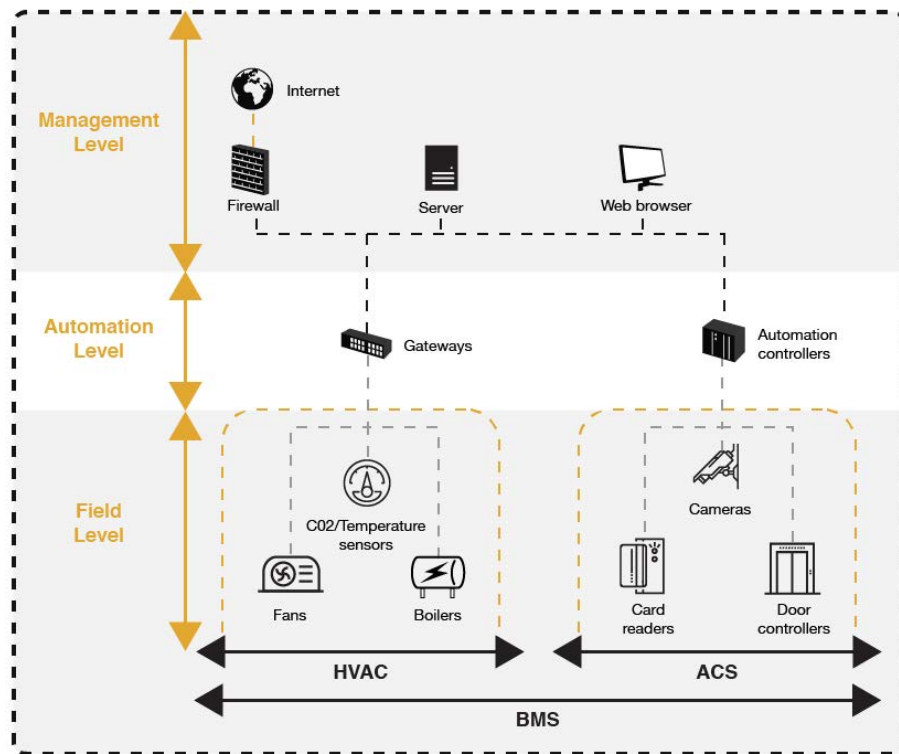


Figura 2 – Relação entre os componentes do BMS e sistemas

Fonte: APPLIED RISK [2019]

A figura 2 mostra um exemplo da relação entre os componentes de um BMS.

2.1.4 Sistema de Controle e Automação de Edifício

Quando um Sistema de Gerenciamento de Edifício passa a satisfazer os requisitos da ISO 16484 [International Organization for Standardization, 2020], tais como suporte a BACnet ele passa a ser um Sistema de Controle e Automação de Edifício (Building Automation Control System, BACS) [European Committee for Standardization, 2006]. Os conceitos de BMS e BACS possuem mais similaridades do que diferenças sendo difícil encontrar na literatura materiais que os diferenciem bem, sendo assim, para este trabalho, Sistemas de Gerenciamento de Edifícios (BMS) e Sistemas de Controle e Automação de Edifícios (BACS) serão considerados sinônimos.

Assim como o Sistema de Gerenciamento de Edifício, o Sistema de Controle e Automação de Edifício também é responsável por controlar e monitorar os aspectos técnicos de um edifício e seus serviços, o BACS também é dividido em três camadas, como representado na figura 2, sendo elas:

- Camada de Campo (*Field Layer*);

- Camada de Automação (*Automation Layer*); e
- Cada de Gerenciamento (*Management Layer*).

A camada de campo é a camada mais baixa, sendo nela encontrados os sensores e atuadores que fazem a interação com o ambiente. A camada de automação é onde os dados são processados, loops de controle são executados e alarmes ativados. Por último, a camada de gerenciamento é a responsável por apresentar os dados do sistema, criar logs de acontecimentos e intermediar o controle do usuário com o sistema. Sistemas mais modernos tendem a separar a lógica da interface gráfica para o usuário com o objetivo de criar acessos mais flexíveis aos BACS [DOMINGUES et al., 2016].

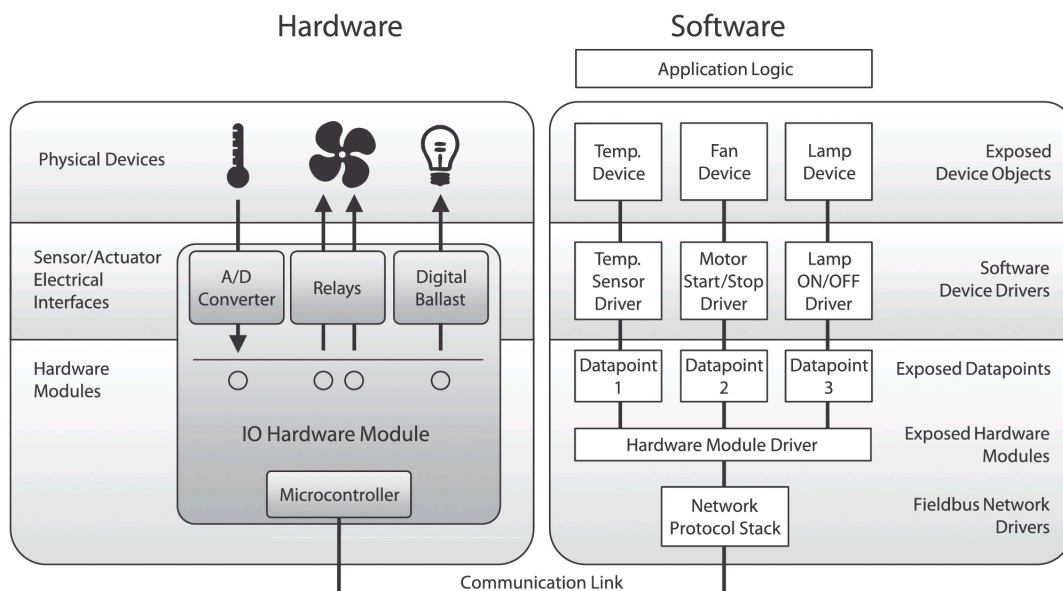


Figura 3 – Pilhas de Hardware e Software envolvidos no Sistema de Controle e Automação de Edifício

Fonte: Domingues et al. [2016]

O caminho da informação em um BACS tem a forma representada pela figura 3, sensores e atuadores interagem com dispositivos físicos através de módulos em hardware, controlados por um microcontrolador responsável por enviar os dados até o BACS através de uma conexão de comunicação (também chamada de barramento de campo ou *fieldbus* em inglês) através de um dos protocolos de baixo nível já citados, como BACnet, KNX, LonWorks, Modbus, ZigBee ou EnOcean. Os dados são recebidos por módulos de hardware que expõem esses dados para o software através de *datapoints* (ou também chamados de *endpoints*, *tags* ou *points*) que interagem com o software para realizar a ação necessária [DOMINGUES et al., 2016].

2.2 Internet das Coisas

Enquanto a internet foi desenvolvida com dados criados por pessoas, a Internet das Coisas é sobre os dados criados por objetos [NORD; KOOHANG; PALISZKIEWICZ, 2019]. Madakam, Ramaswamy e Tripathi definem Internet das Coisas como uma rede de objetos inteligente aberta que tem a capacidade de se auto-organizar, compartilhar informações, dados e recursos [MADAKAM; RAMASWAMY; TRIPATHI, 2015].

Internet das Coisas é muitas vezes referido pela sigla IoT que vem do seu nome em inglês *Internet of Things*. No nome dessa tecnologia temos as duas partes que desempenham os principais papéis: *Internet* e *Coisas*. *Internet* aqui, se refere a mesma internet usadas por bilhões de pessoas ao redor do mundo, já *Coisas*, se refere aos dispositivos com capacidades computacionais atrelados a sensores ou atuadores conectados à rede, o que permite a eles trocarem e consumirem informações.

Em uma rede IoT, dispositivos (muitas vezes chamados de *inteligentes*) tem a habilidade de se comunicarem para monitorar o ambiente em que estão, ou alterar este ambiente. Essas ações podem ser configuradas previamente pelo usuário ou serem definidas na hora com a possibilidade de ser acionadas fora da rede local onde esses dispositivos estão conectados através da internet. Como por exemplo, verificar a temperatura de uma sala sem ter chego nela e solicitar ao sistema que acione as unidades AVAC para que a sala esteja na temperatura desejada ao chegar.

Apesar de ocorrer leves diferenças nos modelos de arquitetura, geralmente um sistema IoT é formado por três camadas, sendo elas:

- Camada Física: responsável por perceber o ambiente físico com sensores, atuadores ou qualquer tipo de interface para perceber e modificar o ambiente físico;
- Camada de Rede: camada de transmissão dos dados através das inúmeras formas de conexões disponíveis como redes sem fio, redes móveis e a internet, a fim de fornecer os dados da Camada Física para a Camada de Aplicação;
- Camada de Aplicação, que oferece os serviços chamados *inteligentes* para os usuários finais.

