

Wagner Augusto Aranda Cotta

**Medição de tempo de comunicação e exibição  
de tempo de resposta para espaços inteligentes  
programáveis**

Brasil

2020

Wagner Augusto Aranda Cotta

**Medição de tempo de comunicação e exibição de tempo  
de resposta para espaços inteligentes programáveis**

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica.

Universidade Federal do Espírito Santo

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Anilton Salles Garcia

Coorientadora: Raquel Frizera Vassallo

Brasil

2020

Ficha catalográfica disponibilizada pelo Sistema Integrado de  
Bibliotecas - SIBI/UFES e elaborada pelo autor

---

C846 Cotta, Wagner Augusto Aranda, 1989-  
m Medição de tempo de comunicação e exibição de tempo de  
resposta para espaços inteligentes programáveis / Wagner  
Augusto Aranda Cotta. - 2020.  
87 f. : il.

Orientador: Anilton Salles Garcia.  
Coorientadora: Raquel Frizera Vassallo.  
Dissertação (Mestrado em Engenharia Elétrica) -  
Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Sistemas Distribuídos. 2. Arquitetura orientada a serviços.  
3. Internet das coisas. 4. Medição de software. I. Garcia, Anilton  
Salles. II. Vassallo, Raquel Frizera. III. Universidade Federal do  
Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 621.3

---

Wagner Augusto Aranda Cotta

## Medição de tempo de comunicação e exibição de tempo de resposta para espaços inteligentes programáveis

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica.

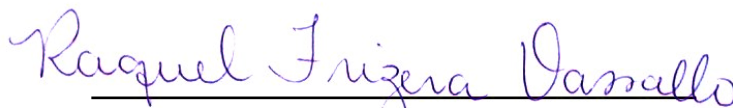
Trabalho aprovado em Vitória, 15 de julho de 2020.

**ANILTON SALLES**  
**GARCIA:39523799720**

Assinado de forma digital por ANILTON SALLES  
GARCIA:39523799720  
Dados: 2020.11.23 16:01:51 -03'00'

---

**Prof. Dr. Anilton Salles Garcia**  
Universidade Federal do Espírito Santo  
Orientador



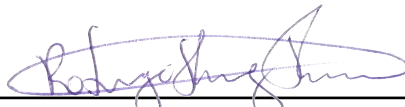
---

**Profa. Dra. Raquel Frizera Vassallo**  
Universidade Federal do Espírito Santo  
Coorientadora



---

**Prof. Dr. Rodolfo da Silva Villaca**  
Universidade Federal do Espírito Santo  
Examinador Externo



---

**Dr. Rodrigo Stange Tessinari**  
Universidade de Bristol  
Examinador Externo

Brasil  
2020

*Trabalho dedicado a todos que habitam meu universo particular.*

# Agradecimentos

Sempre achei esta a pior parte da dissertação para escrever, talvez porque a vida não seja algo exato onde, através de uma fórmula matemática, descobrimos a significância das pessoas na nossa trajetória. Mas, espero representar nestes próximos parágrafos um simbolismo da minha gratidão.

Primeiro de tudo, gostaria de agradecer a Deus por me guiar, iluminar e me dar determinação para sempre seguir em frente com os meus objetivos e não desistir face as dificuldades.

Agradeço ao meu irmão e, principalmente, aos meus pais, que sempre me motivaram, entenderam as minhas faltas, momentos de afastamento e reclusão e me mostraram o quanto era importante estudar. Agradeço também por todas as lições de amor, companheirismo, amizade, caridade, dedicação, compreensão e perdão que me dão a cada novo dia. Sinto-me orgulhoso e privilegiado por ter uma família tão especial.

Ao meu orientador, Anilton Salles Garcia, agradeço pela confiança antes mesmo de ser aprovado no processo seletivo do mestrado, por me guiar nos primeiros passos da pós-graduação e por todos os ensinamentos compartilhados.

Ao Alexandre Pereira do Carmo e à Raquel Frizera Vassallo, primeiramente pelo meu acolhimento na família do laboratório (até então chamado de VIROS) e pela oportunidade de realizar este trabalho. Além disso, agradeço pela orientação, competência, profissionalismo e dedicação que tiveram comigo. Obrigado por acreditarem em mim e pelos incentivos. Tenho certeza que não chegaria neste ponto sem o apoio de vocês. Não posso deixar de agradecer também pelas festas, passeios e programas de índio que, certamente, foram ótimos momentos para curtir e aliviar a cabeça. Vocês contribuíram muito para quem eu sou hoje, tanto como pessoa, quanto como profissional e posso afirmar que, independente dos caminhos da vida, sempre manteremos contato.

A todos os outros membros da família do laboratório, que vivenciaram comigo momentos de aprendizado, de trabalho, de escrita de artigo, de tensão e de divertimento. Em especial, a todos que trabalharam e/ou me ajudaram diretamente no decorrer desta jornada.

Ao AVES (Aventureiros do ES), por me proporcionar, através das aventuras, momentos de lazer e relaxamento durante esse período. Fazer rapel, trilhas e viagens com vocês foi (e ainda é) a melhor válvula de escape da rotina. São tantas pessoas queridas para agradecer que seria injusto não mencionar alguém, portanto, sintam-se todos abraçados em agradecimento.

À Blizzard e Steam por me proporcionarem momentos de distração e lazer com amigos, em meio à pandemia. E, em especial, ao core do WoW (da criticância, do leveling, das míticas cansadas, "das preda quebrada"): Lucas - o bêbado tank que avisa, faz contagem e até manda dbm antes do pull (coisas que não faz quando sóbrio). Jerry - o cara que sabe tudo de WoW, entretanto, "hope you die, rogue"; Samuel - parceria de longa data e responsável por me trazer de volta,. Se tem uma frase que eu posso falar sobre ele é: "everybody suc\* as\*, except grinde"; Thay - a mage que faz pull extra e aggra tudo o que vê pela frente, dos lados, atrás... Aggra até mesmo os mobs friendly, que são impossíveis de aggrar. Sempre contando o número de pulls para pegar o loot certo na hora dela; Dani - a paladina pata que finge que estuda, nunca joga e só aparece para falar: "AaALôÔô" no voice; Por último, mas não menos importante, Morandi - o healer mais calmo que existe e o único a falar: "se curem aí", sendo essa a função dele.

"Ty all a todos!"

Por fim, agradeço ao CNPq e ao PPGEE que me concederam a bolsa de mestrado e a todas outras pessoas que passaram por minha vida nessa fase e contribuíram de alguma forma.

Muito obrigado!

*Wagner Augusto Aranda Cotta*

*"We have developed speed, but we have shut ourselves in.  
Machinery that gives abundance has left us in want.  
Our knowledge has made us cynical.  
Our cleverness, hard and unkind.  
We think too much and feel too little.  
More than machinery we need humanity.  
More than cleverness we need kindness.  
Without these qualities, life will be violent and all will be lost."  
(Charlie Chaplin, The Great Dictator)*



# Resumo

O conceito de cidades inteligentes como área urbanizada onde vários setores cooperam para alcançar resultados sustentáveis traz agregado, direta e indiretamente, uma grande quantidade de tecnologias e conceitos. Uma cidade inteligente pode ser estudada através da correspondência com um espaço inteligente programável, que é um espaço físico equipado com uma rede de sensores, que obtém informações sobre o mundo que observa, e uma rede de atuadores, que permite sua interação com os usuários e a modificação do próprio ambiente através de serviços de computação. Para enfrentar os desafios emergentes da implementação desse espaço, os paradigmas de sistemas distribuídos, juntamente com a arquitetura de Internet das Coisas, vem sendo amplamente utilizados devido às suas vantagens. Assim, no Laboratório de Visão Computacional e Robótica da Universidade Federal do Espírito Santo (UFES) foi criado um Espaço Inteligente Programável utilizando-se uma arquitetura IoT baseada em *broker* e um sistema de forma distribuída, onde os conceitos de cidades inteligentes encontram-se aplicados. Algumas aplicações desenvolvidas para esse espaço inteligente utilizam câmeras como principal sensor e possuem, como um dos seus requisitos, o funcionamento em tempo real. Dessa forma, é levantada a questão da sensibilidade ao tempo de resposta, onde, para um perfeito funcionamento, o sistema tem uma janela de tempo para uma execução de todos os serviços em cadeia. O tempo de resposta pode ser observado utilizando ferramentas de *tracing*, quando se trata de processamento, contudo no que diz respeito à comunicação, há uma ausência de ferramentas específicas, o que deixa claro a necessidade do desenvolvimento de uma maneira capaz de medir a comunicação, especialmente para microsserviços. Sabendo da importância das medições na depuração de tais aplicações e da falta de ferramentas que realizam uma medição completa do tempo de resposta, nesta dissertação de mestrado é apresentada uma proposta de solução que realiza a medição da comunicação em um sistema distribuído baseado em microsserviços. Além disso, tal informação é agregada junto do tempo medido no processamento por outros meios, para que o tempo de resposta seja mais fielmente mensurado e exibido em uma ferramenta de *tracing*. Para fins de validação, a proposta foi aplicada em duas situações: utilizando serviços criados para depuração e utilizando uma aplicação real do espaço inteligente. Ao final, uma discussão é feita envolvendo as conclusões alcançadas com este trabalho e melhorias para a solução proposta, enumerando possíveis trabalhos futuros. **Palavras-chave:** Cidades inteligentes, sistemas distribuídos, internet das coisas, microsserviços.

