UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Ciência e Tecnologia de Sorocaba
Campus de São João da boa vista

RICARDO PASQUATI PONTAROLLI

COMPOSIÇÃO DE SERVIÇOS E MECANISMOS DE SEGURANÇA PARA ARQUITETURAS ORIENTADAS A MICROSSERVIÇOS NA INDÚSTRIA 4.0

Sorocaba
2020

RICARDO PASQUATI PONTAROLLI

COMPOSIÇÃO DE SERVIÇOS E MECANISMOS DE SEGURANÇA PARA ARQUITETURAS ORIENTADAS A MICROSSERVIÇOS NA INDÚSTRIA 4.0

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica, junto ao Programa de Pós-Graduação em Engenharia Elétrica, Interunidades, entre o Instituto de Ciência e Tecnologia de Sorocaba e o Campus de São João da Boa Vista da Universidade Estadual Paulista "Júlio de Mesquita Filho".

Área de concentração: Automação
Orientador: Prof. Dr. Eduardo Paciência Godoy

Sorocaba

2020

P811c

Pontarolli, Ricardo Pasquati
Composição de serviços e mecanismos de segurança para arquiteturas orientadas a microsserviços na indústria 4.0 / Ricardo Pasquati Pontarolli. -- Sorocaba, 2020
103 p. : il., tabs., fotos

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Ciência e Tecnologia, Sorocaba
Orientador: Eduardo Paciência Godoy

1. Engenharia elétrica. 2. Indústria. 3. Controle de processo. 4. Arquitetura orientada a serviços. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Ciência e Tecnologia, Sorocaba. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Sorocaba

CERTIFICADO DE APROVAÇÃO

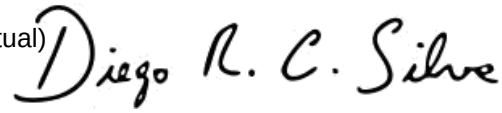
TÍTULO DA DISSERTAÇÃO: COMPOSIÇÃO DE SERVIÇOS E MECANISMOS DE SEGURANÇA PARA ARQUITETURAS ORIENTADAS A MICROSSERVIÇOS NA INDÙSTRIA 4.0

AUTOR: RICARDO PASQUATI PONTAROLLI
ORIENTADOR: EDUARDO PACIÊNCIA GODOY

Aprovado como parte das exigências para obtenção do Título de Mestre em ENGENHARIA ELÉTRICA, área: Automação pela Comissão Examinadora:

Prof. Dr. EDUARDO PACIÊNCIA GODOY (Participação Virtual)
Departamento de Engenharia de Controle e Automação
Instituto de Ciência e Tecnologia - UNESP - Campus de Sorocaba

Prof. Dr. DIEGO RODRIGO CABRAL SILVA (Participação Virtual)
Escola de Ciências e Tecnologia / UFRN



Prof. Dr. PAULO JOSÉ AMARAL SERNI (Participação Virtual)
Departamento de Engenharia de Controle e Automação
Instituto de Ciência e Tecnologia - UNESP - Campus de Sorocaba

Sorocaba, 01 de dezembro de 2020

RICARDO PASQUATI PONTAROLLI

COMPOSIÇÃO DE SERVIÇOS E MECANISMOS DE SEGURANÇA PARA ARQUITETURAS ORIENTADAS A MICROSSERVIÇOS

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica, junto ao Programa de Pós-Graduação em Engenharia Elétrica, Interunidades, entre o Instituto de Ciência e Tecnologia de Sorocaba e o Campus de São João da Boa Vista da Universidade Estadual Paulista "Júlio de Mesquita Filho".

Comissão Examinadora

Prof. Dr. Eduardo Paciência Godoy
UNESP – Instituto de Ciência e Tecnologia de Sorocaba
Orientador

Prof. Dr. Diego Rodrigo Cabral Silva
UFRN – Escola de Ciências e Tecnologia

Prof. Dr. Paulo José Amaral Serni
UNESP – Instituto de Ciência e Tecnologia de Sorocaba

Sorocaba
2020

“Pensar é o trabalho mais difícil que existe. Talvez por isso tão poucos se dediquem a ele.”

Henry Ford
Fundador da Ford Motor

AGRADECIMENTO

Primeiramente gostaria de agradecer a Deus por ter me mantido na trilha certa durante este projeto de pesquisa com saúde e forças para chegar até o final. Agradeço ao meu orientador Prof. Dr. Eduardo Paciência Godoy por aceitar e conduzir o meu trabalho de pesquisa sempre para a direção certa, pelo incentivo e pela dedicação do seu escasso tempo ao meu projeto e pelas valiosas contribuições dadas durante todo o processo. Ao Prof. Dr. Paulo José Amaral Serni e Prof. Dr. Diego Rodrigo Cabral Silva pelas orientações e sugestões para deixar o trabalho mais didático possível, a todos os meus professores do curso de Mestrado em Engenharia Elétrica da Universidade Estadual Paulista pela excelência da qualidade técnica de cada um. Também agradeço a meus novos colegas Prof. Dr. Jeferson A. Bigheti, Me. Michel M. Fernandes, Felipe O. Domingues, que sempre me ajudaram com sua vasta experiência desde o início deste projeto de pesquisa. Agradeço ao Instituto Federal de São Paulo pelo incentivo a qualificação de seus colaboradores. Agradeço a Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 2018/19984-4 pelo apoio a realização dessa pesquisa. Sou grato à minha família pelo apoio que sempre me deram durante toda a minha vida. Aos meus pais que sempre estiveram ao meu lado me apoiando ao longo de toda a minha trajetória. À minha querida esposa Mikely pelo seu amor incondicional, pela compreensão e paciência demonstrada durante o período do projeto. Deixo aqui meu sincero obrigado a todos aqueles que, de alguma maneira, me ajudaram a chegar até aqui.

RESUMO

Novas aplicações e soluções na Indústria 4.0 se concentram na combinação de informática industrial e tecnologias de automação, incluindo a Internet Industrial das Coisas, Sistemas de Controle em Rede, Arquiteturas Orientadas a Serviços e Computação em Nuvem. O grande desafio dessas aplicações é promover a integração entre essas tecnologias, equipamentos e sistemas alocados em diferentes níveis hierárquicos dos sistemas industriais, tornando a automação colaborativa através do uso e compartilhamento de serviços para obtenção de uma arquitetura flexível, distribuída e totalmente integrada através de redes de comunicação no cenário industrial. Diante desse contexto, este trabalho enfoca no desenvolvimento de uma planta piloto industrial de controle de processos utilizando uma arquitetura orientada a microsserviços (MOA) baseada no *framework Moleculer.* Essa planta piloto, com controle de nível, pressão e vazão da tubulação e pressão de reservatório, é usada como base para o desenvolvimento e testes de microsserviços e mecanismos de segurança para o controle das malhas de processo. Os microsserviços desenvolvidos são: Aquisição de Dados (DAQ), Controle PIDPlus, Rastreador para métricas, Base de dados e monitoramento de processo, e Segurança de acesso (Guarda). Diferentes mecanismos de segurança para a arquitetura são implementados, como acesso do desenvolvedor aos microsserviços via chave criptografada, requisições HTTPS, autenticação de usuários com *token*, opções de conexão com o transportador de mensagens NATS, serviço de guarda de controle de acesso entre microsserviços com JSON *Web Token*. Resultados experimentais analisam o desempenho de dois tipos de composição de microsserviços numa aplicação de controle de processo em malha fechada, sendo que a Coreografia (execução sequencial predeterminada dos microsserviços) é executada na metade do tempo da Orquestração (execução dos microsserviços gerenciada por um elemento central) e com menor variabilidade. Além disso, compara-se o desempenho da comunicação entre os microsserviços usando três tipos de transportadores de mensagem: TCP, NATS e MQTT. Os resultados demonstram que o uso de microsserviços, em ambas as composições por Coreografia e Orquestração, é compatível e confiável, cumprindo requisitos de tempo de resposta e de segurança para aplicações de automação e controle de processos, além de fornecer novos requisitos, como modularidade, escalabilidade e interoperabilidade, necessários para essas aplicações no contexto da Indústria 4.0.

Palavras-chave: Indústria 4.0. Controle de Processos. Arquitetura Orientada a Serviços. *Framework Moleculer*. Orquestração. Coreografia. Mecanismos de Segurança.

ABSTRACT

New applications and solutions in Industry 4.0 focus on the combination of industrial computing and automation technologies, including the Industrial Internet of Things, Network Control Systems, Service-Oriented Architectures and Cloud Computing. The great challenge of these applications is to promote the integration between these technologies, equipment and systems allocated at different hierarchical levels of industrial systems, making the collaboration collaborative through the use and sharing of services to obtain a flexible, distributed and fully integrated architecture through of communication networks in the industrial scenario. In this context, this work focuses on the development of an industrial pilot plant for process control using a microservice oriented architecture (MOA) based on the Moleculer framework. This pilot plant, with level control, pipeline pressure and flow and reservoir pressure, is used as a basis for the development and testing of microservices and safety mechanisms for the control of process loops. The microservices developed are: Data Acquisition (DAQ), PIDPlus Control, Tracker for metrics, Database and process monitoring, and Access Security (Guard). Different security mechanisms for the architecture are implemented, such as developer access to microservices via encrypted key, HTTPS requests, user authentication with token, connection options with the NATS message carrier, guard service for access control between microservices with JSON Web Token. Experimental results analyze the performance of two types of microservice composition in a closed-loop process control application, with choreography (predetermined sequential execution of microservices) being performed in half the time of the Orchestration (execution of managed microservices by a central element) and with less variability. In addition, the communication performance between microservices is compared using three types of message transporters: TCP, NATS and MQTT. The results demonstrate that the use of microservices, in both compositions by Choreography and Orchestration, is compatible and reliable, fulfilling response time and security requirements for automation and process control applications, in addition to providing new requirements , such as modularity, scalability and interoperability, necessary for these applications in the context of Industry 4.0.

Keywords: Industry 4.0. Process control. Service Oriented Architecture. Moleculer Framework. Orchestration. Choreography. Security mechanisms.

LISTA DE FIGURAS

Figura 1 - Integração entre Dispositivos de Controle com a Computação e Comunicação em um sistema cyber físico. 17
Figura 2 – Os quatro estágios da revolução industrial 21
Figura 3 - Evolução da automação industrial: tradicional da automação ISA-95 (pirâmide no lado esquerdo) com uma infraestrutura plana baseada em informações para composição de serviços e aplicativos (lado direito). 25
Figura 4 - Comparativo de Estruturas Monolíticas e de Microsserviços 27
Figura 5 - Composição de serviços por Orquestração 29
Figura 6 - Composição de serviços por Coreografia 30
Figura 7 - Modelo representativo de um aplicativo baseado em microsserviços. 33
Figura 8 - Um modelo de referência para implantação segura de microsserviços. 33
Figura 9 - Arquitetura Monolítica 35
Figura 10 - Arquitetura Orientada a Microsserviços 36
Figura 11 – Arquitetura mista 36
Figura 12 - Criação de um Microsserviço usando Função *createService()* 40
Figura 13 - Contêiner ServiceBroker do Moleculer 41
Figura 14 - Configuração padrão e personalizada do ServiceBroker 41
Figura 15 - Exemplo da Propriedade Settings de um Microsserviço no Moleculer 42
Figura 16 - Método de Comunicação RPC das Chamadas Externas de Ações 43
Figura 17 - Chamada da Ação get via *broker.call* de um Microsserviço do Moleculer 44
Figura 18 - Exemplo da Propriedade *Methods* de um Microsserviço do Moleculer 45
Figura 19 - Diagrama de Evento Balanceado para Microsserviço no Moleculer 46
Figura 20 – Nome do Grupo do Evento de um Microsserviço do Moleculer 47
Figura 21 - Exemplo da Função *broker.emit* de um Microsserviço do Moleculer 47
Figura 22 - Diagrama de um Evento Broadcast de um Microsserviço no Moleculer 48
Figura 23 - Exemplo do Método *broker.broadcast* de um Microsserviço do Moleculer 48
Figura 24 - Fluxo de solicitação do usuário 49
Figura 25 - Planta piloto industrial 52
Figura 26 - P&ID da planta piloto industrial. 53
Figura 27 - Legenda do P&ID da planta piloto industrial 54
Figura 28 – Painel de controle com *Raspberry Pi* 3B + e MegaIO Industrial 55
Figura 29 - Painel de acionamento 56

Figura 30 - Microsserviço transportador 58
Figura 31 - Microsserviço API Gateway 59
Figura 32 - Estrutura do controlador PIDPlus 60
Figura 33 - Esquema de criação de um banco de dados 63
Figura 34 - Escrita das variáveis dos sensores da planta-piloto no banco de dados 64
Figura 35 - Dashboard *Grafana* para Supervisão das Variáveis de Processo 65
Figura 36 - Intermediador de serviços com o parâmetro *metrics* habilitado 65
Figura 37 - Exemplo do serviço de rastreamento 66
Figura 38 - Replica de serviço 66
Figura 39 - Comandos do PM2. 67
Figura 40 - Esquema geral do hardware, com seus respectivos softwares, meios de comunicação e serviços. 69
Figura 41 - Camadas de segurança 70
Figura 42 - Comunicação entre cliente e servidor utilizando SSH 71
Figura 43 - Chave criptografada rsa-key-public. 72
Figura 44 - Cliente SSH Putty 72
Figura 45 - Acesso ao terminal da Raspberry Pi pelo PC com cliente Putty SSH 73
Figura 46 - Listas de chaves inicializadas pelo Pageant 73
Figura 47 - Acesso em um site com HTTP e HTTPS 74
Figura 48 - Esquema do serviço API com HTTPS 75
Figura 49 - Autenticação no Api Gateway 76
Figura 50 - Arquivo *nats-server.conf* 77
Figura 51 - Inicialização do servidor *NATS* com seu respectivo arquivo de configuração 77
Figura 52 - Esquema de um serviço com usuário e senha para conexão com o NATS 78
Figura 53 - Ferramenta *mkpasswd* para encriptar a senha 78
Figura 54 - Arquivo *nats-server.conf* com senha criptografada 78
Figura 55 - Log de conexão de um serviço com o transporter NATS 79
Figura 56 - Geração chave com o método guard.generate com o parâmetro daq 80
Figura 57 - Esquema do serviço DAQ com JWT 80
Figura 58 - Painel frontal do LabVIEW ilustrando os componentes da planta piloto 82
Figura 59 - Resposta de controle de todas as malhas da planta 83
Figura 60 - Malha de controle de pressão da tubulação 84
Figura 61 - GET.vi 85
Figura 62 - Painel frontal do LabVIEW utilizado para orquestração dos serviços 86

Figura 63 - Diagrama de bloco do LabVIEW contendo uma requisição. 86
Figura 64 - Diagrama da sequência de orquestração 87
Figura 65 - Diagrama da sequência de orquestração externa 88
Figura 66 - Diagrama da sequência de orquestração interna 89
Figura 67 - Diagrama da sequência de coreografia externa 90
Figura 68 - Diagrama da sequência de coreografia interna 91
Figura 69 - Gráfico do tempo de controle de todas as composições testadas. 92
Figura 70 - Moleculer Benchmark 93
Figura 71 - Comparação de diferentes tipos de *transporters* mantendo a estrutura do experimento de orquestração com aplicação externa (subseção 5.2.1) analisando o tempo total. 94
Figura 72 - Comparação de diferentes tipos de mecanismos de segurança usando como referência a mesma padronização do experimento de orquestração com aplicação externa (subseção 5.2.1) 95
Figura 73 - Painel de Frontal de Controle da Planta Piloto 103

LISTA DE TABELAS

Tabela 1 - Esquema das Propriedades dos Microsserviços 40
Tabela 2 - Configurações no ServiceBroker..... 41
Tabela 3 - Quantidade e descrição dos principais itens da planta-piloto industrial 57
Tabela 4 - Orquestração externa com transportador NATS 88
Tabela 5 - Orquestração interna com transportador NATS..... 89
Tabela 6 - Coreografia externa com transportador NATS 90
Tabela 7 - Coreografia interna com transportador NATS 91

LISTA DE SIGLAS E DEFINIÇÕES

SIGLA	DESCRIÇÃO
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
CLI	*Command Line Interface*
DAQ	*Data Acquisition*
DB	*Data Base*
HTTP	*Hyper Text Transfer Protocol*
HTTPS	*Hyper Text Transfer Protocol Secure*
I2C	*Inter Integrated Circuit*
I4.0	*Industry 4.0*
IIoT	*Industrial Internet of Things*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISA	*International Society of Automation*
JSON	*JavaScript Object Notation*
JWT	JSON *Web Token*
MOA	*Microservice Oriented Architecture*
MQTT	*Message Queuing Telemetry Transport*
MV	*Manipulated Variable*
NATS	*Messaging System*
NPM	*Node Package Manager*
P&ID	*Piping and Instrumentation Diagram/Drawing*
PID	*Proportional Integral Derivative*
PM2	*Process manager*
PV	*Process Variable*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
RSA	*Rivest-Shamir-Adleman*
SOA	*Service Oriented Architecture*
SSH	*Secure Shell*
SSL	*Secure Socket Layer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
URL	*Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO E JUSTIFICATIVA 16
1.1 Objetivos 19
1.2 Estrutura do Trabalho 19
2 REVISÃO DA LITERATURA 21
2.1 Revolução Industrial 21
2.2 Indústria 4.0 22
2.3 Arquitetura Orientada a Serviços 24
2.4 Arquitetura Orientada a Microsserviços 26
2.4.1 Orquestração 28
2.4.2 Coreografia 29
2.5 Aplicações Industriais 30
2.6 Segurança 32
3 FRAMEWORK MOLECULER 35
3.1 Serviço 39
3.2 Transporter 42
3.3 Gateway 42
3.4 Ações 42
3.5 Métodos 45
3.6 Eventos 45
3.7 Exemplificação fluxo de solicitação do usuário 49
4 DESENVOLVIMENTO DO TRABALHO 51
4.1 Planta-Piloto Industrial baseada em Serviços 51
4.2 Hardware 54
4.3 Softwares 57
4.3.1 Serviços 57
4.3.1.1 Transportador 57
4.3.1.2 Gateway 58
4.3.1.3 Controle 59
4.3.1.4 Aquisição de dados 62
4.3.1.5 Base de dados e Monitoramento 62
4.3.1.6 Rastreador 65
4.3.2 Replicação de serviços 66

4.3.3 Gerenciador de processos 67
4.4 Diagrama de Conexão 68
4.5 Segurança 70
4.5.1 Acesso à Raspberry Pi com Cliente SSH 70
4.5.2 HTTPS e Autenticação/Autorização 74
4.5.3 Autenticação ao conectar-se com transportador NATS 76
4.5.4 Microsserviço de Guarda com JSON *Web Token* 79
5 RESULTADOS E DISCUSSÕES 82
5.1 Controle das Malhas de Processo 83
5.2 Orquestração 85
5.2.1 Aplicação Externa 85
5.2.2 Aplicação Interna 89
5.3 Coreografia 90
5.3.1 Aplicação Externa 90
5.3.2 Aplicação Interna 90
5.4 Comparação 91
5.4.1 Composição de serviços 91
5.4.2 Protocolos de Mensagens (*Transporters*) 92
5.4.3 Mecanismos de Segurança 94
6 CONCLUSÃO 97
7 REFERÊNCIAS 99
8 APÊNDICE 103
8.1 Procedimento de Operação da Planta Piloto 103

1 INTRODUÇÃO E JUSTIFICATIVA

A manufatura tem evoluído até chegar em sua versão 4.0, integrando novas tecnologias objetivando maior eficiência. Para Lu (2017), a I4.0 (Indústria 4.0) retrata uma evolução dos sistemas produtivos a partir da convergência entre tecnologias da automação (TA) e tecnologia da informação (TI). A evolução e o desenvolvimento das tecnologias digitais propiciaram a criação de novos métodos de produção nas indústrias globais baseados na automação, robótica, inteligência artificial, Internet das Coisas e inteligência de dados, dentre outras inovações. A utilização dessas tecnologias no contexto industrial, coordenadas de modo a conferir competitividade ao negócio, otimizar a eficiência da cadeia produtiva, adicionar valor ao produto, racionalizar o uso dos recursos e customizar as soluções tecnológicas é chamada de I4.0.

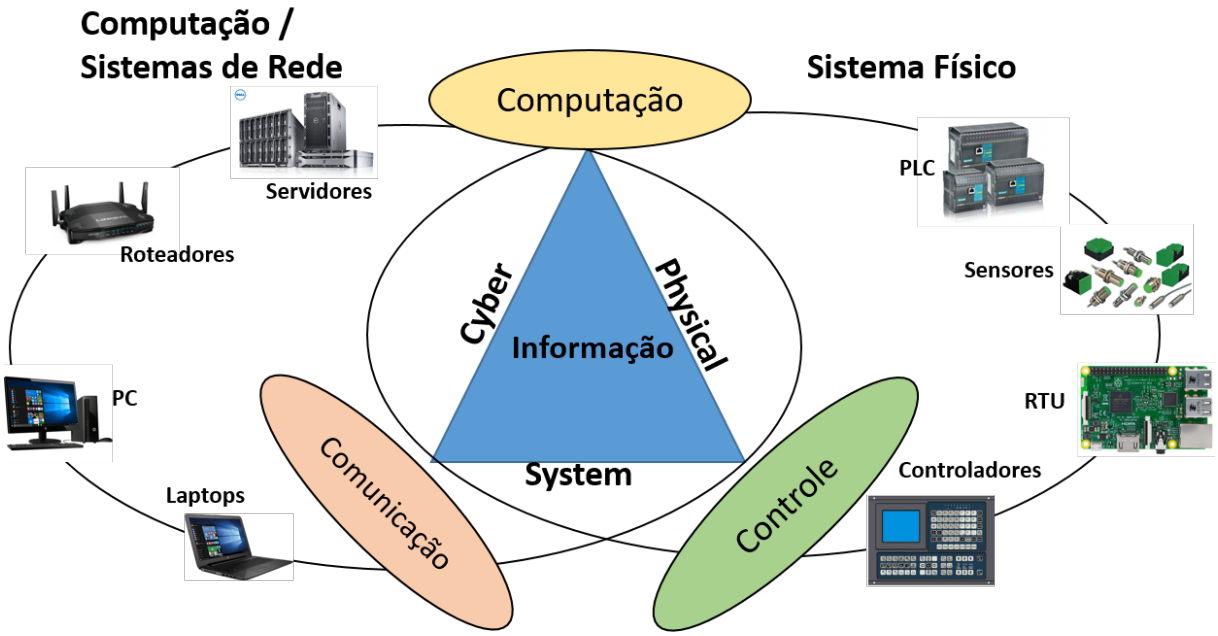
Um primeiro exemplo dessa convergência entre TI e TA foi a criação do gêmeo digital, cuja primeira definição foi feito em 2002 por Michael Grieves no contexto de uma apresentação da indústria sobre a gestão do ciclo de vida do produto. O gêmeo digital representa a interconexão e convergência entre um sistema físico e sua representação digital, incluindo de forma otimizada todas as informações relativas ao sistema que poderiam ser obtidas do mundo real (KRITZINGER et al., 2018; RAZA et al. 2020).

Um segundo exemplo dessa junção entre TI e TA, é o surgimento do termo “Sistema Cyber-Físico (CPS)” que foi proposto em 2006 por um grupo de pesquisadores da fundação americana NSF (*National Science Foundation*), com o propósito de descrever a integração entre dispositivos de controle com a computação e redes de comunicação como pode ser visto na Figura 1. Ao lado esquerdo temos a computação ou sistemas de rede que englobam computadores, roteadores, servidores e ao lado direito o sistema físico, com os controladores lógicos programáveis (PLC), sensores e controladores. Na parte central temos o espaço cyber físico fazendo uma afluência entre os dois mundos, tendo seus três principais pilares, computação, comunicação e controle, dos quais compartilham todas as informações.

Os CPS são sistemas com capacidade computacional que estão conectados com dispositivos do mundo físico e os processos que os cercam, ao mesmo tempo que fornecem e consomem serviços de dados disponíveis na Internet (KANG et al., 2016). De acordo com Colombo et al. (2017), os dispositivos de campo, máquinas, módulos de produção e produtos são compreendidos como CPS que trocam informações de forma autônoma, acionando ações e controlando umas às outras de forma independente. Para Hermann et al. (2016), o CPS,

para se adequar aos requisitos da I4.0, deve abranger clientes, máquinas, produtos, estoques e prestadores de serviço, de forma que interajam executando ações autonomamente.

Figura 1 - Integração entre Dispositivos de Controle com a Computação e Comunicação em um sistema cyber físico.



Fonte: BIGHETI (2020).

Os CPS dependem da integração perfeita de algoritmos computacionais e componentes físicos. A fim de tornar esses sistemas interoperáveis entre si para suportar as novas aplicações da Indústria 4.0, novas arquiteturas industriais têm sido desenvolvidas. Um paradigma recente para a indústria tem sido o desenvolvimento de soluções colaborativas por meio do uso e compartilhamento de serviços para obtenção de uma arquitetura flexível, escalável, interoperável, distribuída e totalmente integrada em rede. Nesse sentido, o desenvolvimento de arquiteturas orientadas a serviços (SOA – *Service Oriented Architecture*) para a aplicação da I4.0 tem se destacado conforme descrito em *Jammes* et al. (2014).

Colombo et al. (2014) discutem o conceito de uma arquitetura de automação baseada no uso da Computação em Nuvem e de Serviços Web (*Web Services*) para comunicação e realização das tarefas entre os diferentes componentes e equipamentos de um sistema industrial. Nessa arquitetura, um sistema de armazenamento de informações em nuvem é compartilhado pelos equipamentos e sistemas, tornando possível o uso de serviços padronizados para comunicação entre os mesmos. Os serviços podem ser acessados por aplicações, sistemas e outros serviços independentemente de onde eles estão alocados (hardware, software e rede de comunicação), propiciando uma arquitetura colaborativa.

Delsing (2017) exibe o conceito de nuvem de automação local, no qual os componentes e dispositivos de automação são disponibilizados como serviços, provendo interoperabilidade total para aplicações industriais. Neste projeto toda a infraestrutura de integração e comunicação foi realizada através de uma Arquitetura Orientada a Serviços (SOA), desenvolvida com o *Framework Arrowhead*. Leitão et al. (2016) discute algumas iniciativas financiadas pela União Europeia sobre aplicações industriais de SOA, onde são apresentadas as arquiteturas SOA desenvolvidas e discutidas implementações e casos de estudo onde estas arquiteturas foram validadas.

Ciavotta et al. (2017) apresentam uma proposta de Arquitetura Orientada a Microsserviços (MOA) para fomentar a implantação do conceito de fábrica digital. Microsserviços representam uma evolução do conceito de SOA. O foco é o desenvolvimento de aplicações de I4.0 e possui como diferencial o suporte para a simulação de processos industriais e criação de gêmeos digitais.

O panorama apresentado demonstra a importância da pesquisa e desenvolvimento de SOA e MOA, para fornecer os novos requisitos das aplicações industriais no contexto da I4.0. Mais significativo é a discussão de resultados experimentais obtidos com o desenvolvimento e implementação dessas arquiteturas, os quais permitam avaliar questões práticas relacionados ao desempenho, confiabilidade e dificuldades deste tipo de aplicação.

Diante desse contexto, esta pesquisa propõem complementar o estudo desenvolvido por Bigheti, Fernandes e Godoy (2019) que propõem uma nova abordagem para Indústria 4.0 com a uma arquitetura de automação baseada em microsserviços para aplicações da I4.0. Uma arquitetura de microsserviços consiste em uma coleção de pequenos serviços autônomos, onde cada serviço é independente e implementa uma única funcionalidade. Este trabalho visa desenvolver serviços e analisar os resultados dessa arquitetura usando diferentes tipos de composição de microsserviços em aplicações de controle de processos no cenário da Indústria 4.0. Pretende-se realizar testes reais sobre essa arquitetura com diferentes tipos de composição, questões de segurança, estabilidade, escalabilidade e redundância aplicados em uma planta piloto industrial, bem como o desenvolvimento de serviços para o controle de processos de todas as suas malhas de controle: nível tanque, vazão de linha, pressão linha e pressão reservatório.

1.1 Objetivos

Este trabalho objetiva o desenvolvimento e testes operacionais de microsserviços e mecanismos de segurança para aplicações de controle de processos na Indústria 4.0, além da avaliação de desempenho da composição de microsserviços por Orquestração e Coreografia. O desenvolvimento de uma planta piloto industrial de controle de processos utilizando uma arquitetura orientada a microsserviços (MOA) baseada no framework *Moleculer*, a qual é utilizada para o desenvolvimento experimental do trabalho, também é descrita.

1.2 Estrutura do Trabalho

Este trabalho de mestrado possui mais sete capítulos, além deste capítulo introdutório que contém o contexto, justificativa e objetivos desta pesquisa. No Capítulo 2 é apresentada uma revisão da literatura sobre a revolução industrial, conceitos e tecnologias relacionadas à Indústria 4.0, bem como são discutidos as arquiteturas SOA e MOA, diferentes composição de serviços, coreografia e orquestração, as principais aplicações industriais que contem esse tipo de arquitetura, além de aspectos de segurança em arquiteturas baseadas em serviços.

O Capítulo 3 apresenta o *framework Moleculer*, utilizado para a criação de microsserviços na plataforma *Node.js*, explicando seus principais pontos: o que é um serviço; Para que é necessário um transportador de mensagens; Serviço *api gateway* padrão; Ações, métodos e eventos de um microsserviço; Exemplo de fluxo de solicitação do usuário para uma visualização geral de como o *framework* trabalha e interage com todos os elementos de sua estrutura.

No Capítulo 4 é descrita a proposta desse trabalho de mestrado, que envolve o desenvolvimento de uma planta piloto industrial de controle de processos com o intuito de realizar testes mais aprofundados da arquitetura orientada a microsserviços na indústria 4.0. Todos os *hardwares* e *softwares* utilizados na planta são apresentados. O desenvolvimento dos microsserviços Aquisição de Dados (DAQ), Controle PID*Plus*, Rastreador para métricas, Base de dados e monitoramento de processo, e Segurança de acesso (Guarda) é detalhado, além de como fazer a replicação dos serviços e gerenciá-los e na sequencia uma visão geral de como o *hardware* e *software* estão distribuídos e realizam sua comunicação exibidos em um diagrama de conexão. Alguns mecanismos de segurança são aplicados na estrutura: Acesso do desenvolvedor aos microsserviços via chave criptografada; Requisições HTTPS; Autenticação de

usuários com *token*; Opções de conexão com o transportador de mensagens NATS; Serviço de guarda de controle de acesso entre microsserviços com JSON *Web Token*.

O Capítulo 5 apresenta os resultados dos diversos experimentos realizados nesse trabalho, contemplando discussões sobre a comparação entre a composição de microsserviços por Orquestração e Coreografia, aplicações externas e internas, comparação entre diferentes tipos de transportador de mensagens na arquitetura e sobre o impacto da implementação dos mecanismos de segurança.