[YAN; ZHANG; VASILAKOS, 2014].

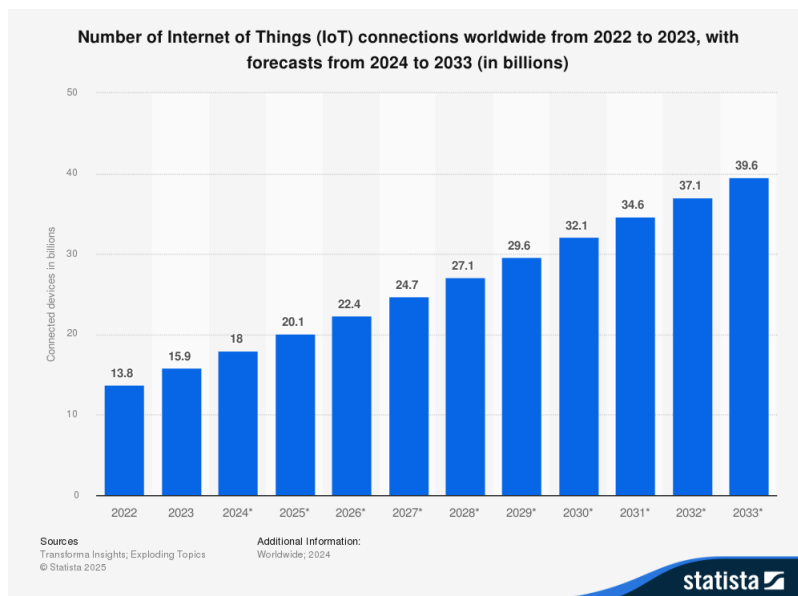
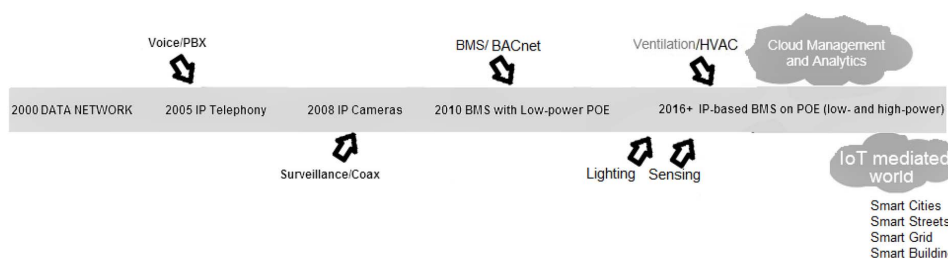


Figura 4 – Números de dispositivos IoT conectados no mundo de 2022 a 2023, com previsões de 2024 a 2033

Fonte: Statista [2024]

Segundo a empresa especializada em coleta de dados, Statista, a quantidade de dispositivos IoT conectados no mundo passa de 18 bilhões em 2024 com projeção para quase o dobro deste número até o fim da década [STATISTA, 2024] como mostrado na figura 4.

Isso se deve muito ao fato de que sensores e microcontroladores ficaram mais baratos, o preço médio dos sensores caiu 200% entre 2004 e 2018 [MICROSOFT, 2019] o que significa mais produtos finais voltado para IoT já que ficou possível embarcar sensores em itens do dia a dia com custos acessíveis.



Note: BACnet is an ASHRAE, ANSI, and ISO 16484-5 standard communications protocol for building automation and control (BACnet was subsumed in ASHRAE/ANSI Standard 135 in 1995, and in ISO 16484-5 in 2003.) The BACnet protocol defines several services that are used to communicate between control devices typically utilized in building (including HVAC, lighting control, access control, and fire detection systems). It specifies a number of network, data link, and physical layer protocols, including but not limited to standards such as IP/Ethernet.

Figura 5 – Convergência da tecnologia de sistemas inteligentes para edifícios nos últimos anos

Fonte: Minoli, Sohraby e Occhiogrosso [2017]

Alguns anos atrás, mesmo em edifícios com BACS, não era incomum encontrar diferentes protocolos, redes e cabos, o que gerava ineficiência tanto na implantação como nos sistemas de gerenciamento. Dessa forma algumas soluções migraram para o uso de

cabo e conjunto de protocolos comuns entre as aplicações, como o cabo de par trançado categoria 6 e o protocolo TCP/IP. Com a convergência para protocolos como TCP/IP facilitou a entrada do IoT em BACS, como mostrado na figura 5 [MINOLI; SOHRABY; OCCHIOGROSSO, 2017].

Já é possível encontrar equipamentos preparados para uso em BACS com suporte a protocolos IoT, como o PFC200 da WAGO que suporta o protocolo MQTT (citado na seção 2.3.2.2).



Figura 6 – Controlador PFC200

Fonte: WAGO [2023]

2.3 Tecnologias

2.3.1 Tecnologias utilizadas em automação de edifícios

Nesta sessão serão apresentados algumas tecnologias utilizadas na automação de edifícios

2.3.1.1 BACnet

Protocolo de Rede de Automação e Controle de Edifícios (Building Automation and Control Network Protocol) ou BACnet é um protocolo mantido e desenvolvido pela ASHRAE com o objetivo de atender as necessidades de comunicações dos sistemas de controle e automação dos edifícios com seus sensores e atuadores. O protocolo fornece meios para que equipamentos troquem informações não importando o serviço que eles realizem [ASHRAE, 2016]. A forma com que o BACnet funciona, permite que usuários não fiquem presos a sistemas proprietários, criando um padrão de comunicação entre os dispositivos como sistemas de AVAC, alarmes de incêndio e etc. uma das suas grandes vantagens é a habilidade de se adaptar a novas tecnologias de rede [BUSHBY; NEWMAN, 2002]. o BACnet define um modelo de informação, criando objetos para os dispositivos conectados, esses objetos representam seus serviços, assim como suas entradas e saídas. Esses objetos chamados de *device object* definem as propriedades do dispositivo como: nome do modelo, fabricante, status e a lista de outros objetos BACnet associados ao dispositivo [DOMINGUES et al., 2016].

2.3.1.2 KNX

KNX ou Konnex é um padrão aberto desenvolvido para ser utilizado em automação de edifícios ou residências [SAPUNDZHI, 2020], da mesma forma que o BACnet, o KNX integra sensores, atuadores e controladores dentro de uma rede, os dispositivos são controlados através de um barramento e são chamados de *Bus Access Unit* ou BAU, sendo possível controlar mais de 65 mil BAUs. No KNX, cada fabricante pode implementar sua própria especificação nos dispositivos. BAUs podem ser separadas em grupos menores chamados de *areas* que podem ser separadas em *lines* de dispositivos. As mensagens trocadas pelos dispositivos KNX pode ser transmitida através de cabos de par trançado, cabos de energia, frequências de rádio ou internet, sendo a forma mais comum o KNX.TP que usa os cabos de par trançado [KRAUS; VIERTEL; BURGERT, 2020]. Um ponto chave do KNX é seu método de comunicação, pouco similar ao MQTT, o KNX utiliza o padrão *observador* na hora de trocar informações, neste padrão múltiplos dispositivos “observadores” serão notificados através de uma única mensagem em *multicast* de

quando os dados em uma fonte sofrerem alterações, dessa forma fica mais fácil a criação de relacionamentos um para muitos (1:N). [DOMINGUES et al., 2016].

2.3.1.3 ZigBee

ZigBee é um dos mais populares padrões de redes *mesh* sem fio. O protocolo é aberto e baseado em pacotes, projetado para ser de fácil uso, baixo consumo e seguro [TOMAR, 2011]. O modelo ZigBee é principalmente definido em três camadas.

- *Application Support Sublayer*: Responsável por vincular *endpoints*, transmitir mensagens entre os dispositivos e o gerenciamento de grupos.
- *ZigBee Device Object*: Responsável pelo gerenciamento de dispositivos; definir o modo de operação do dispositivo; descobrir novos dispositivos e quais serviços de aplicação ele provê; lidar com os pedidos de vinculamento de outros dispositivos.
- *Application Framework*: Abriga as aplicações dos dispositivos.

[DOMINGUES et al., 2016]

2.3.2 Tecnologias utilizadas no desenvolvimento deste trabalho

2.3.2.1 Tasmota

Tasmota é um software de código aberto que atua como firmware para SoC's (System-on-a-Chip) ESP32 da Espressif e possibilita controle e comunicação personalizado com alguns dispositivos comerciais que são baseados nesse SoC, como interruptores inteligentes, lâmpadas inteligentes ou controles infravermelho inteligentes. A comunicação com o firmware é realizada através do protocolo MQTT, permitindo integrar o hardware comercial a qualquer sistema com suporte ao protocolo [TASMOTA, 2020].