# Abstract

The concept of smart cities as an urbanized area where various sectors cooperate to achieve sustainable results brings, directly and indirectly, a large amount of technologies and concepts. A smart city can be studied through correspondence with a programmable intelligent space, which is a physical space equipped with a network of sensors, which obtains information about the observed environment, and a network of actuators, which allows its interaction with users and the environment through computer services. To deal with the emerging challenges of this type of space, the distributed systems paradigms, along with the Internet of Things architecture, have been widely used due to their advantages. Thus, at the Vision and Robotic Systems Laboratory of the Federal University of Espírito Santo, a Programmable Intelligent Space was created using a broker-based IoT architecture and a distributed system, where the concepts of smart cities are applied. Some applications developed for this intelligent space should operate in real time. In this way, the matter of sensibility to response time is raised, where, for perfect functioning, the system has a time window to run all services in the chain. The response time can be observed using tracing tools, when it comes to processing, however with regard to communication, there is a lack of specific tools, which makes clear the need to develop a way to measure communication, especially for microservices. Knowing the importance of measurements in the debugging of such applications and the lack of tools that perform a complete measurement of the response time for the smart space, this master's thesis proposes a solution that performs the measurement of communication in a distributed system based on microservices. Furthermore, this information is combined with the processing time measured by other tools, so that the response time can be more accurately measured and displayed in the tracing tool. For validation purposes, the proposal was applied in two situations: using services created for debugging and using a real application of the intelligent space. At the end, a discussion is made involving the conclusions reached with this work and improvements to the proposed solution, enumerating possible future works.

**Keywords:** Smart cities, distributed systems, internet of things, microservices.

# Lista de ilustrações

Figura 1 – Representação simplificada em camadas da arquitetura do PIS. . . . .	24
Figura 2 – A Internet: exemplo de sistema distribuído físico e virtual. . . . .	30
Figura 3 – Exemplo de <i>cluster</i> . . . . .	31
Figura 4 – Representação das arquiteturas monolíticas e microserviço . . . . .	32
Figura 5 – Exemplo de <i>trace</i> no tempo . . . . .	34
Figura 6 – Exemplo de <i>trace</i> em cadeia no tempo . . . . .	35
Figura 7 – Exemplo de árvore de <i>span</i> . . . . .	35
Figura 8 – Diagrama de eventos no tempo em situação normal e com erro . . . . .	37
Figura 9 – Hierarquia dos serviços de fornecimento de sincronização NTP via Internet	40
Figura 10 – Sincronismo entre mestre e escravo utilizando PTP . . . . .	42
Figura 11 – Exemplo de funcionamento da ferramenta <i>PING</i> . . . . .	45
Figura 12 – Exemplos de utilização de RTT para medições. . . . .	46
Figura 13 – Medição passiva de OWD . . . . .	48
Figura 14 – Visão geral da medição do tempo de resposta de um sistema após a aplicação desta metodologia . . . . .	58
Figura 15 – Exemplo de <i>trace</i> após a aplicação desta metodologia . . . . .	58
Figura 16 – Estrutura de servidores utilizada para sincronização dos relógios . . . . .	65
Figura 17 – Arquitetura em camadas do PIS . . . . .	66
Figura 18 – Testes de funcionamento da Solução 1 . . . . .	67
Figura 19 – Serviço de medição de comunicação de <i>traces</i> por recebimento de times- tamps . . . . .	68
Figura 20 – Modificação da camada <i>middleware</i> de bibliotecas para medição de tempo de comunicação . . . . .	69
Figura 21 – Modificação da camada <i>middleware is-wire</i> para medição de tempo de comunicação . . . . .	70
Figura 22 – Lacunas após as medições do tempo de comunicação . . . . .	71
Figura 23 – Programa criado para detalhamento do tracing . . . . .	72
Figura 24 – Adição do serviço extra para publicação dos <i>traces</i> de comunicação . . . . .	73
Figura 25 – Exibição do <i>trace</i> sem a utilização do serviço de publicação . . . . .	73
Figura 26 – Exibição do <i>trace</i> com a utilização do serviço de publicação . . . . .	74
Figura 27 – Esquemático do funcionamento dos três primeiros experimentos . . . . .	75
Figura 28 – Exemplo de uma amostra do resultado do primeiro experimento . . . . .	76
Figura 29 – Resultado do quarto experimento . . . . .	80
Figura 30 – Exemplo de mapa de ocupação . . . . .	81
Figura 31 – Representação de todos os serviços necessários para gerar o mapa de ocupação . . . . .	81

Figura 32 – Rastreamento realizado em todos os serviços da aplicação mapa de  
ocupação . . . . . 83

# Lista de quadros

Quadro 1 – Comparativo entre as ferramentas de <i>tracing</i> . . . . .	36
Quadro 2 – Comparativo entre os métodos de sincronização de relógio . . . . .	43
Quadro 3 – Configurações pré-definidas do broker . . . . .	63
Quadro 4 – Nomenclaturas convencionadas para os experimentos . . . . .	74
Quadro 5 – Médias das medições dos serviços no primeiro experimento . . . . .	77
Quadro 6 – Médias das medições dos serviços no segundo experimento . . . . .	77
Quadro 7 – Médias das medições dos serviços no terceiro experimento . . . . .	78
Quadro 8 – Médias das medições dos serviços na aplicação mapa de ocupação . . . . .	82

# Lista de abreviaturas e siglas

AMQP	<i>Advanced Message Queuing Protocol</i>
API	Interfaces de Programação de Aplicação
BMC	<i>Best Master Clock</i>
CPU	Unidade Central de Processamento
GLONASST	<i>GLONASS Time</i>
GNSS	<i>Global Navigation Satellite Systems</i>
GPS	Sistema de posicionamento global
GPST	<i>GPS Time</i>
GST	<i>Galileo System Time</i>
HTTP	<i>Hyper Text Transfer Protocol</i>
ICMP	<i>Internet Control Message Protocol</i>
IDE	Ambiente de Desenvolvimento Integrado
IIoT	IoT para a Indústria
IP	Protocolo da Internet
ISP	<i>Internet Service Provider</i>
IoT	Internet das Coisas
JSON	<i>JavaScript Object Notation</i>
LANs	<i>Local Area Networks</i>
MASER	Amplificador de Micro-ondas por Emissão Estimulada de Radiação
NTP	<i>Network Time Protocol</i>
OWD	<i>One-Way Delay</i>
PIS	Espaço Inteligente Programável
PPGEE	Programa de Pós-Graduação em Engenharia Elétrica

PTP	<i>Precision Time Protocol</i>
REST	<i>Representational State Transfer</i>
RTT	<i>Round-Trip Time</i>
TAI	Tempo Atômico Internacional
TIC	Tecnologia da Informação e Comunicação
UDP	<i>User Datagram Protocol</i>
UFES	Universidade Federal do Espírito Santo
UTC	Tempo Universal Coordenado

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>21</b>
<b>1.1</b>	<b>Definição do problema</b>	<b>25</b>
<b>1.2</b>	<b>Motivação</b>	<b>26</b>
<b>1.3</b>	<b>Objetivos e proposta da dissertação</b>	<b>27</b>
1.3.1	Objetivos Gerais	27
1.3.2	Objetivos Específicos	27
1.3.3	Proposta	27
<b>1.4</b>	<b>Estrutura da dissertação</b>	<b>28</b>
<b>2</b>	<b>REFERENCIAL TEÓRICO E ESTADO DA ARTE</b>	<b>29</b>
<b>2.1</b>	<b>Referencial Teórico</b>	<b>29</b>
2.1.1	Sistemas Distribuídos	29
2.1.2	Microserviços	32
2.1.3	<i>Tracing</i>	33
2.1.3.1	Ferramentas para <i>Tracing</i>	36
2.1.4	Sincronização de relógios	36
2.1.4.1	<i>Global Navigation Satellite Systems</i> (GNSS)	38
2.1.4.2	<i>Network Time Protocol</i> (NTP)	39
2.1.4.3	<i>Precision Time Protocol</i> (PTP)	41
2.1.4.4	Comparativo entre os métodos	42
2.1.5	Medição de latência	43
2.1.5.1	Round-Trip Time (RTT)	44
2.1.5.2	<i>One-Way Delay</i> (OWD)	47
<b>2.2</b>	<b>Estado da Arte</b>	<b>49</b>
2.2.1	Considerações	51
<b>3</b>	<b>PROPOSTA DE SOLUÇÃO</b>	<b>53</b>
<b>3.1</b>	<b>Metodologia</b>	<b>53</b>
3.1.1	Definição preliminar	53
3.1.1.1	Método de medição a ser adotado	53
3.1.1.2	Camada da arquitetura na qual a medição será implementada	54
3.1.1.3	Aspectos da medição em relação ao elemento coordenador das trocas de mensagens	54
3.1.1.4	Ferramenta de <i>tracing</i> a ser utilizada	55
3.1.2	Sincronização de relógios	55
3.1.3	Modificação do ambiente e ferramentas de <i>tracing</i>	56
3.1.4	Avaliação das medições	59



3.1.5	Mitigação de imprevistos na ferramenta de <i>tracing</i> . . . . .	59
<b>4</b>	<b>IMPLEMENTAÇÃO, EXPERIMENTOS E RESULTADOS . . . . .</b>	<b>61</b>
<b>4.1</b>	<b>Implementação . . . . .</b>	<b>61</b>
4.1.1	Recursos . . . . .	61
4.1.2	Definição preliminar . . . . .	62
4.1.2.1	Método de medição a ser adotado . . . . .	62
4.1.2.2	Camada da arquitetura na qual a medição será implementada . . . . .	63
4.1.2.3	Aspectos da medição em relação ao elemento coordenador das trocas de mensagens . . . . .	63
4.1.2.4	Ferramenta de <i>tracing</i> a ser utilizada . . . . .	64
4.1.3	Sincronização de relógios . . . . .	64
4.1.4	Modificação do ambiente e ferramentas de <i>tracing</i> . . . . .	66
4.1.4.1	Solução 1 . . . . .	67
4.1.4.2	Solução 2 . . . . .	68
4.1.4.3	Solução 3 . . . . .	70
4.1.5	Avaliação das medições . . . . .	71
4.1.6	Mitigação de imprevistos na ferramenta de <i>tracing</i> . . . . .	72
<b>4.2</b>	<b>Experimentos . . . . .</b>	<b>74</b>
<b>4.3</b>	<b>Aplicação: Serviços de Depuração . . . . .</b>	<b>75</b>
<b>4.4</b>	<b>Aplicação: Mapa de ocupação . . . . .</b>	<b>80</b>
<b>5</b>	<b>CONSIDERAÇÕES FINAIS . . . . .</b>	<b>85</b>
<b>5.1</b>	<b>Conclusão . . . . .</b>	<b>85</b>
<b>5.2</b>	<b>Trabalhos Futuros . . . . .</b>	<b>86</b>
5.2.1	Implementação em novas linguagens . . . . .	86
5.2.2	Medição através do elemento coordenador da troca de mensagens . . . . .	86
5.2.3	Continuação da Solução 1 . . . . .	87
5.2.4	Criação da biblioteca de medição OWD . . . . .	87
	<b>REFERÊNCIAS . . . . .</b>	<b>89</b>