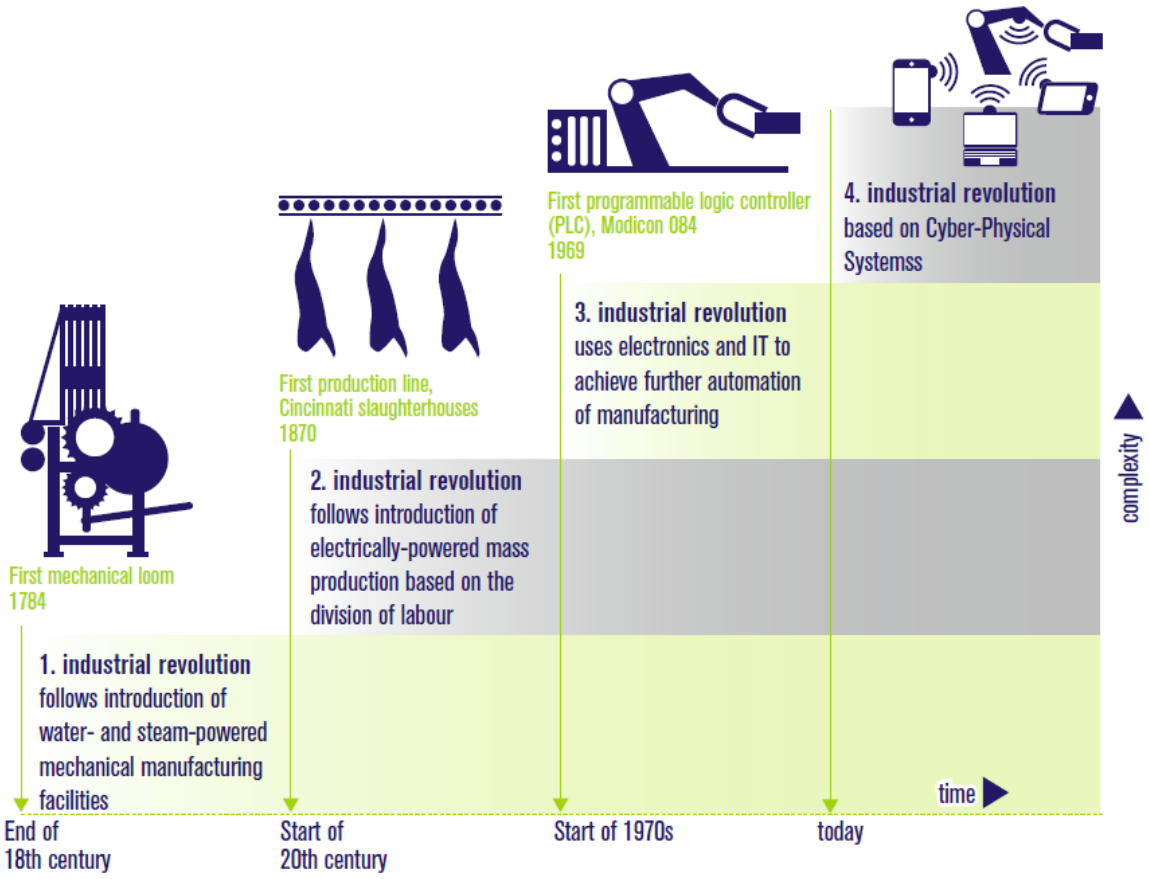
As conclusões são apresentadas no Capítulo 6. No Capítulo 7 estão as referências bibliográficas e por último Capítulo 8 os procedimentos básicos de Operação da Planta Piloto.

2 REVISÃO DA LITERATURA

2.1 Revolução Industrial

A primeira revolução industrial iniciou-se com a introdução da mecanização dos processos, que era realizado de forma manual. Dessa forma surgiram as primeiras fábricas da manufatura, sendo realizado a separação entre as relações dos donos de meios de produção e os trabalhadores assalariados. Foi nessa primeira etapa da revolução que o método do saber fazer, por parte dos profissionais que muitas vezes trabalhavam de modo individual deixou de ser o principal meio de trabalho, para ser substituído pela rápida produção de consumo por parte das fábricas (LU, 2017). Todas as revoluções industriais podem ser observadas de forma resumida na Figura 2 (KAGERMANN; WAHLSTER; HELBIG, 2013).

Figura 2 – Os quatro estágios da revolução industrial



Fonte: KAGERMANN; WAHLSTER; HELBIG (2013).

O período de transição da primeira para a segunda revolução industrial se deve a eletricidade, que passou a ser o componente fundamental da produção em massa. Determinada

produção era realizada sobre diferentes bens de consumos, que contribuíram para o crescimento da economia no século XX. É nesse processo, que modelos de trabalho como o proposto por Henry Ford, denominado de Fordismo, obtém sucesso junto a indústria automobilística, reforçando os produtos padronizados e a produção em massa (LU, 2017).

Na terceira revolução industrial, o computador passa a ser considerado como a principal máquina nas atividades produtivas. Por meio da união de hardware e software, essa ferramenta de trabalho apresenta grande destaque a tecnologia digital, permitindo maior adequação das tarefas produtivas por meio das equipes de controle (STOCK; SELIGER, 2016).

A partir do desenvolvimento das práticas apresentadas pela terceira revolução industrial associado ao crescimento e avanço de tecnologias, surge a quarta revolução industrial, também denominada como indústria 4.0 (LU, 2017). O desenvolvimento das atividades da manufatura junto aos apontamentos das tendências ao longo do tempo propiciou em um novo modelo de trabalho as indústrias, no qual as tecnologias são criadas para integração cada vez maior entre as máquinas e os humanos.

2.2 Indústria 4.0

Para Wang e Wang (2016), a Indústria 4.0 está associada a tecnologias digitais que detêm grande relevância no processo de fabricação, mas que não as limitam em suas respectivas utilizações. O conceito surgiu na Alemanha em 2011 (JAZDI, 2014) e ganhou visibilidade global, sendo atualmente utilizado para abordar o uso de tecnologias de Internet para elevar a eficiência de produção através de serviços inteligentes em fábricas inteligentes. Segundo Sanders, Elangeswaran e Wulfsberg (2016), a Indústria 4.0 representa a aplicação de conceitos dos sistemas ciberfísicos (CPS) e tecnologias que visam a construção de fábricas inteligentes, nas quais a dependência dos seres humanos diante o comando das máquinas seja cada vez menor.

Do ponto de vista de aplicação, é uma mistura da ideia de IoT com a ideia de Sistemas *Cyber*-Físicos (SISINNI et al., 2018). Sistemas *Cyber*-Físicos (CPS - *Cyber-Physical Systems*) são aqueles nos quais há integração da computação com os processos físicos, ou seja, uma "intersecção" entre o físico e o virtual.

Alguns princípios fundamentais são recorrentes nas diversas definições de Indústria 4.0 introduzidas ao longo do tempo (BASSI, 2017). São eles:

- ✓ Uso extensivo da Internet: o uso de serviços de comunicação cabeada ou sem fio nos dispositivos inteligentes possibilita acesso direto às camadas superiores de processos e

serviços. Isto eleva o valor agregado, promove suporte para o uso otimizado de recursos e o controle inteligente do processo de fabricação (JAZDI, 2014). Uma das principais vertentes é o uso de *big data* para a predição de falhas, de forma que se faça a manutenção antes que ocorra a quebra do equipamento. Isto reduz drasticamente o tempo de parada da planta e os prejuízos decorrentes da interrupção da linha (BASSI, 2017);

- ✓ Flexibilidade: a capacidade de operar linhas produtivas flexíveis, com capacidade de customização e minimização do tempo de reprogramação (*setup*), de forma a atender as necessidades individuais do consumidor sem custo adicional por parada e reprogramação da linha de produção (JAZDI, 2014). Algumas tecnologias habilitadoras típicas são a impressão 3D e a rastreabilidade e identificação do produto na linha de produção (BASSI, 2017);
- ✓ Comunicação, Virtualização e CPS: a criação de modelos unificados em contraposição aos diversos protocolos industriais atuais (AS-i, PROFIBUS, PROFINET, CAN, etc.) que em geral são incompatíveis entre si ou de difícil interoperabilidade. A disponibilização de um *framework* comum de comunicação permite o uso de virtualização, conectando os sistemas físicos a contrapartes virtuais (BASSI, 2017).

Segundo Kagermann, Wahlster e Helbig (2013), o potencial da I4.0 fica evidente devido à algumas características e possibilidades, como:

- ✓ Atender aos requisitos individuais do cliente: A linha de produção pode se modificar para atender critérios individuais, específicos do cliente. Mudanças de última hora e volumes de produção muito baixos (tamanho de lote de 1), também são possíveis;
- ✓ Flexibilidade: É possível configurar dinamicamente diferentes aspectos dos processos de negócios, tais como qualidade, tempo, risco e preço. Isso possibilita a otimização do uso de materiais, os processos de engenharia podem ser tornados mais ágeis, processos de fabricação podem ser alterados, a escassez temporária de algum material (devido a problemas de fornecimento, por exemplo) pode ser compensada e grandes aumentos no tamanho dos lotes produzidos podem ser alcançados em um curto espaço de tempo;
- ✓ Tomada de decisão otimizada: A I4.0 permite a verificação antecipada das decisões de projeto, respostas mais flexíveis à interrupção e otimização global em todos os setores de uma empresa;
- ✓ Produtividade e eficiência dos recursos: A mesma motivação que resultou nas revoluções industriais anteriores impulsiona a I4.0, a obtenção da maior produção possível a partir de um determinado volume de recursos (produtividade dos recursos) e utilização da menor quantidade possível de recursos para produzir uma determinada quantidade de produtos (eficiência de recursos).

- ✓ Os processos de fabricação podem ser otimizados para cada caso individual, até mesmo sem parar a produção, sendo continuamente otimizados em termos de consumo de recursos e energia ou de redução de suas emissões.

As aplicações da I4.0 são baseadas na ideia de que onde tudo estará conectado para que as melhores decisões de produção, custo e segurança sejam tomadas, tudo sob demanda e em tempo real. Para que isso aconteça, é necessário a interconexão de dados de automação e infraestrutura de redes de TI de forma segura e confiável (WOLLSCHLAEGER; SAUTER; JASPERNEITE, 2017).

As arquiteturas industriais de automação e controle tradicionais não fornecem, principalmente, estes requisitos de interoperabilidade e integração vertical e horizontal de equipamentos e sistemas (BORANGIU et al., 2019; SISINNI et al., 2018). Em consequência, torna-se necessário o desenvolvimento de novas arquiteturas industriais que forneceram tais requisitos e suportem as demandas tecnologias relacionadas à I4.0. Nesse sentido a utilização da Arquitetura Orientada a Serviços (SOA) surge com uma solução para a integração das tecnologias já existentes para a I4.0.

2.3 Arquitetura Orientada a Serviços

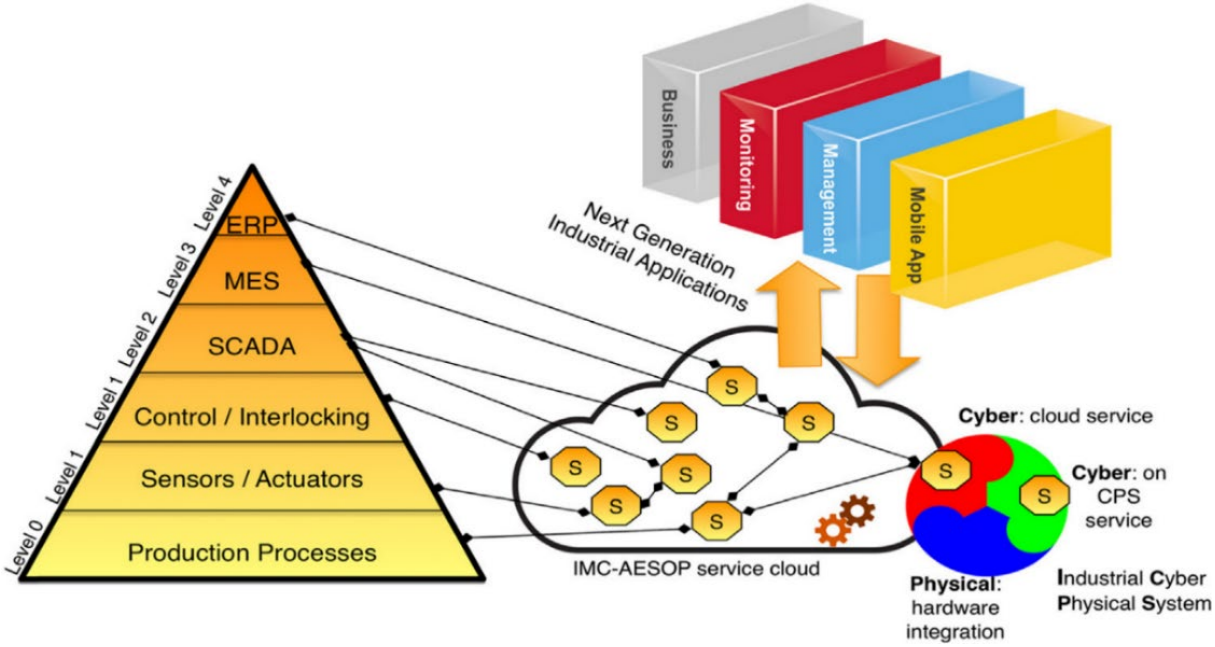
A arquitetura orientada a serviço foi apresentada pela primeira vez, no artigo "*Service Oriented Architectures*", em abril de 1996 por *Roy Schulte* e *Yefim Natis* do *Gartner Group*. SOA não é definida como uma tecnologia, nem como metodologia ou serviço, mas como um conceito de arquitetura para sistemas corporativos, com a finalidade de promover a integração entre o modelo de negócio e a tecnologia da informação por meio de serviços (XIAO et al., 2016.

Esta ponte permite expor as funcionalidades dos aplicativos em serviços padronizados e inter-relacionados. Sendo esse realizado por meio de interfaces de serviços fracamente acoplados, onde os serviços não necessitam de detalhes técnicos da plataforma dos outros serviços para a troca de informações ser realizada.

Pesquisas, como a de Leitão et al. (2016) têm focado no desenvolvimento de SOA com a promessa de oferecer uma arquitetura para suportar a propagação e a utilização de serviços virtuais reutilizáveis. Com o uso de SOA, informações provenientes de diversos sistemas heterogêneos podem ser obtidas de forma transparente ao usuário ou aplicação por meio de serviços. Dessa forma, como mostrado na Figura 3, ocorre uma migração da tradicional

arquitetura em camadas ISA-95 (*International Society of Automation*) para uma arquitetura SOA em nuvem.

Figura 3 - Evolução da automação industrial: tradicional da automação ISA-95 (pirâmide no lado esquerdo) com uma infraestrutura plana baseada em informações para composição de serviços e aplicativos (lado direito).



Fonte: Leitão et al. (2016)

Na SOA, cada elemento dos diferentes níveis hierárquicos da arquitetura ISA-95 não mais se comunica somente com as camadas adjacentes e é somente um fornecedor de dados para a camada superior. O conceito de serviços permite que esses elementos passem a ser clientes ou servidores, a depender da necessidade do processo ou aplicação, e que haja interação, baseada em troca de mensagens, entre quaisquer elementos dos níveis hierárquicos do processo industrial (MORAES, 2017). Além disso, a SOA permite que qualquer aplicação industrial, possam ser rapidamente compostas pela seleção e combinação de novos serviços e funcionalidades disponibilizadas como um serviço em nuvem.

A SOA estabelece uma arquitetura que permite que os serviços sejam publicados, descobertos e consumidos por aplicações ou outros serviços. Portanto, a ideia básica é assegurar um ambiente uniforme para oferecer, descobrir, interagir e utilizar as aplicações.

Nesse caso a SOA permite definir a arquitetura distribuída com o conceito de serviço, onde eles podem ser invocados de forma independente, por qualquer cliente (externos ou internos) que solicitou o serviço, para processar funções simples ou trabalhar em conjunto por meio de implementações coordenadas, com o objetivo de desenvolver novas funcionalidades para os processos existentes.

2.4 Arquitetura Orientada a Microsserviços

O grande marco dessa arquitetura foi o trabalho escrito por Fowler (2014), onde estão descritas todas as características que definem a estrutura de microsserviços. Xiao et al. (2014) descrevem em seu trabalho que o microsserviço é um aplicativo flexível, escalável, Inter operável, distribuído e totalmente integrado através de redes. Numerosos conceitos têm sido discutidos e novas formas de se organizar e construir sistemas computacionais vêm sendo colocadas em prática, deixando de lado as formas tradicionais de se desenvolver uma aplicação. A arquitetura baseada em microsserviços surge como uma alternativa ao tradicional padrão arquitetural monolítico.

Em uma arquitetura monolítica, uma única aplicação é responsável por todos os processos, sendo que essa aplicação é autossuficiente e independente de outras aplicações. A ideia é que a aplicação seja responsável não apenas por uma determinada tarefa, podendo também executar todos os passos necessários para completar uma macro função. Todos os serviços estão alocados no mesmo recurso computacional, indicando certo nível de dependência de um serviço para outro e provendo baixa modularidade para a aplicação.

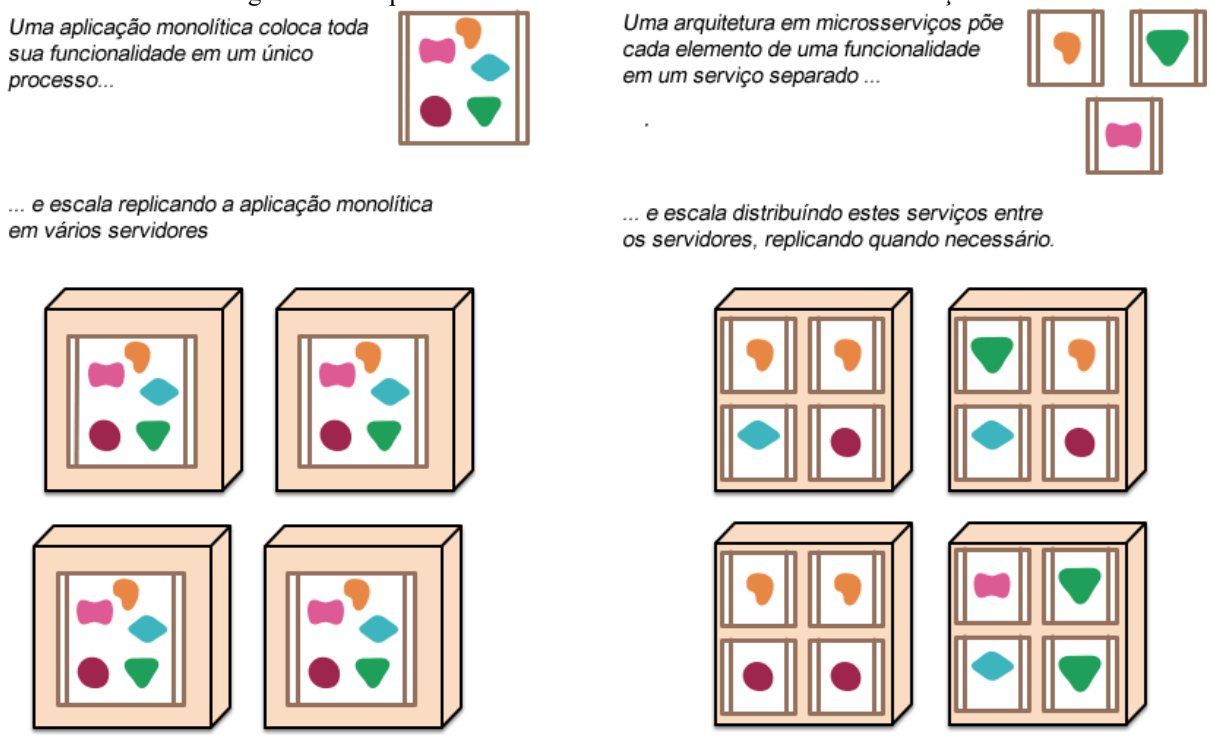
Com o crescimento da aplicação e sua complexidade, havendo um aumento dos serviços acoplados, a dependência entre os serviços começa a dificultar a manutenção e o desenvolvimento de novas funcionalidades (RICHARDSON, 2016).

A arquitetura baseada em Microsserviços (MOA) surge como uma alternativa ao tradicional padrão arquitetural monolítico. Os microsserviços são um estilo arquitetônico em que as aplicações são decompostas em serviços, oferecendo modularidade, tornando as aplicações mais fáceis de desenvolver, testar, implantar e, o mais importante, alterar e manter. Microsserviços também possuem algumas complexidades, como a dificuldade de manutenção de uma base de dados consistente, devido ao fato delas serem distribuídas por cada processo.

Newman (2017) e Richardson (2016) definem os microsserviços como pequenos serviços autônomos que trabalham juntos. Esses serviços rodam em seus próprios processos e se comunicam por meio de mecanismos leves tanto síncronos (*request/response*), quanto assíncronos (*request/callback*), geralmente por meio de REST(*Representational State Transfer*).

Os microsserviços são implantados e escalados de forma independentes e possuem fronteiras ou limites bem definidos, além de poderem ser escritos em diferentes linguagens e utilizar diferentes recursos para armazenamento de dados. Um comparativo entre as arquiteturas monolíticas e microsserviços é apresentado na Figura 4.

Figura 4 - Comparativo de Estruturas Monolíticas e de Microsserviços



Fonte: Lewis & Fowler (2014)

Observa-se no lado esquerdo na Figura 4, que um ambiente construído em meio à abordagem monolítica dificulta a escalabilidade de serviços específicos, enquanto no lado direito os serviços podem ser clonados de forma a facilitar o aumento de acesso a eles diminuindo a carga de requisições em um mesmo servidor, como também ser capaz de lidar com falhas. Os recursos podem ser mais bem aproveitados, evitando desperdício de recursos com alocação de serviços não ou pouco utilizados.

Fowler (2014) e Newman (2015) descrevem todas as características que definem a estrutura de microsserviço. Não é obrigatório encontrar todas essas características em todos os sistemas que seguem esta arquitetura, mas a maioria delas estarão presentes.

As principais características que definem um microsserviço são: cada microsserviço implementa um único recurso de negócios ou funcionalidade; um microsserviço suficientemente simples para ser desenvolvido e mantido por um único programador; microsserviços são executados em processos separados, comunicando-se por meio de padrões de mensagens ou APIs bem definidas; Microsserviços não compartilham armazenamentos de dados. Cada qual é responsável por gerenciar seus próprios dados; Microsserviços têm bases de código separadas e não compartilham código-fonte. No entanto, eles podem usar bibliotecas de utilitários comuns; cada microsserviço pode ser implantado e atualizado de forma independente de outros serviços.

Numa MOA, a saída de um serviço é usada como uma entrada para outro em uma coreografia de serviços independentes. Ao ser agnóstico de dispositivos e plataformas, os microsserviços fornecem flexibilidade para o sistema desenvolvido. Os microsserviços têm diferentes formas de comunicação e processamento, sendo mais um dos atributos que os distinguem do SOA tradicional.

Uma MOA oferece os seguintes benefícios de aplicação: Aumento da resiliência: usando microsserviços toda a aplicação é descentralizada e desacoplada em serviços que atuam como entidades separadas; Escalabilidade: A escalabilidade é um aspecto chave dos microsserviços, como cada serviço é um componente separado, permite-se expandir uma única funcionalidade (ação) ou serviço sem ter que dimensionar toda uma aplicação; Disponibilidade: os serviços críticos para a aplicação podem ser implantados em vários nós ou servidores para aumentar a disponibilidade e o desempenho de um serviço sem impactar o desempenho de outros serviços; Flexibilidade de desenvolvimento: usando microsserviços, o desenvolvimento da aplicação não fica limitado a um único *software* ou linguagem.

Dessa forma, é possível escolher a ferramenta certa para cada tarefa; Facilidade de desenvolvimento e modularidade: a criação dos microsserviços é mais simples e rápida por executarem funcionalidades específicas, fornecendo maior modularidade para a aplicação.

2.4.1 Orquestração

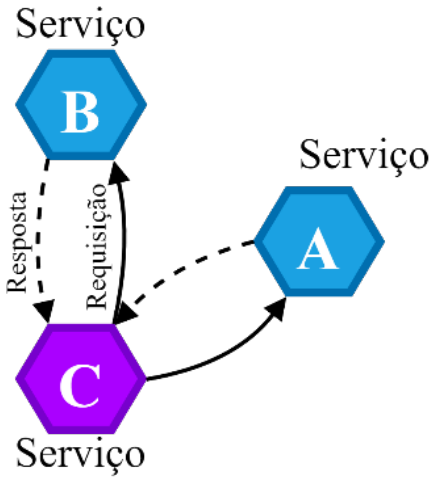
Orquestração de serviços é a composição de serviços para criar um serviço, para resolver uma tarefa de um processo de negócio ou para realizar uma atividade de uma aplicação. Neste caso, sempre há a figura de um ponto central, como um maestro de uma orquestra. O orquestrador coordena a chamada de outros serviços para compor uma função de maior granularidade.

Para Sheng et al. (2014), a orquestração pode ser considerada uma abordagem baseada no modelo *workflow* (Fluxo de Trabalho), contendo os nós e o fluxo de dados entre eles, onde um processo central controla os *Web Services* envolvidos e coordena a execução das diferentes operações.

A Figura 5 apresenta um exemplo de orquestração de serviços onde representa um único processo de negócios executável centralizado (o orquestrador) representado pela letra "C" que coordena a interação entre os diferentes serviços "A" e "B". O orquestrador é responsável por invocar e combinar os serviços. A orquestração inclui o gerenciamento de transa-

ções entre serviços individuais e ela emprega uma abordagem centralizada para a composição do serviço.

Figura 5 - Composição de serviços por Orquestração



Fonte: Autor

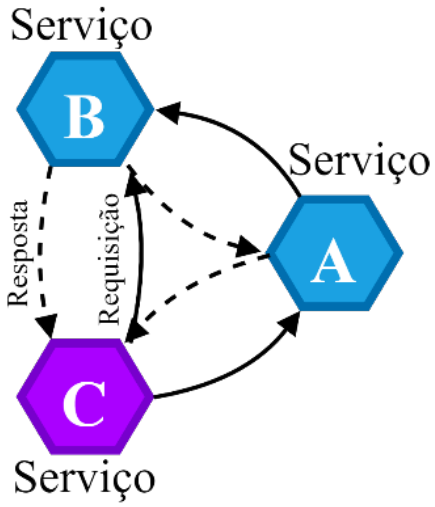
2.4.2 Coreografia

Na coreografia, a sequência de execução dos serviços já está pré-determinada antes da sua execução e os serviços se conhecem e conversam entre si diretamente sem um intermediador no meio. Por exemplo, quando um serviço é acionado e envia uma mensagem, outros serviços podem estar programados de antemão para receber ou não essa mensagem e dispararem outras ações.

Essa coreografia de serviço é uma descrição global dos serviços participantes, que é definida pela troca de mensagens, regras de interação e acordos entre dois ou mais terminais. A coreografia emprega uma abordagem descentralizada para a composição do serviço, descrevendo as interações entre múltiplos serviços. Isso significa que uma coreografia difere de uma orquestração, quando relacionado com a lógica que controla as interações entre os serviços envolvidos.

Os serviços são acionados conforme a classe de eventos que ocorrem, sendo isso uma característica básica da arquitetura orientada a eventos, onde por meio de um *middleware* é possível atribuir essa característica mediante a criação de fluxos como apresentado na Figura 6.

Figura 6 - Composição de serviços por Coreografia



Fonte: Autor

Para Fattori et al. (2011), a coreografia de serviço é mais colaborativa e permite que cada parte envolvida possa descrever seus serviços na interação. A coreografia representa uma descrição global do comportamento de cada um dos serviços participantes da interação, o que é definido pela troca pública de mensagens, regras de interação e acordos entre dois ou mais processos de negócios (SHENG et al.,2014).

Souit (2013) afirma que a coreografia é a interação entre componentes distribuídos sem a existência de uma entidade controlando a lógica de colaboração, como acontece no mecanismo de orquestração. A coreografia é normalmente associada com interações que ocorrem entre múltiplos Serviços Web, ao passo que apenas as trocas de mensagens públicas são consideradas relevantes e cada serviço conhece apenas sobre suas interações e comportamentos.

2.5 Aplicações Industriais

Nos últimos tempos, diversos trabalhos desenvolveram arquiteturas SOA e avaliaram sua a utilização em aplicações industriais no contexto de CPS, IIoT e I4.0. Leitão et al. (2016) apresentam uma revisão de algumas iniciativas financiadas pela União Europeia, com destaque ao projeto IMC-AESOP, que é bastante relacionado à proposta deste trabalho. O projeto IMC-AESOP (*ArchitecturE for Service-Oriented Process*) (IMC-AESOP, 2011) desenvolveu uma arquitetura SOA em nuvem para aplicações industriais focada principalmente em aplicações de controle de processos, sistemas SCADA (*Supervisory Control and Data Acquisition*) e DCS (*Distributed Control Systems*). O trabalho de Karnouskos et al. (2014) descreve a concepção dos serviços e o mecanismo de composição de serviços para obtenção de outras funci-

onalidades, destacando a comunicação via *Web Service* entre os serviços através do Perfil de Dispositivos para Serviços da *Web* (DPWS).

Os resultados do projeto IMC-AESOP foram base para o desenvolvimento do projeto ARROWHEAD (ARROWHEAD, 2014), de 2013 a 2017. Este projeto desenvolveu o conceito de nuvem de automação local (DELSING, 2017), no qual os componentes e dispositivos de automação eram disponibilizados como serviços, provendo interoperabilidade total para aplicações industriais. Neste projeto, toda a infraestrutura de integração e comunicação era realizada através de uma SOA, estruturada no formato de um *framework*. O *framework* Arrowhead atualmente é usado nos projetos *Arrowhead Tools* (ARROWHEAD TOOLS, 2019), com vigência de 2019 a 2022, e *Productive 4.0* (PRODUCTIVE 4.0, 2019), com vigência de 2017 a 2020.

O conceito de serviços permite que os dispositivos, equipamentos e sistemas industriais passem a ser clientes ou servidores de dados, a depender da necessidade do processo ou aplicação, e que haja interação entre eles, de forma transparente, não importando os níveis hierárquicos do processo industrial a que eles pertençam (MORAES, 2017). Além disso, a arquitetura SOA permite que qualquer aplicação industrial, no contexto de CPS, IIoT e I4.0, possam ser rapidamente compostas pela seleção e combinação de novos serviços e funcionalidades disponibilizadas como um serviço em nuvem.

Dois projetos europeus estão focando no desenvolvimento de MOA para aplicações no contexto de CPS, IIoT e I4.0. Ciavotta et al. (2017) apresentam uma proposta de MOA para fomentar a implantação do conceito de fábrica digital. Esta arquitetura faz parte do projeto MAYA (*Multi-disciplinArY integrated simulAtion and forecasting tools)* (MAYA, 2019), cujo foco é o desenvolvimento de aplicações de CPS e I4.0 e possui como diferencial o suporte para a simulação de processos industriais e criação de gêmeos digitais (*Digital Twin*). Na arquitetura são propostos cinco grupos principais de serviços que se comunicam via padrão REST com HTTP e *WebSocket* via TCP/IP. Um destaque pode ser dado aos microsserviços orquestrador (*Orchestrator*) e Agendador (*Scheduler*), os quais coordenam e organizam os outros serviços para permitir a composição e criação de serviços de alto nível e aplicações de processo. Innerbichler et al. (2017) apresentam uma proposta de MOA em nuvem para construção de uma plataforma colaborativa para a I4.0, que permita o monitoramento, a otimização e a negociação em tempo real com base em IoT nas cadeias de fornecimento (*Supply Chain*) de manufatura. Esta arquitetura faz parte do projeto NIMBLE (NIMBLE, 2018).

Bigheti et al. (2018) apresentam uma proposta de arquitetura orientada a microsserviços (MOA) em nuvem para aplicações de IIoT e I4.0, discutindo as características, detalhes

operacionais, vantagens e potencias de aplicação. Os autores explicam que a integração de diferentes tecnologias usando uma arquitetura SOA através do uso e compartilhamento de microsserviços fornece interoperabilidade para as aplicações. Por fim, demonstram que os testes operacionais iniciais da arquitetura realizados com a implementação de uma malha de controle e supervisão de uma variável de processo usando microsserviços validam a proposta.

2.6 Segurança

Com aumento significativo dos microsserviços em diferentes áreas, uma preocupação em proteger tais serviços em diferentes níveis e estágios tem surgido, levando em consideração o ciclo de vida típico de um projeto, design, desenvolvimento, testes, manutenção, verificação, monitoramento, infraestrutura e interfaces.