2.3.2.2 MQTT

MQTT ou Message Queuing Telemetry Transport [International Organization for Standardization, 2016] é um protocolo de transporte para mensagens no formato *publicar/assinar*, ele é simples, leve, aberto e projetado para ser fácil de implementar, seu foco de uso são ambientes onde recursos são geridos com cuidado, como comunicação *Machine to Machine* (M2M) e Internet das Coisas [OASIS, 2019]. O foco principal do projeto desse protocolo é minimizar o consumo de banda e recursos do dispositivo, sendo assim, ele é capaz de transmitir dados com uma banda baixa e conexões instáveis. O protocolo funciona sobre TCP/IP ou outros protocolos de rede que possam prover as mesmas capacidades do TCP/IP como conexões bi-direcionais. O MQTT oferece três qualidades de serviços (*Quality of Service*, ou QoS) na entrega de mensagens.

- QoS 0: “No máximo uma vez”, mensagens são entregues no máximo uma vez.
- QoS 1: “Pelo menos uma vez”, mensagens são garantidas de serem entregues pelo menos uma vez, pode ocorrer duplicatas.
- QoS 2: “Exatamente uma vez”, mensagens são entregues exatamente uma vez.

Uma estrutura que suporte o protocolo MQTT deve ter pelo menos três componentes,

1. *Publisher* ou Produtor, um cliente MQTT
2. *Broker*, o servidor MQTT que gerencia o recebimento e entrega das mensagens
3. *Subscriber* ou Consumidor, um cliente MQTT

No padrão *publicar/assinar*, o produtor cria uma mensagem e envia para o servidor, consumidores então leem essa mensagem. No MQTT um produtor publica mensagens nos tópicos do *broker*, consumidores interessados em receber essas mensagens se inscreverão nesse tópico e o *broker* se encarregará de entregá-las [MISHRA; KERTESZ, 2020].

3 Sistema Opus

Buscando o objetivo do trabalho, foi proposto um sistema denominado de **Opus**, dividido em três partes, *Opus* (Servidor Local), *Maestro* (Servidor Remoto) e *Conductor* (Interface de Usuário), ele tem o objetivo principal de permitir que múltiplos usuários possam controlar dispositivos IoT.

O sistema **Opus** foi desenvolvido pensando em edifícios de uso coletivo, como escritórios, coworking e salas de aula, onde a automação pode ser utilizada para facilitar a vida dos usuários, como controlar a iluminação e climatização.

Inicialmente foi planejado um sistema apenas com Servidor e Interface de Usuário, porém, visando a autonomia de edifícios e privacidade dos dados, o Servidor foi dividido em dois, Servidor Local (que manteve o nome *Opus*) e Servidor Remoto (que recebeu o nome de *Maestro*), onde o Servidor Local é responsável por todo o gerenciamento de dispositivos e permissões de usuários, enquanto o Servidor Remoto é responsável pela autenticação de usuários e sincronização de informações entre os Servidores Locais, assim como ser uma ponte entre a Interface de Usuário e os Servidores Locais através da internet, permitindo o controle remoto dos dispositivos em qualquer lugar do mundo.

A arquitetura do sistema é exemplificada na figura 7.

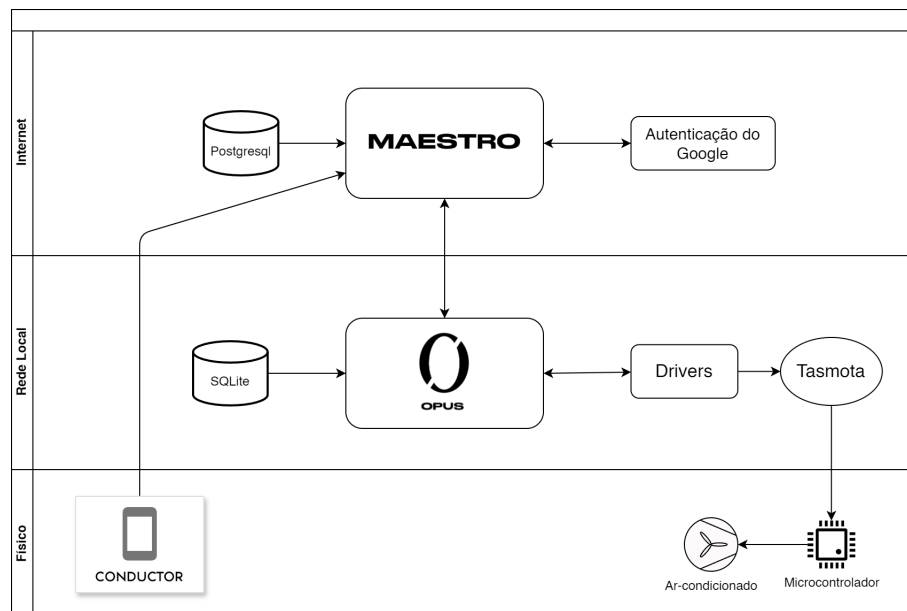



Figura 7 – Arquitetura do Sistema

Fonte: Elaborado pelo Autor (2025)

A comunicação entre as partes do sistema é realizada através do protocolo MQTT. Este protocolo foi escolhido por ser leve e fácil de implementar utilizando bibliotecas dis-

poníveis para Python, a linguagem de programação escolhida para o desenvolvimento do sistema. Dada a natureza “Publicar e Assinar” do MQTT, ele não é ideal para aplicações do tipo “Requisição e Resposta”, para contornar este detalhe, foi implementado um padrão na comunicação entre *Opus* e *Maestro*.

Todas as mensagens devem ser do tipo JSON, contendo sempre um campo “callback” que indicará ao destinatário da mensagem o tópico MQTT onde a resposta deve ser enviada ao remetente. O remetente deve se inscrever no tópico “callback” para receber a resposta, como ilustrado na figura 8, onde “data” é o dado que será mandado do *Maestro* para o *Opus*, note que, além do campo do conteúdo da mensagem (“device_id”), também é enviado um campo “callback” que indica o tópico MQTT onde a resposta deve ser enviada, esse campo possui um identificador aleatório, garantindo que várias respostas possam ser recebidas ao mesmo tempo sem conflitos.



```
async def get_device_state(local_server: OpusServer,
device_id: str) → json:
    """Get the state of a device."""
    request_id = str(uuid.uuid4())
    data: dict = {
        "device_id": device_id,
        "callback": env('CLOUD-MQTT') +
"/devices/get_device_state/callback/" + request_id
    }

    send_task = asyncio.create_task(send_msg_to_server(
local_server, f"devices/get_state/{device_id}", data))
    receive_task = asyncio.create_task(receive_callback(env(
'CLOUD-MQTT') + f"/devices/get_device_state/callback/{
request_id}"))
```

Figura 8 – Exemplo de Comunicação entre *Opus* e *Maestro*

Fonte: Elaborado pelo Autor (2025)

3.1 Servidor Local

O controle de dispositivos, gerenciamento de permissões de usuário e abstração da estrutura de um edifício, são responsabilidades assumidas pelo Servidor Local, chamado de *Opus*.

O *Opus* é uma aplicação escrita em Python visando convergir a operação de dispositivos IoT de vários fabricantes e protocolos em um único ponto de controle, tal funcionalidade é obtida a partir da natureza modular do Python que foi aqui aproveitada para criar partes expansíveis do sistema, chamadas de *drivers* e *interfaces*.

A aplicação é organizada em quatro módulos principais, como ilustrado na figura ??, são eles:

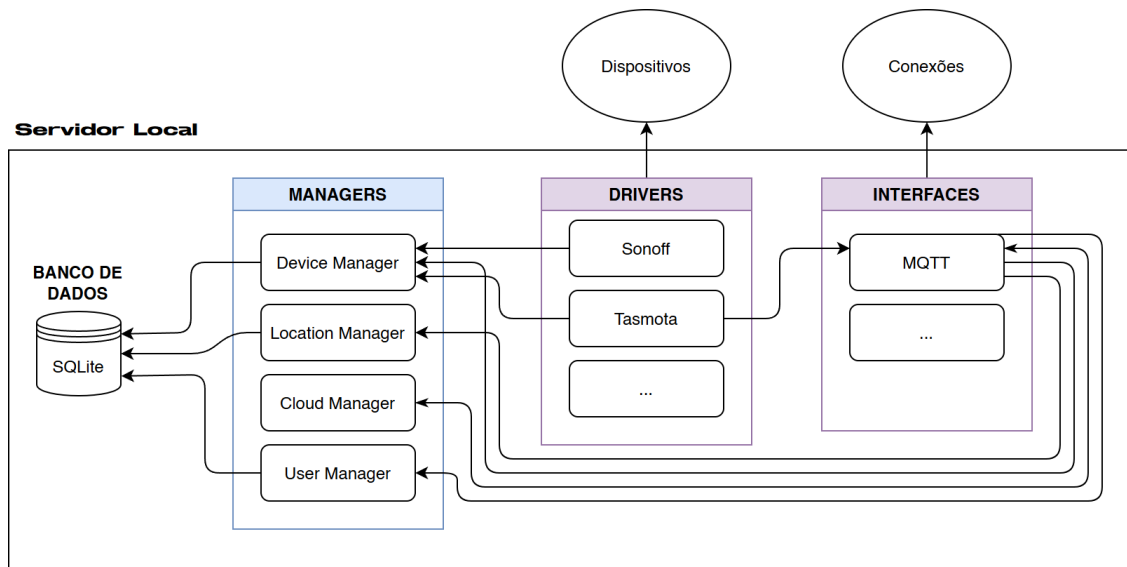


Figura 9 – Módulos do Opus

Fonte: Elaborado pelo Autor (2025)