# 1 Introdução

Na década passada, o conceito de cidades inteligentes despertou grande interesse, tanto da academia quanto da indústria, como uma maneira de superar os desafios relacionados à rápida e crescente urbanização. Segundo (GARTNET, 2012), uma cidade inteligente é uma área urbanizada onde vários setores cooperam para alcançar resultados sustentáveis, por meio da análise de informações contextuais em tempo real. Tal conceito também pode ser enunciado como o uso da Tecnologia da Informação e Comunicação (TIC), que escala, organiza e integra processos e informações, para melhorar a infraestrutura, otimizar a mobilidade urbana e criar soluções sustentáveis que resultem em um aumento da qualidade de vida (LAW; LYNCH, 2019). Além disso, as cidades inteligentes também possuem o desafio de trabalhar os diferentes perfis de cidadãos e a viabilidade de projetos, tendo em vista principalmente o futuro e não somente os benefícios a curto prazo.

Com o objetivo de solucionar problemas urbanos, a utilização de estratégias e tecnologias dentro do conceito de cidades inteligentes tem sido aplicada em várias partes do mundo. Devido à democratização da Internet, essa temática ganhou forças e, atualmente, é um dos assuntos mais comentados entre os governos, pois, além do planejamento urbano, tornou-se necessário o investimento em soluções tecnológicas que possam ser aceitas e utilizadas pelos residentes de cada cidade.

Uma cidade inteligente é fortemente baseada em sensores que fornecem informações atualizadas sobre diversas variáveis, incluindo temperatura, umidade, poluição e até mesmo imagens. Os sistemas de aquisição de dados, juntamente com os estados do ambiente e suas configurações, é que determinam o comportamento de uma aplicação ou evento que seja interessante para o usuário. Com isso, entende-se que a ampla utilização de sensores dentro desse âmbito pode, também, descrever a cidade inteligente como um grande sistema de sistemas.

Embora a rápida urbanização proporcione diversas oportunidades para a aplicação do conceito supracitado, a implementação da infraestrutura de cidades inteligentes ainda apresenta muitos desafios que demandam pesquisas, tais como redes de sensoriamento em larga escala, necessidade de alto poder de processamento, necessidade de baixa latência em determinadas aplicações, escalonamento de recursos, comunicação máquina-a-máquina, etc.

Paralelamente a isso, nos últimos anos, o rápido avanço tecnológico nas áreas de eletrônica, computação e sistemas de comunicação mudou a maneira como o homem interage com o ambiente em sua volta. Dispositivos cada vez mais modernos e autônomos foram desenvolvidos e dotados com capacidade de inferir, tomar decisões e realizar tarefas

cada vez mais complexas. Este mesmo avanço tecnológico propiciou que dispositivos simples como medidores, microfones ou interruptores, que antes não possuíam nenhuma inteligência, ganhassem a capacidade de processamento e comunicação, aumentando sua interação com o usuário.

Com isso, iniciaram-se pesquisas que têm buscado associar o sensoramento e inteligência ao ambiente, permitindo o uso de tais dispositivos, com alguma capacidade interativa, para a realização de tarefas mais amplas e complexas no ambiente. Desse modo, a inteligência, a percepção e os componentes de ação (ou atuadores) são distribuídos em rede para construir um ambiente que forneça informações a alguns dispositivos complexos permitindo-lhes responder a uma ampla gama de eventos.

Tais ambientes são chamados de Espaços Inteligentes ([LEE; ANDO; HASHIMOTO, 1999](#)), cuja inspiração nasceu da computação ubíqua. A filosofia da computação ubíqua é que a computação esteja imersa no ambiente ocupado por pessoas, permitindo que sua interação seja realizada de forma natural ([WEISER, 1991](#)).

Assim, um Espaço Inteligente pode ser definido como um espaço físico equipado com uma rede de sensores, que obtém informações sobre o mundo que observa, e uma rede de atuadores, que permite sua interação com os usuários e a modificação do próprio ambiente através de serviços de computação. Dessa forma, sensores, atuadores e serviços de computação são governados por uma infraestrutura capaz de coletar e analisar as informações obtidas pelos sensores e tomar decisões. Diante disso, pode-se citar a utilização de Espaços Inteligentes como uma alternativa tecnológica para se corresponder ao conceito de Cidades Inteligentes.

Ainda nesse contexto, para enfrentar os desafios emergentes da implementação de tal infraestrutura, os paradigmas de sistemas distribuídos vem sendo amplamente utilizados devido às suas vantagens básicas: diminuição do tempo de execução; aumento da disponibilidade e do grau de confiabilidade; organização dos componentes por função específica ([TANENBAUM, 2008](#)).

Os sistemas distribuídos destacam-se com base na confiabilidade, no desempenho e escalonamento de recursos de acordo com a necessidade da aplicação ([TANENBAUM, 2008](#)). O conjunto atua de maneira tão eficaz que o usuário final é abstraído da segmentação, seja ela de hardware ou software, e acaba sendo notório apenas a melhoria de desempenho ao executar as tarefas.

Além disso, outra omissão do sistema para com o usuário se dá no que diz respeito às falhas. Devido a uma enorme tolerância a erros, o sistema possivelmente apresentará alguma perda de desempenho devido à nova alocação de recursos ou congestionamento na rede, entretanto, sem prejuízos no funcionamento, muitas vezes tal comportamento acaba sendo imperceptível ou resulta apenas em um atraso no recebimento da resposta

pelo usuário final.

Para que ocorra dessa forma, dentro de um sistema distribuído, deve-se haver, em diferentes servidores, várias instâncias de um mesmo serviço que rodam de maneira independente, focados em um único e pequeno conjunto de atividades dentro de um conjunto de serviços maior, ou seja, ter a arquitetura baseada em microsserviços. Assim, caso ocorra problemas em alguma dessas instâncias, o sistema como um todo não deixa de realizar suas funções. Pois, as requisições direcionadas ao serviço defeituoso são redirecionadas a outro serviço capaz de assumi-las de forma transparente.

Visando contribuir para o enfrentamento dos desafios emergentes da implementação de um Espaço Inteligente juntamente com os paradigmas de sistema distribuído, vê-se a arquitetura de Internet das Coisas como um conceito altamente compatível.

A Internet das Coisas (IoT) é vista por diversos pesquisadores ([ANDREEV et al., 2015](#))([ALI et al., 2017](#))([MROUE et al., 2018](#)) como principal ascendente na indústria de TIC, onde incorpora a visão de conectar virtualmente qualquer coisa com tudo e disseminar uma extensa quantidade de informação em curtos espaços de tempo ([LIBERG et al., 2018](#)). Múltiplos sensores conectados, tratados também como Coisas conectadas, geram uma grande quantidade de dados que, posteriormente, se transformam em conhecimento e informação. Diante dessa visão, não é interessante que esse processo de criação do conhecimento seja tratado de forma totalmente centralizada e, por isso, deve ser combinada com a computação distribuída para que o custo da transmissão das informações seja reduzida pelo balanceamento da carga de processamento entre os dispositivos ([PRIETO et al., 2018](#)). Sendo assim, as distintas características das tecnologias IoT e os sistemas distribuídos apoiam-se mutuamente para encontrar um ponto ótimo em relação ao processamento e comunicação desses dispositivos.

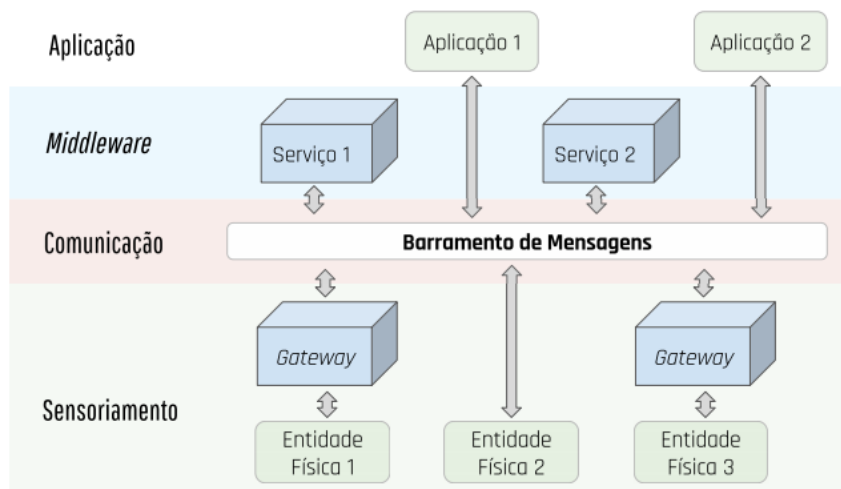
Para tornar esse processo de criação do conhecimento viável, são necessários dispositivos e meios de comunicações que realizem a conexão de todos os elementos. Assim torna-se necessária uma infraestrutura compatível e dimensionável que seja capaz de absorver o aumento exponencial do número de dispositivos e gerenciar a grande quantidade de informações transportadas ([MROUE et al., 2018](#)). Tal necessidade pode ser suprida ao se utilizar, conjuntamente, uma arquitetura IoT e um sistema distribuído, pois enquanto o sistema distribuído busca cumprir os requisitos de processamento, a arquitetura IoT fica responsável pela comunicação entre os dispositivos através de elementos denominados *brokers*, que são unicamente responsáveis pela coordenação da comunicação entre os clientes, sejam estes objetos ou serviços. Em geral, os *brokers* permitem a comunicação entre seus clientes no formato *publish-subscribe*, onde são recebidas mensagens dos publicantes (*publishers*) e reencaminhadas para os assinantes (*subscribers*) cadastrados em determinado tópico.