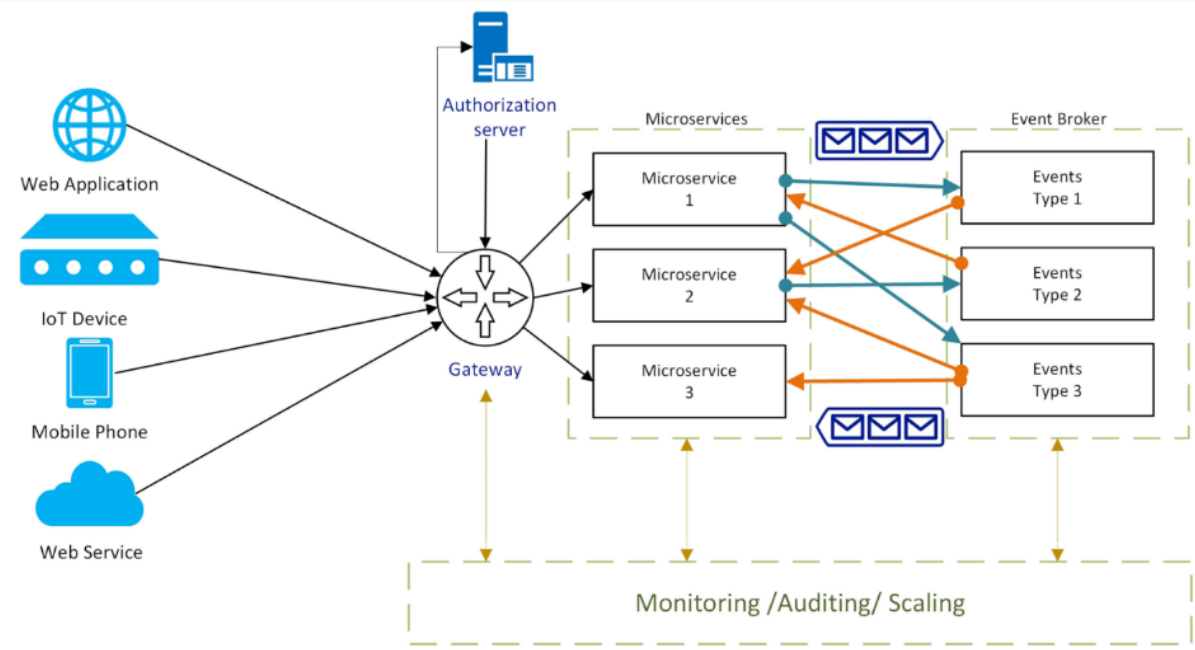
Os microsserviços apresentam problemas de segurança desafiadores. Primeiro, a complexidade da rede introduzida pelo grande número de microsserviços aumenta bastante a dificuldade de monitorar a segurança de todo o aplicativo. Segundo, os microsserviços geralmente são projetados para confiar completamente um no outro; portanto, o comprometimento de um único microsserviço pode reduzir o aplicativo inteiro (SUN; NANDA; JAEGER, 2016).

Pahl e Donini (2018) propõem a proteção de microsserviços IoT com certificados, fornecendo serviços de IoT com autenticação, responsabilidade e integridade do desenvolvedor ao ambiente de execução. Ele permite que todas as entidades participantes verifiquem a operação correta das entidades anteriores na cadeia de processamento.

Nehme et al. (2019) mostra uma estrutura básica de ponta a ponta que pode ser vista na Figura 7, onde o gateway lida com solicitações de uma diversidade de clientes externos. Por ter uma posição central, ele pode retornar respostas personalizadas de acordo com o tipo de cliente. Ele também gerencia o gerenciamento de acesso, comunicando-se com um servidor de autorização.

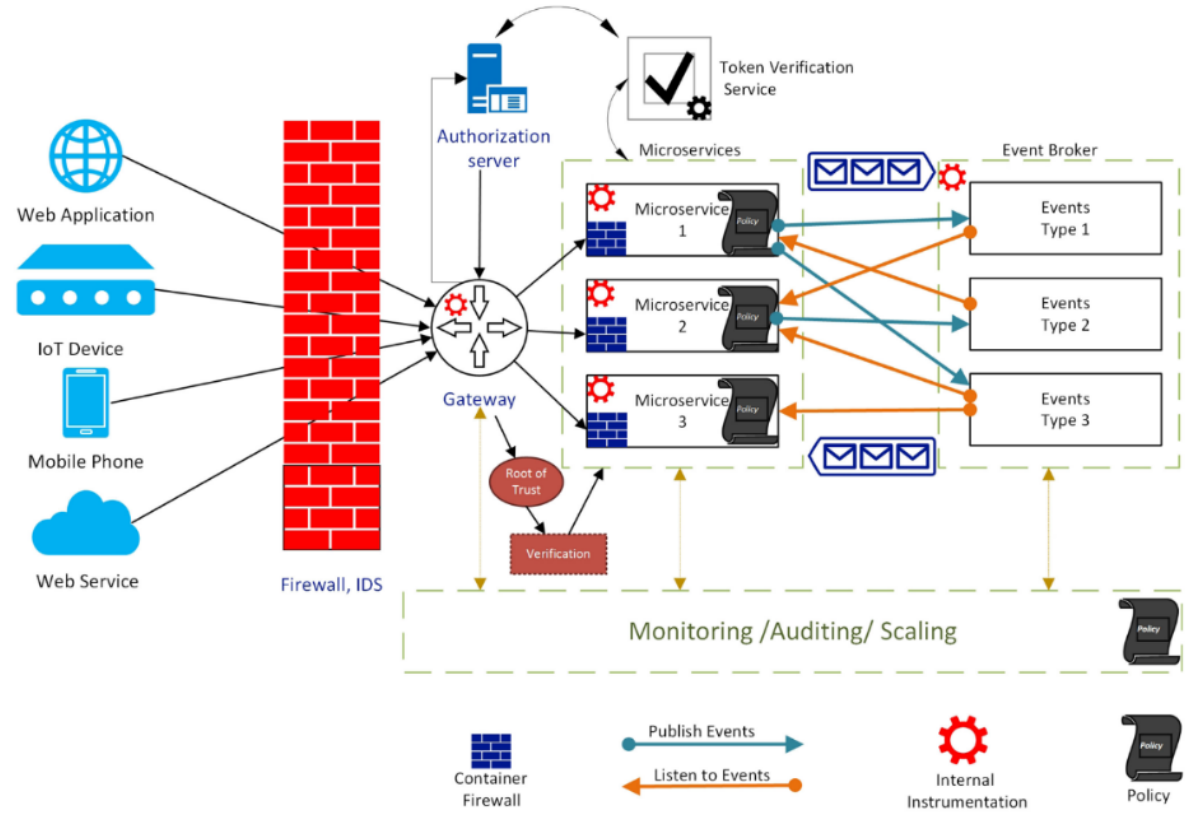
O gateway encaminha solicitações para serem processadas por microsserviços. Para obter uma funcionalidade comercial específica, os microsserviços se comunicam produzindo e consumindo eventos usando um *eventbroker*, cuja função é publicar e armazenar eventos de mudança de estado. Cada microsserviço pode publicar e assinar eventos de diferentes categorias. Um agente de eventos é um aplicativo autônomo e, devido à sua função, é essencial na auditoria e monitoramento.

Figura 7 - Modelo representativo de um aplicativo baseado em microsserviços.



Fonte: Nehme et al, (2019)

Figura 8 - Um modelo de referência para implantação segura de microsserviços.



Fonte: Nehme et al, (2019)

Com base no modelo de referência Figura 7, é proposto uma modificação para incorporar a segurança, que pode ser visto na Figura 8.Os elementos de rede são inseridos para

aplicar políticas no nível da rede, desde regras simples de tráfego até inspeção profunda de pacotes, procurando tráfego malicioso ou extraindo informações inteligentes. As políticas também devem ser definidas, aplicadas e verificadas para segregar a comunicação e o acesso entre vice-vice-versa.

A verificação do token verifica a validade do token de acesso em vez de confiar totalmente no gateway, e as políticas podem definir os direitos de acesso do token. Esta é uma atenuação contra vulnerabilidades em potencial que surgem do gateway, sendo um representante confuso se comprometido.

Um subsistema de monitoramento, teste e verificação é adicionado. Esses componentes devem interagir diretamente com os componentes de instrução no nível do microsserviço (representado por engrenagens). Esses agentes são, idealmente, parte integrante do esqueleto de qualquer serviço e devem ser enriquecidos com métricas específicas de serviço. Os próprios contêineres devem ser monitorados, mas também se espera suporte do sistema operacional subjacente.

Uma raiz de confiança suporta os processos de *bootstrap-ping*, mantendo imagens de contêiner, material criptográfico e configurações. Isso é usado para garantir a autenticidade dos componentes de software e configurações de segurança.

Qualquer solicitação do mundo externo deve passar por um *firewall* e *IDS* (*Intrusion Detection* System), e os *firewalls* de contêiner devem inspecionar solicitações do gateway ou qualquer tráfego interno em potencial. Os *tokens* de acesso também devem ser verificados quanto à autenticidade no nível de microsserviços e processados para o controle de acesso pelas regras de política de microsserviços. Além disso, todas as partes críticas do sistema devem ser sistematicamente monitoradas, e as imagens e configurações dos contêineres devem ser avaliadas em relação às imagens confiáveis de hardware e software.

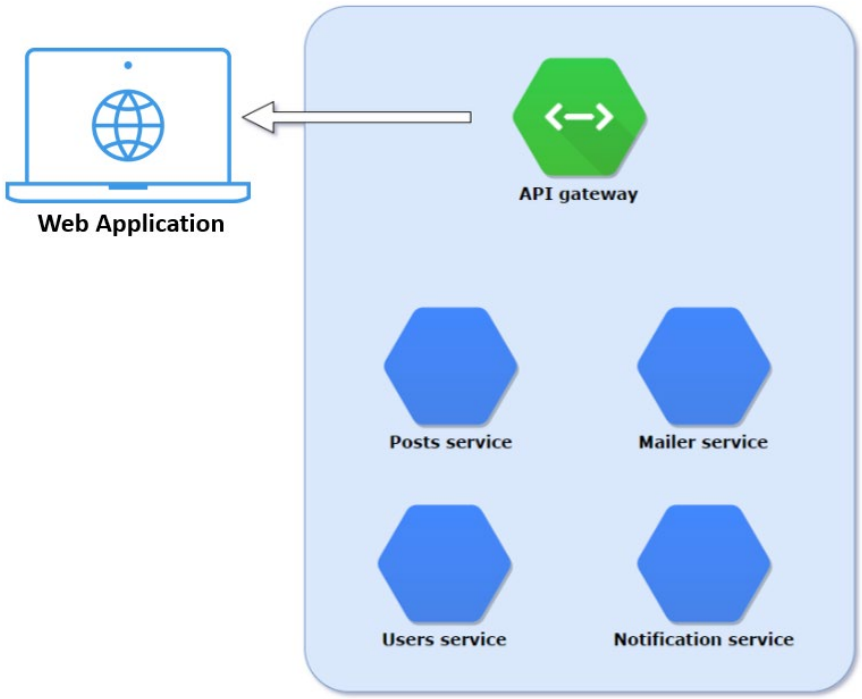
A base para o desenvolvimento de uma solução SOA ou MOA se encontra no framework, ou conjunto de soluções de software, utilizado. O framework *Moleculer* de microsserviços, utilizado neste trabalho é apresentado n próxima seção.

3 FRAMEWORK MOLECULER

Um *framework* representa uma estrutura base ou uma plataforma de desenvolvimento, como um arcabouço, que contém ferramentas, bibliotecas, sistemas e componentes que agilizam o processo de desenvolvimento de uma solução específica. O *framework Moleculer* é uma estrutura de desenvolvimento com a linguagem *JavaScript* para desenvolvimento de aplicações orientadas a microsserviços (MOLECULER, 2020). O *framework* é de código aberto (*open source*) e roda sobre a plataforma Node.js. O framework *Moleculer* suporta três tipos de arquitetura de software: Monolítica, Microsserviço e Mista.

Na estrutura monolítica, Figura 9, todos os serviços estão sendo executados no mesmo nó como um monólito. Não há latência de rede e nenhum módulo transportador. As ligações locais são as mais rápidas.

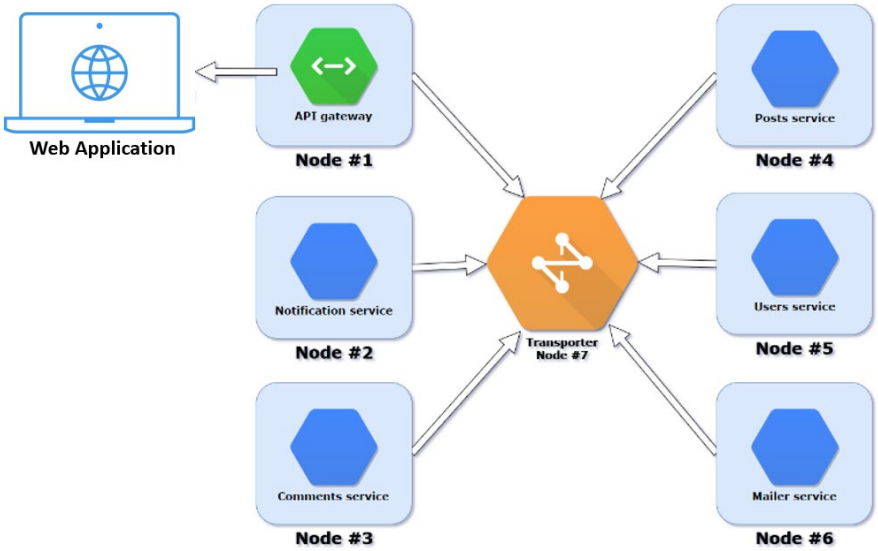
Figura 9 - Arquitetura Monolítica



Fonte: Adaptado Moleculer (2020)

A Arquitetura orientada a microsserviço, Figura 10, os serviços são executados em nós individuais que se comunicam via protocolo de comunicação (*transporter*) por meio de um *broker* de mensagens. Nesta arquitetura distribuída, a latência da comunicação entre os serviços não é insignificante, mas os serviços podem ser replicados para proporcionar resiliência e tolerância a falhas.

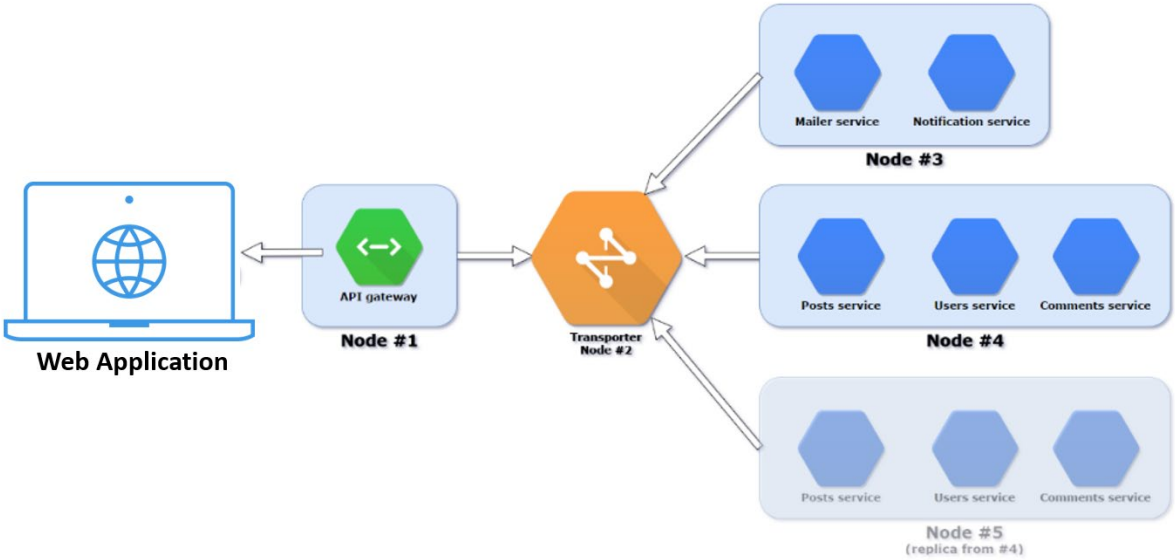
Figura 10 - Arquitetura Orientada a Microsserviços



Fonte: Adaptado Moleculer (2020)

Na arquitetura mista, Figura 11, estamos executando serviços coerentes em um grupo no mesmo nó. Ele combina as vantagens das arquiteturas monolítica e microsserviços. Por exemplo, se o serviço de postagens chama o serviço de usuários várias vezes, nós os colocamos no mesmo nó, para que possamos reduzir a latência de rede entre esses serviços. Se o nó estiver sobrecarregado, iremos aumentá-lo.

Figura 11 – Arquitetura mista



Fonte: Adaptado Moleculer (2020)

Na arquitetura do *Moleculer*, todos os microsserviços são iguais, não havendo nenhum tipo de hierarquia ou prioridade. Uma grande vantagem desse *framework* é que todos os microsserviços desenvolvidos possuem disponível um recurso automático de registro e descoberta.

Dessa forma, todos os serviços existentes são informados após a criação de um novo serviço ou a disponibilização de uma nova funcionalidade em um serviço. Outro recurso importante é o balanceamento automático de carga, o qual tem função de distribuir dinamicamente a carga da comunicação entre os microsserviços uniformemente.

Os quadrados de cor azul na arquitetura (Figura 10) representam os nós de gerenciamento do *Moleculer* para suporte ao desenvolvimento, descoberta e configuração dos (micro) serviços. Cada nó pode conter um ou mais serviços e é responsável pelo gerenciamento dos seus dados, possuindo seu próprio banco de dados, seguindo o padrão de um banco de dados por serviço. O *framework Moleculer* possui um conjunto oficial de adaptadores de banco de dados, o adaptador padrão da *Moleculer* é baseado em *NeDB*, tendo suporte também ao *MongoDB, Mongoose, Sequelize, CouchDB, TypeORM, DynamoDB, GunDB, RethinkDB, MacroMeta e orientDB*.

Cada serviço disponibilizado na arquitetura é representado com um hexágono azul. Os serviços podem oferecer e executar diferentes tarefas que são chamadas de ações (exemplo: um serviço de aquisição de dados pode oferecer uma ação de aquisição de dados de entrada e uma ação de atualização de dados de saída). Na Figura 10 cada serviço disponibilizado na arquitetura é representado com um hexágono azul.

O hexágono verde representa o (micro) serviço de *gateway* (API Gateway), o qual tem a função de interface de conexão dos serviços internos (hexágonos azuis) com aplicações externas por meio de chamadas via rede ou nuvem por meio do protocolo REST. A comunicação entre os microsserviços é realizada via um serviço de mensagens (*Transporter*) também conhecido como *Broker* de mensagens, representado pelo hexágono laranja, existentes em transportes como o MQTT e o NATS, por exemplo, que requerem instalação, ou seja, tem dependência.

O serviço de mensagens (*Transporter*) é um dos componentes mais importantes da arquitetura e numa infraestrutura para a mensageria, pois é o mecanismo que coordena o envio e a recepção de mensagens em uma fila. O *Transporter* é capaz de enfileirar e manter as mensagens até serem processadas por consumidores. O produtor simplesmente envia a mensagem para o *Transporter*, sem a necessidade de um mecanismo de descoberta de serviços.

Alguns protocolos de mensagens, como o TCP, requerem a utilização do mecanismo de descoberta de serviços, pois não tem dependência e configuração zero. Ele usa o protocolo *Gossip* para disseminar o status do nó, a lista de serviços e os batimentos cardíacos. Ele contém um recurso de descoberta UDP integrado para detectar nós novos e desconectados na rede. Não sendo necessário listar todos os nós remotos. É necessário que pelo menos um nó que esteja online. Por exemplo, crie um nó de fofoca "sem serviço", que não faz nada, apenas compartilha outros endereços de nós remotos por mensagens de fofoca. Portanto, todos os nós devem saber apenas o endereço do nó *gossiper* para poder se comunicar com todos os outros nós.

A comunicação baseada em mensagens também suporta uma variedade de padrões de comunicação, incluindo os pedidos de caminho único (*one-way requests*) e *publish-subscribe*. Uma vantagem desse serviço no *Moleculer* é que se um mesmo serviço é disponibilizado em múltiplas instâncias em diferentes nós (maior disponibilidade), as requisições recebidas pelo *Transporter* serão automaticamente balanceadas entre os nós que estão online naquele momento e que podem executar o serviço requisitado. Além disso, com o uso do *Transporter,* a mudança de um mecanismo de comunicação para outro (entre os diferentes disponíveis no *Moleculer*) é transparente.

Na arquitetura de microsserviços do *framework Moleculer,* utiliza-se o protocolo de comunicação para mensageria, ou seja, para se comunicar com os outros microsserviços, sendo o grande benefício dessa abordagem, o desacoplamento entre produtores e consumidores de eventos e dados. O desacoplamento simplifica o desenvolvimento e aumenta a disponibilidade, em comparação ao uso de transações distribuídas. Caso nenhum consumidor esteja disponível para processar um evento, o *Transporter* irá enfileirar o evento até que a mensagem possa ser processada.

O serviço *Transporter* de mensageria é uma técnica que visa solucionar a comunicação entre sistemas completamente diferentes de uma maneira confiável. *Broker* de mensagem faz com que seja possível integrar tais sistemas para que eles possam trocar dados de forma desacoplada. A troca de informações entre os microsserviços utilizando o *Transporter* é realizado pelo módulo *Serializers,* responsável pela serialização dos dados das mensagens.

Os principais recursos para microsserviços que compõem o framework *Moleculer* são (MOLECULER, 2020): Conceito de requisição-resposta; Arquitetura baseada em eventos; Suporte a *middlewares* para Node.js; Suporte ao armazenamento em cache de variável; Diversas opções de comunicação (*Transporters);* Diversas opções de *serializers*: JSON (*JavaScript Object Notation*), *MsgPack, Protocol Buffer* etc.; Suporte ao desenvolvimento de múltiplos

serviços em um mesmo nó; Suporte nativo para o registro de serviços; Descoberta automática de serviços; API para interface com aplicações externas.

O *framework Moleculer* é um estilo arquitetônico em que as aplicações são decompostas em serviços de baixo acoplamento, oferecendo modularidade, tornando as aplicações mais fáceis de desenvolver, testar, implantar e, o mais importante alterar e manter. As principais classes de componentes operacionais do *framework Moleculer* são: *ServiceBroker*; *Service*.

3.1 Serviço

Um serviço é um módulo *JavaScript* simples que contém alguma parte de um aplicativo complexo. Ele é isolado e autocontido, o que significa que, mesmo se ficar offline ou travar, os serviços restantes não serão afetados.

O componente *Service* representa um microsserviço no *framework Moleculer*. Para criar um microsserviço, deve-se definir um esquema (*schema*) com as seguintes propriedades: *Name*: nome do microsserviço. É a primeira parte do nome da ação de quando é chamado o microsserviço. *Version***:** É usada para executar várias versões do mesmo microsserviço, com ações diferentes. *Settings:* configurações do microsserviço, como porta de comunicação.

Essas configurações são enviadas durante o procedimento de descoberta do serviço; *Actions:* ações ou funções que compõem o funcionamento do microsserviços, podendo ser chamadas internamente por outros microsserviços ou externamente via API REST; *Methods:* métodos são funções privadas do microsserviço que somente são executadas internamente; *Events:* eventos podem ser disparados conforme execução das ações do microsserviço.

Cada microsserviço publica um evento sempre que necessário e outros microsserviços podem se inscrever em eventos publicados. Na Figura 12 é apresentado o objeto *broker* criado a partir da classe *ServiceBroker* que utiliza a função *createServive*({*schema*}). Na Tabela 1 é descrito o funcionamento de cada propriedade do *schema* (esquema) da função *createService()*.

Figura 12 - Criação de um Microsserviço usando Função *createService()*

```
const {ServiceBroker} = require("moleculer");
const = new ServiceBroker ({
    nodeID: "node-1"
    logger: true,
    logLevel: "info",
});

broker.createService({
    name: "posts",
    actions: {
        get(ctx){
        this.logger.info("Log message via service logger");
        }
    }
});
```

Fonte: Adaptado Moleculer (2020)

Tabela 1 - Esquema das Propriedades dos Microsserviços

Propriedade	Descrição
name	O *name* é uma propriedade obrigatória na criação do microsserviço, portanto, deve ser definido. É nome do microsserviço. Que neste caso o *name* é *"posts"*.
actions	A *actions* é a propriedade para execução de uma ação do microsserviço podendo ter uma ou mais ações. O nome da ação é *get* que envia uma mensagem para o console "*Log message via Service logger*". Que neste caso *actions* é *"get"* .

Fonte: Autor

Nó

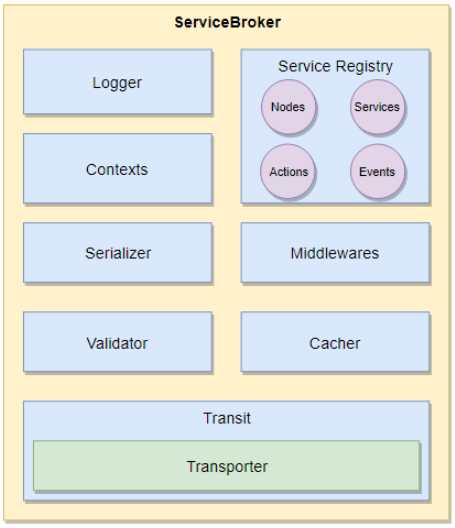
Um nó é um processo de sistema operacional simples executado em uma rede local ou externa. Uma única instância de um nó pode hospedar um ou vários serviços.

Intermediário de Serviço

O *ServiceBroker* é o principal componente da estrutura do *Moleculer*. É um contêiner com recursos embarcados para simplificação do desenvolvimento e mediação dos microsserviços, sendo responsável pela configuração dos nós como: nome do nó, registro de serviços, descoberta automática de serviços, *cache* de variáveis, *serializer*, *middlewares* e *Transporter*, entre outros recursos como pode ser visto na Figura 13.

Para instanciar um *Node* (nó), é necessário carregar o módulo *Moleculer* no Node.js. Após isso, pode-se criar um objeto de *ServiceBroker* e em seguida configurar as operações de funcionamento do microsserviço.

Figura 13 - Contêiner ServiceBroker do Moleculer



Fonte: Moleculer (2020)

O objeto *ServiceBroker*, utiliza um serviço de mensagens (*Transporter*)**,** para se comunicar com outros nós. A Tabela 2 apresenta as explicações da configuração de cada comando do *ServiceBroker* da Figura 14.

Figura 14 - Configuração padrão e personalizada do ServiceBroker

```
// Standard ServiceBroker configuration
const {ServiceBroker} = require("moleculer");
const = new ServiceBroker ({
// Custom configuration
    nodeID: "node-1"
    transporter: "nats://localhost:4222",
    logLevel: "info",
    requestTimeout: 5*1000,
    requestRetry: 3
});
```

Fonte: Adaptado Moleculer (2020)

Tabela 2 - Configurações no ServiceBroker

Opção	Descrição
nodeID	O nome node (nó) é “node-1”.
transporter	Configuração do *transporter* utilizado, que neste caso é NATS(*Messaging System*) que está se comunicando os outros microsserviços pela porta 4222.
logLevel	Nível de *log* para mensagens de console dos *logs* de informação do funcionamento do microsserviço, como (*trace, debug, info, warn, error, fatal*). Que neste caso é “debug”
requestTimeout	Tempo em milissegundos de espera de uma resposta a uma requisição.
requestRetry	Número de tentativas de solicitação de requisição de comunicação com um microsserviços.

Fonte: Autor

3.2 Transporter

O transportador é um barramento de comunicação que os serviços usam para trocar mensagens. Ele transfere eventos, solicitações e respostas.

3.3 Gateway

O API Gateway expõe os serviços do *Moleculer* aos usuários finais. O gateway é um serviço *Moleculer* regular rodando um servidor. Ele lida com as solicitações de entrada, mapeia-as em chamadas de serviço e retorna as respostas apropriadas.

A propriedade *Settings*, Figura 15, tem a função de configuração dos recursos disponíveis nos microsserviços. Por exemplo, a configuração da porta de comunicação dos comandos API REST que serão enviados por meio do microsserviço Gateway.

Figura 15 - Exemplo da Propriedade Settings de um Microsserviço no Moleculer

```
let ApiService = require("moleculer-web");

module.exports = {
    name: "api",
    mixins: [ApiService]
    settings: {
        // Change port settings
        port: 8080
    },
    actions: {
        myAction() {
            // Add a new action to apiService
        }
    }
}
```

Fonte: Adaptado Moleculer (2020)

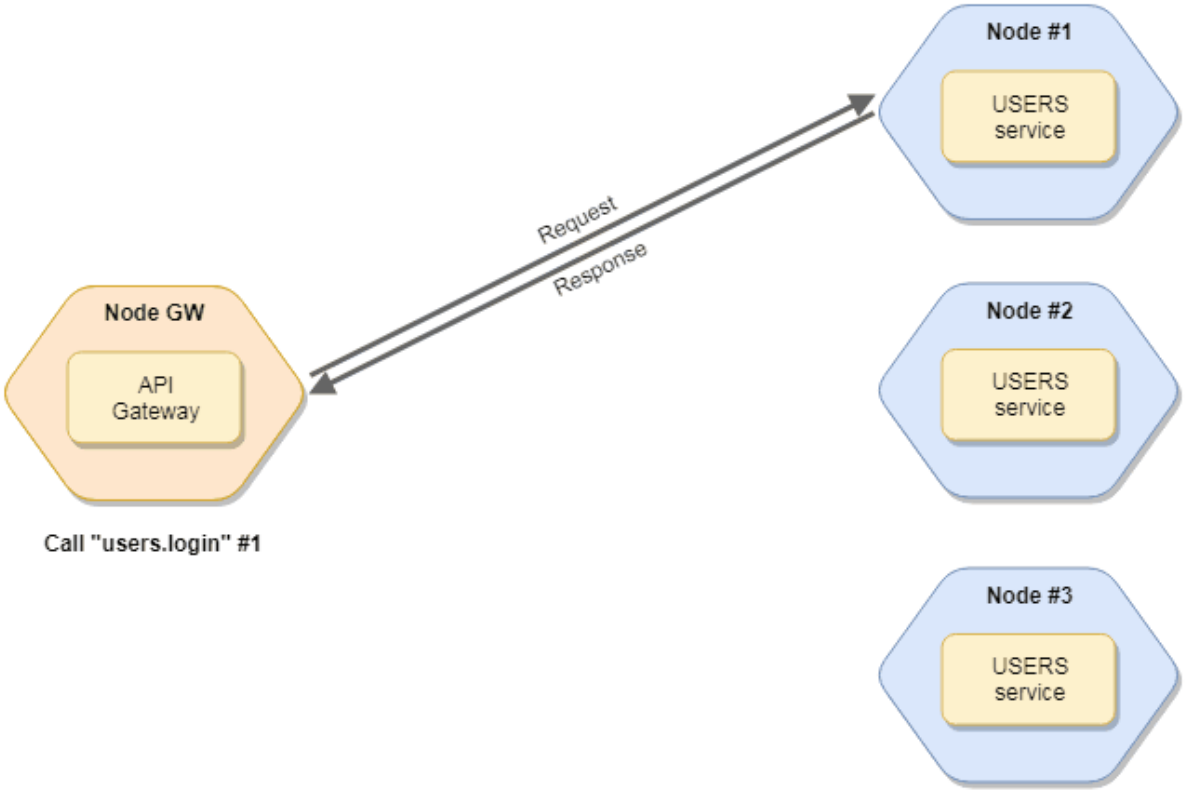
3.4 Ações

A propriedade *Action* pode ter dois tipos de chamadas. As chamadas internas que são executadas pela função *broker.call* ou externas via API REST por meio do microsserviço *Gateway*. As chamadas externas das ações utilizam métodos públicos que podem ser chamados por qualquer objeto programado no microsserviço.

O método utilizado para as chamadas das ações é RPC(*Remote Procedure Call*), que são solicitadas via requisições HTTP externamente por um navegador por exemplo e interna-

mente utilizando o transporter escolhido, podendo ser NATS, MQTT entre outros suportados pelo framework. O diagrama da Figura 16 apresenta o método de comunicação por RPC do *Moleculer*.

Figura 16 - Método de Comunicação RPC das Chamadas Externas de Ações



Fonte: Moleculer (2020)

Essas chamadas externas das ações são executadas via API REST por meio do microsserviço Gateway, por exemplo, usando o comando GET “http://localhost:3000/get” e o resultado apresentado via console pode ser verificado na Figura 17.

RPC são chamadas remotas de procedimento, que são uma tecnologia de comunicação entre processos que permite a um programa de computador chamar um procedimento em outro espaço de endereçamento (geralmente em outro computador, conectado por uma rede).

O programador não se preocupa com detalhes de implementação dessa interação remota do ponto de vista do código, portanto a chamada se assemelha as de procedimentos locais.

RPC é uma tecnologia popular para a implementação do modelo cliente-servidor de computação distribuída. Uma chamada de procedimento remoto for iniciada pelo cliente enviando uma mensagem para um servidor remoto para executar um procedimento específico. Uma resposta é retornada ao cliente.