Interfaces, módulos que adicionam uma forma de comunicação ao *Opus*, como, por exemplo, uma *interface* para comunicação com dispositivos ZigBee[2.3.1.3], ou KNX[2.3.1.2]; **Drivers**, responsáveis por traduzir os comandos do *Opus* para o protocolo do dispositivo, podendo ou não utilizar alguma *interface* própria para esta finalidade; **Managers**, são módulos da aplicação que gerenciam seu funcionamento, recebem, enviam e processam comandos vindos do Servidor Remoto e por fim, o banco de dados.

3.1.1 Arquivo de Configuração

Afim de personalizar a execução do *Opus*, um arquivo de configuração é utilizado para definir parâmetros de execução, como interfaces, drivers e configurações específicas

de cada um. O arquivo de configuração é escrito em YAML, um formato de fácil leitura e escrita, que permite a definição de estruturas de dados complexas de forma simples e legível.

O arquivo de configuração é dividido em seções, cada uma com um propósito específico, como interfaces, drivers e configurações gerais. Na figura 10 é possível ver um exemplo de arquivo de configuração do *Opus*.

```
name: opus-server-exemplo

# Interfaces
interfaces:
  # A interface MQTT<Maestro> é obrigatória
  # para o funcionamento do sistema
  - mqtt<maestro>:
      log_id: "mqtt<maestro>"
      host: "maestro.alicino.com.br"
      port: 1883
      username: ""
      password: ""
      client_id: "opus-server-exemplo"

  - Interface1:
      dado1: "valor1"
      dado2: "valor2"

  - Interface2:
      dado1: "valor1"
      dado2: "valor2"

# Drivers
drivers:
  - Driver1
  - Driver2
```

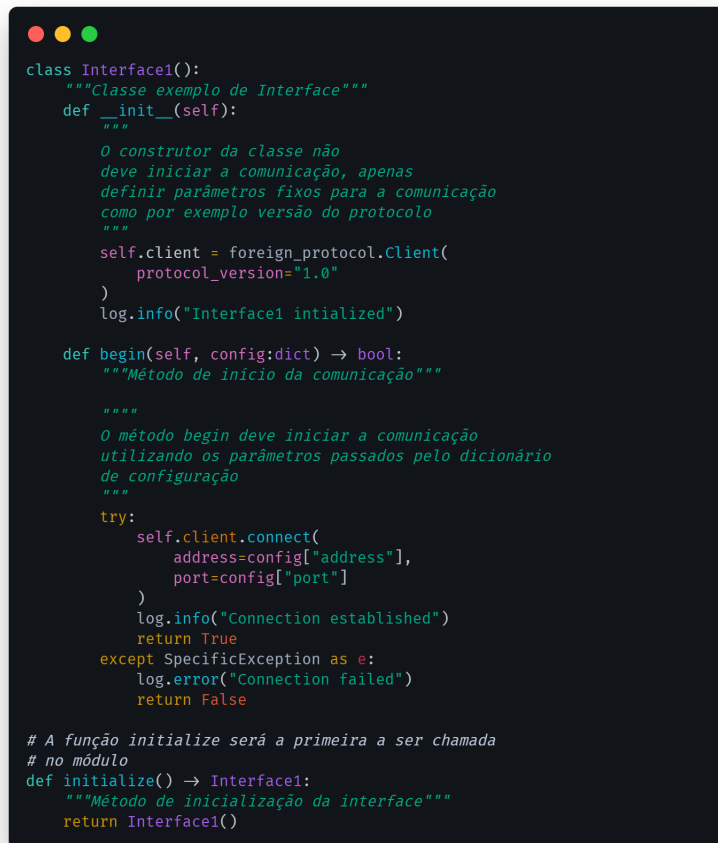
Figura 10 – Exemplo de Arquivo de Configuração

Fonte: Elaborado pelo Autor (2025)

3.1.2 Interfaces

A fim de ser uma aplicação modular, que pode se comunicar com diversos dispositivos, foi necessário levar em conta a vasta gama de protocolos existentes hoje no mercado, sejam de dispositivos IoT com fins de automação residencial, ou equipamentos industriais de automação de edifícios. Para preencher essa lacuna, foi criado o conceito de *Interfaces*. Este conceito permite que novos meios de comunicação sejam adicionados sem a necessidade de alterar o código-fonte dos módulos já existentes, criando um sistema expansível.

Uma *Interface* é um módulo Python que implementa um conjunto de métodos



```

class Interface1():
    """Classe exemplo de Interface"""
    def __init__(self):
        """
        O construtor da classe não
        deve iniciar a comunicação, apenas
        definir parâmetros fixos para a comunicação
        como por exemplo versão do protocolo
        """
        self.client = foreign_protocol.Client(
            protocol_version="1.0"
        )
        log.info("Interface1 intialized")

    def begin(self, config:dict) → bool:
        """Método de início da comunicação"""
        """
        O método begin deve iniciar a comunicação
        utilizando os parâmetros passados pelo dicionário
        de configuração
        """
        try:
            self.client.connect(
                address=config["address"],
                port=config["port"]
            )
            log.info("Connection established")
            return True
        except SpecificException as e:
            log.error("Connection failed")
            return False

# A função initialize será a primeira a ser chamada
# no módulo
def initialize() → Interface1:
    """Método de inicialização da interface"""
    return Interface1()

```

Figura 11 – Exemplo de Interface

Fonte: Elaborado pelo Autor (2025)

necessários para a comunicação com um protocolo específico. Cada interface encapsula a lógica de comunicação para um método de comunicação e expõe aos módulos somente os métodos necessários para realizar a comunicação. Todas as *Interfaces* recebem um dicionário de configurações, onde o usuário pode especificar parâmetros para a comunicação, desta forma cada método de comunicação pode ser configurado conforme as necessidades do usuário e de forma dinâmica.

Para um módulo Python ser considerado uma *Interface* pelo *Opus*, ele deve implementar uma função “ initialize ()” que retorna o objeto principal da classe implementada no módulo, este objeto será adicionado ao ambiente de execução do *Opus* e estará então acessível para todos os módulos da aplicação. Também é necessário a implementação de um método “begin(self, config: dict) → bool” membro da classe principal, este método receberá o dicionário de configurações da *Interface* e então começar a comunicação com o protocolo especificado. A figura 11 apresenta um exemplo prático da implementação de uma *Interface*, destacando os métodos e função essenciais para o funcionamento.

As *Interfaces* são carregadas na aplicação em tempo de execução, de forma dinâmica no momento da inicialização dos módulos. Para que uma *Interface* seja carregada, é ne-

cessário que seu nome esteja listado no arquivo de configuração do *Opus* seguido de um identificador único (exemplo: “interface1 <identificador>”), como também é necessário que o arquivo contendo a implementação da *Interface* esteja no diretório de *Interfaces* com o mesmo nome mencionado no arquivo de configuração (exemplo: “./ interfaces / interface1 .py”), em caso da mesma *Interface* ser mencionada mais de uma vez no arquivo de configuração com o identificador diferente, será carregada uma instância diferente da *Interface* para cada identificador. Por fim, se ocorrer uma falha durante a inicialização de uma *Interface* o *Opus* não prosseguirá e encerrará a execução com um erro marcado como crítico no arquivo de log, sendo necessário corrigir o erro e reiniciar a aplicação.

3.1.3 Drivers

Sem uma padronização da comunicação entre dispositivos IoT, cada fabricante pode adotar protocolos distintos e estruturas variadas de mensagens. Diante deste cenário, foi adotado o conceito de *Drivers*, criado para abstrair e padronizar a comunicação entre dispositivos e o *Opus*.