No Laboratório de Visão Computacional e Robótica do Programa de Pós-Graduação

em Engenharia Elétrica (PPGEE) da Universidade Federal do Espírito Santo (UFES) foi criado um Espaço Inteligente Programável (PIS) utilizando-se uma arquitetura IoT baseada em *broker* e um sistema de forma distribuída, onde os conceitos de cidades inteligentes, anteriormente expostos, encontram-se aplicados.

O referido Espaço Inteligente Programável adota o modelo de camadas apresentado na Figura 1, onde a camada de aplicação provê uma interface para o desenvolvimento das aplicações desse espaço (QUEIROZ, 2016). Na camada de *middleware*, estão os diversos serviços oferecidos às aplicações, podendo ser do tipo que oferecem comunicação ou ainda serviços específicos de processamento. A camada de comunicação é responsável por permitir, através de um *broker*, a troca de mensagens entre os microserviços e, por fim, a camada de sensoriamento é responsável por expor os recursos do mundo físico para o mundo virtual.

Figura 1 – Representação simplificada em camadas da arquitetura do PIS.



Fonte: (QUEIROZ, 2016).

Em algumas aplicações desenvolvidas neste PIS (RAMPINELLI et al., 2014)(CARMO et al., 2019)(ALMONFREY et al., 2018)(PICORETI et al., 2018), as câmeras são os principais sensores do ambiente, o que caracteriza a utilização de um número menor de sensores, porém cada um desses gera um grande volume de dados, os quais devem ser coletados, agregados e analisados para extrair as informações necessárias para a atuação no ambiente. Diante dessas aplicações, cujo sensores são do tipo câmera, percebe-se que o requisito comum a elas é a resposta em tempo real e, para atingir esse objetivo, por exemplo, há a necessidade de prover alta capacidade de processamento computacional, bem como eficiente transmissão dos dados.

Um tempo de resposta pode ser definido como a composição do tempo de processamento dentro dos diversos serviços adicionado ao tempo de comunicação entre eles. Sendo que o tempo de processamento é o tempo gasto pelo programa para executar um bloco definido de instruções dentro de um serviço e o tempo de comunicação é o

intervalo de tempo decorrido do instante em que a função de envio de dados é chamada até o recebimento completo dos dados pelo destinatário. Com isso, há trabalhos, por exemplo (PICORETI et al., 2018), onde o tempo gasto no processamento das funções é influenciado pela infraestrutura através de mecanismos de orquestração com *feedback* de métricas da própria aplicação. De uma maneira geral, para medir o tempo de processamento dentro de um serviço, há diversas ferramentas e técnicas como *monitoring*, *logging* e *tracing* para observar e gerar registros dos trabalhos realizados em um sistema baseado em microsserviços.

O *monitoring* consiste em medir aspectos como utilização da Unidade Central de Processamento (CPU), o uso do disco rígido, latência da rede e outras métricas de infraestrutura em torno dos componentes e do sistema. Geralmente, os dados provenientes do *monitoring* são exibidos em forma de gráficos ou diagramas (BEYER et al., 2016).

O *logging* disponibiliza os registros de algum evento discreto ocorrido em determinado momento. Normalmente, é apresentado ao usuário de forma textual e com uma quantidade crescente de informações (JANAPATI, 2017).

O *tracing* é semelhante ao *logging*, mas se concentra no registro do fluxo de execução do programa, à medida que as requisições viajam através de vários módulos. Além disso, o *tracing* distribuído, quando o *tracing* é aplicado a sistemas distribuídos, também pode preservar os relacionamentos de causalidade quando o sistema é dividido em várias *threads*, processos, máquinas e até localizações geográficas (OPENTRACING.IO, 2020). Há algumas ferramentas, como *Zipkin* e *Jaeger* que utilizam o padrão *OpenTracing*, para exibição do *tracing* realizado de uma forma humanamente legível. Os dados gerados pelo *tracing* distribuído são, particularmente, adequados para depurar e monitorar arquiteturas de sistemas distribuídos, como microsserviços. Esse tipo de dados contém informações relevantes sobre o caminho percorrido por uma solicitação e o tempo gasto em seu processamento, dessa forma, é possível detectar comportamentos incomuns durante as solicitações.

Face ao exposto, pelo fato do *tracing* ser realizado somente de forma distribuída neste trabalho, o termo *tracing* distribuído será referenciado somente como *tracing*.

## 1.1 Definição do problema

Com os constantes avanços tecnológicos, a cada dia, surgem novas aplicações baseadas em microsserviços sensíveis ao tempo de resposta, onde, para um perfeito funcionamento, o sistema tem uma janela de tempo para a execução de todos os serviços em cadeia. Em um cenário ideal, o tempo gasto para processamento dos serviços juntamente com a comunicação entre eles deve ficar compreendido dentro dessa janela.

Como os microsserviços são, por definição, serviços distribuídos por muitos servidores, físicos ou virtuais, para permitir um dimensionamento horizontal, há uma forma de comunicação entre eles e, consequentemente, técnicas para medição do tempo de comunicação, porém cada grupo interessado descreve a sua conforme suas necessidades. Isso, muitas vezes, ocasiona conflito de ideias e torna o processo de padronização mais complexo. Dessa forma, ao comparar com o tempo de processamento, torna-se evidente a falta de padronização e, consequentemente, de ferramentas específicas baseadas em um padrão para medição do tempo de comunicação.

Tendo isso em vista, uma vez que as plataformas nem as ferramentas de *tracing* possuem suporte nativo para medição do tempo gasto na troca de mensagens dos microsserviços, uma alternativa para realizar a medição de tempo de comunicação seria a plataforma na qual os microsserviços estão sendo executados fornecer o suporte necessário às ferramentas de monitoramento. Assim, mesmo não sendo desenvolvidas com tal objetivo, as ferramentas de monitoramento poderiam utilizadas para esse fim.

## 1.2 Motivação

Pode-se perceber em (CARMO et al., 2019), que possui aplicações sensíveis ao tempo de resposta, uma grande dificuldade em se medir o tempo de comunicação corretamente. Para a medição e otimização do tempo de processamento, técnicas e ferramentas previamente citadas nesse trabalho são utilizadas. Entretanto, para efetuar a medição do tempo de comunicação, é realizado, após a finalização da aplicação, um cálculo baseado em um arquivo contendo registros dos horários de envio e recebimento das mensagens, ou seja, a obtenção do tempo de comunicação não é feita de forma automática. Com isso, pode-se dizer, que tal medição fica dificultada devido à falta de ferramentas específicas.

A mesma estrutura do trabalho previamente citado, entretanto com alguns serviços diferentes, é utilizada em (COTTA et al., 2019). Nesse trabalho, o principal objetivo é utilizar serviços de visão computacional para guiar uma pessoa, através de um robô cão-guia, até um local desejado dentro de um prédio público. Uma vez que tal aplicação foi desenvolvida para utilização no Espaço Inteligente, que é o estudo de caso do presente trabalho, encontra-se mais uma vez a dificuldade de se medir o tempo de resposta com fins de depuração. Nesse caso, devido à utilização de microsserviços que trabalham paralelamente em vez de linearmente, a observância do arquivo de registro, em conjunto do cálculo manual, acaba não sendo um bom método para se determinar o tempo de comunicação.

Com isso fica evidente a necessidade do desenvolvimento de uma maneira capaz de realizar medições do tempo de comunicação para microsserviços em diversos cenários e aplicações, de forma automática e precisa.

## 1.3 Objetivos e proposta da dissertação

### 1.3.1 Objetivos Gerais

O trabalho apresentado nesta dissertação tem como objetivo geral o desenvolvimento de uma metodologia para medição do tempo de comunicação em sistemas distribuídos baseados em microsserviços, para que, juntamente com a medição do tempo de processamento já realizada por outros meios, o tempo de resposta possa ser mais fielmente mensurado e exibido ao seu operador.

Como contextualização, vale destacar que apesar de se buscar o desenvolvimento de um mecanismo mais geral possível, este trabalho está focado em apresentar uma solução funcional para o ambiente disponível, ou seja, o PIS montado no laboratório de Visão Computacional e Robótica da UFES, que, conforme já foi explicado, é um sistema distribuído baseado em microsserviços.

Com isso, todo o desenvolvimento da solução está restrito às diretrizes do projeto do espaço inteligente, isto é, utilizam-se linguagens de programação, bibliotecas e padrões já estabelecidos. Entretanto, a proposta aqui apresentada pode ser prontamente utilizada em qualquer arquitetura de sistema distribuído, contanto que seja adotada um modelo de divisão de camadas. Além disso, vale frisar que, embora tenha sido feita uma revisão bibliográfica para busca de novas ferramentas no que tange a realização de *tracing*, optou-se pela utilização das ferramentas já presentes no PIS.

Por fim, vale mencionar que não constitui objeto de estudo no escopo deste trabalho as otimizações do tempo de processamento nem de tempo de comunicação. Sendo o foco mantido no desenvolvimento da metodologia de medição.