Se a aplicação tiver várias instâncias de serviços, o *ServiceBroker* fará o balanceamento de carga da solicitação entre as instâncias. O método de balanceamento automático de carga tem função de distribuir dinamicamente a carga da comunicação entre os microsserviços uniformemente.

Figura 17 - Chamada da Ação get via *broker.call* de um Microsserviço do Moleculer

```
const {ServiceBroker} = require("moleculer");
const = new ServiceBroker ({
    nodeID: "node-1"
    logger: true,
    logLevel: "info",
});

broker.createService({
    name: "posts",
    actions: {
        get(ctx){
        this.logger.info("Log message via service logger");
        }
    }
});

broker.start()
    .then(() => broker.call ("posts.get"))
    .then(() => broker.logger.info ("Log message via broker logger"))
```

Fonte: Adaptado Moleculer (2020)

Para chamar uma ação interna, é usado o método *broker.call*. Quando o método *broker.call* é chamado, o *ServiceBroker* procura o microsserviço em um nó que tem uma determinada ação e o chama. A função retorna um *Promise*. Em JavaScript *Promise* é um objeto usado para processamento assíncrono.

Um *Promise* (de "promessa") representa um valor que pode estar disponível agora ou no futuro. Isso permite a associação de métodos de tratamento para eventos da ação assíncrona num caso eventual de sucesso ou de falha. Isto permite que métodos assíncronos retornem valores como métodos síncronos: ao invés do valor final, o método assíncrono retorna uma promessa ao valor em algum momento no futuro.

Na Figura 17 é apresentada a função *action* do microsserviço, que neste caso é chamado de *get*. Para chamada interna dessa função, é preciso iniciar o microsserviço via comando *broker.start()* e depois chamar ação via comando *broker.call*.

3.5 Métodos

A propriedade *Methods* (métodos) é função privada do microsserviço que somente são executadas internamente. Para criação de métodos no microsserviço, os comandos devem ser colocados entre chaves dentro da função *methods*, como mostrado na Figura 18, e serem chamadas a partir das funções de ação *action*. As funções privadas, não podem ser chamadas via *broker.call*. Para executar chamadas internas dos manipuladores de ações e de eventos, deve-se realizar por meio da função *methods*.

Figura 18 - Exemplo da Propriedade *Methods* de um Microsserviço do Moleculer

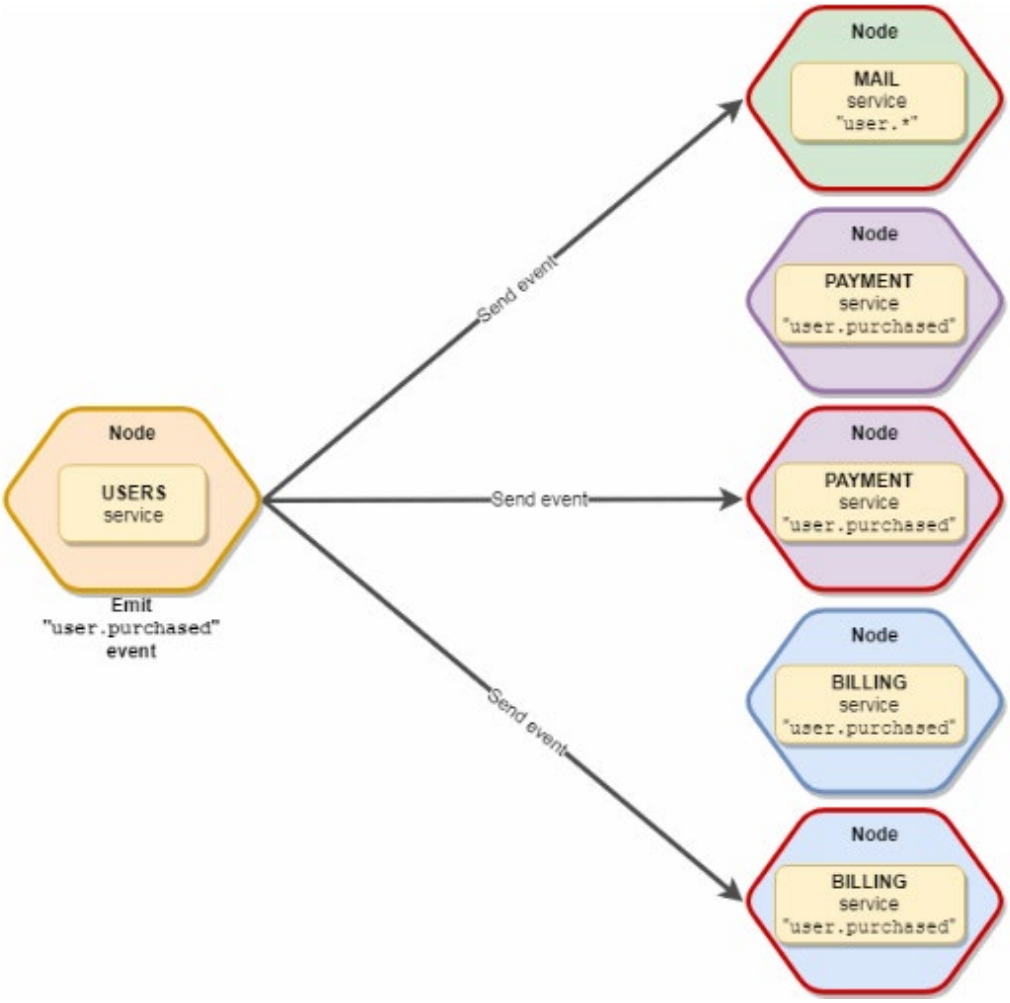
```
// mailer.service.js
module.exports = {
    name: "mailer",
    actions: {
        send(ctx) {
            // Call the `sendMail` method
            return this.sendMail(ctx.params.recipients, ctx.params.subject,
ctx.params.body);
        }
    },

    methods: {
        // Send an email to recipients
        sendMail(recipients, subject, body) {
            return new Promise((resolve, reject) => {
                ...
            });
        }
    }
};
```

Fonte: Adaptado Moleculer (2020)

3.6 Eventos

O *ServiceBroker* possui um barramento integrado para suportar uma arquitetura orientada a eventos. Quando ocorrer um evento, na Figura 19 pode ser enviado uma mensagem para os nós locais e remotos.

Figura 19 - Diagrama de Evento Balanceado para Microsserviço no Moleculer



Fonte: Moleculer (2020)

A propriedade *events* (eventos) consiste em produtores de eventos que geram um fluxo de dados, e consumidores dos eventos que os escutam. Os eventos são entregues após a execução da propriedade, para que os consumidores possam responder imediatamente conforme os eventos ocorram.

Os produtores são separados dos consumidores. Um produtor não sabe quais consumidores estão ouvindo. Os consumidores também são separados uns dos outros e cada consumidor vê todos os eventos. Isso difere do padrão de Consumidores Concorrentes, onde os consumidores removem mensagens de uma fila e uma mensagem é processada apenas uma vez.

As arquiteturas de microsserviços baseados em eventos usam o modelo de *publish/subscribe*. A infraestrutura de mensagens acompanha o controle de assinaturas. Quando um evento é publicado, ele envia o evento para cada assinante.

Depois que um evento é recebido, ele não pode ser reproduzido e não será exibido para assinantes novos. No *Moleculer* há dois tipos de eventos como podem ser verificados: *Balanced events; Broadcast event.*

No *Balanced events* (eventos balanceados), os microsserviços são organizados em grupos lógicos ouvintes dos eventos. Isso significa que apenas um microsserviço ouvinte é acionado em cada grupo, como pode observado na Figura 19.

O nome do *group* (grupo) vem do nome do microsserviço, mas pode ser definido na programação das propriedades do evento como pode ser visto na Figura 20.

Figura 20 – Nome do Grupo do Evento de um Microsserviço do Moleculer

```
module.exports = {
    name: "payment",
    events: {
        "order.created": {
        // Register handler to the "other" group instead of "payment" group.
        group: "other",
        handler(payload){
            // ...
        }
        }
    }
}
```

Fonte: Adaptado Moleculer (2020)

Para enviar um evento balanceado deve ser usado a função *broker.emit*. Nessa função o primeiro parâmetro é o nome do evento e o segundo parâmetro é o *payload* de acordo com a Figura 21.

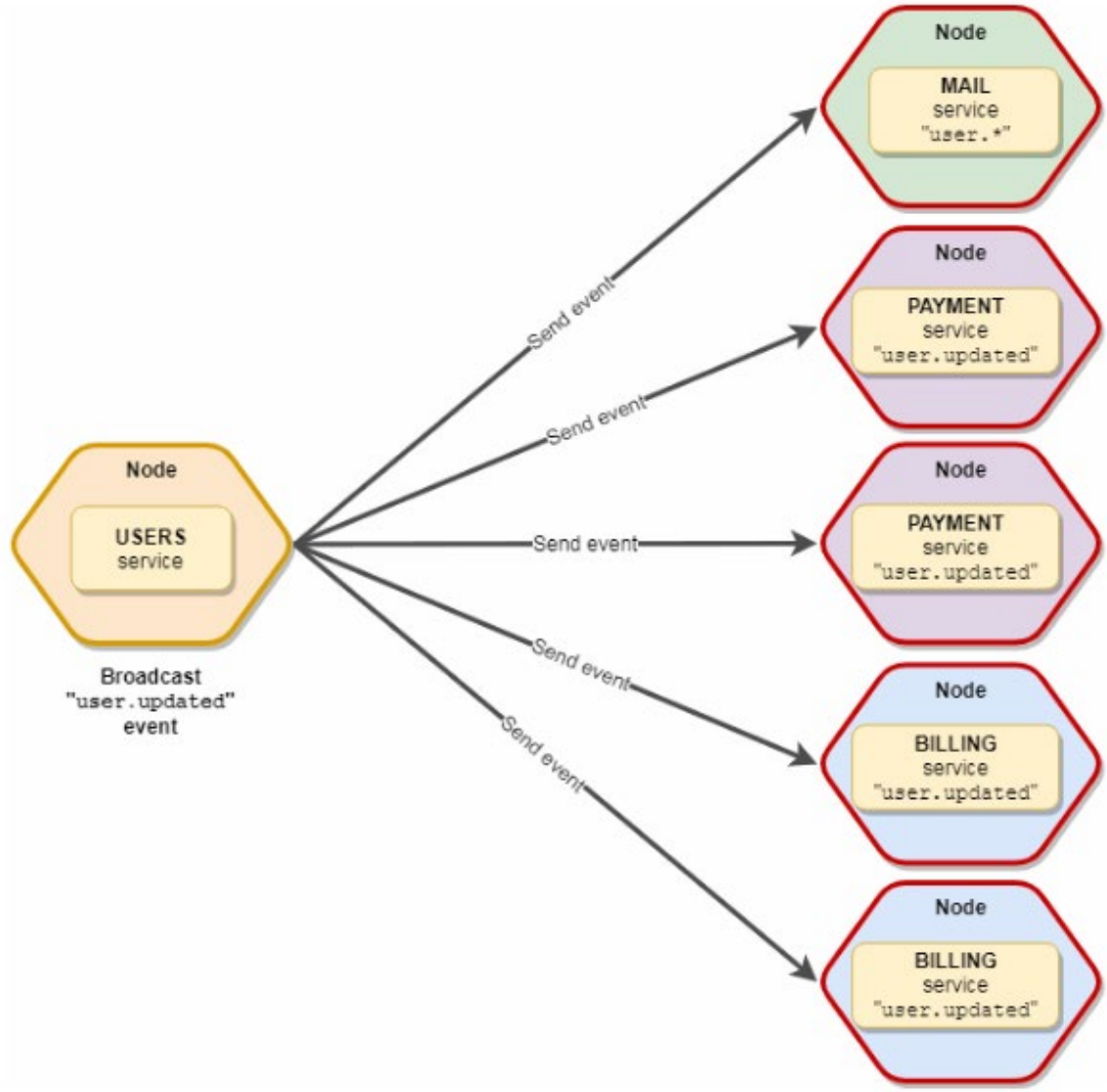
Figura 21 - Exemplo da Função *broker.emit* de um Microsserviço do Moleculer

```
// The 'user' will be serialized to transportation
broker.emit("user.created",user);
```

Fonte: Adaptado Moleculer (2020)

No *Broadcast event,* a transmissão dos eventos é enviada para todos os serviços locais e remotos disponíveis. Na Figura 23 é apresentado o método *broker.broadcast* para transmissão de eventos para todos os microsserviços. A transmissão dos eventos não é balanceada, sendo que todas as instâncias de serviço recebem a mensagem do evento ocorrido como pode ser vista na Figura 22.

Figura 22 - Diagrama de um Evento Broadcast de um Microsserviço no Moleculer



Fonte: Moleculer (2020)

Figura 23 - Exemplo do Método *broker.broadcast* de um Microsserviço do Moleculer

```
broker.broadcast("config.changed", config);
```

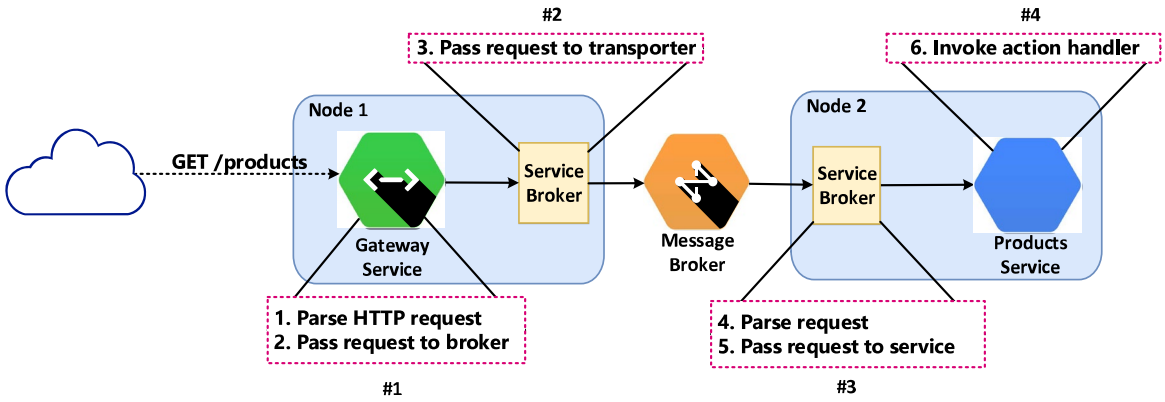
Fonte: Adaptado Moleculer (2020)

No *Moleculer*, todos os módulos principais possuem uma instância personalizada de um registrador de eventos relevantes à aplicação (*logs*). Também possui um registrador de console embutido que é o *logger* padrão (MOLECULER, 2020).

3.7 Exemplificação fluxo de solicitação do usuário

Para um maior entendimento e uma visão global resumida sobre o framework considera-se uma hipotética loja online que lista apenas seus produtos para um exemplo simples na Figura 24.

Figura 24 - Fluxo de solicitação do usuário



Fonte: Moleculer 2020

Do ponto de vista arquitetônico, a loja online pode ser vista como uma composição de 2 serviços independentes: o serviço de produtos e o serviço de gateway. O primeiro é responsável pelo armazenamento e gerenciamento dos produtos, enquanto o segundo simplesmente recebe as solicitações dos usuários e as encaminha para o serviço de produtos.

Para garantir que o sistema seja resiliente a falhas, executa-se os produtos e os serviços de gateway em nós dedicados (nó-1 e nó-2). A execução de serviços em nós dedicados significa que o módulo transportador é necessário para a comunicação entre serviços.

A maioria dos transportadores suportados pelo *Moleculer* dependem de um corretor de mensagens para comunicação entre serviços. No geral, a arquitetura interna da loja está representada na Figura 24.

Primeiro, a solicitação (GET / *products*) é recebida pelo servidor HTTP em execução no node-1. A solicitação de entrada é simplesmente passada do servidor HTTP para o serviço de gateway que faz todo o processamento e mapeamento.

A solicitação do usuário é mapeada em uma ação *listProducts* do serviço *products*. Em seguida, a solicitação é passada ao *broker*, que verifica se o serviço do produto é local ou remoto. Nesse caso, o serviço do produto é remoto, portanto, o corretor precisa usar o módulo transportador para entregar a solicitação.

O transportador simplesmente pega a solicitação e a envia pelo barramento de comunicação. Como ambos os nós (nó-1 e nó-2) estão conectados ao mesmo barramento de comunicação (intermediário de mensagem), a solicitação é entregue com sucesso ao nó-2. Após a recepção, o corretor do node-2 analisará a solicitação recebida e a encaminhará ao serviço de produtos.

Por fim, o serviço de produtos chama a ação *listProducts* e retorna a lista de todos os produtos disponíveis. A resposta é simplesmente encaminhada de volta ao usuário final.

4 DESENVOLVIMENTO DO TRABALHO

A proposta deste trabalho que foca no desenvolvimento de uma planta piloto industrial de controle de processos industriais, como controle de nível, pressão de linha, pressão de reservatório, e vazão no contexto da indústria 4.0 para desenvolvimento e testes operacionais da composição de microsserviços, diferentes tipos de transportes e mecanismos de segurança, usando uma arquitetura orientada a microsserviços que é apresentada nesta seção. Inicialmente são apresentadas de forma detalhada todas as informações sobre o desenvolvimento de hardware e software da planta piloto. Após isso, o desenvolvimento dos microsserviços e dos mecanismos de segurança implementadas neste trabalho são apresentados.

4.1 Planta-Piloto Industrial baseada em Serviços

A planta piloto industrial foi desenvolvida através de uma parceria entre a FAPESP, Unesp Sorocaba, SENAI de Lençóis Paulista e a empresa *Emerson Automation Solutions*. A proposta principal da planta piloto é o controle de processos das seguintes variáveis: nível, pressão, vazão e monitoramento da temperatura. Plantas industriais de automação utilizam o tradicional padrão ISA-95, ilustrado na Figura 3, sendo o grande diferencial da planta piloto industrial proposta neste trabalho o fato de sua arquitetura ser baseada em microsserviços.

Pode-se observar uma vista frontal da planta piloto na Figura 25, onde é possível ver todos os seus componentes como atuadores e sensores, além de seu PI&D (*Piping and Instrumentation Diagram*) na Figura 26 e sua respectiva legenda na Figura 27 para uma melhor visualização de suas malhas de controle.

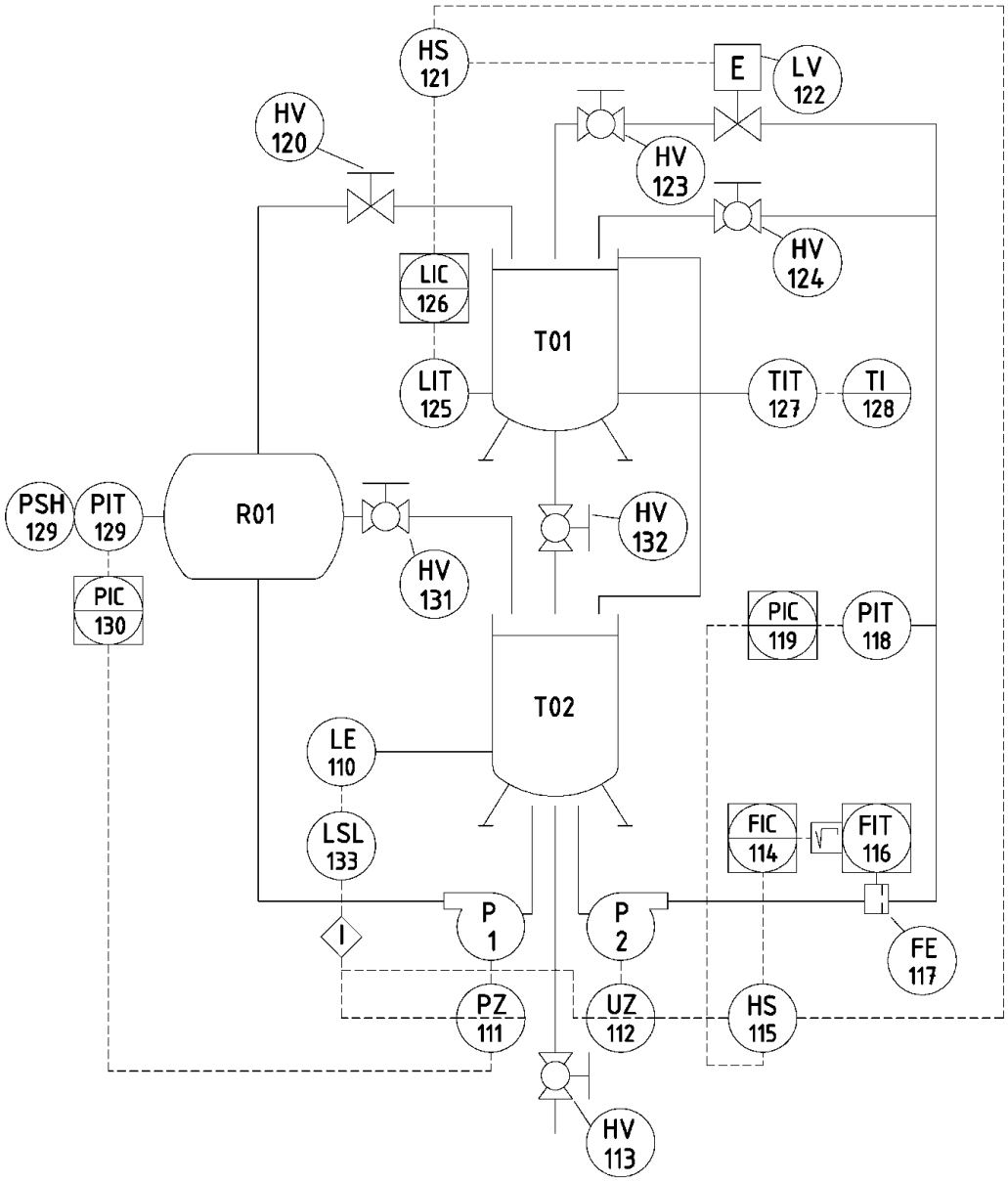
A parte física da planta piloto possui um tanque em inox principal (TQ02) de 83 litros na parte inferior da planta para armazenamento de água. O liquido pode ser bombeado através de tubos de inox de ½” para parte superior da planta em duas rotas distintas, tanto para o tanque em acrílico (TQ01) de 38 litros com a bomba (P2), quanto para o reservatório de pressão em inox (R01) de 15 litros com a bomba (P1).

Dois painéis elétricos são usados para organização, sendo o inferior para circuitos de potência e comando e o superior para alocação das interfaces de conexão de IO e placas de programáveis. O painel superior também possui um display LCD *touch* de 10” para supervisão de informações da planta.

Figura 25 - Planta piloto industrial

Fonte: Autor

Figura 26 - P&ID da planta piloto industrial.



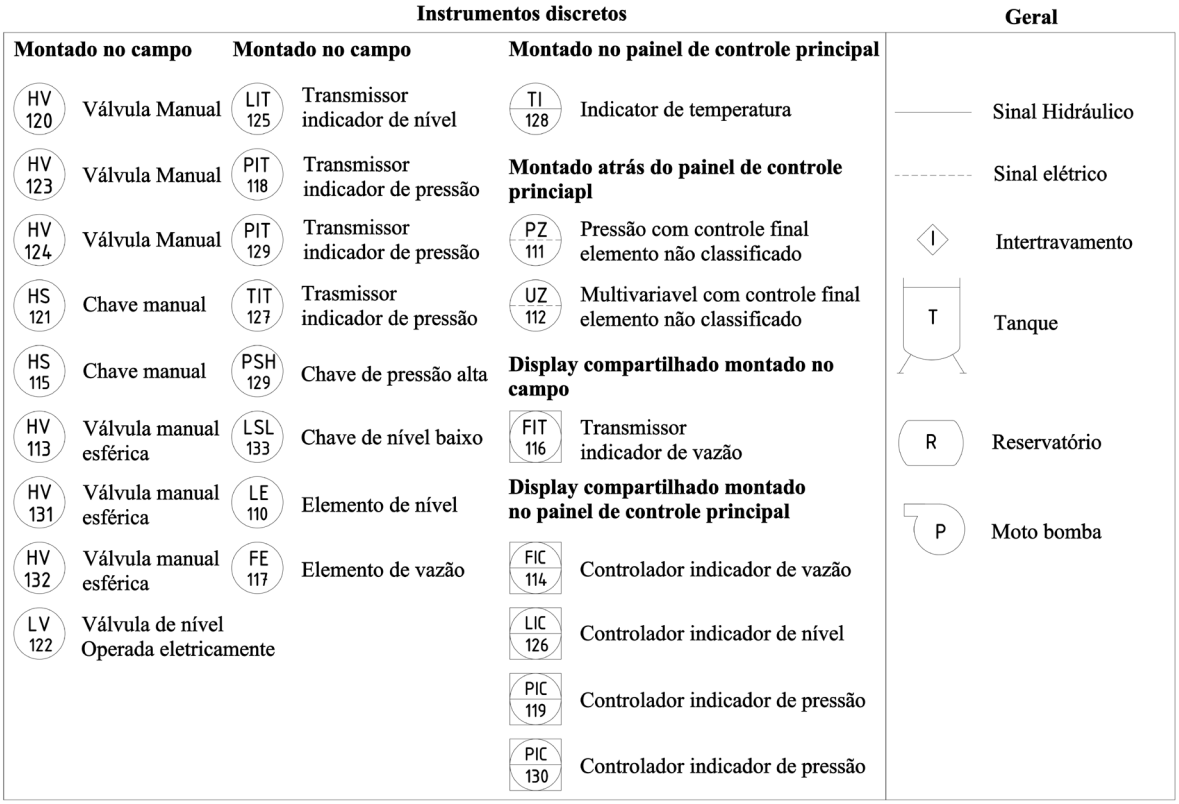
Fonte: Autor

As variáveis de processos disponibilizadas na planta e seus respectivos instrumentos de medição da Emerson são: nível do tanque (LIT125) e vazão (FIT116) com o transmissor de pressão coplanar de 0 a 5000 mmH_2O da ROSEMOUNT modelo 2051CD, pressão de linha (PIT118) e pressão do reservatório (PIT129) com o pressostato eletrônico de 0 a 2,5 bar da WIKA modelo PSD-4, e temperatura (TIT127). Para medição de vazão é usado uma placa de orifício junto ao transmissor de pressão.

Como variáveis de comando, temos duas bombas e uma válvula proporcional. As bombas de baixo ruído Dancor modelo Pratika CP-4R (P1 e P2) têm as mesmas características de potência de 0,5CV e uma vazão de 4,24 m³/h e são controladas por inversores de fre-

quência da WEG modelo CFW300. Para evitar que as bombas funcionem sem água no reservatório adicionou-se uma chave de nível (LSL110) da SITRON. A válvula proporcional motorizada (LV 122) de ½" é da BUSCHJOST modelo 8288200.9650.

Figura 27 - Legenda do P&ID da planta piloto industrial

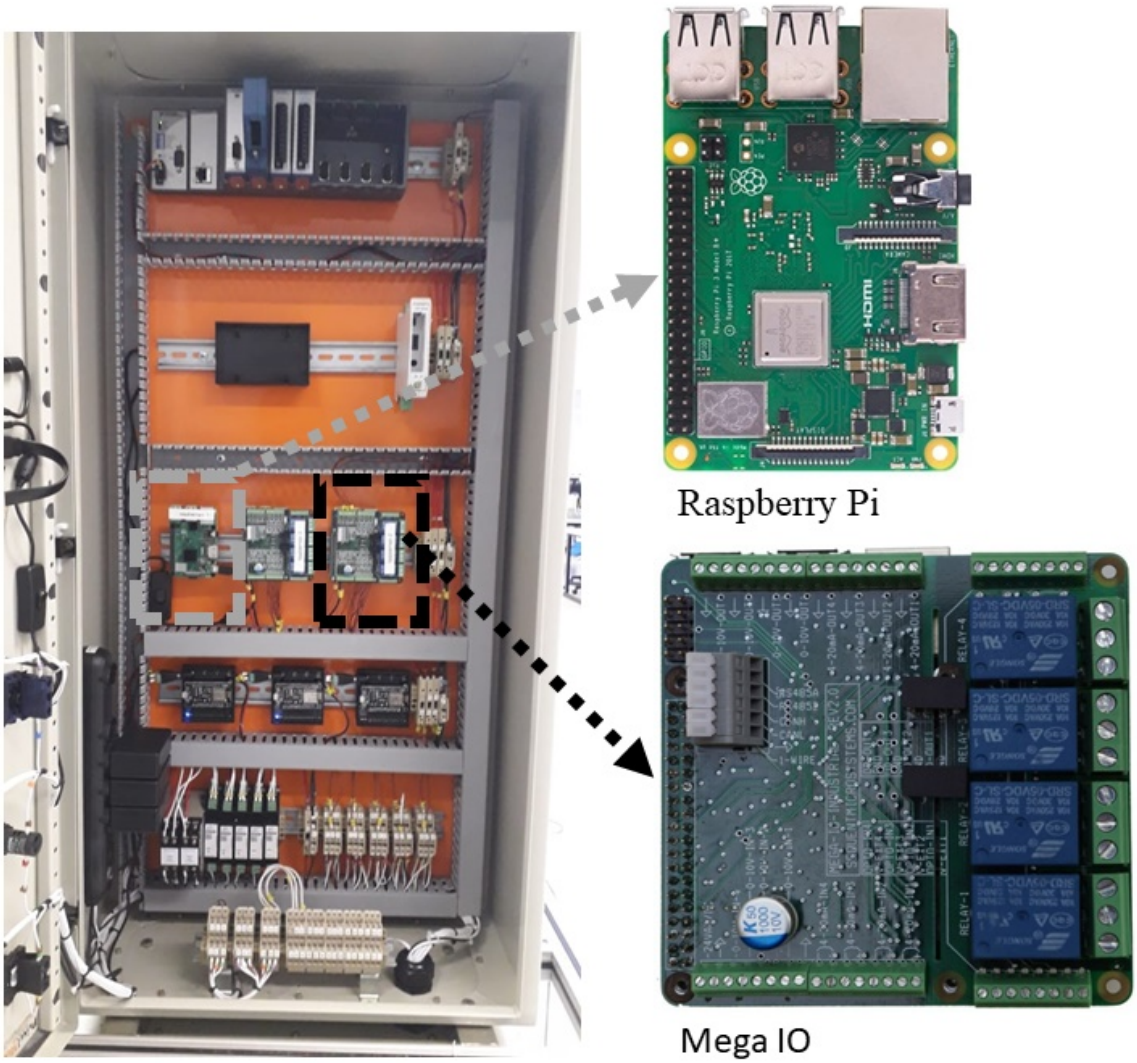


Fonte: Autor

4.2 Hardware

Os principais hardwares utilizados na planta são apresentados nesta seção. No painel de controle, utilizou-se a placa *Raspberry Pi* 3B+, que pode ser vista na Figura 28, para a parte lógica de programação e microsserviços, com a placa de expansão *MegaIO* industrial da *Sequent Microsystems* (MegaIO, 2020), que pode conectar-se como um *shield* na *Raspberry Pi.*

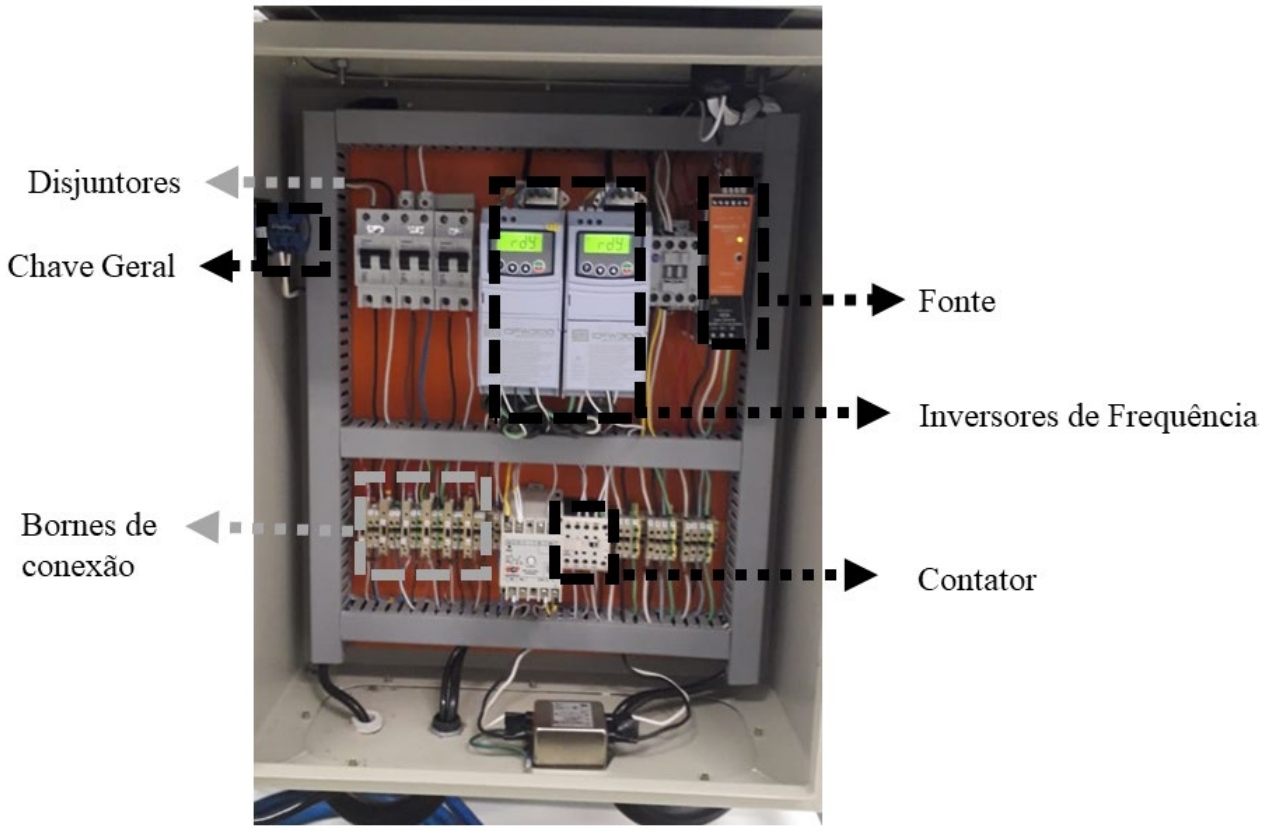
Figura 28 – Painel de controle com *Raspberry Pi* 3B + e MegaIO Industrial



Fonte: Autor

Através da placa *MegaIO*, é possível fazer a coleta de corrente analógica dos sensores de 4 a 20mA, bem como o acionamento das bombas gerando um sinal de 0 a 10V para os inversores de frequência, Figura 29, ou para a válvula proporcional. Este conjunto de placa *Raspberry Pi* com *MegaIO* foi utilizado para o desenvolvimento do Microsserviço de Aquisição de Dados (DAQ) criado neste trabalho.