Drivers são pacotes escritos em Python e carregados no momento da inicialização do *Opus*. Sua função é atuar como um tradutor para um tipo específico de dispositivo, garantindo que seus comandos e dados sejam corretamente interpretados e processados. Além disso, caso o método utilizado pelo *Driver* suporte descoberta automática, ele pode relatar novos dispositivos encontrados. Por conta disso, é necessário que *Drivers* e o módulo de gerenciamento de dispositivos do *Opus* trabalhem em conjunto, garantindo a integração de uma ampla gama de dispositivos sem a necessidade de modificar os módulos do sistema. Como resultado, o sistema se torna modular e escalável.

Quando um dispositivo utiliza um protocolo não suportado nativamente pelo Python, como aqueles que requerem uma camada física diferente do TCP/IP (exemplo: comunicação Serial), ou que utilizam TCP/IP sem suporte HTTP (exemplo: MQTT), o *Driver* pode acessar as *Interfaces* para realizar a comunicação.

```

from core.devices.__generic import OpusDevice
from core.device_manager import DeviceManager
from core.devices.light import OpusLight

from typing import override
from uuid import UUID, uuid1

# O Opus irá preencher o valor da chave com a Interface
# solicitada neste dicionário
interfaces: dict[any, str] = {
    "Interface1": None,
    "Interface2": None
}

class Driver1Device(OpusDevice):
    """Dispositivo genérico exemplo para o Driver1"""
    def __init__(self):
        super().__init__()
        self.specific_address: str = ""
        self.id = uuid1()

class Driver1Light(OpusLight):
    """Dispositivo do tipo Luz para o Driver1"""
    def __init__(self, name: str, driver1_device: Driver1Device):
        super().__init__(
            name=name,
            uuid=UUID(str(driver1_device.id)),
            driver="Driver1"
        )
        self.specific_address: str = driver1_device.specific_address

    @override
    def on(self):
        """
        Reimplementação do método Ligar, de acordo
        com os requisitos do Driver
        """
        interfaces['Interface1'].enviar_mensagem(
            {"address": self.specific_address, "ação": "ligar_luz"}
        )

class Driver1:
    """Driver de exemplo para o Opus"""
    def __init__(self, device_manager: DeviceManager):
        self.opus_device_manager: DeviceManager = device_manager

    def translate_message_to_devices(self,
                                     message: str) → list[Driver1Device]:
        """
        Método exemplo para receber a mensagem da Interface
        e retornar quais novos dispositivos foram descobertos
        """
        # Implementar a lógica de acordo com o protocolo
        # de comunicação entre o Driver e a Interface
        pass

    def find_devices(self):
        """
        Drivers podem encontrar e reportar novos
        dispositivos para o Opus
        """
        global interfaces
        # Exemplo de uso de uma interface em um Driver
        interfaces['Interface2'].enviar_mensagem(PROCURAR_DISPOSITIVOS)
        msg = interfaces['Interface2'].receber_mensagem()
        new_devices: list[Driver1Device] = self.translate_message_to_devices(msg)
        # Adiciona dispositivos descobertos a lista de dispositivos disponíveis
        for device in new_devices:
            self.opus_device_manager.new_device(device)

    def new_light(name: str, base_device: Driver1Device) → Driver1Light:
        """
        Ao registrar um dispositivo, o Opus irá chamar a função
        de acordo com o Driver que o dispositivo pertence
        """
        return Driver1Light(name, base_device)

    def start(dirs: dict,
              config: dict,
              drivers: dict,
              interfaces: dict,
              managers: dict) → None:
        """Função chamada pelo Opus para iniciar o Driver"""
        finder = Driver1(managers['devices'])

```

Figura 12 – Exemplo de Driver

Fonte: Elaborado pelo Autor (2025)

Para ser considerado um *Driver*, o pacote deve seguir um padrão de nomenclatura e implementar algumas funções esperadas. O pacote e o módulo principal devem ter o mesmo nome, pois o Opus busca pelo nome do *Driver* dentro do diretório de *Drivers*. Caso um *Driver* chamado “exemplo” seja encontrado no diretório de *Drivers*, o Opus carregará o módulo principal do pacote, localizado em “drivers/exemplo/exemplo.py”. O módulo principal deve conter uma função “start()” sem retornos, que será chamada no momento em que o *Driver* for carregado. Além disso, deve possuir um dicionário global de *Interfaces* com a seguinte assinatura: “interfaces: dict[str, Any]”. Caso o *Driver* necessite de uma *Interface*, seu nome deve estar presente nesse dicionário, e o Opus injetará o objeto correspondente, permitindo que o *Driver* o acesse.

Como *Drivers* comandam toda a comunicação com os dispositivos, toda a estrutura de um dispositivo, assim como a lógica de controle, ficam a cargo do *Driver*. Eles são autônomos e podem funcionar como aplicações independentes, permitindo a implementação de lógicas específicas, independentemente de sua complexidade. Todavia, é importante que o *Driver* siga o padrão esperado pelo Opus em classes voltadas para a comunicação com dispositivos.

3.1.3.1 A Classe de Dispositivos

A forma mais simples de abstrair um dispositivo é via uma classe. *Drivers* têm liberdade total para implementar a classe dos dispositivos, sendo necessário apenas que essa classe herde da classe genérica de dispositivos do Opus, que contém os métodos esperados para o controle dos dispositivos através da aplicação.

A classe genérica “OpusDevice” é utilizada para representar dispositivos recém-descobertos, cujo tipo ainda não foi determinado. Após a identificação, o dispositivo deve ser instanciado como um tipo mais específico, herdado de “OpusDevice”.

3.1.3.2 O Registrando de Dispositivos

Durante a etapa de registro de dispositivos descobertos, o Opus irá buscar pela função que retorna uma instância daquele tipo de dispositivo para o *Driver* em questão. O nome desta função deve ser “new_” seguido do tipo do dispositivo.

Por exemplo, para um dispositivo do tipo “light” no *Driver* de exemplo, chamado Driver1, a assinatura da função seria:

```
def new_light() -> Driver1Light:
    return Driver1Light()
```

Um exemplo de um *Driver* que implementa um dispositivo teórico do tipo “light”, com os requisitos mínimos é mostrado na figura 12.

3.1.4 Gerenciadores

No “núcleo” do *Opus*, estão os *Managers*, ou Gerenciadores, eles são responsáveis por coordenar o fluxo da aplicação *Opus*, se comunicando utilizando a *Interface* MQTT. Divididos em quatro módulos principais:

- CloudManager, que gerencia a comunicação com o *Maestro*;
- UserManager, que gerencia os usuários do *Opus* em questão.
- LocationManager, que gerencia a estrutura do edifício no sistema;
- DeviceManager, que gerencia os dispositivos;

3.1.4.1 CloudManager

CloudManager é o módulo responsável pela comunicação entre o Servidor Local (*Opus*) e Servidor Remoto (*Maestro*), garantindo a sincronização de informações e o controle remoto dos dispositivos cadastrados no sistema. Essa comunicação é realizada principalmente via MQTT, permitindo que o *Opus* envie e receba mensagens do *Maestro* de maneira assíncrona e eficiente.

Funcionalidades Principais

- Login no *Maestro*: ao ser inicializado, o *CloudManager* realiza um procedimento de login no *Maestro*, enviando informações como o identificador do Servidor Local, dado pelo nome do Servidor Local definido pelo usuário no arquivo de configuração; seu endereço IP público e o tópico MQTT de *callback*. Caso a autenticação falhe ou o *Maestro* não responda em um tempo limite, a aplicação se encerra para evitar estados inconsistentes.
- Acesso das informações internas ao *Opus*: o controle do que usuários podem ou não controlar é todo gerenciado por cada Servidor Local, em casos do *Maestro* requerer informações sobre algum usuário, como quais dispositivos ele está autorizado a controlar, o *CloudManager* irá fazer a integração com outros módulos do “núcleo” do *Opus* para obter essas informações e enviar ao *Maestro* via MQTT.

3.1.4.2 UserManager

UserManager é o módulo responsável pelo gerenciamento de usuários dentro do *Opus*, garantindo que apenas os usuários registrados e com as devidas permissões possam acessar dispositivos cadastrados no sistema.

Funcionalidades Principais

- Sincronização de Usuários do Maestro: o *UserManager* recebe informações de usuários cadastrados no Maestro e os atribui localmente no *Opus*. Esse processo inclui a atribuição daquele usuário a nível de acesso chamado de *role* e a persistência dos dados no banco de dados local.
- Gerenciamento de Permissões: o módulo é responsável por checar níveis de permissão de usuários, entregando esse dados a outros módulos como o *DeviceManager* que utiliza essas informações para permitir ou não o acesso a um dispositivo.
- Comunicação via MQTT: o *UserManager* também estabelece um canal de comunicação via MQTT com o Maestro, para troca de informações sobre usuários, como por exemplo, a atribuição de um novo usuário ao *Opus*.