### 1.3.2 Objetivos Específicos

Para atingir o objetivo geral, os seguintes objetivos específicos foram considerados:

- Utilizar o padrão de medida de tempo de processamento como base para desenvolver uma metodologia para medição de tempo de comunicação, que mais se aproxima de um padrão já estabelecido;
- Implementar e validar a proposta através da utilização em microsserviços no Espaço Inteligente Programável da UFES.

### 1.3.3 Proposta

O presente trabalho tem como proposta realizar modificações necessárias na estrutura de software do PIS, modificar bibliotecas *opensource* específicas e criar microsserviços paralelos, quando necessário. Tais ações visam habilitar a utilização de ferramentas de



*tracing* na própria arquitetura de software do espaço inteligente para realizar a medição do tempo de comunicação.

Sendo assim, como é possível simplificar o resultado da aplicação deste trabalho como a medição de um tempo de resposta, propõe-se também a inserção do tempo de comunicação junto do tempo de processamento na exibição do *tracing*, ou seja, será exibido na interface gráfica da ferramenta o tempo de resposta completo.

Para isso, os seguintes procedimentos foram adotados:

- Revisar bibliograficamente sobre: sistemas distribuídos, *tracing* distribuído, medição de tempo de comunicação, sincronização de relógios;
- Entender na íntegra a implementação atual do Espaço Inteligente Programável utilizado;
- Realizar a medição de tempo de processamento e tempo de comunicação para, conseqüentemente, obter o tempo de resposta;

## 1.4 Estrutura da dissertação

Esta dissertação está dividida em 5 capítulos.

O [Capítulo 1](#) corresponde à introdução, contendo uma contextualização do tema abordado, as justificativas para o trabalho desenvolvido, além dos objetivos e a proposta desta pesquisa.

Em seguida, o [Capítulo 2](#) apresenta os conceitos teóricos e os trabalhos relacionados que abrangem os temas relacionados a este trabalho.

O [Capítulo 3](#) e [Capítulo 4](#) compreendem juntos o desenvolvimento deste trabalho, onde, no [Capítulo 3](#), são descritos os detalhes sobre a metodologia da solução proposta para o problema da falta de ferramentas para medição do tempo de comunicação e, no [Capítulo 4](#), são mostrados todos os detalhes da implementação e dos experimentos realizados, bem como os resultados obtidos.

Por fim, o [Capítulo 5](#) traz as conclusões alcançadas com o desenvolvimento deste trabalho e também discute melhorias para a solução proposta, enumerando possíveis trabalhos futuros.

## 2 Referencial Teórico e Estado da Arte

Este capítulo, primeiramente, apresenta os aspectos teóricos dos temas abordados neste trabalho. Conceitos importantes, como sistemas distribuídos e *tracing*, são explicados de uma maneira mais aprofundada, de forma a fornecer uma base para o entendimento do funcionamento da solução proposta. Na segunda parte do capítulo, é apresentado o estado da arte em relação à solução proposta. De uma forma mais específica, os trabalhos discutidos estão mais estritamente ligados à medição do tempo de resposta, incluindo tempo de comunicação, para sistemas distribuídos baseados em microsserviços, implementados em quaisquer ambientes.

### 2.1 Referencial Teórico

#### 2.1.1 Sistemas Distribuídos

Na década de 80, com o alto nível de sofisticação que os protocolos e as redes de comunicação em geral atingiram, juntamente com o desenvolvimento tecnológico dos microprocessadores, surge uma nova tecnologia: a dos Sistemas Distribuídos.

De acordo com (BAGCHI, 2010), um sistema pode ser definido como um conjunto de entidades com interconexões que interagem entre si com um objetivo comum, formando um todo integrado. Sendo assim, (COLOURIS et al., 2011) diz que um sistema distribuído é um sistema computacional no qual os componentes, de hardware ou software, conectados em rede se comunicam e coordenam suas ações utilizando troca mensagens entre si, tendo como principal motivação o compartilhamento de informações e recursos.

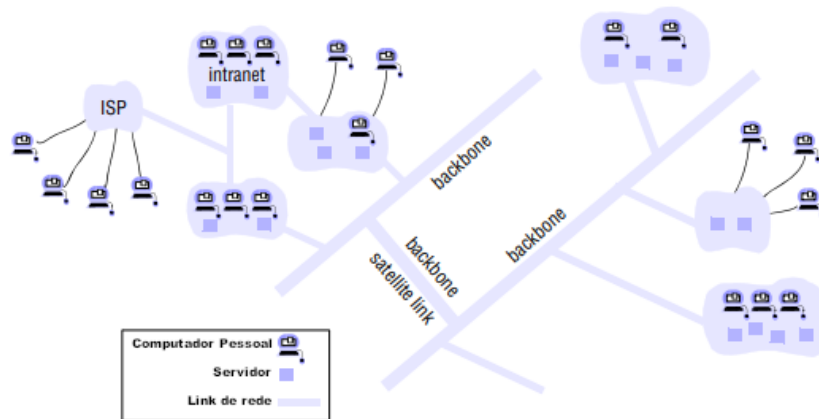
Para (TANENBAUM, 2008), um sistema distribuído é também um conjunto de computadores independentes que se apresentam como um sistema único e coerente. Com essa definição, surge um ponto bem relevante: um sistema distribuído é composto por componentes autônomos que, de alguma forma, precisam cooperar entre si para que o usuário tenha a impressão de estar utilizando um sistema único. O estabelecimento dessa cooperação é o núcleo do desenvolvimento de sistemas distribuídos.

Para os sistemas distribuídos, não há requisitos em relação ao tipo da entidade, podendo variar desde computadores de alto desempenho até pequenos nós em rede de sensores e, segundo (TANENBAUM, 2008), também não há premissa em relação à interconexão desses dispositivos.

É comum citar a Internet ou alguma aplicação de grande empresa como exemplo de tais sistemas, pois a maioria de seu conteúdo é distribuído, inclusive, geograficamente, como

pode ser visto na Figura 2. No meio empresarial, dificilmente os sistemas são centralizados, pois a distribuição de recursos elimina parte do risco de erros críticos que poderiam paralisar os sistemas. Outra presença de sistema distribuído ocorre em grandes redes, onde os dados e o processamento deles pode ser executado por qualquer um nó presente na rede, não se restringindo apenas ao sistema isolado onde os dados são adquiridos.

Figura 2 – A Internet: exemplo de sistema distribuído físico e virtual.



Fonte: Adaptado de (COLOURIS et al., 2011)

Na Figura 2, pode-se visualizar, além dos links de comunicação entre todos os componentes, as conexões dos terminais, computadores de uso pessoal, com os servidores de rede que, por sua vez, se conectam nos provedores ou *Internet Service Provider* (ISP). Todos os elementos estão situados em diferentes localidades, configurando um sistema distribuído.

Os sistemas distribuídos têm como objetivo principal fornecer, aos usuários e aplicações, o acesso aos recursos de hardware e software de maneira eficiente, simplificada, segura e controlada. Para isso, são aplicados graus de transparência de distribuição de recursos, com o intuito de abstrair a distribuição geográfica, bem como são utilizados padrões e métodos normalizados, para prover aos programadores a possibilidade de integração de suas aplicações ou expansão de suas funcionalidades, mesmo sem o conhecimento da infraestrutura utilizada.

Nesse quesito, para (COLOURIS et al., 2011), a arquitetura de um sistema distribuído pode ser interpretada como a divisão de responsabilidades entre os componentes do sistema. Assim, nesse contexto, introduz-se o conceito de *cluster*, que é um sistema computacional composto por um conjunto de dispositivos independentes e que trabalham de forma integrada, analogamente a um sistema único (BUYAYA, 1999).

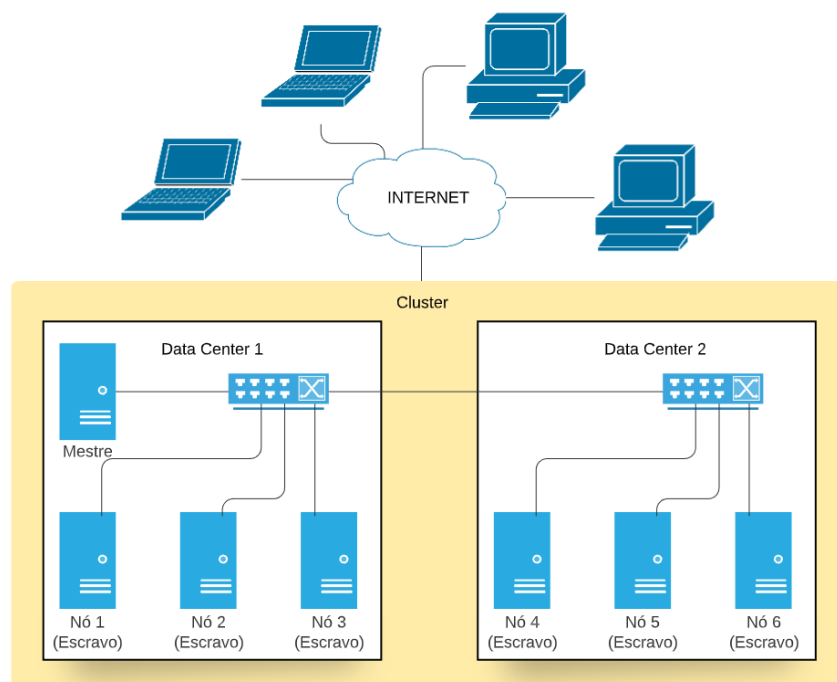
Os *clusters*, em geral, são tipificados por um agrupamento físico ou virtual de vários elementos para execução de aplicações (DANTAS, 2005). Tais elementos podem ser desde máquinas virtuais conectadas por algum dispositivo virtual de interconexão até mesmo

computadores fisicamente separados e conectados através de rede. Seu principal objetivo é construir e habilitar a expansibilidade de um sistema dotado de alto poder computacional composto por vários dispositivos de diferentes poderes de processamento. Dessa forma, qualquer outro elemento pode ser adicionado ao sistema posteriormente que este será automaticamente expandido e seu poder computacional ampliado.