Figura 29 - Painel de acionamento



Fonte: Autor

Em resumo, esse microsserviço realiza as leituras das entradas e atualiza as saídas da placa de expansão *MegaIO*, disponibilizando essa funcionalidade (execução dessas tarefas) como um serviço (interface padronizada) que pode ser requisitado numa aplicação.

Os esquemáticos de toda a parte elétrica de acionamento e controle não foram apresentados, porém uma visão geral de todo o funcionamento da planta bem como sua comunicação pode ser vista com um pouco mais de detalhes na subseção 4.4 Diagrama de conexão. A quantidade e descrição dos principais componentes utilizados na planta-piloto industrial podem ser vistos na Tabela 3.

Tabela 3 - Quantidade e descrição dos principais itens da planta-piloto industrial

Planta-piloto Industrial	
Quantidade.	Descrição
1	Fonte 24V
5	Fonte 5V
3	Disjuntor bipolar
8	Fusível
2	Botoeira
2	Indicador
4	Chave seletora
1	Relé de Nível
2	Contator
2	Inversor de Frequência
2	Motor elétrico trifásico 220V
1	Eletroválvula
6	Válvula manual
1	Sensor de nível
1	Sensor de vazão
2	Sensor de pressão
1	Sensor de temperatura
4	*Raspberry Pi* 3B+
2	*Megaio* Industrial
1	Display

Fonte: Autor

4.3 Softwares

Os sistemas operacionais utilizados foram Windows 10 no computador, além de ter o software LabVIEW para facilitar o controle, monitoramento e analisar os dados estatísticos. Sistema operacional Linux na *Raspberry Pi*, mais especificamente o sistema operacional oficial *Raspbian*, onde o *Raspberry Pi* ficará com a parte lógica de todos os serviços. Esses serviços são desenvolvidos com o *framework Moleculer*, A estrutura é de código aberto e é executada na plataforma Node.js em conjunto com seu gerenciador de pacotes oficial NPM (*Node Package Manager*), simplificando a criação de serviços eficientes, confiáveis e escaláveis.

4.3.1 Serviços

4.3.1.1 Transportador

O *framework Moleculer* disponibiliza uma série de mecanismos diferentes para comunicação entre os microsserviços que são conectáveis como: TCP (*Transmission Control*

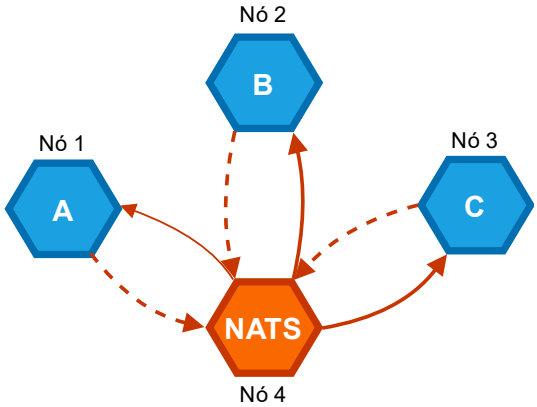
Protocol)/IP, NATS, MQTT (*Message Queuing Telemetry Transport*), AMQP (*Advanced Message Queuing Protocol*) etc. Utilizou-se no projeto o transportador NATS, que é o padrão do framework *Moleculer* e representa um sistema de mensagens de código aberto simples e de alto desempenho para arquiteturas nativas de aplicativos, mensagens de Internet das Coisas e microsserviços.

A configuração do tipo de transportador é feita no intermediário de serviço indicando o tipo de transportador desejado. Após ser configurado o tipo de transportador, a comunicação entre os microsserviços ocorre de forma transparente, sendo assim, não importa qual o mecanismo de comunicação é utilizado, não é necessário nenhum tipo de especificação ou configuração adicional para a comunicação entre os microsserviços.

O serviço transportador fica alocado em uma *Raspberry Pi* utilizando uma rede sem fio e é responsável pela troca de mensagens entre os microsserviços da arquitetura MOA, conforme é apresentado na Figura 30, onde cada nó representa um microsserviço A, B ou C, que para se comunicar com o outro utiliza o transportador.

As chamadas internas das ações utilizando qualquer tipo de transportador, são executadas pelo intermediador de serviço de cada microsserviço que dispara a solicitação para o outro serviço desejado que também possui um intermediador de serviço distinto e próprio, que então retorna a chamada solicitada.

Figura 30 - Microsserviço transportador



Fonte: Autor

4.3.1.2 Gateway

O serviço API *Gateway* é responsável pela interface padronizada de conexão dos serviços do *Moleculer* com aplicações externas, ou seja, na publicação dos serviços utilizando o protocolo REST. Possui diversos recursos entre eles o HTTP (*Hyper Text Transfer Protocol*

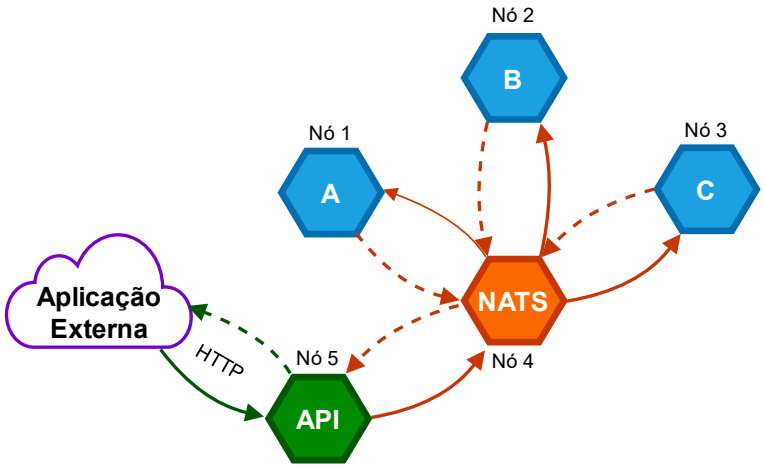
Secure) e HTTPS (*Hyper Text Transfer Protocol Secure*), arquivos estáticos, múltiplas rotas, analisadores de corpo JSON e suporte a autorização.

Para acessar um microsserviço A, B ou C, visto na Figura 31 basta utilizar um navegador ou qualquer aplicação externa que tenha disponível o padrão HTTP, e então digitar o endereço IP (*Internet Protocol*) de onde o microsserviço *Gateway* está alocado e escutando na porta 3000, e passar algum parâmetro caso a ação para aquele serviço necessite, obtendo um caminho como este: http://IP:3000/serviço/ação?parametro=valor.

Alguns exemplos de URLs(*Uniform Resource Locator*) disponíveis em microsserviços básicos de testes são descritos a seguir:

- Chamada da ação olá do microsserviço de comprimento: http://IP:3000/greeter/hello retornando olá.
- Chamada da ação soma com parâmetros do microsserviço matemática: http://IP:3000/math/add?a=2&b=3 retornando 5.

Figura 31 - Microsserviço API Gateway



Fonte: Autor

4.3.1.3 Controle

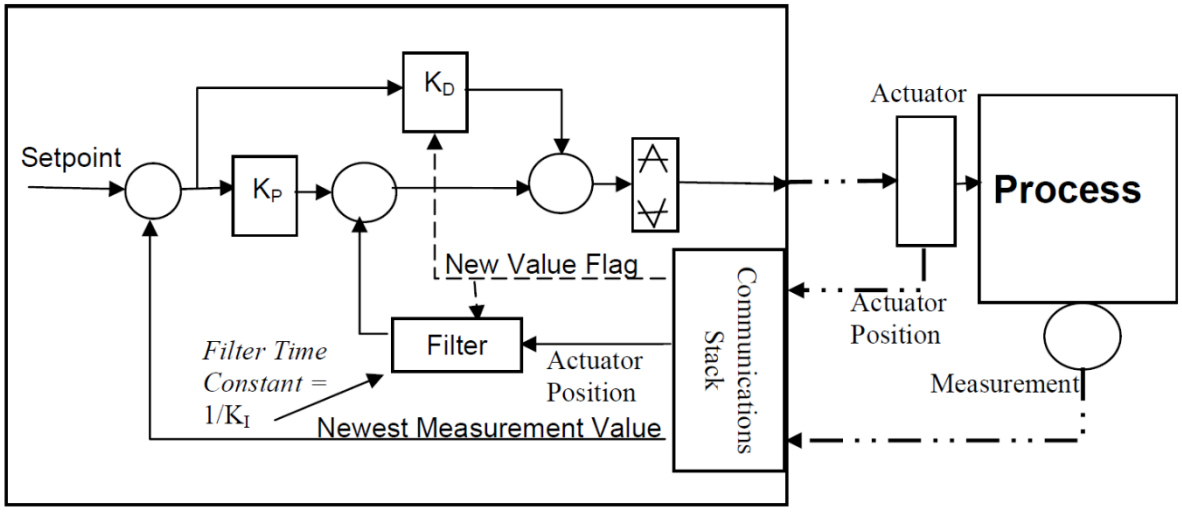
O microsserviço Controle PIDPlus (BIGHETI; FERNANDES; GODOY, 2019) foi usado e tem como funcionalidade principal o controle de processos, sendo responsável por calcular, a partir de um algoritmo de controle, um sinal de controle a ser aplicado no processo a ser controlado. Cada instância desse serviço é responsável pela implementação de uma malha fechada de controle. Este serviço pode ser hospedado ou rodar em três tipos de plataforma: computador, sistema embarcado ou nuvem. Este serviço necessita operar em conjunto com outros serviços (DAQ) para obtenção das variáveis do processo a ser controlado.

O microsserviço Controle PIDPlus utiliza como base o algoritmo de controle PID-Plus (BLEVINS; NIXON; WOJSZNIS, 2014; SONG et al., 2006). Este algoritmo foi criado para aplicações de controle em malha fechada usando redes de comunicação, mas neste trabalho foi modificado para trabalhar com o conceito de requisições e operar como um serviço. Apesar desse serviço usar especificamente o algoritmo de controle PIDPlus, um ponto importante é que seria possível criar outros microsserviços de controle que implementassem outros algoritmos e técnicas de controle diferentes, fornecendo flexibilidade e modularidade para o desenvolvimento e implementação.

O microsserviço Controle PIDPlus possui basicamente ações. As ações estão relacionadas às funcionalidades principais do serviço: Atualização de variáveis do controlador (*Setpoint* e Parâmetros de configuração do controlador como ganho proporcional - *K*, Tempos integrativo - *Ti* e Derivativo – *Td*, Constante *DeltaT*), Monitoramento de variáveis do controlador (*Setpoint*, Variável do processo e Variável Manipulada), Aquisição de entrada - variável do processo (*PV – Process Variable*) e Atualização de saída – variável manipulada (*MV – Manipulated Variable*).

O PIDPlus é um controlador PID (*Proportional Integral Derivative*) modificado para operar de forma adequada com grandes períodos de amostragem do sensor, atualizações não periódicas da variável e perda de mensagens transmitidas (BLEVINS; NIXON; WOJSZNIS, 2014). A implementação PIDPlus (SONG et al., 2006) é ilustrada na Figura 32, que mostra que o *reset* das partes integrais/derivativas do PID são determinadas com base no tempo entre recepções de mensagens (*New Value Flag*) pela pilha de comunicações.

Figura 32 - Estrutura do controlador PIDPlus



Fonte: SONG et al. (2006).

O PIDPlus mantém o sinal de controle no último nível calculado até que uma nova medida seja recebida. Uma observação é que sua sintonia é independente do período de amostragem, dependendo apenas das características físicas da planta. A realimentação (*Newest Measurement Value*) e o filtro de 1ª ordem (*Filter*) são modificados para criar a contribuição de reposição com o seguinte comportamento:

- Manter a última saída do filtro calculado (FN-1), até uma nova medição ser informada (New Value Flag);
- Quando uma nova medição é recebida (New Value Flag), utilize a nova saída do filtro como contribuição da realimentação (FN).

A principal diferença do PID e PIDPlus está na parte integrativa que é substituída por um filtro de 1ªordem. Para contabilizar a resposta do processo, a saída do filtro é calculada da seguinte maneira quando uma nova medição é recebida:

$$F_N = F_{N-1} + (O_{N-1} - F_{N-1}) * \left(1 - e^{\frac{-\Delta T}{T_{reset}}}\right) \quad (1)$$

onde: F_N = nova saída do filtro, F_{N-1} = saída do filtro na última execução, O_{N-1} = saída do controlador na última execução, ΔT = tempo decorrido desde que o último valor medido foi recebido, T_{reset} = constante de tempo da planta somado ao tempo morto.

A parte derivativa do algoritmo modificado (limitação de taxa não aplicada) é determinada pela seguinte equação (2):

$$O_D = K_D * \frac{e_N - e_{N-1}}{\Delta T} \quad (2)$$

onde: O_D = termo derivativo do controlador, K_D = ganho derivativo, e_N= erro atual, e_{N-1} = último erro, ΔT = tempo decorrido desde que o último valor medido foi recebido.

Considere a contribuição da parte derivativa quando as entradas forem perdidas por vários períodos. Quando a comunicação é restabelecida, e_N - e_{N-1} na equação (2) seria o mesmo para os algoritmos originais e modificados. No entanto, para o algoritmo PID padrão, o divisor na parte derivada seria o período, enquanto que no novo algoritmo é o tempo decorrido entre duas medições recebidas com sucesso, sendo assim o algoritmo modificado produziria uma ação derivada menor que o algoritmo PID padrão (SONG et al., 2006).

Na implementação do PIDPlus, o cálculo de reposição compensa automaticamente a alteração da medição e taxa de atualização da medição. Os cálculos do termo derivativo para um novo valor de medição não estão disponíveis a cada execução do PID. Assim, não há necessidade de modificar a sincronização para o controle sem fio, ou seja, o ajuste é baseado estritamente no ganho e dinâmica do processo.

Alterou-se a saída desse serviço, que é a variável manipulada para 0 a 100%, com o intuito de padronização em conjunto com o serviço de aquisição de dados.

4.3.1.4 Aquisição de dados

O Microsserviço de Aquisição de dados (DAQ) é responsável pela aquisição de dados de variáveis usando módulos de hardware alocados no processo. A *Raspberry Pi* onde esse serviço está alocado, se comunica com a placa *Meagio* Industrial através do protocolo I2C (*Inter Integrated Circuit*).

O microsserviço DAQ possui uma ação de leitura de entradas de corrente de 4 a 20mA, *riin,* e outra ação *ruin* leitura das entradas de tensão de 0 a 10V, utilizada para a aquisição dos valores medidos pelos sensores da planta industrial: sensor de pressão do tanque PIT129, sensor de pressão do tubo PIT118, sensor de temperatura da água TIT127, sensor de nível do reservatório LIT125 e sensor de fluxo do tubo FIT116.

O mesmo serviço também tem outra ação para atualizar as saídas, *wuout*, com valores de 0 a 10V, que é usada para ajustar os valores de comando dos inversores de frequência das bombas P1 ou P2 ou da válvula proporcional.

4.3.1.5 Base de dados e Monitoramento

O Microsserviço de Banco de Dados é responsável por armazenar os dados das variáveis de interesse da planta piloto, como sinais dos sensores e comando dos atuadores. Este serviço executa periodicamente uma rotina realiza requisições ao serviço DAQ para obtenção dos dados das variáveis e escrita num banco de dados da planta piloto. Este serviço e o banco de dados criado rodam numa *Raspberry Pi*.

Utilizou-se um banco de dados de séries temporais de código aberto desenvolvido pela *InfluxData*®, *InfluxDB,* que tem a estrutura de dados composta por medições, séries e pontos. Todos os pontos estão relacionados com uma data e hora. Essa base de dados permite que as consultas possam ser em séries, agrupamento de valores a partir de valores-chaves

também denominadas de etiquetas, e o agrupamento de séries pelo identificador de sequência que resulta em uma medição. O esquema de criação de um banco de dados pelo serviço pode ser visto na Figura 33.

Contendo o local onde será armazenado, seu nome e o tipo de variável que será armazenada. No campo, as denominações PIT129, PIT118, LIT125 e FIT116, referem-se aos sensores que medem as variáveis do reservatório, pressão da tubulação, nível e vazão, respectivamente. Verifica-se a existência do banco de dados com seu respectivo nome para evitar duplicidade, e caso não exista o banco de dados com o nome criado no esquema será criado um.

Figura 33 - Esquema de criação de um banco de dados

```
// influxDB.service.js

// Schema database
const influx = new Influx.InfluxDB({
  host: 'localhost',
  database: 'daq_response',
  schema: [{
    measurement: 'inputs',
    fields: {
        PIT129: Influx.FieldType.INTEGER,
        PIT118: Influx.FieldType.INTEGER,
        LIT125: Influx.FieldType.INTEGER,
        FIT116: Influx.FieldType.INTEGER,
    },
    tags: ['variable']
  }]
})

// Create database
influx.getDatabaseNames()
  .then(names => {
    if (!names.includes('daq_response'))
    {return influx.createDatabase('daq_response');}
  })
  .catch(err => {console.error('Error creating Influx database!');})
```

Fonte: Autor

Os dados das variáveis dos sensores são requisitados através de requisição feita ao serviço de aquisição de dados via protocolo HTTP na seguinte rota, http://192.168.1.201:3000/api/daq/riin, que pode ser vista na Figura 34. Caso o serviço de aquisição de dados não esteja disponível, nenhum dado é armazenado no banco, caso contrário todos dados serão escritos no banco de dados com medições denominada entradas, segundo a sua etiqueta que identifica o nome da variável medida. Neste caso da pressão do reserva-

tório coletada do sensor PIT129, e o campo que armazena o valor bruto verificado da pressão, onde esse processo se repetirá a cada 500ms. O armazenamento dos dados do banco de dados ficará, primordialmente, em um único embarcado, *Raspberry Pi,* e em trabalhos futuros poderá ser alocado em Nuvem.

Uma vez com os dados das variáveis armazenados, cada qual sendo um ponto carimbado com hora e data, sendo medições estruturadas em séries com suas devidas etiquetas que identificam sua origem, sensores, a base de dados poderá ser consultada utilizando-se uma interface de interação com o usuário que seja capaz de mostrar esses dados através de tabelas e/ou gráficos visualmente compreensivos.

Figura 34 - Escrita das variáveis dos sensores da planta-piloto no banco de dados

```
// influxDB.service.js

const request = require('request');
const Influx = require('influx');

// Path to request variables from plant sensors
const url = 'http://192.168.1.201:3000/api/daq/riin'

// Writes to the database
const wdb = () => {
  request({ url: url, json: true}, (error, response) => {
    if(error){console.log('Unable to connect to daq service!')}
    else{
      influx.writeMeasurement('inputs',[{
          tags: {variable: 'reservoir'},
          fields: {PIT129: Number(response.body.riin[0])}
      ])
    }
  }
}      ...
```

Fonte: Autor

Para exibição dos dados armazenados no banco de dados utilizou-se na planta industrial o serviço de monitoramento e visualização *Grafana* (GRAFANA, 2020). *Grafana* é uma solução de análise e monitoramento de código aberto com plugins de acesso à os tipos bancos de dados, simplificando os procedimentos de escrita/leitura no banco de dados para criação de interfaces de supervisão.

Para supervisão da planta piloto industrial desenvolvida, algumas interfaces de visualização ou dashboards foram criadas. A Figura 35 apresenta um dashboard criado para monitorar os valores das quatro variáveis de processo controláveis da planta piloto (pressão de linha, pressão do reservatório, vazão na tubulação e nível do tanque).

Figura 35 - Dashboard *Grafana* para Supervisão das Variáveis de Processo

Fonte: Autor

4.3.1.6 Rastreador

O *Moleculer* possui um serviço de rastreamento interno que coleta informações de rastreamento dentro de um serviço chamado *console-tracer*. As métricas são registradas como um *middleware*, ou seja, envolvem chamadas de ação. Portanto, o *middleware* de métricas é chamado logo antes da execução de uma ação e depois que a resposta é recebida de volta, fazendo a subtração e tendo o tempo total de execução daquela ação.

Para que esse modulo funcione, habilitou-se o parametro *metrics*, Figura 36, em todos os intermediarios de serviço de cada serviço, apenas no modo de desenvolvimento, para analisar o desempenho dos mesmos.

Figura 36 - Intermediador de serviços com o parâmetro *metrics* habilitado

```
let broker = new ServiceBroker({
    metrics: true,
});
```

Fonte: Autor

O codigo fonte deste serviço, *index.js,* possui um objeto chamado *requests* o qual armazena todas as metricas dos serviços e depois exibe-as no console, um exemplo ilustrativo pode ser visto na Figura 37, que mostra o tempo de execução de cada ação, como *votes.count* que demorou 12ms para ser concluida. Estendeu-se esse codigo, adicionando-se uma nova

função para gravar todas essas mestricas em um arquivo JSON que será analisado estaticamente pelo *software LabVIEW* no capitulo de resultados deste trabalho.

Figura 37 - Exemplo do serviço de rastreamento

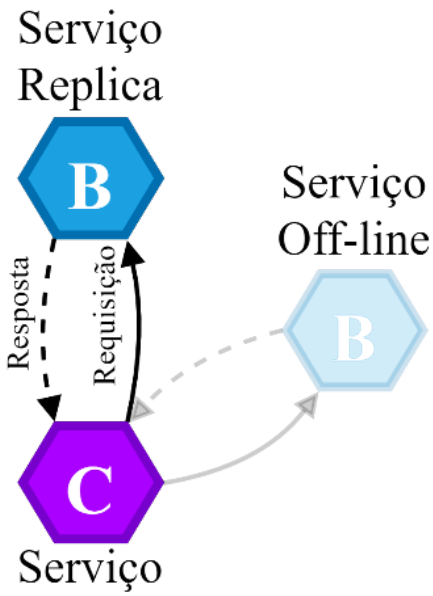
```
INFO  -   Call 'posts.find' action
INFO  -
INFO  -   ID: 98cc6c53-f797-47aa-b99a-f2f8bab56172                Depth: 4 Total: 11
INFO  -
INFO  -   api-gw.rest                  17ms [                                    ]
INFO  -     posts.find                 15ms [..                                ...]
INFO  -       users.get                 5ms [..                                   ]
INFO  -         friends.count *         2ms [.........                          ]
INFO  -       votes.count               13ms [....                              .....]
INFO  -       users.get                  5ms [....                                 ]
INFO  -         friends.count *          1ms [..........                         ]
INFO  -       votes.count               12ms [......                            .....]
INFO  -       users.get                  4ms [......                               ]
INFO  -         friends.count *          1ms [..........                         ]
INFO  -       votes.count               12ms [......                            .....]
INFO  -
INFO  - <= 200 GET /posts/find?limit=5 [+17.152 ms]
```

Fonte: Moleculer (2020)

4.3.2 Replicação de serviços

O *Moleculer* possui várias estratégias de balanceamento de carga integradas. Se os serviços tiverem várias instâncias em execução, o registro de serviço usará essas estratégias para selecionar um nó entre todos os nós disponíveis, ilustrado na Figura 38.

Figura 38 - Replica de serviço



Fonte: Autor

Duplicamos o serviço DAQ para redundância de aquisição de dados. Se um dos serviços do DAQ ficar offline, o balanceador de carga chamará o outro, garantindo que o sistema seja resiliente a falhas. Esse exemplo demonstra uma vantagem proporcionada pelo uso da arquitetura orientada a serviços para implementação de redundância em aplicações industriais de automação e controle.

4.3.3 Gerenciador de processos

Utilizou-se o *process manager* (PM2, 2020) para administrar todos os serviços disponíveis no desenvolvimento do projeto. O PM2 gerencia os processos de produção para os aplicativos *Node.js*, balanceia as cargas internas, permite manter os aplicativos sempre ativos, recarregá-los sem tempo de inatividade e facilita tarefas comuns de administração do sistema. Oferece uma CLI (*Command Line Interface*) simples e intuitiva, instalável através do NPM.

Na Figura 39 pode-se ver um trecho de código que resume a forma de como utilizou-se o PM2 para os serviços da aplicação do trabalho proposto.

Figura 39 - Comandos do PM2.

```
Raspberry Pi

# Install
$ npm install pm2@latest -g
# Start an app
$ pm2 start app.js --watch
[PM2] Starting /home/pi/app.js in fork_mode (1 instance)
[PM2] Done.
┌──────────┬──┬─────────┬────┬───┬──────┬───────┬──────┬──┬────────┬────┬─────────┐
│App name  │id│version  │mode│pid│status│restart│uptime│cpu│  mem   │user│watching│
├──────────┼──┼─────────┼────┼───┼──────┼───────┼──────┼──┼────────┼────┼─────────┤
│ app      │0 │  N/A    │fork│738│online│   0   │  0s  │ 0%│21.8 MB│ pi │ enable  │
└──────────┴──┴─────────┴────┴───┴──────┴───────┴──────┴──┴────────┴────┴─────────┘
# Managing process
$ pm2 restart app.js
$ pm2 reload app.js
$ pm2 stop app.js
$ pm2 delete app.js
# List managed applications
$ pm2 ls
# Startup Script Generator
$ pm2 startup systemd
[PM2] Init System found: systemd
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u pi --hp
/home/pi
$ sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd -u pi --hp
/home/p
$ pm2 save
```

Fonte: Autor.

O comando *$ pm2 start app.js* (simboliza um microsserviço) é um jeito simples de iniciar, *daemonize* (rodar no plano de fundo) e monitorar sua aplicação, tendo ainda como opção de parâmetro o *–watch*, que observa quando algum arquivo foi modificado e reinicia o aplicativo automaticamente, sendo útil na etapa de desenvolvimento.

Pode-se reiniciar, recarregar, parar ou deletar uma aplicação, bem como ver o status de cada uma com *$ pm2 ls*. Despois que todos os serviços já foram carregados pode-se gerar um script, *$ pm2 startup systemd*, que é carregado assim que o sistema operacional é iniciado ou reiniciado, subindo todos os serviços salvos automaticamente.

Garante-se que sempre que o sistema sofrer uma reinicialização, por falta de energia por exemplo, todos os serviços serão carregados automaticamente quando ela for reestabelecida.

4.4 Diagrama de Conexão

Após a apresentação da planta e seus componentes de hardware e software detalhado nos itens anteriores, um resumo na Figura 40 com a interação entre hardware e software é apresentado. Os softwares instalados em cada dispositivo seguem a seguinte distribuição:

- Computador: OS Windows 10, LabVIEW, Framework *Moleculer*, PM2 e Node.js;
- Raspberry Pi: OS Raspbian, Grafana, Influxdb, Node.js, Framework Moleculer, PM2, *Transporter* (NATS, TCP ou MQTT).

Alocação dos serviços: os microsserviços foram implantados de forma independente, cada um em um nó do *Moleculer*. Optou-se por essa distribuição para balanceamento das cargas de comunicação (requisição/resposta) e também obtenção de uma arquitetura totalmente distribuída e modular para composição das aplicações. Logo, a distribuição ficou da seguinte maneira:

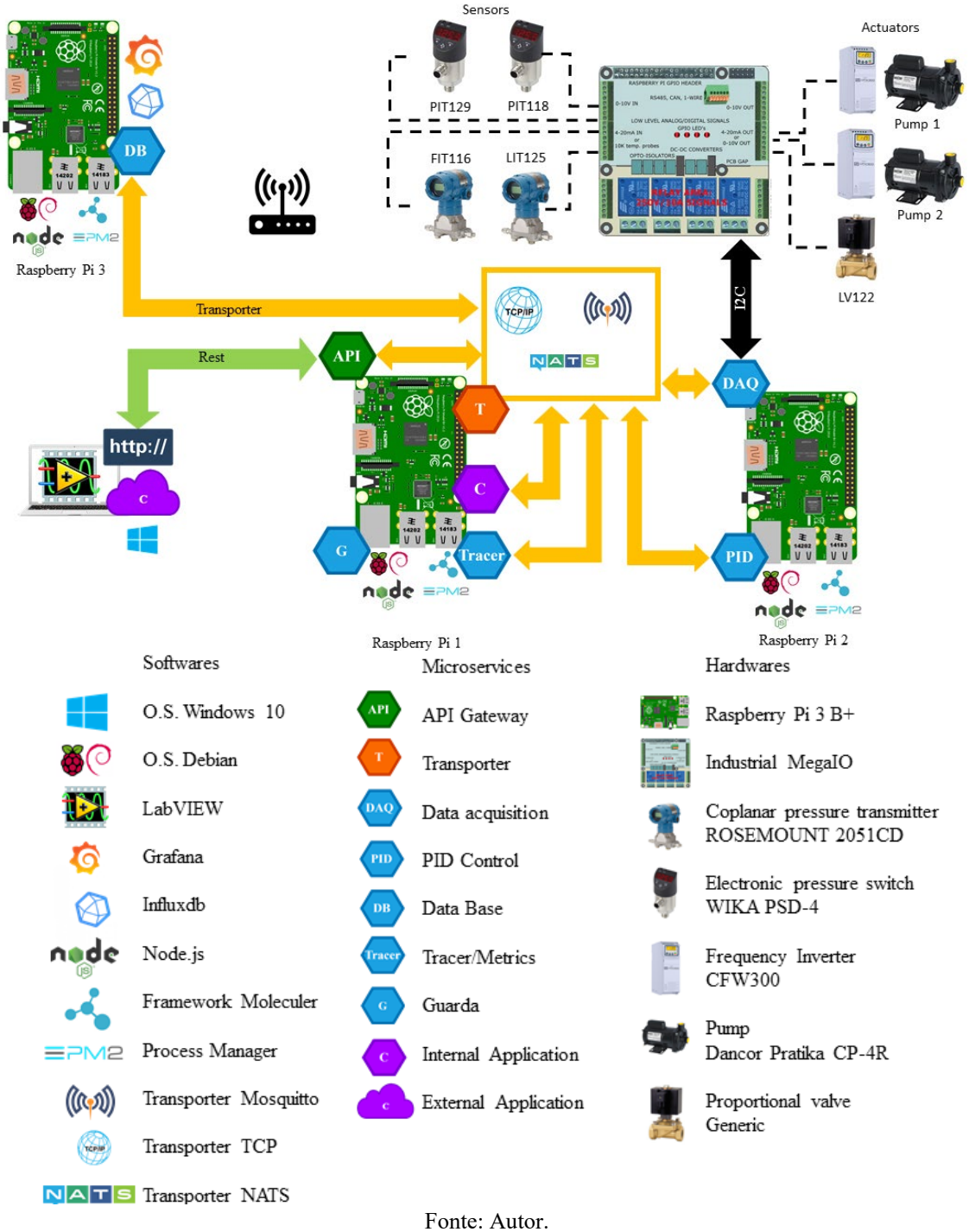
- Raspberry Pi 1: Serviços API Gateway, *Transporter* (TCP, NATS ou MQTT), Rastreador de métricas, Guarda e Aplicação Interna;
- Raspberry Pi 2: Serviços DAQ e PID (replicados, sendo um serviço para cada malha de controle);
- Raspberry Pi 3: Serviço Banco de Dados e Monitoramento via Grafana
- Computador 1: Aplicação externa no LabVIEW.