3.1.4.3 LocationManager

LocationManager é o módulo responsável pela organização espacial dos dispositivos dentro do *Opus*, permitindo estruturar o ambiente em uma hierarquia composta por prédios, espaços e salas. Essa estrutura foi pensada para facilitar a implementação em vários tipos de edifícios, mantendo um balanço entre modularidade e simplicidade.

Funcionalidades Principais

- Gerenciamento da Estrutura do Edifício: o *LocationManager* organiza os dispositivos do *Opus* em uma estrutura hierárquica composta por:
 - Prédios (Buildings): representam grandes unidades físicas, como edifícios propriamente ditos, ou setores, como blocos de um campus;
 - Espaços (Spaces): são subdivisões dentro dos edifícios, como andares em um prédio, sendo um agrupamento de salas;
 - Salas (Rooms): representam ambientes específicos em um espaço, onde dispositivos podem ser instalados.
- Criação e Registro de Estruturas: o módulo realiza toda a manutenção das estruturas no banco de dados local e também mantém as estruturas em memória durante a execução.
- Carregamento de Estrutura Existentes: durante a inicialização, o *LocationManager* consulta o banco de dados do *Opus* e carrega todas as estruturas previamente registradas em memória, onde ocorrem todas as operações de gerenciamento.

- Configuração e Comunicação via MQTT: o módulo estabelece comunicação via MQTT, permitindo que o Maestro e outros clientes solicitem informações ou realizem operações. O *LocationManager* processa mensagens em tópicos MQTT específicos para:
 - Criar novos prédios, espaços e salas conforme solicitado.
 - Fornecer a lista atual de locais cadastrados no sistema.

3.1.4.4 DeviceManager

DeviceManager é o módulo responsável pelo gerenciamento de dispositivos dentro do *Opus*. Ele controla a descoberta, registro, armazenamento e operação de dispositivos conectados, garantindo a integração entre diferentes drivers e a comunicação entre o *Opus* e o *Maestro*.

Funcionalidades Principais

- Gerenciamento de Dispositivos: o *DeviceManager* mantém um catálogo de dispositivos disponíveis e registrados em memória, esse catálogo é dividido nos tipos de dispositivos suportados pelo *Opus*;
- Registro de Dispositivos: o módulo permite registrar dispositivos no sistema, vinculando cada um a um driver específico e a uma sala dentro da abstração dos edifícios cadastrados no *Opus*. Esse processo inclui:
 - Verificação de Disponibilidade: o dispositivo precisa estar previamente detectado na lista de dispositivos disponíveis, encontrados pelos *Drivers*;
 - Verificação do Local: o dispositivo apenas pode ser registrado em salas previamente cadastradas no *LocationManager*;
 - Criação no Banco de Dados: após a validação, o dispositivo é adicionado a lista de dispositivos registrados e também é adicionado ao banco de dados para persistência dos dados em caso de reinicialização do sistema.
- Classificação por Tipo e Driver: o *DeviceManager* categoriza os dispositivos de acordo com seu tipo e *Driver*, para os fins deste trabalho, estão suportados os seguintes tipos de dispositivos:
 - LIGHT: Dispositivos de iluminação controláveis;
 - HVAC: Sistemas de Ar-Condicionado.

Outros dispositivos podem ser adicionados através da implementação de novos drivers.

- Comunicação via MQTT: o módulo assim como os outros gerenciadores, também está integrado ao sistema de mensagens MQTT, permitindo o controle remoto dos dispositivos via mensagens enviadas pelo *Maestro*.

3.1.5 Banco de Dados

A escolha do banco de dados para o Servidor Local *Opus* foi tomada a partir dos seguintes critérios:

- Banco autocontido: Não requer a instalação de um servidor de banco de dados separado, tornando a aplicação mais simples e portátil;
- Desempenho eficiente: Para um ambiente local, SQLite oferece operações rápidas de leitura e escrita, atendendo bem às necessidades do sistema;
- Segurança dos dados: Com um banco de dados local, as informações de dispositivos de um edifícios permanecem restritas para acesso externo.

Com base nesses critérios, foi escolhido o SQLite como banco de dados para o *Opus*. O SQLite é um banco de dados relacional que não requer um servidor separado para funcionar, também tem uma boa integração com Python através do ORM SQLAlchemy, o que facilita a modelagem e manipulação dos dados.

3.1.5.1 Estrutura do Banco de Dados

O banco de dados do *Opus* segue um modelo relacional estruturado para representar a hierarquia de um edifício inteligente, usuários, dispositivos registrados e permissões de acesso. A estrutura inclui as seguintes tabelas principais:

- Building: Representa os edifícios cadastrados no sistema;
- BuildingSpace: Define espaços dentro dos edifícios, como andares em um prédio;
- BuildingRoom: Representa salas dentro dos espaços;
- Device: Contém os dispositivos IoT cadastrados, armazenando informações como tipo de dispositivo, nome e dados específicos de configuração que variam de acordo com o *Driver* utilizado;
- Role: Define os diferentes papéis dos usuários no sistema, com níveis de segurança específicos;
- User: Armazena os usuários registrados;

- **role_device** (tabela associativa): Relaciona os dispositivos que cada papel pode acessar.

A modelagem utiliza UUIDs como chaves primárias para garantir identificadores únicos globalmente.

O ERD (Diagrama de Relacionamento de Entidades) do banco de dados do *Opus* é mostrado na figura 13.

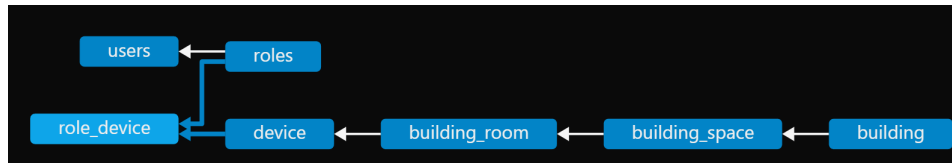


Figura 13 – Diagrama de Relacionamento de Entidades do Banco de Dados do Opus

Fonte: Elaborado pelo Autor (2025)

O diagrama da figura 13 ilustra muito bem como a hierarquia de um edifício é representada no banco de dados, dispositivos são contidos em salas que são contidas em espaços que são contidos em prédios. Um ponto importante a ser observado é a relação entre a tabela **role** e a tabela **device**, que é feita através da tabela associativa **role_device**. **role** define os níveis de acesso dos usuários, cada usuário tem uma **role** que é representada por um valor numérico onde quanto menor o valor, maior o nível de acesso, sendo 0 o nível de administrador. A tabela associativa **role_device** relaciona os dispositivos que cada **role** pode acessar, isso garante que o acesso a cada dispositivo seja controlado de acordo com o nível de permissão do usuário.

O banco de dados do *Opus* tem as seguintes informações em cada tabela:

- **Building**

- **building_pk**: UUID - Identificador único do edifício;
- **building_name**: str - Nome do edifício.

- **BuildingSpace**

- **building_space_pk**: UUID - Identificador único do espaço;
- **building_fk**: UUID - Chave estrangeira para o edifício;
- **space_name**: str - Nome do espaço.

- **BuildingRoom**

- **building_room_pk**: UUID - Identificador único da sala;
- **building_space_fk**: UUID - Chave estrangeira para o espaço;

- `room_name`: `str` - Nome da sala.

- **Device**

- `device_pk`: `UUID` - Identificador único do dispositivo;
- `room_fk`: `UUID` - Chave estrangeira para a sala;
- `device_name`: `str` - Nome do dispositivo;
- `device_type`: `str` - Tipo do dispositivo;
- `driver_name`: `str` - Nome do *Driver* utilizado;
- `driver_data`: `JSON` - Dados específicos do *Driver*.

- **Role**

- `role_pk`: `UUID` - Identificador único do papel;
- `role_name`: `str` - Nome do papel;
- `security_level`: `int` - Nível de segurança do papel.

- **User**

- `user_pk`: `UUID` - Identificador único do usuário;
- `given_name`: `str` - Nome do usuário;
- `email`: `str` - E-mail do usuário;
- `fk_role`: `UUID` - Chave estrangeira para o papel.

3.1.6 Registro e Aquisição de Dispositivos

```
"""Generic Device Class"""
from uuid import UUID

class DeviceType(enum.Enum):
    """Enumerate for Device Types"""
    UNKNOWN = "UNKNOWN"
    LIGHT = "LIGHT"
    HVAC = "HVAC"
    PROJECTOR = "PROJECTOR"
    GATE_OPENER = "GATE_OPENER"
    DOOR_LOCK = "DOOR_LOCK"

class OpusDevice():
    """Generic Device Class"""
    def __init__(self):
        self.name: str
        self.id: UUID | None
        self.room_id: UUID | None
        self.space_id: UUID | None
        self.building_id: UUID | None
        self.driver: any
        self.type: DeviceType

    def get(self):
        """Get the device"""
        return {
            "device_name": self.name,
            "device_pk": str(self.id),
            "device_type": self.type,
            "building_room_pk": str(self.room_id),
            "building_space_pk": str(self.space_id),
            "building_pk": str(self.building_id)
        }
```

Figura 14 – Classe Genérica de Dispositivo do Opus

Fonte: Elaborado pelo Autor (2025)

Para garantir modularidade e flexibilidade na integração de novos dispositivos, o *Opus* adota um modelo baseado em classes genéricas, que servem como base para representar diferentes tipos de dispositivos e padronizar a interação com os *Drivers*. Ao criar suas próprias classes genéricas de dispositivo. Na figura 14 é possível visualizar a classe genérica de dispositivo do *Opus*, já na figura 15 é possível ver uma implementação de uma classe genérica de dispositivo implementada pelo *Driver* Tasmota.