Já no que se refere à infraestrutura de software de um *cluster*, (DANTAS, 2005) ressalta que os pacotes utilizados possuem funções cruciais para o perfeito funcionamento das aplicações que serão executadas. São responsáveis por abstrair o hardware e software do sistema, detectar falhas automaticamente e redirecionar as funções para outros nós. Além disso, determinam quem é o mestre do *cluster*, que realiza a distribuição homogênea da carga de processamento das solicitações para os nós escravos.

Como ilustração, a Figura 3 mostra um *cluster* composto por sete dispositivos, sendo um mestre e dois grupos de três nós escravos geograficamente separados, que, pelo ponto de vista da Internet, é considerado como um único sistema.

Figura 3 – Exemplo de *cluster*



Fonte: Produção do próprio autor.

Comumente, a arquitetura é implementada baseada em serviços ou microsserviços. Isso significa que um cliente, seja físico ou outro serviço, pode realizar uma requisição através de mensagem para o serviço que irá recebê-la, tratá-la e, em seguida, devolvê-la à entidade de origem ou propagar para um próximo serviço. Dessa forma, um usuário externo, ao utilizar o sistema, não fica ciente de cada etapa envolvida na sua requisição e recebe apenas o resultado dessas.

### 2.1.2 Microsserviços

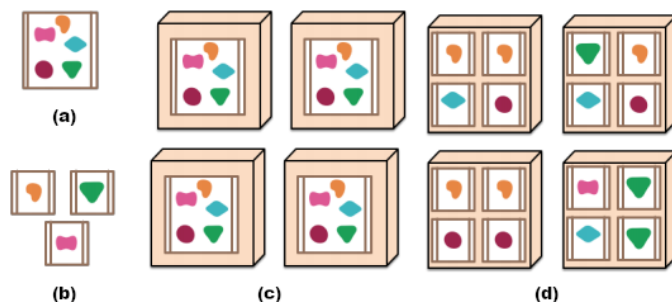
Os microsserviços podem ser definidos como um estilo arquitetural que estrutura uma aplicação como uma coleção de serviços vagamente acoplados (DRAGONI et al., 2017). Esse estilo vem sendo utilizado e aprimorado devido aos progressos recentes da engenharia de software no que tange às infraestruturas de computação distribuída em nuvem, melhorias nas interfaces de programação de aplicação (API), metodologia de desenvolvimento ágil e o surgimento do fenômeno de aplicações em *container*, que é uma unidade padrão de software que encapsula o código e todas as dependências para que a aplicação seja executada de forma rápida e segura, comunicando-se com outros através de uma API (PAHL et al., 2019).

Os microsserviços, considerados uma evolução em sistemas distribuídos, são especificamente desenvolvidos e configurados para serem minimalistas, resilientes, autônomos e para desempenhar poucas ou apenas uma função. Dessa forma, essa estrutura provê rápidas iterações, automação, depuração, testes e desenvolvimento contínuo, o que permite às equipes criarem produtos e implementarem códigos exponencialmente mais rápido e seguro que antes (NEWMAN, 2015).

Até a popularização dos microsserviços, a arquitetura de sistema distribuído mais utilizada era do estilo monolítico. Tal modelo consiste no desenvolvimento de todo o software em um único serviço, ou seja, todas as características e recursos eram agrupadas, empacotadas e implementadas em uma aplicação de camada única, usando um único código como base. Uma ilustração comparativa entre a arquitetura monolítica e de microsserviços pode ser vista na Figura 4.

Na aplicação monolítica, todas as funcionalidades, Figura 4a, estão presentes em único processo, que se escalam replicando-as para todos os servidores, como pode ser visto na Figura 4c. Por sua vez, a estrutura de microsserviços separa cada funcionalidade em um serviço separado, Figura 4b, que são escalados de forma distribuída pelos servidores, replicando somente os necessários, Figura 4d.

Figura 4 – Representação das arquiteturas monolíticas e microsserviço



### 2.1.3 *Tracing*

O termo *tracing* pode ser definido como a maneira que um sistema é monitorado e como as medidas de seu desempenho são coletadas. Pode-se também dizer que *tracing* distribuído é um método proveniente do *tracing* tradicional, mas aplicado a sistemas distribuídos (SAMBASIVAN et al., 2016).

Devido à natureza altamente distribuída, as falhas nas aplicações baseadas em microsserviços são notoriamente trabalhosas de se identificar (ALSHUQAYRAN; ALI; EVANS, 2016). Muitas vezes, os erros são difíceis de se reproduzir, dado que um sistema desse tipo se divide em vários recursos, físicos ou virtuais, e se comunica através de vários protocolos que podem falhar. Dessa forma, o *tracing* é largamente utilizado por desenvolvedores para indicar onde uma falha ocorre num sistema e porquê.

A semântica do *tracing* distribuído define (SIGELMAN et al., 2010):

1. *Annotations*: é a unidade básica de informação que é propagada. Existem dois tipos de *annotations*:
  - a) *Timing*: é associada a um registro de um *timestamp* que corresponde ao tempo de início ou tempo decorrido de uma medição. Um *timestamp* é o registro de um instante de tempo representado em alguma unidade temporal;
  - b) *Tags*: é associada a um mapeamento do tipo "chave-valor", que pode ser utilizada para inserção de informações pelo desenvolvedor;
2. *Spans*: é associada à unidade básica de trabalho, isto é, cada *span* pode ser associada a uma *thread*, função ou serviço. Basicamente, é um conjunto de *annotations* do tipo *tag* e *timing* que contém informações específicas de uma aplicação e infraestrutura. Possuem uma relação de causalidade, isto é, um *span* pode ser considerado causador de outro e, por isso, ser considerado pai do que foi causado.
3. *Trace*: é uma coleção de *spans*, onde são agregadas todas as relações de causalidade e a representação do fluxo de execução do sistema. Em casos onde há somente um *span*, o *trace* é considerado o próprio *span*.

Para se utilizar o *tracing*, é necessário a inserção de fragmentos de códigos dentro do serviço que se deseja medir o tempo de execução, mais especificamente no início e no final do bloco de comandos que se deseja coletar métricas. Esse processo de adição do código extra, que será executado quando o sistema estiver realizando suas funções, é chamado de instrumentação (GEIMER et al., 2009).

Um ponto relevante a ser mencionado é que, independente de quão leve seja, toda instrumentação sempre será uma carga extra no funcionamento do sistema.

A partir desse contexto, encontra-se uma gama de ferramentas e padrões, onde o mais popular é o padrão *OpenTracing* (OPENTRACING, 2020a), que segue o modelo proposto por (FONSECA et al., 2007).

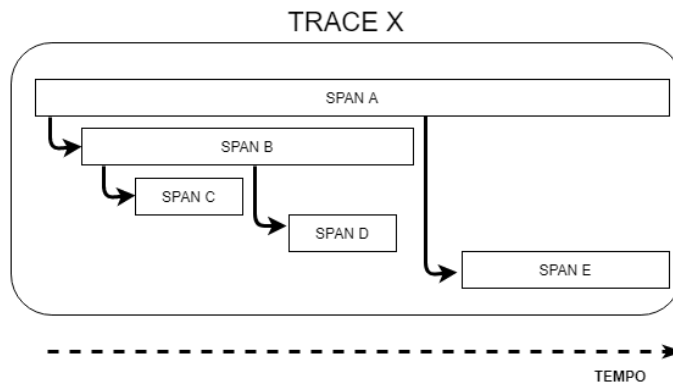
Tal padrão utiliza metadados dinâmicos para propagar a relação de causalidade entre os *spans*, isto é, cada *span* possui, além das *annotations*:

- *TraceID*: identificador de *trace* que é comum a todos os *spans* do mesmo *trace*;
- *SpanID*: identificador único do *span*;
- *ParentID*: identificador reservado para representar a relação pai-filho entre os *spans*;

Além disso, de um ponto de vista técnico, o padrão define o formato para os *spans* e a convenção semântica para seu conteúdo e *annotations* (OPENTRACING, 2020a) (OPENTRACING, 2020b);

A Figura 5 proporciona uma visão clara de um *trace*, isto é, como os *spans* estão relacionados no tempo e a relação de causalidade entre si.

Figura 5 – Exemplo de *trace* no tempo



Fonte: Produção do próprio autor.

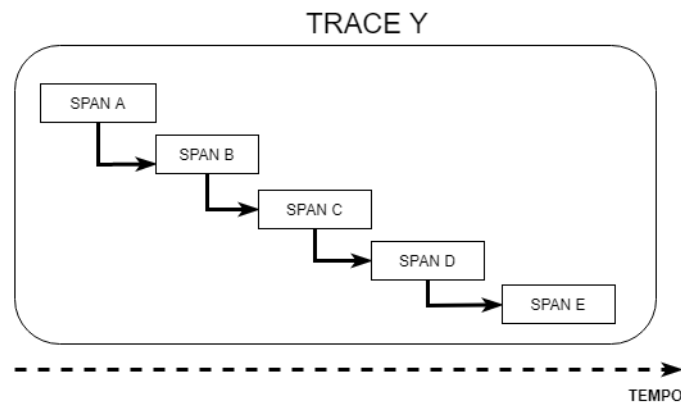
Conforme Figura 5, o "*Span A*" é o causador do "*Span B*" e "*Span E*". Por sua vez, o "*Span B*" causa o "*Span C*" e o "*Span D*". Dessa forma, diz-se que "*Span A*" é o pai de "*Span B*" e "*Span E*", bem como "*Span C*" e "*Span D*" são filhos do "*Span B*". Nesse caso, "*Span A*" não possui pai, portanto, ele é a origem de todo *trace*.