As principais formas de comunicações na arquitetura são:

- Comunicação REST (Wi-Fi ou Ethernet cabeada), representada pela seta na cor verde, através de requisições HTTP(S) para o acesso via API;

- Comunicação via transporter (Wi-Fi ou Ethernet cabeada), representado pela seta laranja, podendo utilizar o protocolo TCP, broker NATS ou broker MQTT para fazer comunicação entre os serviços;
- Comunicação I2C para integração entre a Raspberry Pi e a placa de conexão MegaIO de dispositivos de entrada (sensores) e saída (atuadores). A placa MegaIO utiliza conexão com os sensores e atuadores via padrão industrial de 4 a 20 mA ou 0 a 10 VDC.

Figura 40 - Esquema geral do hardware, com seus respectivos softwares, meios de comunicação e serviços.



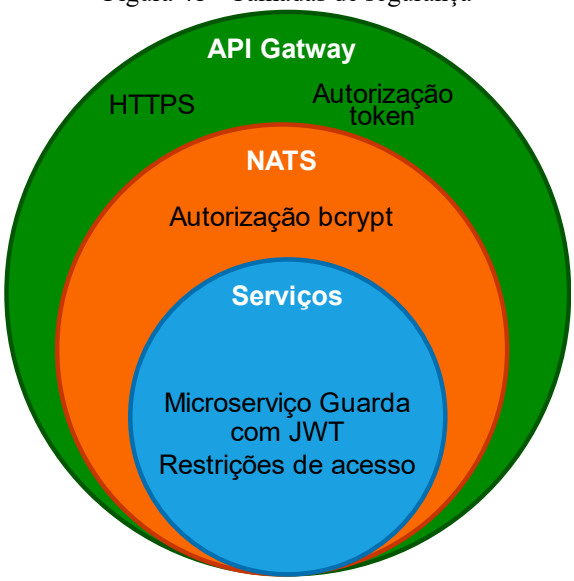
Fonte: Autor.

4.5 Segurança

Buscando adequar a aplicação da arquitetura orientada a microsserviços em relação às necessidades de segurança de aplicações industriais, neste trabalho foram desenvolvidos diferentes mecanismos de segurança. Esses mecanismos proporcionam diferentes funcionalidades de segurança como controle de acesso aos microsserviços, comunicação segura com encriptação e autenticação de usuário, controle da comunicação entre serviços e nas requisições via API Gateway.

Na Figura 41 tem-se o resumo dos mecanismos se segurança desenvolvidos neste trabalho, onde a segurança do acesso aos serviços por aplicações externas é garantida com o uso do HTTPS e token incorporadas na URls. Já o acesso dos serviços por aplicações internas que se conectam no transportador de mensagem também só é possível com autorização criptografada. E por último, o serviço de guarda garante que apenas serviços que tenham relação tenham acesso, caso contrário são barrados. Uma descrição desses mecanismos e sua implementação são apresentados nas subseções a seguir.

Figura 41 - Camadas de segurança



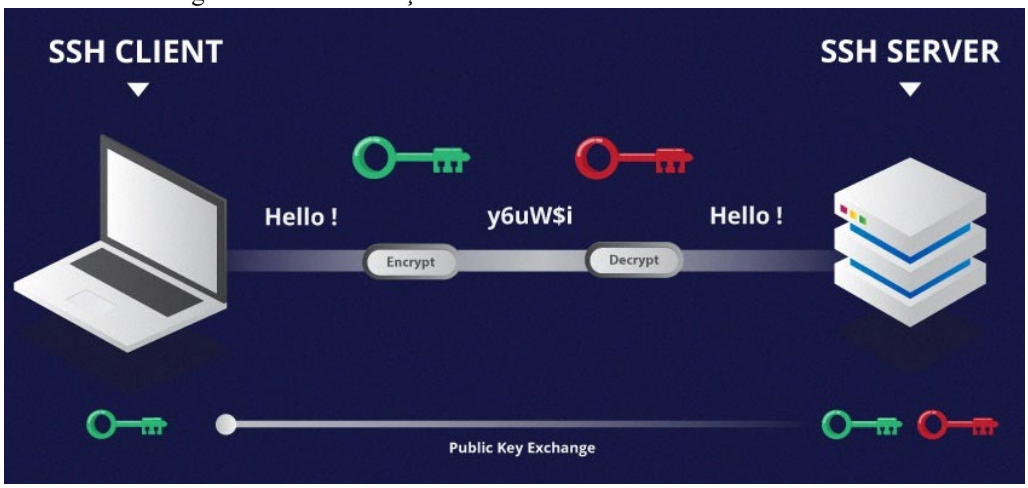
Fonte: Autor

4.5.1 Acesso à Raspberry Pi com Cliente SSH

Secure Shell (SSH) é um protocolo de administração remota que permite aos usuários controlar e modificar seus servidores pela Internet de maneira segura, utilizando a criptografia de mensagens. A criptografia assimétrica usa duas chaves separadas para criptografia e

descriptografia, como pode ser visto na Figura 42, essas duas chaves são conhecidas como chave-pública e chave-privada que garantem a integridade das informações.

Figura 42 - Comunicação entre cliente e servidor utilizando SSH



Fonte: Hostinger, 2020.

Essa funcionalidade de segurança permitirá que apenas quem possui tal chave pública tenha acesso aos servidores (Raspberry Pi) para então ler/escrever o código dos microsserviços, evitando que alguém conectado à rede consiga excluir algum código por engano por exemplo.

Para acessar ou editar os serviços na *Raspberry Pi* empregou-se o software (PuTTY, 2020), cliente *SSH* ou *Telnet*, desenvolvido para a plataforma *Windows*, software de código aberto que está disponível com seu código-fonte e é desenvolvido e suportado por um grupo de voluntários.

Para estabelecer uma comunicação *SSH*, utilizou-se o software *Puttygen* para gerar um par de chaves *RSA* criptografadas, *rsa-key-public* e *rsa-key-private*, onde a chave pública foi transferida para o arquivo *authorized_keys* criado em cada *Raspberry Pi* como mostrado na Figura 43, onde criou-se uma pasta com o nome *ssh,* e dentro desse diretório criou-se o arquivo *authorized_keys,* e então foi copiado e colado a chave que foi gerada no *Puttygen*.

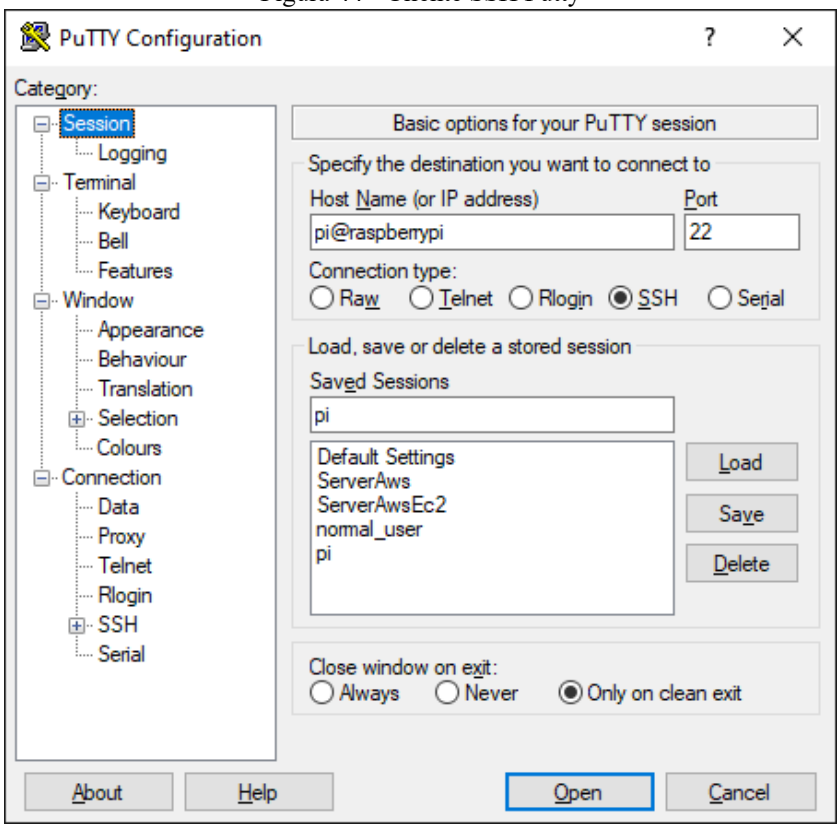
Com o software *Putty*, adicionou-se a chave privada nas configurações de autenticação, e acessou-se a *Raspberry Pi* desejada com o nome de usuário e seu respectivo endereço de IP ou nome, Figura 44, e a autenticação foi feita comparando-se a chave pública com a chave privada, estabeleceu-se assim uma conexão segura de comunicação entre o computador e a Raspberry na porta 22 na Figura 45, ou seja, entre desenvolvedor e servidor.

Figura 43 - Chave criptografada rsa-key-public.

```
Raspberry Pi

# Create the directory
$ mkdir ~/.ssh
$ chmod 700 ~/.ssh
# Create file
$ touch ~/.ssh/authorized_keys
$ chmod 600 authorized_keys
$ cd ~/.ssh/nano authorized_keys
# ssh-rsa
AAAAB3NzaC1yc2EAAAABJQAAAQEAz+APNidkcK5AsHNeKh5jUtmmC73re5rnlHV75oEg2qZNtj3WBRw5CDc2f5+IqIW
0QD0qIFFt8N4RGB5cBdZnC0RmS81HkvCtPugTM9bUfQlU2KLiqWtfPCMwM0cvykvXC5rKB3lXcvpq9negrBmhPU1buy
EdCjh8HAAjtPvN/BVcjTGUQXReXqmbqjg60w0t+gTKP68x1jmVXE9ifREDJMFKBHCfXBNOVIEveD2DGocN58UMVk2fY
qVAXAKSJbDBNfKaWpRsL/Va5xK8h2HyIdPQ+CbBDzRlgZFYbuwCpADGGnT1R2ZNV4EX5GPUHVmAPkq3tfOncAgHxXxG
VqwFrQ== rsa-key-1
```

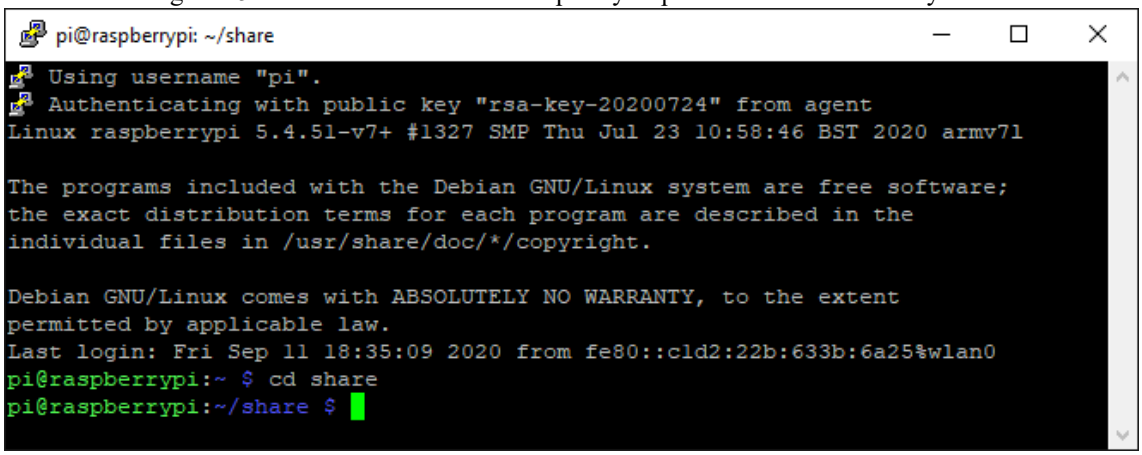
Fonte: Autor.

Figura 44 - Cliente SSH Putty



Fonte: Autor.

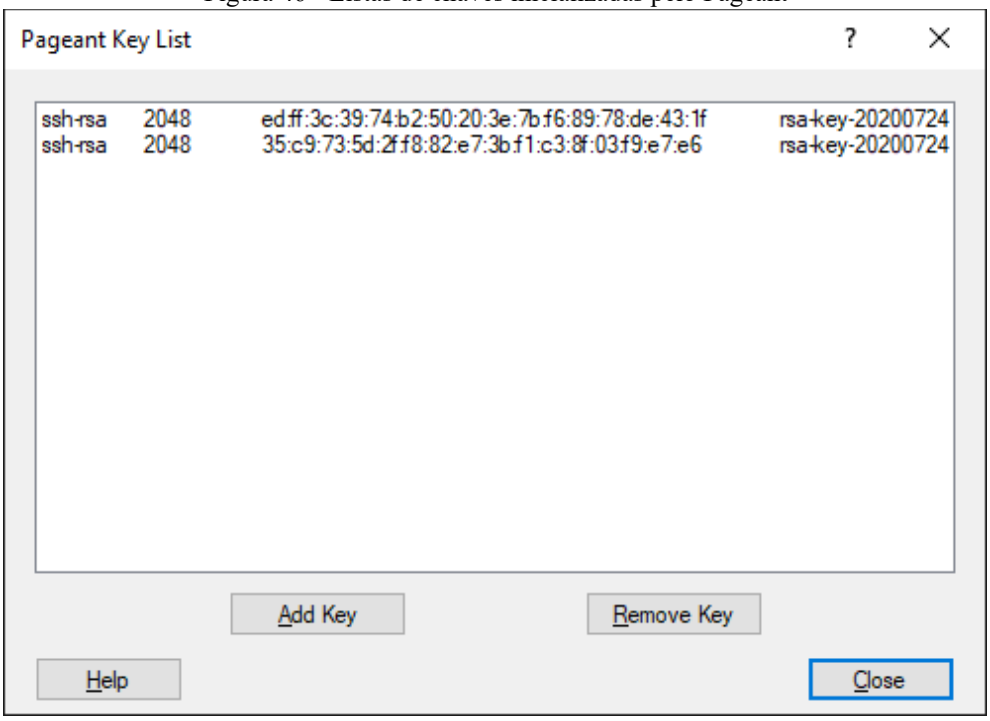
Figura 45 - Acesso ao terminal da Raspberry Pi pelo PC com cliente Putty SSH



Fonte: Autor.

Opcionalmente, pode-se configurar um agente para inicialização automática da chave privada, *pagent,* onde sempre que o sistema for inicializado todas as chaves cadastradas são automaticamente gerenciadas por esse agente para a conexão, Figura 46, não sendo mais necessária a inserção de usuário e senha. Garante-se assim, o acesso restrito, apenas aos detentores da chave criptografada.

Figura 46 - Listas de chaves inicializadas pelo Pageant

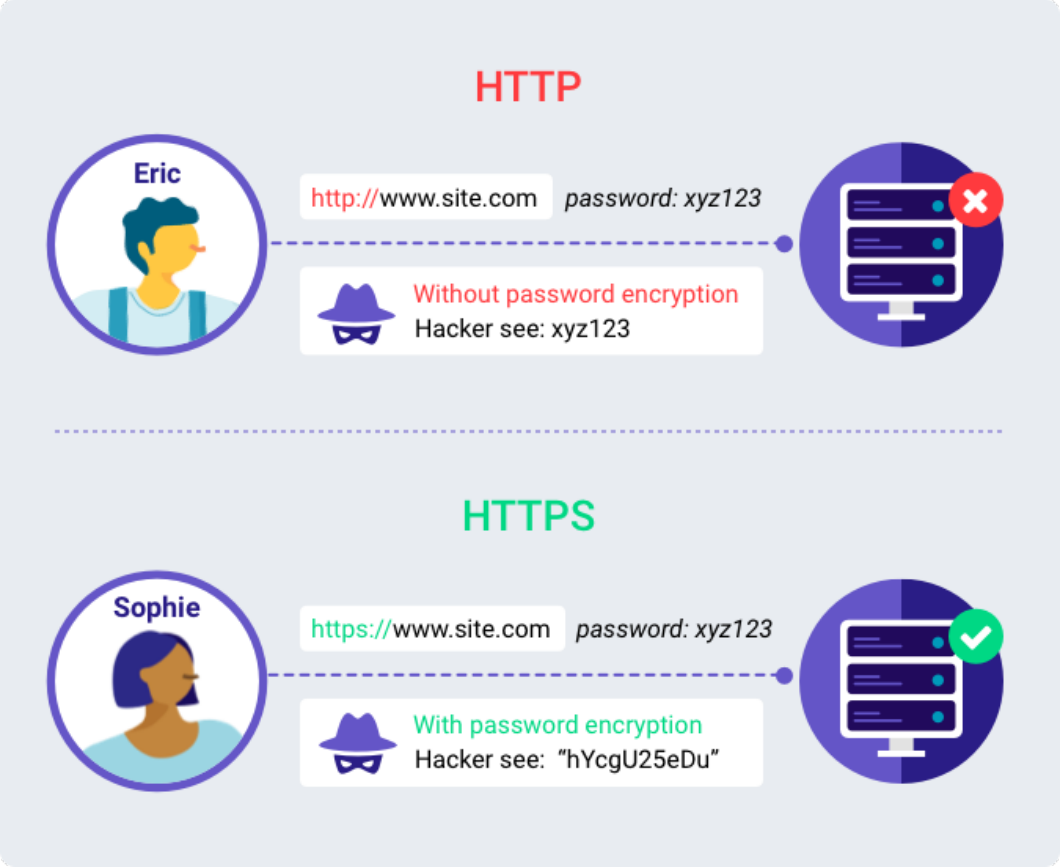


Fonte: Autor.

4.5.2 HTTPS e Autenticação/Autorização

Hyper Text Transfer Protocol Secure (HTTPS) é uma implementação do protocolo HTTP sobre uma camada adicional de segurança que utiliza o protocolo *Secure Socket Layer* (SSL)/*Transport Layer Security* (TLS), que permite que os dados sejam transmitidos por meio de uma conexão criptografada e que se verifique a autenticidade do servidor e do cliente por meio de certificados digitais. Isso garante que alguma informação sensível, como senhas, não sejam vistas na integra, Figura 47, mas sim de maneira criptografada, como o caso de senhas dos usuários.

Figura 47 - Acesso em um site com HTTP e HTTPS



Fonte: Botify (2020).

A aplicação dessa funcionalidade na arquitetura, trará mais segurança em acesso ao banco de dados por exemplo, que requerer uma senha. As aplicações externas acessam os serviços da MOA via protocolo *REST* pelo serviço API *Gateway,* que redireciona essa chamada para o serviço desejado. Como por exemplo temos o acesso dos valores dos sensores através do navegador por meio desse link *http://ip_raspberrypi/api/daq/riin*.

Para adicionar a funcionalidade do HTTPS, utilizou-se o software *OpenSSL* no *Windows* para a geração tanto da chave, *key.pem*, e o certificador local, *certificate.pem*, que utiliza o algoritmo *RSA* com 2048 *bits* e duração de 365 dias. Configurou-se o esquema do serviço do *api gateway* conforme Figura 48, onde para leitura das chave e do certificado utilizou-se o *file system*, sendo possível agora acessar os serviços com o segurança, e mantendo-se todos os caminhos anteriormente criados. Acessou-se o exemplo citado anteriormente agora com segurança *https://ip_raspberrypi/api/daq/riin*.

Figura 48 - Esquema do serviço API com HTTPS

```
module.exports = {

    name: "api",

    mixins: [ApiGateway],

    settings: {

        // openssl software generate key and certificate
        // command : $ openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365
-out certificate.pem

        https: {
            key: fs.readFileSync("ssl/key.pem"),
            cert: fs.readFileSync("ssl/certificate.pem")
        },
        ...
    }
}
```

Fonte: Autor.

Não é interessante para alguns serviços que se tenha uma rota publica, onde para acessá-lo é necessária uma senha ou autenticação. Adicionou-se mais segurança ao aplicar a autenticação antes de acessar os serviços, onde para isso é necessário habilitar essa função no serviço do *Gateway*, Figura 49, *authentication: true.* Onde qualquer serviço só pode ser acessado com o *token*.

Verificou-se a autenticação adicionando mais um método no serviço, *authenticate*, que verifica se o *token,* digitado para acessar qualquer serviço bate com o registrado, neste caso o *token* é 12345.

Para continuar acessando o mesmo exemplo citado anteriormente que antes era *https://ip_raspberrypi/api/daq/riin*, agora passa a ser acessado como *https://ip_raspberrypi/api/daq/riin?access_token=12345*

Figura 49 - Autenticação no Api Gateway

```
// api.service.js

module.exports = {

    name: "api",
    mixins: [ApiGateway],
    settings: {
        routes: [{
            authentication: true // Enable authentication
        }]
    },

    methods: {
        // Authenticate the user from request
        authenticate(ctx, route, req, res) {
            let accessToken = req.query["access_token"];
            if (accessToken) {
                if (accessToken === "12345") {
                    return Promise.resolve({ id: 1, username: "john.doe", name: "John
Doe" });
                } else {
                    return Promise.reject();
                }
            } else {
                return Promise.resolve(null);
            }
        }
    }
};
```

Fonte: Autor.

Outra maneira de gerar esses tokens seria utilizando o JWT para gerar essas senhas de maneira mais seguras, criando um banco de dados de cadastro de usuários autorizados, onde seria gerado um JWT randômico vinculado ao nome deste usuário e passando o *token* no cabeçalho da mensagem, seguindo a seguinte sintaxe *Authorization: <tipo de criptografia> <token>*, não expondo a senha na URL da requisição.

4.5.3 Autenticação ao conectar-se com transportador NATS

A autenticação de cliente (microsserviço) é o processo pelo qual os clientes acessam de forma segura um servidor(*transporter*) ou computador remoto utilizando uma identificação e senha. Para a arquitetura proposta, isso impedirá que algum microsserviço não cadastrado

possa conectar-se ao *transporter*, impedindo que se comunique ou envie mensagens para os outros serviços.

O serviço do *transporter* NATS, *nats-server*, inicia-se por padrão sem um usuário e senha, onde qualquer serviço que utilize o NATS para comunicar-se com outros serviços pode plugar-se nessa rede de comunicação sem uma autenticação bastando para isso apontar o endereço onde está hospedado o servidor de comunicação de mensagens.

A fim de dar maior segurança nessas conexões criou-se um arquivo de configuração *nats-server.conf*, Figura 50, no mesmo local onde encontra-se o arquivo executável do *nats-server,* contendo uma lista de todos os usuários e suas respectivas senhas, que podem conectar-se ao barramento de comunicação, e esse arquivo será passado como parâmetro para o servidor assim que iniciado, Figura 51.

Figura 50 - Arquivo *nats-server.conf*

```
authorization: {
    users: [
        {user: admin, password: "1234"},
        {user: a, password: b},
        {user: b, password: a},
    ]
}
```

Fonte: Autor

Figura 51 - Inicialização do servidor *NATS* com seu respectivo arquivo de configuração

```
$ sudo pm2 start "./nats-server -c nats-server.conf" --name nats-server
```

Fonte: Autor

O esquema de qualquer serviço que deseje utilizar esta comunicação deve ser alterado, conforme Figura 52, contendo o endereço onde o servidor *NATS* está sendo executado, neste caso localmente e escutando na porta 4222, e também o mesmo usuário, admin, e senha cadastrados no servidor do NATS, 1234, pois toda vez que algum serviço tentar conectar-se, o usuário e a senha serão solicitados, e chegados com a lista cadastrada no servidor.

Para aumentar ainda mais o nivel de segurança, o servidor do NATS, suporta hash de senhas e tokens de autenticação usando bcrypt. Para tirar proveito disso, substitui-se a senha de texto sem formatação na configuração da Figura 50, 1234, por seu hash bcrypt, e o servidor utilizará automaticamente o bcrypt conforme necessário.

Figura 52 - Esquema de um serviço com usuário e senha para conexão com o NATS

```
let broker = new ServiceBroker({
    namespace: "",
    nodeID: "",
    transporter: {
        type: "NATS",
        options: {
            url: "nats://localhost:4222",
            user: "admin",
            pass: "1234"
        }
    },
});
```

Fonte: Autor

A ferramenta de criptografia do próprio *nats-server, mkpasswd*, vista na Figura 53, é quem faz todo o trabalho, bastando apenas passar a senha utilizada, nesse caso 1234, e o retorno é a senha criptografada, *$2a$11$7lWUScD1smGd1IbKn2sqwO7tggRy6LYK.y2euAyqUn08ymgksseXq,* que deve ser substituído no arquito *nats-server.conf* conforme Figura 54, e reiniciar o servidor novamente. Nesse caso o cliente ainda usa a versão em texto sem formatação em seu esquema de serviço, Figura 52, não necessitando nenhuma configuração a mais para conectar-se.

Figura 53 - Ferramenta *mkpasswd* para encriptar a senha

```
$ cd go/src/github.com/nats-io/nats-server/util/mkpasswd
$ sudo go run mkpasswd.go -p
Enter Password: 1234
Reenter Password: 1234
bcrypt hash: $2a$11$7lWUScD1smGd1IbKn2sqwO7tggRy6LYK.y2euAyqUn08ymgksseXq
```

Fonte: Autor

Figura 54 - Arquivo *nats-server.conf* com senha criptografada

```
authorization: {
    users: [
        {user: admin, password: "$2a$11$7lWUScD1smGd1IbKn2sqwO7tggRy6LYK.y2euAyqUn08ymgksseXq"},
        {user: a, password: b},
        {user: b, password: a},
    ]
}
```

Fonte: Autor

Valida-se tal proteção tentando conectar qualquer serviço ao servidor do NATS e observando seu respectivo *log,* na Figura 55. Onde inicia-se o serviço, *name.service.js*, e caso

suas credenciais de usuário e senha estejam corretas, exibe-se a mensagem que foi conectado com sucesso, caso contrário, um alerta de violação de segurança é exibido.

Figura 55 - Log de conexão de um serviço com o transporter NATS

```
$node name.service.js
[2020-09-03T17:30:18.269Z] INFO  raspberrypi-2-1116/TRANSIT: Connecting to the
transporter...

# Credenciais corretas
[2020-09-03T17:30:19.160Z] INFO  raspberrypi-2-1116/TRANSPORTER: NATS client is connected.

# Credenciais incorretas
[2020-09-03T17:33:31.958Z] ERROR raspberrypi-2-1136/TRANSPORTER: NATS error. 'Authorization
Violation'
```

Fonte: Autor

4.5.4 Microsserviço de Guarda com JSON *Web Token*

O microsserviço guarda monitora as mensagens que são transmitidas entres os microsserviços, onde sempre que alguma mensagem é envida, é emitido um evento para o serviço de guarda antes de chegar ao seu destinatário, essa funcionalidade garante á aplicação que microsserviços não autorizados, não consigam conversar com outros microsserviços, mesmo que tenham sidos autenticados no *transporter.* O microsserviço *guard* trabalha em conjunto com o *middleware SeviceGuard* para proteger as ações dos serviços com JWT. Possui duas ações principais, *check* e *generate*, e dois métodos *generateJWT* e *verifyJWT*.

Gerou-se as chaves utilizando o console de desenvolvedor interativo *REPL*, Figura 56, com o seguinte comando *call guard.generate --service name,* onde a ação *generate* do serviço *guard* recebeu como parâmetro o nome do serviço que deseja-se gerar o *token*, e em seguida chamou-se o método *generateJWT* que utiliza a biblioteca *NPM* (*jsonwebtoken*, 2020) que por padrão utiliza o algoritmo *HS256*.

O segredo ou chave privada sempre é o parâmetro *moleculer* definido internamente neste método, ao finalizar retorna o *token* codificado *(JSON Web Tokens*, 2020), possuindo três partes: a primeira é o cabeçalho que tem o algoritmo (*HS256*) e o tipo de *token* (*JWT*), a segunda tem os dados, no caso o nome do serviço e a última parte a verificação da assinatura. Esse processo foi repetido para todos os serviços, onde cada serviço tem um token *distinto*.

Figura 56 - Geração chave com o método guard.generate com o parâmetro daq

```
mol $ call guard.generate --service daq
{
  options: { service: 'daq' },
  actionName: 'guard.generate',
  rawCommand: 'call guard.generate --service daq'
}
>> Call 'guard.generate' with params: { service: 'daq' }
>> Execution time:
>> Response:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzZXJ2aWNlIjoiZGFxIiwiaWF0IjoxNTgxNjkzNDA3fQ.0Jly6RwDGoZjJRIgaM6bKiUFMGJFeUruyiCF3hxMAbw'
```

Fonte: Autor

Colocou-se o *token* recebido no esquema dos serviços na propriedade *authToken* Figura 57. Definiu-se a restrição na definição da ação, onde caso a propriedade *restricted* tiver seu atributo como *null* ou não for definida isso significa que a ação pode ser chamada por qualquer serviço.

Caso a restrição seja colocada para um nome ou mais de serviços, somente esses podem acessá-la. No caso, Figura 57, a função de leitura dos sensores, *riin,* pode ser acessada por qualquer serviço, pois foi definida como *null.* Já a função de escrita para atualização das saídas, *ruout,* pode ser acessada apenas pelo serviço *api* e *pid*, caso algum outro serviço tente acessá-lo um erro é gerado.

Figura 57 - Esquema do serviço DAQ com JWT

```
module.exports = {

    name: "daq",

    authToken:
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzZXJ2aWNlIjoiZGFxIiwiaWF0IjoxNTgxNjkzNDA3fQ.0Jly6R
wDGoZjJRIgaM6bKiUFMGJFeUruyiCF3hxMAbw',

    actions: {

        riin: {
            // It can be called by everyone.
            restricted: null,
            handler(ctx) {}
        },

        ruout: {
            // It can be called by "api" & "pid" service.
            restricted: [
                "api",
                "pid"
            ],
            handler(ctx) {}
        }
    },
}
```

Fonte: Autor.

Para observar as chamadas de ações colocou-se o *middleware ServiceGuard* em cada *broker* de cada serviço. Quando um serviço chama a ação de outro serviço, o *ServiceGuard* chama a ação *check* do serviço *guard*, que verifica se o *token* é valido e se ele está ou não restrito a acessar a ação do serviço desejado, caso positivo a ação é acessada normalmente, caso negativo a ação não pode ser acessada e um erro é retornado.