Observe que ambas as classes não possuem métodos de controle específicos, apenas métodos auxiliares, já que não é esperado que a instância dessa classe seja utilizada para controlar os dispositivos, apenas representá-los.


```

"""Tasmota Device Class"""
from ipaddress import ip_address
from core.devices.__generic import OpusDevice

class TasmotaDevice(OpusDevice):
    """Generic Tasmota Device"""
    def __init__(self):
        super().__init__()
        self.driver = "tasmota"
        self.tasmota_name: str
        self.mqtt_topic: str

    def __str__(self) → str:
        return f"<TasmotaDevice-{self.tasmota_name}>"

```

Figura 15 – Classe Genérica de Dispositivo do Driver Tasmota

Fonte: Elaborado pelo Autor (2025)

Uma vez que dispositivo é registrado, o *Opus* busca pelo método “new_?()” [3.1.3.2] no *Driver* correspondente, onde então, o *Driver* deve retornar uma instância do dispositivo com seu tipo específico.

A figura 16 mostra um exemplo de uma classe genérica de dispositivo do tipo luz, que herda da classe genérica de dispositivo do *Opus*. Já a figura 17 mostra uma implementação de uma classe de dispositivo do tipo luz, específica para o *Driver* Sonoff, que implementa os métodos abstratos para a comunicação com o dispositivo.

```

"""Generic Class for Light"""
from abc import abstractmethod
from __generic import OpusDevice, DeviceType

class OpusHSL(enum.Enum):
    H: float # Hue
    S: float # Saturation
    L: float # Lightness

class OpusLight(OpusDevice):
    """Generic Implementation of a Light"""
    def __init__(self, name: str,
                 uuid: str,
                 driver: any):
        super().__init__()
        self.name = name
        self.id = uuid
        self.room_id = None
        self.space_id = None
        self.building_id = None
        self.driver = driver
        self.type = DeviceType.LIGHT
        self.power_state: bool # Is the HVAC On or Off
        self.hsl_color: OpusHSL # Hue, Saturation, Lightness

    @abstractmethod
    def on(self) → None:
        """Turn the light on"""

    @abstractmethod
    def off(self) → None:
        """Turn the light off"""

    @abstractmethod
    def get_state(self) → bool:
        """Get the state of the light"""
        return {
            "power_state": self.power_state
        }

```

Figura 16 – Classe Genérica de Dispositivo do Tipo Luz

Fonte: Elaborado pelo Autor (2025)

```

class Sonofflight(OpusLight):
    """Sonoff API Light Implementation"""
    def __init__(self,
                 name: str,
                 sonoff_device: SonoffDevice):
        super().__init__(
            name=name,
            uuid=UUID(str(sonoff_device.id)),
            driver="sonoff"
        )
        self.ip_address: ip_address = ip_address(sonoff_device.
ip_address)
        self.hostname: str = str(sonoff_device.hostname)
        self.port: int = int(sonoff_device.port)
        self.device_id: str
        self.startup_info_dump: dict = {}

        # Get the "device_id" which was not possible before
        device_payload = requests.post(
            url=f'http://{self.ip_address}:{self.port}
/zeroconf/info',
            data= json.dumps({
                "deviceid": "",
                "data": {}
            }),
            headers={"Content-Type": "application/json"},
            timeout=5
        )
        device_payload = device_payload.json()

        self.power_state = device_payload['data']['switch']
        self.device_id = device_payload['data']['deviceid']
        self.bssid = device_payload['data']['bssid']

    @override
    def on(self) → None:
        requests.post(
            url=f'http://{self.ip_address}:{self.port}
/zeroconf/switch',
            data= json.dumps({
                "deviceid": self.device_id,
                "data": { "switch": "on"}
            }),
            headers={"Content-Type": "application/json"},
            timeout=5
        )

    @override
    def off(self) → None:
        requests.post(
            url=f'http://{self.ip_address}:{self.port}
/zeroconf/switch',
            data= json.dumps({
                "deviceid": self.device_id,
                "data": { "switch": "off"}
            }),
            headers={"Content-Type": "application/json"},
            timeout=5
        )

```

Figura 17 – Classe de Dispositivo do Tipo Luz do Driver Sonoff

Fonte: Elaborado pelo Autor (2025)

Registro de Dispositivos A fim de oferecer suporte a dispositivos IoT que podem ser descobertos automaticamente por meio de sistemas de anúncio, o *Opus* organiza os dispositivos em dois estados distintos: *Dispositivos Disponíveis* e *Dispositivos Registrados*.

Dispositivos Disponíveis são dispositivos detectados pelos *Drivers*, mas que ainda não possuem uma definição específica dentro do sistema. Eles herdam da classe base *OpusDevice* e representam unicamente a presença de um novo dispositivo compatível, sem que haja, nesse momento, qualquer categorização funcional (como luz, ar-condicionado ou fechadura), configuração ou armazenamento no banco de dados.

Para que um *Dispositivo Disponível* possa ser utilizado ativamente no sistema, ele deve passar pelo processo de *Registro*. Esse processo consiste em atribuir ao dispositivo um tipo específico, vinculá-lo ao banco de dados do *Opus* e habilitar seu controle dentro da plataforma. Uma vez registrado, o dispositivo torna-se um *Dispositivo Registrado*, adquirindo uma identidade única dentro do *Opus* e tornando-se acessível para controle e gerenciamento pelos usuários, e também gravando suas informações no banco de dados local.

3.2 Servidor Remoto

Em um ambiente de edifícios inteligentes, a gestão dos usuários deve ser executada com cautela, pois certos dispositivos não podem ser acessados por qualquer usuário, assim como os usuários devem ser rastreáveis para evitar que as funcionalidades que o sistema adiciona possam se tornar prejudiciais para o edifício e sua gestão, como mal uso de dispositivos, ou até mesmo ações maliciosas. Com estas limitações em mente foi montada a dinâmica de usuários do sistema Opus, onde os servidores Opus [3.1] são responsáveis por gerenciar permissões de acesso a dispositivos e uma entidade centralizada é responsável por autenticar e gerenciar usuários. Com a necessidade dessa entidade centralizada foi criado o Maestro, um servidor remoto responsável pela autenticação e gestão de usuários, bem como a intermediação entre o aplicativo móvel e os servidores locais.

Sendo assim, as duas principais funções do Maestro são:

- Autenticação e gerenciamento de usuários;
- Intermediação entre o aplicativo móvel e os servidores locais.

Essas funções são realizadas através de 2 módulos principais que controlam duas API (Application Programming Interface) distintas, a API em REST (Representational State Transfer), voltada a comunicação com o usuário e a API em MQTT voltada a comunicação com os servidores locais.

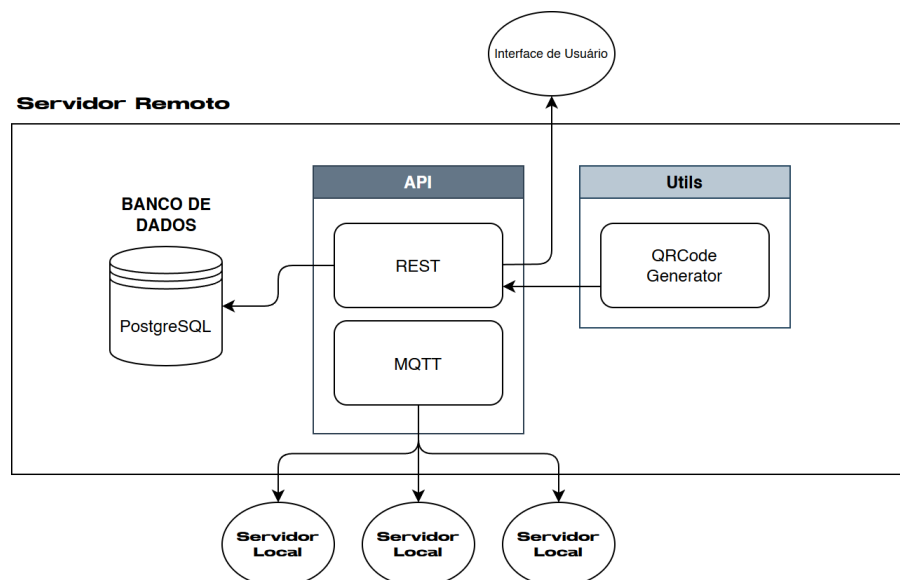


Figura 18 – Módulos do Maestro

Fonte: Elaborado pelo Autor (2025)

3.2.1 API REST

Além de gerenciar a autenticação e cadastro de usuários, a API REST também é responsável por agir como intermediador entre os Servidores Locais *Opus* e a Interface de Usuário, ela faz isso utilizando a API MQTT, que envia comandos utilizando esse protocolo que é o padrão de comunicação entre servidores no sistema *Opus*.