A disposição dos *spans* através do tempo é outro ponto que pode ser claramente observado pela Figura 5. Todos os *spans* possuem em suas *annotations* do tipo *timing* informações de *timestamp* de início e duração. Para esse exemplo, embora os *spans* estejam alocados em diferentes pontos no tempo, permanecem dentro dos limites do *span* pai. Isso significa que o *span*-pai é dependente da finalização dos filhos e, por isso, deve ter o seu início sempre antes e o fim sempre depois do último *span*-filho.

Há também casos onde os *spans* possuem uma propagação em cadeia, conforme pode ser visto na Figura 6. Nesta situação, o "*Span A*" é pai do "*Span B*", que é pai do

"Span C", que é pai do "Span D", que é pai do "Span E". Sendo assim, como não há dependências entre os *spans*, o início e o fim de um *span*-pai deve ocorrer sempre antes do início do *span*-filho.

Figura 6 – Exemplo de *trace* em cadeia no tempo

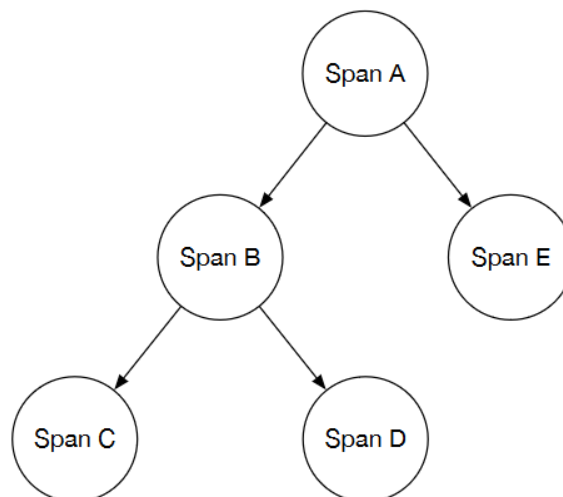


Fonte: Produção do próprio autor.

Caso as condições supracitadas não sejam satisfeitas, muito possivelmente, há um problema de sincronismo de relógios, que é discutido na [Seção 2.1.4](#), entre as entidades do sistema distribuído.

Segundo ([OPENTRACING, 2020a](#)), com alguns *spans*, é possível gerar uma árvore de *spans* e modelar uma representação gráfica de uma porção do sistema. Isto se dá devido à relação de causalidade já explanada. Dessa forma, além de mensurar o tempo de processamento de uma ação dentro de uma aplicação, é possível visualizar o caminho percorrido por uma requisição. A [Figura 7](#) demonstra um exemplo de árvore de *span*.

Figura 7 – Exemplo de árvore de *span*



Fonte: Adaptado de ([OPENTRACING, 2020a](#))



Por exemplo, se cada serviço estiver localizado fisicamente em um terminal diferente representados pelas letras da árvore de *span*, pode-se criar o rastro A-B-D, que significa que uma requisição passou pelas máquinas A,B e D, respectivamente. Com isso, fica facilitada a identificação de possíveis falhas em algum ponto do sistema.

### 2.1.3.1 Ferramentas para *Tracing*

As ferramentas para *tracing*, em sua maioria, são focadas para sistemas distribuídos baseado em microsserviços. Elas, normalmente, recebem dados de *tracing* dos sistemas, tratam as informações e as exibem para o usuário em forma de gráficos e diagramas. O [Quadro 1](#) mostra um comparativo entre as três ferramentas mais utilizadas.

Do [Quadro 1](#), pode-se perceber que todas as ferramentas são bem similares: são de código aberto, permitem ser executadas através de *container* e dispõem de uma interface gráfica para o usuário. *Jaeger* ([JAEGER, 2020](#)) e *Appdash* ([SOURCEGRAPH, 2020](#)) foram desenvolvidas baseadas no *Zipkin* ([ZIPKIN, 2020](#)), entretanto não possuem mais vantagens. Um ponto forte do *Zipkin* é o suporte no transporte dos dados, isto é, com mais protocolos suportados, há uma maior abrangência de aplicações. Todas essas ferramentas são fortemente focadas na coleta e exibição dos *traces* para o usuário, entretanto não fornecem recursos extras que poderiam ser interessantes, como por exemplo a medição de tempo de comunicação entre dois pontos de um sistema.

Quadro 1 – Comparativo entre as ferramentas de *tracing*

Característica	Ferramenta		
	<i>Appdash</i>	<i>Jaeger</i>	<i>Zipkin</i>
Código aberto	Sim	Sim	Sim
Suporte a <i>Docker</i>	Sim	Sim	Sim
Interface de usuário	Sim	Sim	Sim
Protocolo de Transporte	HTTP	HTTP,Thrift	HTTP, <i>Kafka</i> ,AMQP, <i>Scribe</i>
Taxa de amostragem dinâmica	Não	Sim	Não
Suporte ao <i>OpenTracing</i>	Com <i>plugin</i>	Nativo	Nativo

Fonte: Produção do próprio autor.

Sendo assim, tais ferramentas podem ser um ponto de partida para a identificação do caminho percorrido por uma requisição e a medição de tempo de processamento, entretanto há ainda a problemática na medição do tempo de comunicação para a obtenção fiel do tempo de resposta.

## 2.1.4 Sincronização de relógios

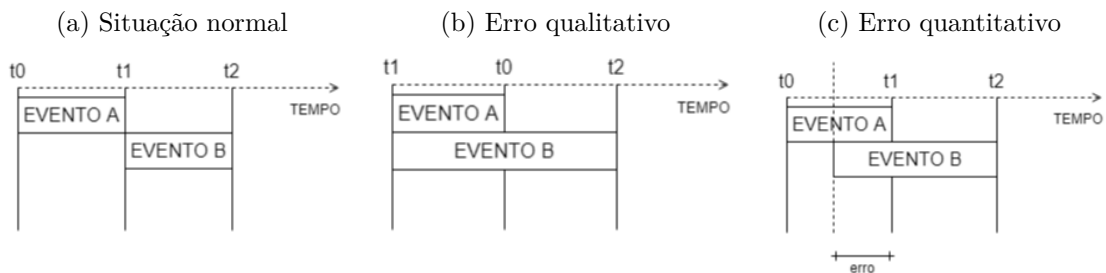
Como já explanado, um sistema distribuído, normalmente, é composto de elementos independentes que cooperam entre si, se comunicam através de mensagens e podem não

possuir um relógio comum. Do ponto de vista do *tracing*, esses sistemas são mais complexos de se instrumentar que os sistemas monolíticos, pois há o paralelismo entre os processadores e nós, atrasos randômicos na comunicação, falhas parciais e, principalmente, a inexistência de um relógio global sincronizado.

Devido a essa ausência, um mesmo algoritmo, ao ser executado, pode gerar resultados em diferentes períodos de tempo, o que caracteriza o comportamento do sistema como imprevisível e não-determinístico. Além disso, a falta de um relógio comum prejudica a determinação das relações de causalidade entre eventos numa análise de *tracing*. Usar relógios não sincronizados, ou com erro na sincronização, para análise de desempenho, resulta numa representação errônea da ferramenta de *tracing* (DOLESCHAL et al., 2008).

Quando o erro de sincronismo causa a violação da ordem lógica dos eventos, por exemplo, quando o evento de recebimento está situado antes do evento de envio, diz-se que o erro é do tipo qualitativo. Na Figura 8b, devido a um erro de sincronismo, o valor de tempo  $t_1$  está situado antes do valor  $t_0$ , portanto, todos os eventos que estiverem relacionados a  $t_1$  apresentarão erro, seja o fim do evento antes mesmo do início ou medição de tempo negativa. Por sua vez, quando há uma distorção na medição dos eventos, por exemplo, um evento, aparentemente, ter a duração na ordem de segundos, enquanto a duração real foi na ordem de milissegundos define-se como erro quantitativo (DOLESCHAL et al., 2008). Conforme mostra a Figura 8c, o erro quantitativo tem um acréscimo na duração do evento, ou seja, o evento que deveria durar  $t_B = t_2 - t_1$  tem a duração  $t_B = (t_2 - t_1) + t_{erro}$ .

Figura 8 – Diagrama de eventos no tempo em situação normal e com erro



Fonte: Produção do próprio autor.

Desde o advento dos sistemas distribuídos até a década de 90, uma grande quantidade de trabalhos sobre protocolos de sincronização de tempo foram desenvolvidos, bem como inúmeros protocolos foram propostos em diversas publicações. Em (YANG; MARS-LAND, 1993), mais de 60 artigos foram listados em uma revisão bibliográfica realizada em 1993 sobre esse assunto. Diante disso e face ao exposto anteriormente, demonstra-se a essencialidade da sincronização do relógio e seus protocolos para um funcionamento correto e consistente de um sistema distribuído para atacar ambos os erros e, conjuntamente, a realização do *tracing*.

Em (LAMPORT, 1978), é provada que a sincronização é possível e apresenta um algoritmo que torna possível sincronizar todos os relógios para produzir um único tempo padrão não ambíguo. O autor afirma que a sincronização dos relógios não precisa ser absoluta. Se dois processos não interagem entre si, não é necessário que os seus relógios estejam sincronizados, pois a falta de sincronização não seria significativa e não causaria problemas. Assim, em alguns casos, o que usualmente importa é se há concordância na ordem dos acontecimentos dos eventos.

Para muitas aplicações, é suficiente que todas as máquinas estejam com o mesmo tempo, mesmo este não estando em concordância com o tempo real, neste caso, diz-se que é importante que haja a sincronização lógica dos relógios (SCHWARZ; MATTERN, 1993). Para alguns sistemas cujo o tempo real é relevante, deve-se utilizar relógios físicos e, conseqüentemente, realizar a sincronização.