5 RESULTADOS E DISCUSSÕES

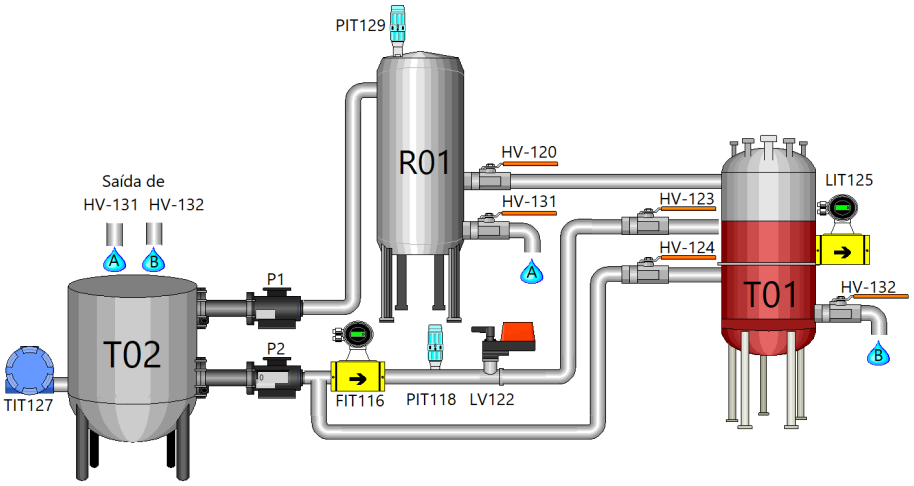
Realizou-se os experimentos com a arquitetura de microsserviços usando a planta piloto de controle de processos, e nesses experimentos foram necessárias algumas configurações para seu correto funcionamento, além de composições de serviços diferentes que serão abordados nos próximos itens.

A latência da rede não é desprezível, principalmente quando usado comunicação sem fio (Wi-Fi). Os serviços podem ser dimensionados para serem resilientes e tolerantes a falhas como o caso do serviço de aquisição de dados que foi duplicado. Os serviços são iguais e não têm hierarquia, mestre ou escravo ou prioridade. Quando colocados no modo de produção, são registrados e podem ser observados através do terminal, quais estão on-line, quanta CPU estão usando, quais ações estão disponíveis, entre outros dados.

A comunicação interna entre os serviços é feita através do transportador de mensagens NATS, que é o padrão do framework *Moleculer,* e para comunicação com redes externas, ele possui seu próprio API gateway que através do protocolo HTTP comunica-se com aplicações web.

A principal aplicação externa utilizada para coleta e análise dos dados foi o LabVIEW, na Figura 58 criou-se os principais componentes da planta industrial para fácil associação com o modelo real, como os tanques (TQ01 e TQ02) e reservatório (R01), bombas (P1 e P2), sensores (PIT118, PIT129, LIT125, FIT116 e TIT127) e a eletroválvula (LV122) com seus respectivos indicadores, além de representações das válvulas manuais (HV-120, HV-123, HV-124 e HV-132).

Figura 58 - Painel frontal do LabVIEW ilustrando os componentes da planta piloto



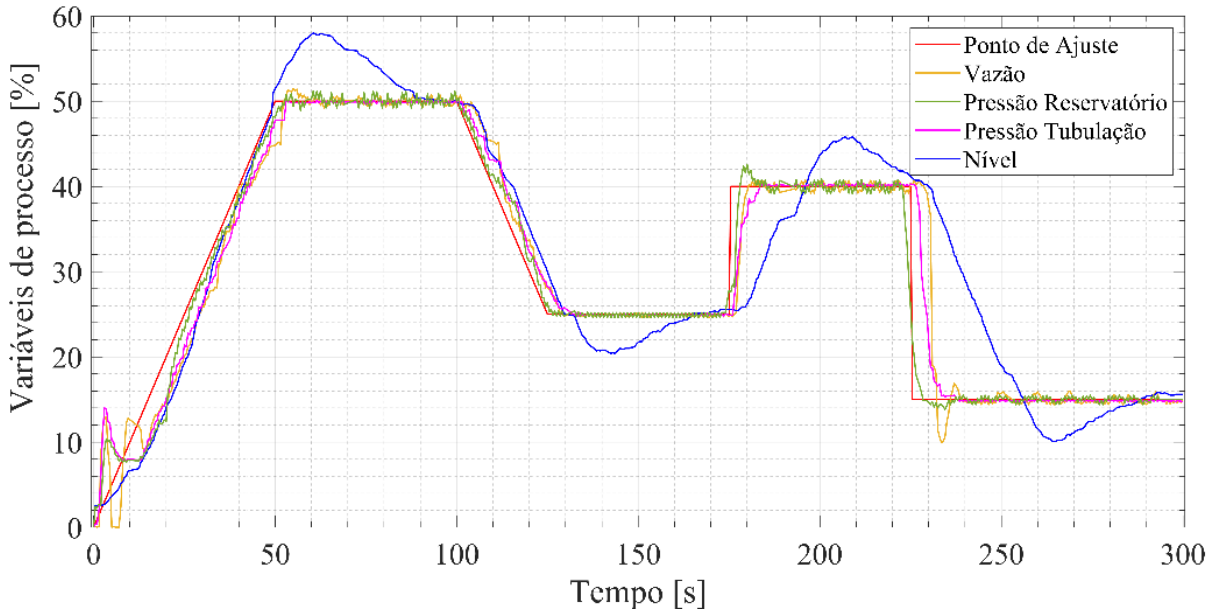
Fonte: Autor

5.1 Controle das Malhas de Processo

Para demonstrar a aplicabilidade da MOA e dos microsserviços desenvolvidos, controlou-se todas as malhas de processo da planta: pressão da tubulação, pressão do reservatório e vazão, que tem uma dinâmica de processo mais rápida, e a malha de nível, que tem uma dinâmica mais lenta. A Figura 59 apresenta as curvas de controle das malhas na planta piloto, a qual apresenta alguns distúrbios, pois foram os primeiros testes realizados, o que foi corrigido com o arredondamento de casas decimais, podendo ser visto a melhora significativa do gráfico na Figura 60 ao final deste capitulo.

A planta possui potencial para testes de controle em cascata ou testes de controle simultâneo, porém para comparação de resultados de métricas de tempos manteve-se apenas uma malha operacional, na qual escolheu-se a malha da pressão da tubulação.

Figura 59 - Resposta de controle de todas as malhas da planta



Fonte: Autor

Realizou-se teste no circuito de controle de pressão da tubulação da planta piloto industrial com o tempo do ciclo de malha fechada de 500 ms, sendo coletados um total de 600 amostras dentro de 300 segundos. As válvulas manuais HV-124 fechada, HV-132 e HV-123 abertas e a eletroválvula LV-122 aberta 20%. A pressão na tubulação foi controlada com a bomba P2, podendo coletar os dados com o sensor de pressão PIT-118.

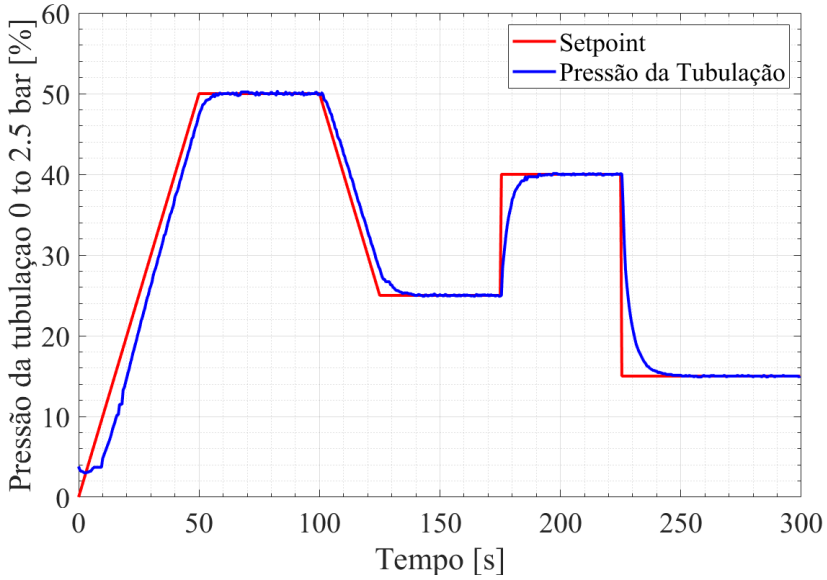
O serviço de controle PID roda em conjunto com o serviço DAQ. Por exemplo, para controlar a pressão da tubulação, é necessário coletar o valor do sensor de pressão PIT-118.

Esse dado e parâmetros apropriados são enviados para o serviço de controle PID. Despois de calculado o controle, o serviço DAQ é chamado novamente e atualiza a frequência do inversor da bomba.

Na Figura 60 tem-se o gráfico da resposta de controle da pressão da tubulação, no qual utilizamos apenas uma imagem, pois os gráficos são muito semelhantes tendo uma pequena variação, onde em vermelho temos o valor desejado (*Setpoint)*, e a variável de processo, pressão da tubulação, na cor azul.

Padronizamos todas as variáveis de processo em porcentagem para facilitar as conversões e visualizações gráficas das quatro possíveis variáveis da planta. No gráfico da Figura 60 quando marcado 50% da pressão da tubulação, representa 1,25bar.

Figura 60 - Malha de controle de pressão da tubulação



Fonte: Autor

Foi possível verificar que tanto a orquestração, quanto a coreografia de serviços são viáveis para o controle de processos em aplicações da I4.0, comprovados nos testes reais de uma planta piloto Industrial. A composição de serviços aprimora a flexibilidade e a modularidade de aplicativos industriais, simplificando a criação de novos serviços e composições variadas. Além disso, tanto a orquestração quanto a coreografia de microsserviços já fornecem acesso a todas as informações do processo, o que facilita o desenvolvimento de aplicativos de monitoramento e manutenção.

Aplicou-se o gerenciador de processos, PM2, para a inicialização de todos os serviços de maneira automática ao inicializar os sistemas embarcados. Em caso de o sistema ser

reiniciado, todos os serviços sobem automaticamente, mantendo sempre todos os serviços online.

O *Moleculer* tem internamente estratégias de balanceamento. Se houver múltiplas instâncias (réplicas) de um mesmo serviço rodando, o registro de serviços irá utilizar essa estratégia para selecionar um nó, ou serviço, disponível de todos que estejam conectados com o mesmo nome ou função. Duplicou-se o serviço DAQ para redundância na aquisição de dados. Se um dos dois serviços DAQ ficar offline o balanceador de cargas irá chamar o outro serviço DAQ fazendo o serviço ficar tolerante a falhas.

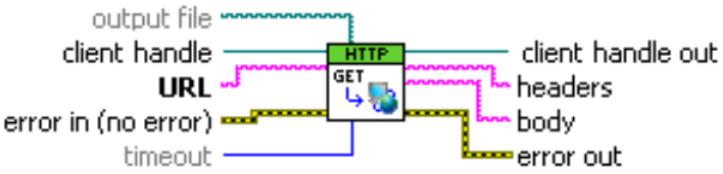
5.2 Orquestração

5.2.1 Aplicação Externa

Antes de descrever como o processo é executado, é interessante uma breve explicação de como uma requisição é realizada no LabVIEW, onde sua estrutura geral é apresentada na Figura 62. Com o GET.vi é possível enviar uma solicitação da Web que retorna cabeçalhos e dados do corpo de um servidor, página da Web ou serviço da Web.

Este VI usa o método GET HTTP, onde o detalhamento das entradas e saídas podem ser vistos na Figura 61, com descrição das principais.

Figura 61 - GET.vi



Fonte: LabVIEW (2020)

Output file: especifica um arquivo para salvar os dados do corpo retornados pelo servidor.

Client handle: especifica o identificador de cliente a ser associado à solicitação da Web.

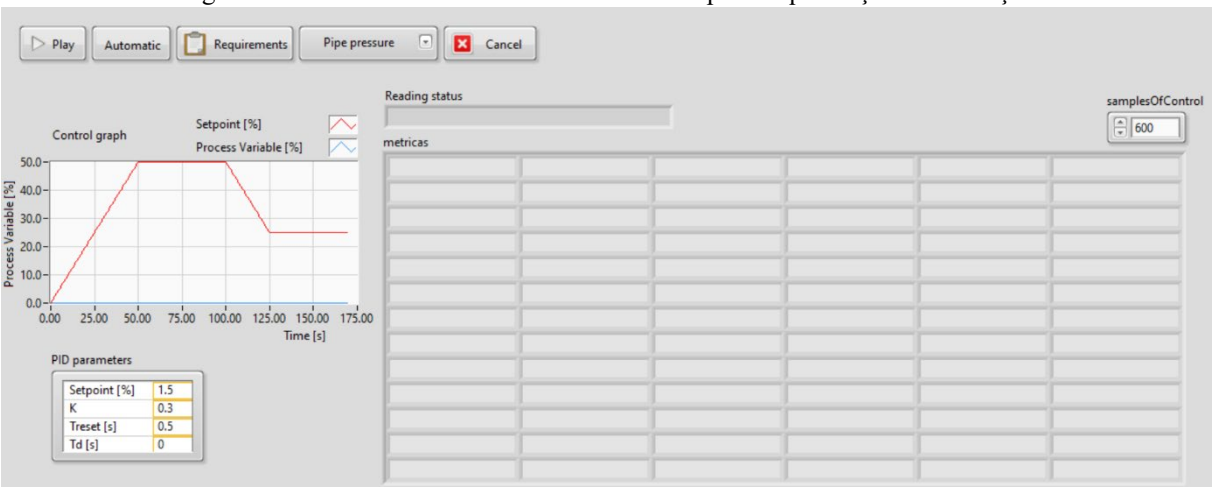
URL: especifica a URL do servidor, página da Web ou serviço da Web para o qual este VI envia a solicitação da Web.

Headers: retorna os campos de cabeçalho retornados pelo servidor. Consulte o site do World Wide Web Consortium em www.w3.org para obter mais informações sobre as definições dos campos de cabeçalho, incluindo cabeçalhos disponíveis, descrições e sintaxe.

Body: retorna os dados do corpo retornados pelo servidor.

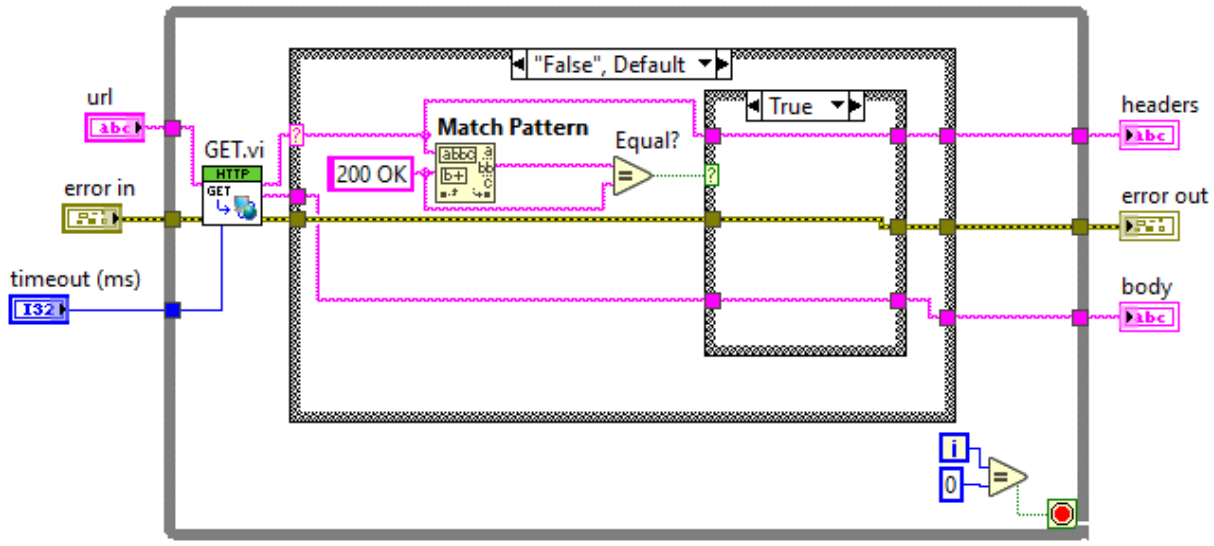
O processo de controle da malha de pressão de linha inicia-se no painel frontal do LabVIEW Figura 62, ao pressionar o botão de *Play,* que requisita ciclicamente, através de requisições HTTP Figura 63, ao serviço DAQ o dado do sensor por meio da URL http://192.168.1.201:3000/api/daq/riin.

Figura 62 - Painel frontal do LabVIEW utilizado para orquestração dos serviços



Fonte: Autor.

Figura 63 - Diagrama de bloco do LabVIEW contendo uma requisição.



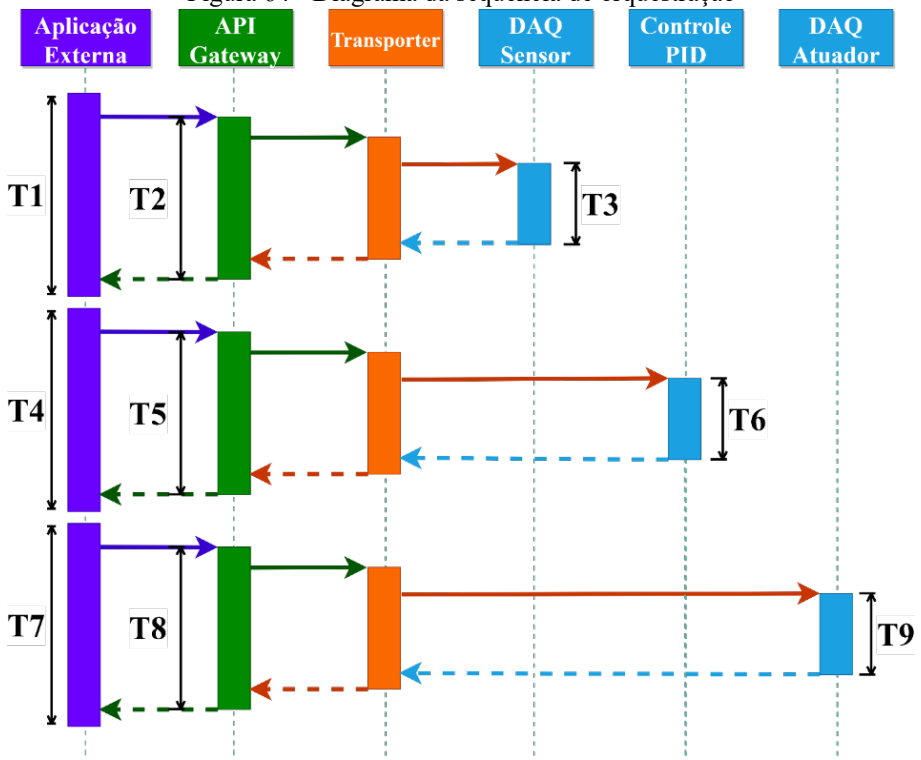
Fonte: Autor

Caso essa requisição seja bem sucedida recebe um cabeçalho com o código “200 OK”, na Figura 63, que é comparado com a *string* “200 OK” utilizando o *Match Pattern* e o *Equal?,* caso seja positivo o valor de todos os sensores da planta são atualizados na variável de saída *body*. Caso negativo o último valor armazenado é mantido utilizando o *shiftregister*.

O dado do sensor é enviado ao serviço de controle PID que calcula o sinal de controle para ser aplicado na planta, pela URL http://192.168.1.201:3000/api/control/pidplus?ti=0.50&td=0.00&kp=0.30&pv=%s&setpoint=%s. Finalmente o sinal de controle é enviado ao serviço DAQ, que é responsável por atualizar o atuador da planta, pela URL http://192.168.1.201:3000/api/daq/wuout?channel=2&value=%s.

Analisou-se o desempenho de execução durante a orquestração com o serviço traçador. O objetivo foi medir a distribuição do tempo e a duração de cada ação de serviço. Para coletar esses tempos, o cálculo de métricas foi habilitado em cada intermediador de serviço de cada serviço. O termo T1 ao T9 na Figura 64 representam o período médio medido durante cada orquestração. Onde cada requisição e resposta emitem um evento ao serem criadas e finalizadas, e seu tempo é subtraído obtendo o tempo total, quem executa esses cálculos é o serviço de métricas, que armazena todos os dados das requisições em um arquivo *requests.json*.

Figura 64 - Diagrama da sequência de orquestração

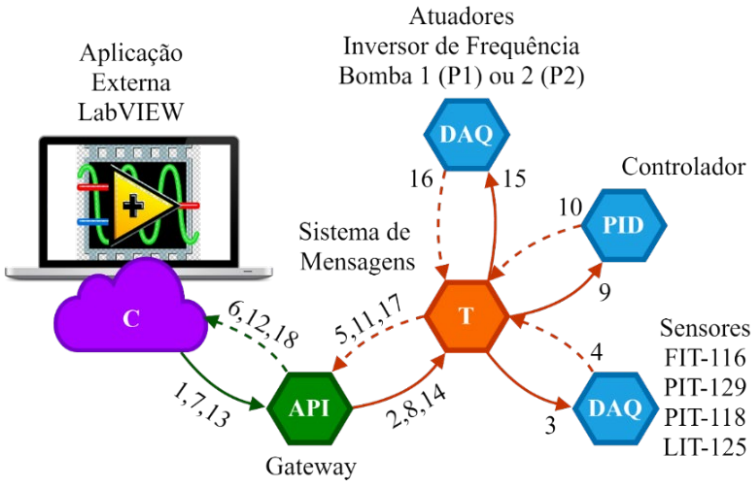


Fonte: Autor

A orquestração simplificada de cada um dos serviços da planta pode ser vista na Figura 64 onde a aplicação externa LabVIEW é representada de roxo com o a letra C, que significa composição, nestes caso composição de orquestração externa de serviços, pois é realizada externamente e acessa os serviços através da API Gateway em verde, e depois passa pelo transportador de mensagens em laranja, representado pela letra T para acesso aos serviços, em azul, DAQ e PID. A numeração de 1 a 18, são a quantidade de passos necessárias para realizar apenas um loop de controle.

Esses tempos coletados no LabVIEW e estão na Tabela 4. Na coluna ação, os termos http representam uma requisição feita pelo LabVIEW utilizando o protocolo REST, como exemplo, *http.daq.rii*, temos uma requisição GET ao serviço *daq* pedindo a ação *riin*, de leitura dos valores de entrada de corrente dos sensores, ou seja, *http.service.action*, que representa o tempo total que durou essa requisição passando por todos os passos de 1 a 6 na Figura 65.

Figura 65 - Diagrama da sequência de orquestração externa



Fonte: Autor

Tabela 4 - Orquestração externa com transportador NATS

T	Ação	Média [ms]	Mínimo [ms]	Máximo [ms]	Desvio Padrão [ms]	Amostras bem sucedidas [%]
1	http.daq.riin	22,9	16,3	111,5	22,3	99,2
2	api.rest.daq.riin	19,0	13,5	346,6	41,7	
3	daq.riin	4,8	2,7	22,7	1,2	
4	http.control.pidplus	23,1	0	109,6	12,2	92,8
5	api.rest.control.pidplus	18,3	10,7	311,3	18,3	
6	control.pidplus	2,4	1,2	17,5	1,5	
7	http.daq.wuout	21,3	0	84,1	8,1	91,8
8	api.rest.daq.wuout	16,2	9,5	48,1	4,2	
9	daq.wuout	1,9	1,1	13,6	0,8	

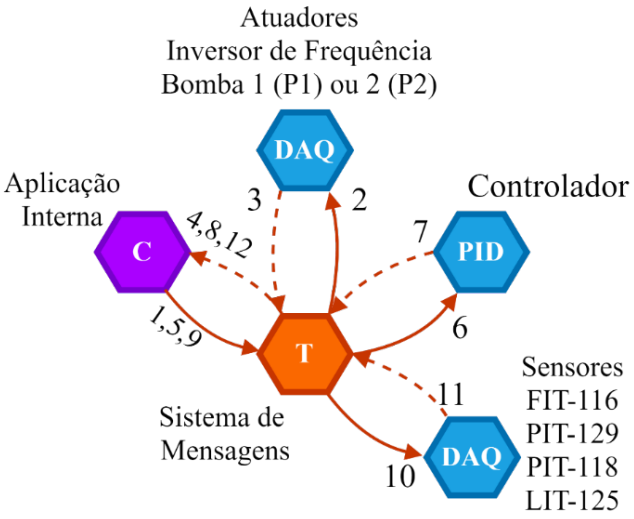
Fonte: Autor

Já o termo *api.rest.daq.riin*, é uma chamada interna do API Gateway ao serviço *daq* com a ação *riin*, passando por todos os passos do 2 a 5 na Figura 65, ou seja, *api.rest.service.action* sendo o tempo de duração da *api* mais o tempo de comunicação do *transporter* e por último o *daq.riin*, *service.action* que é o tempo de duração que o serviço *daq* demorou para executar a ação *riin*, de leitura dos sensores. Essas nomenclaturas foram adotadas nos demais testes.

5.2.2 Aplicação Interna

Criou-se um serviço de composição utilizando a orquestração representado em roxo e com a letra C na Figura 66, onde seu princípio de funcionamento é executar sequencialmente DAQ, PID e DAQ novamente, seguindo o mesmo princípio do LabVIEW, porem neste caso não temos a passagem pelo API Gateway. Os tempos de execução foram compilados e estão na Tabela 5.

Figura 66 - Diagrama da sequência de orquestração interna



Fonte: Autor

Tabela 5 - Orquestração interna com transportador NATS

T	Ação	Média [ms]	Mínimo [ms]	Máximo [ms]	Desvio Padrão [ms]	Amostras bem sucedidas [%]
1	composition.orchestration	35,8	26,8	512,5	60,3	97,2
2	daq.riin	4,3	2,5	15,3	0,7	
3	control.pidplus	1,2	0,7	11,6	0,6	
4	daq.wuout	1,6	0,9	11,4	0,5	

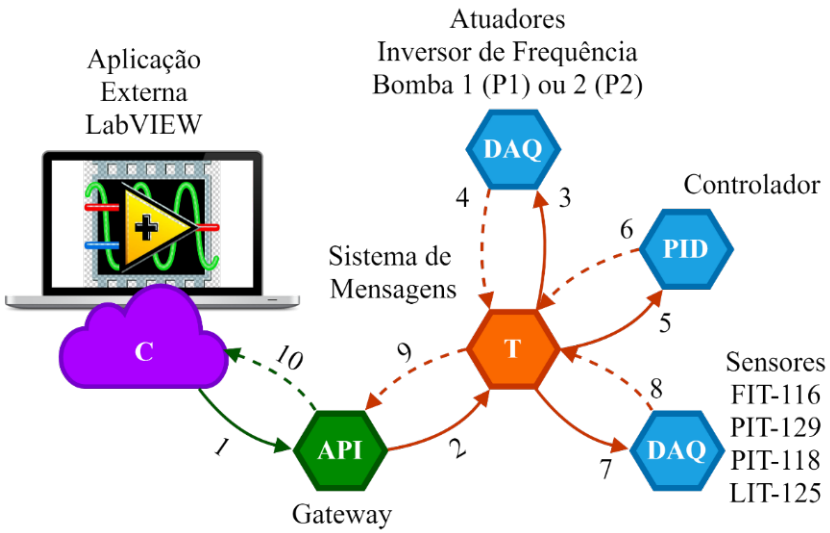
Fonte: Autor

5.3 Coreografia

5.3.1 Aplicação Externa

Ao contrário da orquestração externa que vai e volta diversas vezes centralizando as informações, a coreografia vai e volta apenas uma vez, passos 1 e 10, pois os serviços trabalham de forma colaborativa. A sua sequência de execução pode ser vista na Figura 67 bem como suas métricas de tempo na Tabela 6.

Figura 67 - Diagrama da sequência de coreografia externa



Fonte: Autor

Tabela 6 - Coreografia externa com transportador NATS

T	Ação	Média [ms]	Mínimo [ms]	Máximo [ms]	Desvio Padrão [ms]	Amostras bem sucedidas [%]
1	http.daq.riin	27,3	18,0	206,6	37,7	99,8
2	api.rest	21,7	0	518,5	52,3	
3	daq.riin	4,8	2,2	24,0	1,8	
4	control.pidplus	3,0	1,3	18,0	1,7	
5	daq.wuout	3,7	1,7	19,3	1,6	

Fonte: Autor

5.3.2 Aplicação Interna

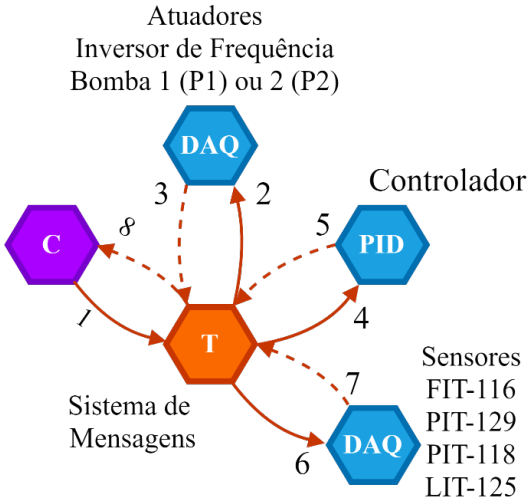
No caso da coreografia com aplicação interna, não tem a passagem pela API Gateway, e quem inicia todo o processo é o serviço de Composição em roxo, tendo a sequência de execução de 1 a 8, na Figura 68. Os tempos de execução foram compilados e estão na Tabela 7.

Tabela 7 - Coreografia interna com transportador NATS

T	Ação	Média [ms]	Mínimo [ms]	Máximo [ms]	Desvio Padrão [ms]	Amostras bem sucedidas [%]
1	composition.choreography	18,9	6,4	147,2	15,8	96,2
2	daq.riin	4,6	2,5	20,7	1,2	
3	control.pidplus	2,2	1,1	15,7	1,3	
4	daq.wuout	2,4	1,2	14,8	1,0	

Fonte: Autor

Figura 68 - Diagrama da sequência de coreografia interna



Fonte: Autor

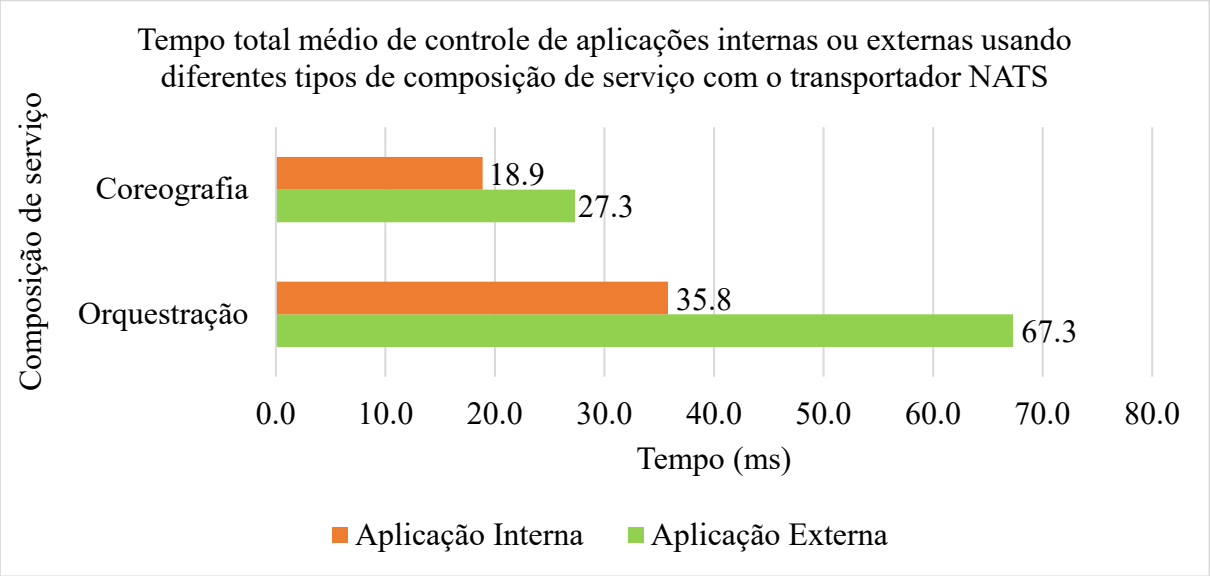
5.4 Comparação

5.4.1 Composição de serviços

Esta comparação tem o objetivo de verificar se o microsserviço API Gateway pode impactar nas aplicações industriais deste projeto. Para isso fizemos testes onde todas as requisições passam pela API Gateway, que são denominadas como aplicações externas. E testes sem a mesma, onde os serviços comunicam-se entre si, que são as aplicações internas. Além disso, comparamos dois tipos de composição, a coreografia e a orquestração.