Referências

APPLIED RISK. *I Own Your Building (Management System)*. 2019. Citado 2 vezes nas páginas 5 e 6.

ASHRAE. *BACnet™ - A Data Communication Protocol for Building Automation and Control Networks*. Georgia, US, 2016. Disponível em: <https://www.ashrae.org/technical-resources/standards-and-guidelines/standards-addenda/standard-135-2016-bacnet-a-data-communication-protocol-for-building-automation-and-control-net>. Citado na página 11.

BELLINI, P.; NESI, P.; PANTALEO, G. Iot-enabled smart cities: A review of concepts, frameworks and key technologies. *Applied Sciences (Switzerland)*, MDPI, v. 12, 2 2022. ISSN 20763417. Citado na página 5.

BERKOBEN, K.; KAED, C.; SODORFF, T. *A Digital Buildings Ontology for Google's Real Estate*. 2020. Disponível em: <https://github.com/google/digitalbuildings>. Citado na página 5.

BROOKS, D. Intelligent buildings: an investigation into current and emerging security vulnerabilities in automated building systems using an applied defeat methodology. 03 2012. Citado na página 4.

BUCKMAN, A. H.; MAYFIELD, M.; BECK, S. B. What is a smart building? *Smart and Sustainable Built Environment*, Emerald Publishing, v. 3, p. 92–109, 9 2014. ISSN 20466102. Citado 3 vezes nas páginas 3, 4 e 5.

BUSHBY, S. T.; NEWMAN, H. M. *BACnet Today Significant New Features And Future Enhancements*. 2002. Citado na página 11.

CLEMENTS-CROOME, D. Sustainable intelligent buildings for people: A review. *Intelligent Buildings International*, v. 3, p. 67–86, 2011. ISSN 17508975. Citado na página 4.

DOMINGUES, P. et al. *Building automation systems: Concepts and technology review*. [S.l.]: Elsevier B.V., 2016. 1-12 p. Citado 3 vezes nas páginas 7, 11 e 12.

European Committee for Standardization. *Energy performance of buildings — Impact of Building Automation Control and Building Management*. [S.l.], 2006. Disponível em: http://www.cres.gr/greenbuilding/PDF/prend/set4/WI_22_TC-approval_version_prEN_15232_Integrated_Building_Automation_Systems.pdf. Citado na página 6.

INTELLIGENT. In: *Oxford Learner's Dictionaries*. 2023. Disponível em: https://www.oxfordlearnersdictionaries.com/definition/american_english/intelligent. Acesso em: 18 de fevereiro de 2023. Citado na página 3.

International Organization for Standardization. *Information technology — Message Queuing Telemetry Transport (MQTT) v3.1.1*. Geneva, CH, 2016. Disponível em: <https://www.iso.org/standard/69466.html>. Citado na página 12.

International Organization for Standardization. *Building automation and control systems (BACS) — Part 6: Data communication conformance testing*. Geneva, CH, 2020. Disponível em: <https://www.iso.org/standard/79630.html>. Citado na página 6.

KAMAL, S. et al. *Smart and Intelligent Buildings Achieving Architectural Concepts*. 2021. 155-163 p. Citado na página 5.

KRAUS, N.; VIERTEL, M.; BURGERT, O. Control of knx devices over ieee 11073 service-oriented device connectivity. In: . [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2020. p. 421–424. ISBN 9781728163895. Citado na página 11.

MACAYA, J. F. M. et al. Ano IX - Nº 2 - Smart Cities. *Cetic.br - Centro Regional para o Desenvolvimento da Sociedade da Informação*, nov. 2017. Disponível em: <https://cetic.br/publicacao/ano-ix-n-2-smart-cities>. Citado na página 1.

MADAKAM, S.; RAMASWAMY, R.; TRIPATHI, S. Internet of things (iot): A literature review. *Journal of Computer and Communications*, Scientific Research Publishing, Inc., v. 03, p. 164–173, 2015. ISSN 2327-5219. Citado na página 8.

MEYER, F. D. *How smart buildings will revolutionize our society* | Frederik De Meyer | *TEDxGlarus*. 2022. Disponível em: <https://www.youtube.com/watch?v=XDP2Gz7fP38>. Acesso em: 16 de setembro de 2023. Citado na página 1.

MICROSOFT. 2019 manufacturing trends report. 2019. Disponível em: <https://info.microsoft.com/rs/157-GQE-382/images/EN-US-CNTNT-Report-2019-Manufacturing-Trends.pdf>. Citado 2 vezes nas páginas 1 e 9.

MINOLI, D.; SOHRABY, K.; OCCHIOGROSSO, B. Iot considerations, requirements, and architectures for smart buildings-energy optimization and next-generation building management systems. *IEEE Internet of Things Journal*, Institute of Electrical and Electronics Engineers Inc., v. 4, p. 269–283, 2 2017. ISSN 23274662. Citado 2 vezes nas páginas 9 e 10.

MISHRA, B.; KERTESZ, A. The use of mqtt in m2m and iot systems: A survey. *IEEE Access*, Institute of Electrical and Electronics Engineers Inc., v. 8, p. 201071–201086, 2020. ISSN 21693536. Citado na página 13.

MORVAJ, B.; LUGARIC, L.; KRAJCAR, S. *Demonstrating smart buildings and smart grid features in a smart energy city*. 2011. Citado na página 1.

NEVES, R. P. A. de A.; CAMARGO, A. R. *Espaços Arquitetônicos de Alta Tecnologia: Os Edifícios Inteligentes*. 2002. Citado na página 3.

NORD, J. H.; KOOHANG, A.; PALISZKIEWICZ, J. *The Internet of Things: Review and theoretical framework*. [S.l.]: Elsevier Ltd, 2019. 97-108 p. Citado na página 8.

OASIS. *MQTT Version 5.0*. [S.l.], 2019. Disponível em: <https://mqtt.org/mqtt-specification/>. Citado na página 12.

PAIVA, A. de; JEDON, R. Short- and long-term effects of architecture on the brain: Toward theoretical formalization. *Frontiers of Architectural Research*, Higher Education Press Limited Company, v. 8, p. 564–571, 12 2019. ISSN 20952635. Citado na página 1.

POWELL, J. A. *Intelligent Design Teams Design Intelligent Buildings*. [S.l.]: Habitat International, 1990. 83-94 p. Citado na página 3.

SAPUNDZHI, F. A survey of knx implementation in building automation. *TEM Journal*, UIKTEN - Association for Information Communication Technology Education and Science, v. 9, p. 144–148, 2 2020. ISSN 22178333. Citado na página 11.

SMART. In: *Oxford Learner's Dictionaries*. 2023. Disponível em: https://www.oxfordlearnersdictionaries.com/definition/american_english/smart_1. Acesso em: 18 de fevereiro de 2023. Citado na página 3.

STATISTA. *Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033*. 2024. Disponível em: <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>. Citado 2 vezes nas páginas 1 e 9.

STUBBINGS, M. *Intelligent buildings. Contracts*, 1986. Citado na página 3.

TASMOTA. 2020. Disponível em: <https://tasmota.github.io/docs/About>. Citado na página 12.

TOMAR, A. *Introduction to Zigbee Technology*. [S.l.]: element14, 2011. Citado na página 12.

WAGO. *Controlador PFC200; 2 geração; 2 x ETHERNET, RS-232/-485*. 2023. Disponível em: <https://www.wago.com/br/controlador/controlador-pfc200/p/750-8212>. Citado na página 10.

WIGGINTON, M.; HARRIS, J. *Intelligent skins*. [S.l.]: Butterworth-Heinemann, 2002. 176 p. ISBN 0750648473. Citado na página 3.

WONG, J. K.; LI, H.; WANG, S. W. Intelligent building research: A review. *Automation in Construction*, Elsevier, v. 14, p. 143–159, 2005. ISSN 09265805. Citado na página 3.

YAN, Z.; ZHANG, P.; VASILAKOS, A. V. A survey on trust management for internet of things. *Journal of Network and Computer Applications*, Academic Press, v. 42, p. 120–134, 2014. ISSN 10958592. Citado na página 8.