Todos os experimentos realizados com aplicação externa e interna, tanto na orquestração quanto na coreografia estavam com todos os mecanismos de segurança desativados. Os dados dos testes das subseções 5.2 e 5.3 foram compilados e apresentados na Figura 69. Pode-se observar que a aplicação externa é sempre mais lenta que a aplicação interna como esperado, e a coreografia sempre é mais rápida que a orquestração.

Figura 69 - Gráfico do tempo de controle de todas as composições testadas.



Fonte: Autor

Analisando o tempo médio total de controle da malha de pressão de cada experimento, o menor tempo conseguido foi de 18,9 ms na coreografia com aplicação interna e o maior tempo de 67,3 ms na orquestração com aplicação externa.

Apenas como referência a especificação técnica da Petrobras número ET-5290.00-2000-800-MDP-001 estipula os seguintes períodos que se deve considerar como tempo máximo de resposta para malhas de controle:

a) Para variáveis de vazão, pressão e pressão diferencial: 500 ms;

b) Para variáveis de temperatura, nível e analítica: 1000 ms.

Desse modo, considerando os resultados apresentados, no pior dos casos que foi o tempo de 67,3ms, demonstra-se que a arquitetura orientada a microsserviços com composição de serviços por orquestração e por coreografia é capaz de cumprir especificações de malhas de controle de processos industriais, usando como base as definidas pela Petrobras.

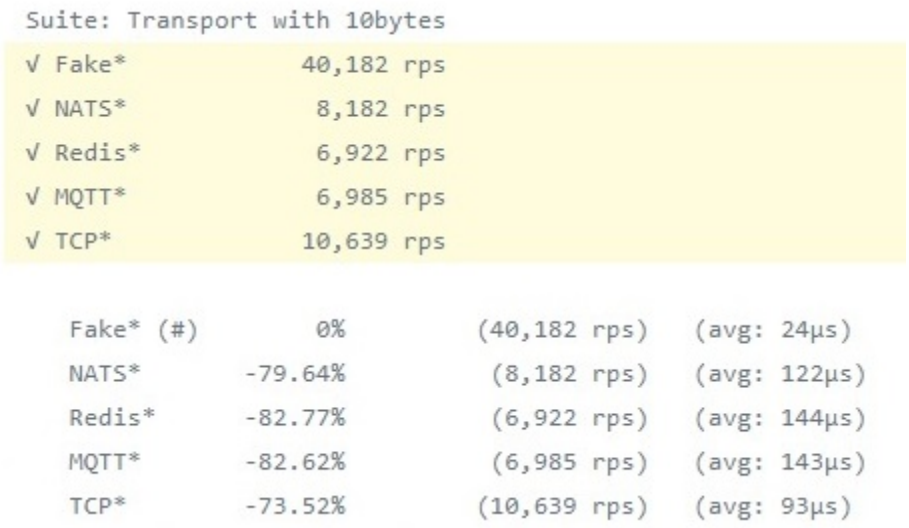
5.4.2 Protocolos de Mensagens (*Transporters*)

Selecionar um protocolo de mensagens eficaz é um desafio e uma tarefa difícil, porque depende da natureza do sistema/solução desenvolvido e de seus requisitos de mensagens. Alguns protocolos de mensagem para aplicações de IoT e arquiteturas orientadas a serviços incluem o CoAP, AMQP, Web Services SOAP/REST, TCP, WebSocket, MQTT, NATS, Redis, HTTP.

Considerando aplicações tradicionais de monitoramento via arquiteturas de IoT, as principais preocupações sobre esses protocolos são a perda de pacotes, o esforço de processamento e o consumo de energia que os dispositivos IoT requerem para transferir dados (geralmente sem fio) de um dispositivo para outro ou para a nuvem. Alguns trabalhos estudam e comparam esses protocolos em diferentes aplicações de IoT (NAIK, 2017; KOKKONIS, CHATZIMPARMPAS; KONTOGIANNIS, 2018; SUEDA, SATO, HASUIKE, 2019).

O framework *Moleculer* representa uma solução para criação de aplicações baseadas em microsserviços focada para a área de Computação. O *Moleculer* oferece suporte a diferentes tipos de protocolos para serem usados para a comunicação entre os serviços (*Transporter*), sendo o NATS o padrão. Um benchmark de testes de operação do *Moleculer* com diferentes protocolos pode ser visto no site oficial do framework, e é apresentado na Figura 70. Nesse benchmark, os fatores considerados estão relacionados à aplicação computacional, sendo o número de requisições por segundo (*rps*) realizadas e a média do tempo de execução dos serviços de teste (*avg*).

Figura 70 - *Moleculer* Benchmark



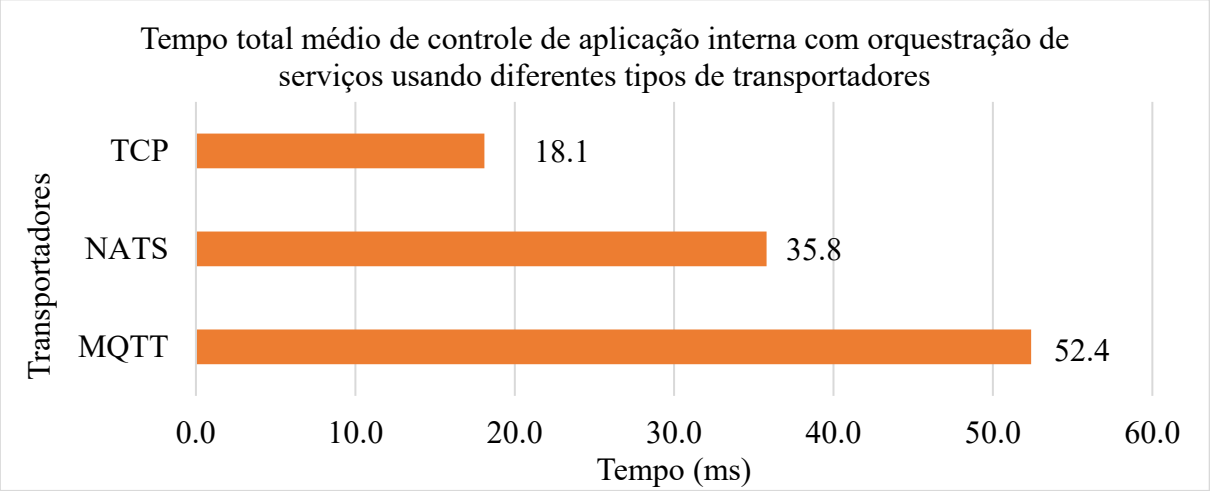
Fonte: Moleculer (2020).

Conforme pode ser visto na Figura 70, o framework *Moleculer* operando com *Transporter* usando protocolo TCP foi o que obteve o melhor desempenho. O *Transporter* TCP do *Moleculer* não necessita de configuração zero e usa o protocolo *Gossip* para divulgar o status do nó, a lista de serviços e a frequência cardíaca de cada serviço. Ele contém um recurso de descoberta UDP integrado para detectar nós novos e desconectados na rede.

Considerando aplicação do *Moleculer* para o cenário industrial neste trabalho, principalmente contemplando aplicações de controle em malha fechada, é importante avaliar outros fatores como o atraso (latência) e *jitter* (variabilidade do atraso) de comunicação. Neste trabalho, escolhemos manter o padrão de métricas dos demais experimentos para ter uma referência comum, que neste caso era a média do tempo total de um ciclo de controle (latência) e seu desvio padrão (variabilidade) ou *jitter*.

Usando a mesma padronização do experimento de orquestração com aplicação externa (subseção 5.2.1) e mudando apenas os protocolos usados no *Transporter* pela estrutura usada, o tempo total de controle para cada transportador específico pode ser visto na Figura 71, onde a mesma sequência de desempenho foi mantida em relação ao teste de orquestração com aplicação externa. Este teste mostra como a escolha do transportador impacta diretamente no tempo total de controle.

Figura 71 - Comparação de diferentes tipos de *transporters* mantendo a estrutura do experimento de orquestração com aplicação externa (subseção 5.2.1) analisando o tempo total.



Fonte: Autor

Neste caso, o protocolo TCP obteve melhor desempenho, pois conforme explicado, a comunicação com outros serviços é feita diretamente, porta a porta. Ao contrário de outros protocolos como MQTT e NATS, que usa seu próprio *broker* para gerenciar mensagens, impactando no tempo de controle final, porém oferecendo mais recursos além da comunicação em si. É importante citar que o NATS é o mecanismo padrão do *Moleculer*, fornecendo suporte a outras funcionalidades como o mecanismo de segurança que foi implementado, descrito na seção 4.5.3.

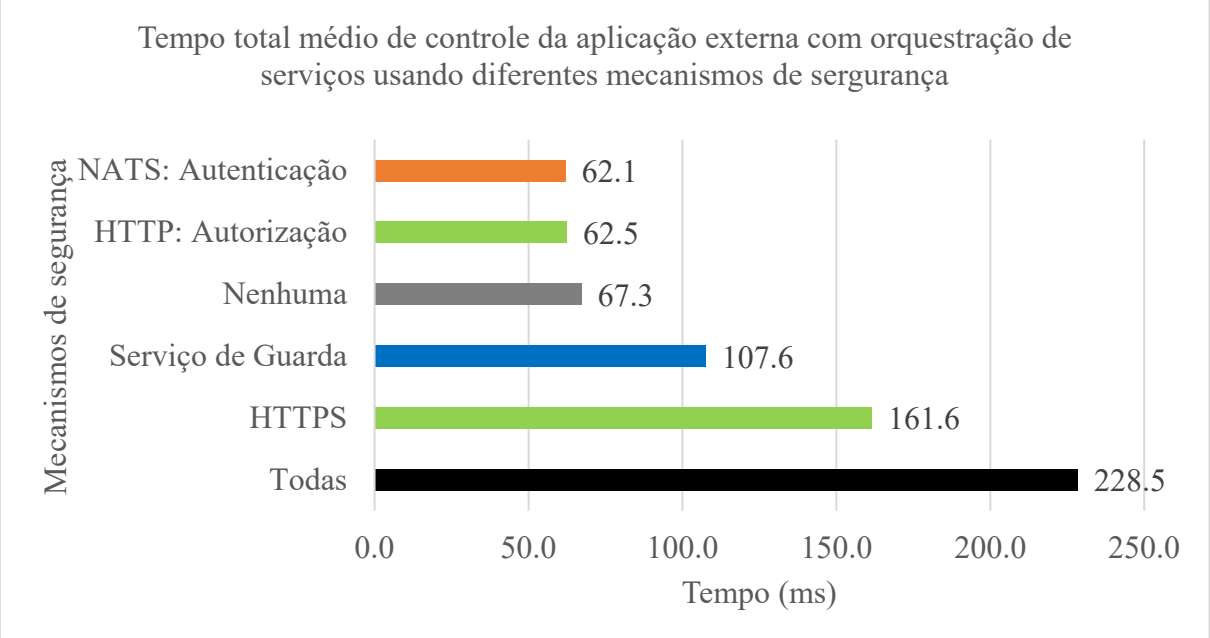
5.4.3 Mecanismos de Segurança

Usando como referência a mesma padronização do experimento de orquestração com aplicação externa (subseção 5.2.1) que não possui nenhum mecanismo de segurança, ativou-se apenas um mecanismo de segurança por vez para testar seu impacto de maneira isolada no tempo total de controle. Na Figura 72 temos o tempo de referência sem nenhum mecanismo de segurança, tendo o tempo de 67,3ms.

Começando da camada mais externa da arquitetura, que é o acesso externo através da API Gateway, ativando apenas a criptografia HTTPS nesse serviço, obteve-se um tempo de 161,6ms. Ainda no mesmo serviço, ativando apenas a autenticação para geração do token que será utilizado para autorização de acesso das rotas, HTTP: Autorização, obteve-se o tempo de 62,5ms.

Já na comunicação interna entre os serviços, utilizando o *transporter* NATS com autenticação, onde todos os serviços que desejam conectar-se a ele devem informar o usuário e senha, obteve-se o tempo de 62.1ms. Além disso tem o mecanismo de um Guarda, que verifica todas as requisições entre os serviços vendo quem está autorizado ou não a realiza-las, como um homem do meio, obteve-se o tempo de 107,6ms.

Figura 72 - Comparação de diferentes tipos de mecanismos de segurança usando como referência a mesma padronização do experimento de orquestração com aplicação externa (subseção 5.2.1)



Fonte: Autor

Os dois mecanismos que mais impactaram no tempo foram o HTTPS e o serviço de Guarda. O aumento de tempo total de controle do HTTPS, deve-se principalmente à constante leitura do certificado e sua validação. Já o serviço de Guarda, tem um aumento também con-

siderável, devido ao uso de um *middleware* que fica a todo momento monitorando as requisições. Mesmo assim, até no pior dos casos, que foi aplicando todos os mecanismos de segurança, tendo um tempo total de 228.5ms, o tempo total de controle ficou dentro do tempo aceitável nos padrões industriais.

6 CONCLUSÃO

Aplicações da indústria 4.0 que empregam a arquitetura orientada a microsserviços garantem a interoperabilidade e integração vertical entre todas as camadas do sistema manufatureiro, tendo como principais características uma estrutura modular e distribuída. Essa arquitetura vem consolidando-se, mostrando interesse em pesquisas e desenvolvimentos. Suas funcionalidades e seus benefícios voltados para aplicações no contexto da I4.0 foram discutidas bem como os detalhes operacionais e da estrutura utilizada, comunicação, composição dos serviços e segurança.

O desenvolvimento dos microsserviços e aplicações propostas neste trabalho foram apresentados, bem como o framework *Moleculer* para a criação dos mesmos, que simplifica a implantação de uma arquitetura orientada a microsserviços, uma vez que toda a infraestrutura computacional não precisou ser desenvolvida, reduzindo o tempo de desenvolvimento permitindo que o foco fosse dado ao desenvolvimento dos microsserviços e aplicações voltadas para automação e controle. Detalhes operacionais de uma planta piloto de controle de processos onde esses microsserviços foram aplicados, com quatro possíveis variáveis de controle com instrumentos e equipamentos industriais usando uma arquitetura orientada a serviços foram discutidos. Foi possível criar os microsserviços para o controle da planta como API Gateway, aquisição de dados, controle, guarda, base de dados para monitoramento e rastreador para métricas de tempo.

Além dos microsserviços, diversos experimentos foram realizados para a validação da arquitetura. A malha de pressão de linha foi utilizada como base, utilizando a plataforma desenvolvida, onde os mesmos serviços utilizados para controle da malha citado, podem ser replicados para o controle de todas as malhas da planta.

O primeiro experimento buscou analisar, se utilizando diferentes tipos de composição, surtiriam algum impacto na performance do controle, como a orquestração e coreografia tanto internamente utilizando apenas o *transporter* como externamente utilizando o api gateway. Os resultados demonstraram que quanto menos rotas os serviços têm que realizar mais eficazes são como no caso da coreografia que apresentaram melhores resultados.

Um segundo teste, verificou se variando os tipos de *transportes,* surtiria algum efeito no tempo de controle. A variação de protocolos de comunicação também impacta o desempenho de comunicação, sendo que *transporter* usando TCP foi o que apresentou o melhor resultado na comunicação serviço a serviço em relação ao NATS e MQTT que utilizam um *broker*.

Por último, testes analisaram o desempenho da comunicação com a implantação dos mecanismos de segurança desenvolvidos, de forma isolada e em conjunto. Apesar de exercerem impacto no desempenho de comunicação, conclui-se que devem ser escolhidos de acordo com o projeto pois também impactam em questões de segurança que são essenciais para aplicações industriais.

Além dos experimentos percebeu-se uma das desvantagens em relação ao modo monolítico no decorrer do projeto, sendo que a arquitetura orientada a microsserviços requer muito mais passos na comunicação para ser executada, pois todas as suas funcionalidades estão desacopladas e distribuídas em microsserviços distintos, aumentando o tempo final de controle, porem nos testes realizados demonstra que mesmo assim, está dentro dos padrões industriais aceitáveis.

Em contrapartida, as vantagens se sobressaem, pois a composição de microsserviços aprimora a flexibilidade e a modularidade de aplicativos industriais, e sua replicação, simplificando a criação de novos serviços. Além disso, já fornece acesso a todas as informações do processo, o que facilita o desenvolvimento de aplicativos de monitoramento e manutenção.

Todos experimentos demonstrados nesta pesquisa indicam que a arquitetura orientada a microsserviços é favorável quanto ao desenvolvimento de soluções para a I4.0. Tópicos relacionados à versionamento, redundância e conteinerização, não bordados, oferecem possibilidades para pesquisas em novos horizontes.

7 REFERÊNCIAS

ARROWHEAD TOOLS. **Arrowhead Tools for Engineering of Digitalisation Solutions**, 2019. Disponível em: https://cordis.europa.eu/project/id/826452. Acesso em: Abril, 2020.

ARROWHEAD. **The Arrowhead Framework**, 2014. Disponível em: https://www.arrowhead.eu/. Acesso em: Abril, 2020.

BANGEMANN T, KARNOUSKOS S, CAMP R, CARLSSON O, RIEDL M, MCLEOD S, HARRISON R, COLOMBO AW, STLUKA P. **State of the art in industrial automation**. Industrial cloud-based cyber-physical systems Springer International Publishing, Switzerland, ch 2, 2014, pp. 23–47.

BASSI, L. **Industry 4.0: Hope, hype or revolution?.** *In*: INTERNATIONAL FORUM ON RESEARCH AND TECHNOLOGIES FOR SOCIETY AND INDUSTRY (RTSI), 3., 2017, Modena. Proceedings. Piscataway: IEEE, 2017. p. 1-6. DOI: 10.1109/RTSI.2017.8065927. Disponível em: https://ieeexplore.ieee.org/abstract/document/8065927. Acesso em: 08 fev. 2019.

BIGHETI, J. A.; FERNANDES, M. M.; GODOY, E. P. **Control as a Service: A Microservice Approach to Industry 4.0**. 2019 IEEE International Workshop on Metrology for Industry 4.0 and IoT, MetroInd 4.0 and IoT 2019 - Proceedings. Anais...Institute of Electrical and Electronics Engineers Inc., 1 jun. 2019

BIGHETI, J. A.; RISSO, S. L.; CALDIERI, M. R.; GODOY, E. P. **Proposta de Arquitetura Orientada a Microsserviços para Aplicações de Internet das Coisas Industrial**. In: XXII Congresso Brasileiro de Automática, 2018, João Pessoa. Anais do CBA 2018. Campinas: SBA, 2018.

BIGHETI, J.A.; CALDIERI, M.R.; GODOY, E.P. **Automação e controle de processos na nuvem**: proposta e estudo de caso. In: Congresso Brasileiro de Automática (CBA), Vitória, ES, 2018.

BLEVINS, T; NIXON, M. AND WOJSZNIS, W. **PID Control Using Wireless Measurements**. American Control Conference (ACC), June 4-6, 2014. p. 1-6.

BORANGIU, T.; TRENTESAUX, D.; THOMAS, A.; LEITÃO, P.; BARATA, J. **Digital transformation of manufacturing through cloud services and resource virtualization**. Computers in Industry, Amsterdam, v. 108, p. 150-162, Jun. 2019. DOI: 10.1016/j.compind.2019.01.006. Disponível em: http://www.sciencedirect.com/science/article/pii/S0166361519300107. Acesso em: 09 set. 2019.

CIAVOTTA, M.; ALGE, M.; MENATO, S.; ROVERE, D.; PEDRAZZOLI, P. **A Microservice-based Middleware for the Digital Factory**. Procedia Manufacturing, Vol. 11, 2017, pp. 931-938.

COLOMBO, A. W, KARNOUSKOS, S. BANGEMANN, T. **Towards the Next Generation of Industrial Cyber-Physical Systems**. In: Bangemann T, Karnouskos S, Delsing J, Stluka P, Harrison R, Jammes F, Lastra JL (eds), Ch 1, Industrial cloud-based cyber-physical systems. Springer International Publishing, Switzerland, 2014, pp 01–22.

COLOMBO, A. W, KARNOUSKOS, S., O. KAYNAK, Y. SHI, S. YIN. **Industrial Cyber-physical Systems**: **A Backbone of the Fourth Industrial Revolution**. In: IEEE Industrial Electronics Magazine, March 2017.

Como funciona o SSH. Disponível em: <https://www.hostinger.com.br/tutoriais/como-funciona-o-ssh/>. Acesso em: 18 nov. 2020.

DELSING, J. **IoT Automation Arrowhead Framework**. CRC Press, 440 p., 2017.

FATTORI, C. C.; JUNQUEIRA, F.; SANTOS FILHO, D. J. DOS; MIYAGI, P. E. **Service composition modeling using interpreted Petri net for system integration**. In: IEEE International Conference on Mechatronics, 2011, pp. 696-701.

FOWLER, M. **Microservices a definition of this new architectural term**, 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: fevereiro, 2020.

Grafana. Disponível em: <https://grafana.com/>. Acesso em: 19 jun. 2020.

HTTPS vs. HTTP: O que é um site seguro e um fator de classificação? | Botify. Disponível em: <https://www.botify.com/blog/what-is-a-secure-website-https-vs-http>. Acesso em: 18 nov. 2020.

IMC-AESOP Project. (2011). **IMC-AESOP Project**. Disponível em: http://www.imcaesop.eu. Acesso em: Novembro, 2020.

INNERBICHLER, I.; GONUL, S.; DAMJANOVIC-BEHRENDT, V.; MANDLER, B.; STROHMEIER, F. **NIMBLE collaborative platform: Microservice architectural approach to federated IoT**. Global Internet of Things Summit (GIoTS), Geneva, 2017, pp. 1-6.

JAZDI, N. **Cyber physical systems in the context of Industry 4.0**. *In:* IEEE INTERNATIONAL CONFERENCE ON AUTOMATION, QUALITY AND TESTING, ROBOTICS, AQTR, 2014, Cluj-Napoca. Proceedings [...]. Piscataway: IEEE, 2014. p. 1–4. Disponível em: https://ieeexplore.ieee.org/abstract/document/6857843. Acesso em: 24 nov. 2018

JSON Web Tokens - jwt.io. Disponível em: <https://jwt.io/>. Acesso em: 14 fev. 2020.

jsonwebtoken - npm. Disponível em: <https://www.npmjs.com/package/jsonwebtoken>. Acesso em: 14 fev. 2020.

KAGERMANN, H.; WAHLSTER, W.; HELBIG, J. **Recommendations for implementing the strategic initiative INDUSTRIE 4.0.** Acatech, n. April, p. 13-78, 2013.

KARNOUSKOS, S.; COLOMBO, A.W.; BANGEMANN, T.; MANNINEN, K.; CAMP, R.; TILLY, M.; SIKORA, M.; JAMMES, F.; DELSING, J.; ELIASSON, J.; NAPPEY, P.; HU, J.; GRAF, M. The IMC-AESOP **Architecture for Cloud-Based Industrial Cyber-Physical Systems**. In: Colombo A. et al. (eds) Industrial Cloud-Based Cyber-Physical Systems. Springer, Cham, 2014.

KOKKONIS, G.; CHATZIMPARMPAS, A.; KONTOGIANNIS, S. **Middleware IoT protocols performance evaluation for carrying out clustered data**. South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference, SEEDA_CECNSM 2018. Anais...Institute of Electrical and Electronics Engineers Inc., 26 nov. 2018.

LEITÃO, P.; COLOMBO, A.W.; KARNOUSKOS, S. **Industrial automation based on cyber-physical system Technologies:** Prototype implementations and challenges. Computers in Industry. Vol. 81, September 2016, pp. 11-25.

LEWIS, J; FOWLER, M. **Microservices**. Martin Fowler, 2014. Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em: 10 abr. 2020.

LU, Y. Industry 4.0: **A survey on technologies, applications and open research issues**. Journal of Industrial Information Integration, v. 6, p. 1–10, 2017.

MAKSUTI, S.; TAUBER, M.; DELSING, J. **Generic Autonomic Management as a Service in a SOA-based Framework for Industry 4.0**. IECON Proceedings (Industrial Electronics Conference). Anais...IEEE Computer Society, 1 out. 2019.

MAYA. (2019). **The MAYA Project**. Disponível em: http://www.maya-euproject.com/. Acesso em: Janeiro, 2020.

MOLECULER. (2020). **Fast & powerful microservices framework for Node.js**. Disponível em: https://moleculer.services/. Acesso em: janeiro, 2020.

MORAES, E. C. **Desenvolvimento de padrões de interfaces para integração de sistemas heterogêneos na manufatura**. Salvador, 2017. 247f. Tese (Doutorado - Engenharia Industrial) – Universidade Federal da Bahia, PEI, 2017.

NAIK, N. **Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP**. 2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings. Anais...Institute of Electrical and Electronics Engineers Inc., 26 out. 2017

NEHME, A. et al. **Securing Microservices**. IT Professional, v. 21, n. 1, p. 42–49, 1 jan. 2019.

NEWMAN, S. **Building Microservices**. O'Reilly Media, Inc. CA: Gravenstein, 2015.

NIMBLE. **The NIMBLE Project**. Disponível em: https://www.nimble-project.org/. Acesso em: janeiro, 2020.

PAHL, M. O.; DONINI, L. **Securing IoT microservices with certificates**. IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018. Anais...Institute of Electrical and Electronics Engineers Inc., 6 jul. 2018.

PM2. Disponível em: <https://pm2.keymetrics.io/>. Acesso em: 19 fev. 2020.

PRODUCTIVE 4.0. **Productive 4.0 Project Website**. Disponível em: https://productive40.eu/. Acesso em: abril, 2020.

Raspberry Pi Stackable Card for Industrial Automation. Disponível em: <https://sequentmicrosystems.com/index.php?route=product/product&path=20&product_id=52>. Acesso em: 19 jun. 2020.

RAZA, M. et al. **A Digital Twin Framework for Industry 4.0 Enabling Next-Gen Manufacturing**. ICITM 2020 - 2020 9th International Conference on Industrial Technology and Management. Anais...Institute of Electrical and Electronics Engineers Inc., 1 fev. 2020

RICHARDSON, C.; SMITH, F. **Microservices From Design to Deployment**. NGINX, 2016. Disponível em: https://www.nginx.com/resources/library/designing-deploying-microservices/. Acesso em: Janeiro, 2020

SANDERS, A.; ELANGESWARAN, C.; WULFSBERG, J. **Industry 4.0 implies lean manufacturing: research activities in industry 4.0 function as enablers for lean manufacturing**. Journal of Industrial Engineering and Management, v. 9, n. 3, p. 811-833, 2016.

SHENG, Q. Z.; QIAO, X.; VASILAKOS, A. V.; SZABO, C.; BOURNE, S.; XU, X. **Web services composition:** A decade's overview. Information Sciences, v. 280, October 2014, pp. 218– 238.

SISINNI, Emiliano et al. **Industrial internet of things: challenges, opportunities, and directions**. IEEE Transactions on Industrial Informatics, Piscataway, v. 14, n. 11, p. 4724-4734, 2018. DOI: http://dx.doi.org/10.1109/tii.2018.2852491. Disponível em: https://ieeexplore.ieee.org/document/8401919 Acesso em: 16 ago. 2018.

SONG, J.; MOK, A.; CHEN, C.; NIXON, M.; BLEVINS, T.; WOJSZNIS, W. **Improving PID control with unreliable communications.** *Paper presented at the ISA EXPO Technical Conference.* Houston, Texas, October 17-19, 2006.

SOUIT, S.; FATTORI, C. C.; JUNQUEIRA, F.; FILHO, D. J. S.; MIYAGI, P. E. **Orchestrating dispersed productive systems**. In: IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, Anais.2013.

STOCK, T.; SELIGER, G. **Opportunities of Sustainable Manufacturing in Industry 4.0**. Procedia CIRP. Anais. Elsevier B.V., 1 jan. 2016.

KOKKONIS, G.; CHATZIMPARMPAS, A.; KONTOGIANNIS, S. **Middleware IoT protocols performance evaluation for carrying out clustered data**. South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference, SEEDA_CECNSM 2018. Anais...Institute of Electrical and Electronics Engineers Inc., 26 nov. 2018.

SUEDA, Y.; SATO, M.; HASUIKE, K. **Evaluation of Message Protocols for IoT**. Proceedings - 2019 IEEE/ACIS 4th International Conference on Big Data, Cloud Computing, and Data Science, BCD 2019. **Anais**...Institute of Electrical and Electronics Engineers Inc., 1 maio 2019.

SUN, Y.; NANDA, S.; JAEGER, T. **Security-as-a-service for microservices-based cloud applications**. Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015. Anais...Institute of Electrical and Electronics Engineers Inc., 1 fev. 2016.

WANG, L.; WANG, G. **Big data in cyber-physical systems, digital manufacturing and Industry 4.0**. International Journal of Engineering and Manufacturing (IJEM), v. 6, n. 4, p. 1-8, 2016.

WOLLSCHLAEGER, M.; SAUTER, T.; JASPERNEITE, J. **The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0.** IEEE Industrial Electronics Magazine, [s.l.], v. 11, n. 1, p.17-27, mar. 2017. Institute of Electrical and Electronics Engineers (IEEE). DOI: 10.1109/mie.2017.2649104.

XIAO, Z.; WIJEGUNARATNE, I.; QIANG, X. **Reflections on SOA and Microservices**. In: 4th IEEE International Conference on Enterprise Systems, 2016, pp. 60-67.

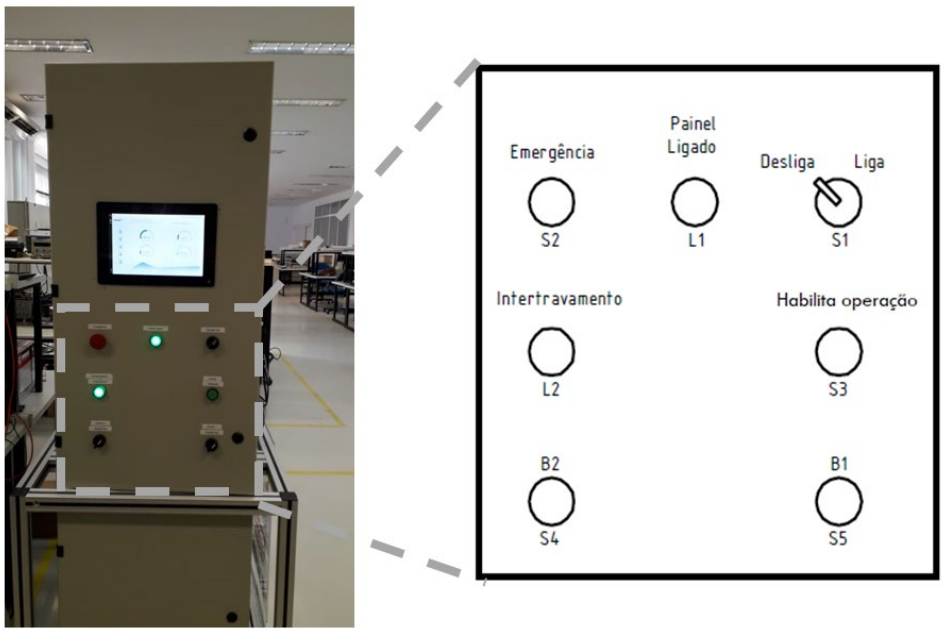
8 APÊNDICE

8.1 Procedimento de Operação da Planta Piloto

A fim de operar a planta piloto, é necessário energizá-la com tensão de alimentação de 220 VAC e acionar a chave geral de energia. Logo após, utilizando-se os botões e lâmpadas indicadoras presentes no painel de controle da planta que pode ser visto na Figura 73, deve-se seguir os passos:

1. Acionar a chave seletora do painel Desliga/Liga. Com essas configurações selecionadas, os indicadores painel ligado (L1) e intertravamento nível mínimo (L2) devem ficar acesos;
2. O nível mínimo garante que as bombas não sejam acionadas sem uma quantidade mínima de água para circular na tubulação, essa proteção é feita com uma chave de nível. Garantido o nível mínimo, basta pressionar o botão Habilita Operação (S3);
3. O controle individual de Liga/Desliga das duas bombas é feito através das duas chaves seletoras da bomba 1 (S5) e bomba 2 (S4). Acionando ambas as chaves na posição de Liga, as bombas irão funcionar em uma frequência de operação baixa (3Hz), estando prontas para rodarem em sua faixa de operação máxima ou mínima (0 a 100%) através de requisições dos microsserviços.

Figura 73 - Painel de Frontal de Controle da Planta Piloto



Fonte: Autor