Uma vez que a utilização de relógios físicos não é a realidade do presente trabalho, somente a sincronização lógica será aqui abordada.

Como a sincronização é primordial para realização do *tracing*, há diversos métodos para alcançar esse objetivo, entretanto as principais são: *Global Navigation Satellite Systems* (GNSS), *Precision Time Protocol* (PTP) e *Network Time Protocol* (NTP).

#### 2.1.4.1 *Global Navigation Satellite Systems* (GNSS)

Um sistema de navegação global por satélite, GNSS, é caracterizado pelo fornecimento de tempos precisos, além do posicionamento e navegação geoespacial autônomo com cobertura global. Atualmente, as três principais sistemas de navegação global já implementadas são: GPS, GLONASS, Galileo.

O sistema de posicionamento global (GPS) (DOD, 2001), operado pela força aérea dos Estados Unidos, tem a capacidade de fornecer informações precisas sobre localização, tempo e frequência (OAKS et al., 1998). Os satélites são equipados com dois relógios atômicos de césio e dois de rubídio. Em geral, diferentemente dos relógios que se baseiam no tempo universal coordenado (UTC), os relógios atômicos nos satélites são configurados para a escala *GPS Time* (GPST). GPST é uma escala de tempo contínua e, teoricamente, com precisão de cerca de 14 nanossegundos (ALLAN; ASHBY; HODGE, 1997). Entretanto, durante a interpretação dos sinais, pode haver perdas na precisão, ficando por volta dos 100 nanossegundos (DANA; PENROD, 1990). Em relação ao UTC, GPST não contém os segundos adicionais ou outras correções pertinentes, pois não é corrigido para corresponder à rotação da Terra. Sendo assim, essas faltas de correções tem como consequência constante uma diferença de tempo em relação ao Tempo Atômico Internacional (TAI).

O GLONASS, mantido pelas forças de defesas aeroespaciais da Rússia, oferece uma alternativa ao GPS e possui uma precisão comparável. Os satélites possuem a bordo

relógios atômicos de césio, sincronizados entre si com precisão superior a 20 nanossegundos, além de um amplificador de micro-ondas por emissão estimulada de radiação (MASER) (HOWARTH, 1959) de hidrogênio, denominado de sincronizador central (CS). A escala de tempo utilizada é o *GLONASS Time* (GLONASST) que é gerado pelo CS-GLONASS e, diferentemente do GPS, implementa os segundos adicionais. As escalas de tempo dos satélites são comparadas e ajustadas, duas vezes por dias, em relação a escala de tempo do CS. Sua precisão em relação ao UTC tem um limiar de 1 milissegundo, embora normalmente seja ligeiramente melhor que 1 microssegundo (DUNN; DISL, 2012).

Por sua vez Galileo, operado conjuntamente pelas Agências GNSS Europeia e Espacial Europeia, é baseado na escala *Galileo System Time* (GST). Essa escala é gerada no Centro de Controle Galileo, em Fucino, na Itália, com base nas médias de diversos tipos de relógios atômicos e é sincronizada com o TAI, com uma discrepância nominal abaixo de 50 nanossegundos (FERNÁNDEZ-HERNÁNDEZ et al., 2014). Cada satélite detém dois MASER de hidrogênio e dois relógios atômicos de rubídio para comparação dos tempos a bordo e terrestre.

Assim, a sincronização através do GNSS ocorre a partir de um dispositivo receptor que calcula o atraso de tempo relativo aos sinais de, no mínimo, quatro satélites e o propaga pela rede. Vale destacar que, além do requisito de número de satélites, outros fatores podem afetar a precisão da sincronização, por exemplo, interferência de caminhos múltiplos do sinal, geometria dos satélites e erros orbitais dos satélite. Entretanto, no geral, as desvantagens dessa sincronização são ofuscadas pela precisão obtida nos resultados.

#### 2.1.4.2 *Network Time Protocol* (NTP)

O NTP é utilizado para construir e manter um conjunto de servidores de relógio e caminhos de transmissão como uma sub-rede de sincronização. O protocolo foi descrito pela primeira vez em (MILLS, 1985), revisado diversas vezes em sucessivas versões e, por fim, definido como um protocolo padrão da Internet (MILLS, 1989). O NTP é construído no Protocolo da Internet (IP) (POSTEL et al., 1981) e no *User Datagram Protocol* (UDP) (POSTEL, 1980), que fornecem um mecanismo de transporte sem conexão, no entanto é facilmente adaptável a outros conjuntos de protocolos.

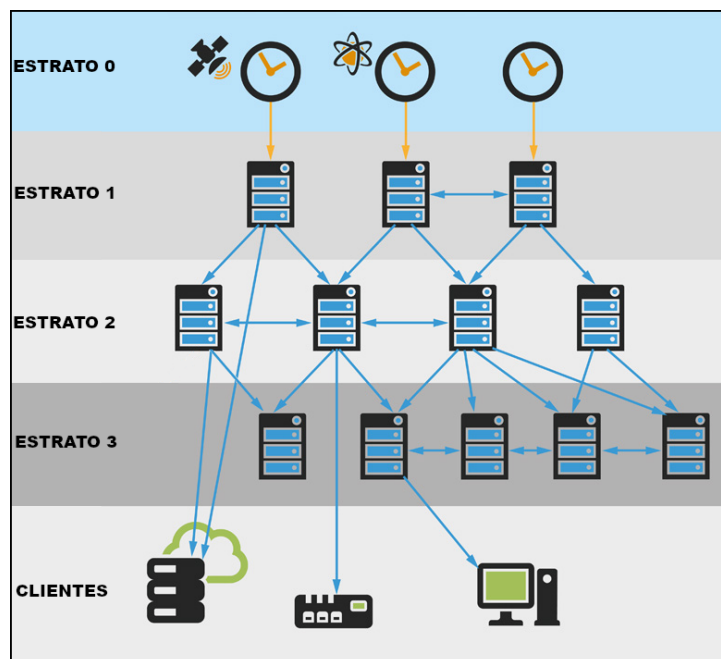
Não há fornecimento de descoberta de pares, aquisição ou autenticação no próprio NTP, embora algumas implementações incluam esses recursos. A integridade dos dados é fornecida pelo IP e UDP, através de somas de verificação (*checksum*). O protocolo opera em diversos modos adequado a diferentes cenários, envolvendo estações de trabalho privadas, máquinas de serviço público e várias configurações de rede. Como apenas um único formato de mensagem NTP é usado, o protocolo é facilmente implementado e pode ser usado em uma variedade de sistemas operacionais e ambientes de rede.

Na implementação mais comum, cliente-servidor, um cliente envia uma mensagem

NTP para um ou mais servidores, que processam a requisição e as respondem assim que recebidas. As informações contidas numa mensagem NTP permitem ao cliente determinar a hora do servidor em relação a hora local e ajustar o relógio de acordo. Além disso, a mensagem inclui informações para calcular a precisão e a confiabilidade, para que os dados imprecisos possam ser descartados e somente os melhores de vários servidores possam ser selecionados. A precisão de sincronização, geralmente, é no intervalo de um milissegundo nas *Local Area Networks* (LANs) que possuem menos fontes de latência na rede.

Em sistemas distribuídos, a situação mais comum de implementação do NTP é a partir de um servidor local, conectado a diversos servidores NTP através da Internet, que é acessado por um conjunto de dispositivos. A hierarquia dos serviços de fornecimento de sincronização NTP via Internet, [Figura 9](#), é composta por 4 níveis, chamados de Estratos. Os ditos servidores primários estão no estrato 1 e são conectados diretamente a vários serviços de horário nacional no estrato 0 via satélite, rádio ou rede. Normalmente, o serviço de tempo no estrato 0 pode ser um relógio atômico, um receptor de rádio sintonizado com os sinais transmitidos por um relógio atômico ou um receptor de um GNSS usando os sinais de relógio altamente precisos transmitidos pelos satélites.

Figura 9 – Hierarquia dos serviços de fornecimento de sincronização NTP via Internet



Fonte: Adaptado de ([BURNICKI, 2020](#))

Muitas organizações que possuem uma elevada quantidade de dispositivos sincronizados configuram seus próprios servidores de horário, de modo que apenas um servidor local acesse os servidores de horário do estrato 2. Assim, em seguida, os dispositivos de rede restantes são configurados para usar o servidor de horário local que nesse caso passa a ser considerado um servidor situado no estrato 3.

### 2.1.4.3 Precision Time Protocol (PTP)

O PTP, padrão IEEE 1588-2008, é um protocolo aplicável a sistemas, que se comunicam através de pacotes, que teve seu desenvolvimento focado para ser utilizado em sistemas de controle e medição, implementados com tecnologias de rede, computação e objetos distribuídos (LEE; ELDSON, 2004).

O protocolo permite uma sincronização dos relógios com precisão de poucos microssegundos e é otimizado para o uso mínimo da largura de banda da rede. Dessa forma, é utilizado em aplicações onde é necessária uma precisão maior do que a que pode ser alcançada com o NTP e a utilização de sincronização por GNSS não é possível. De acordo com (MILLS, 2012), esse tipo de protocolo é mais comum em sistemas com fins específicos, onde os elementos são interconectados por *switches*, em uma rede dedicada em alta velocidade.

O sincronismo ocorre em duas etapas: a primeira é uma organização dos relógios na rede numa hierarquia mestre-escravos, onde o relógio que disponibiliza o tempo para uma sub-rede é denominado de mestre e o restante de escravos. Em seguida, os nós começam a trocar mensagens contendo *timestamps* entre os mestres e escravos para ajuste dos relógios.

Na versão mais recente do IEEE 1588 (TECHNOLOGY, 2008), as mensagens são divididas em: mensagens de eventos que são críticas em relação ao tempo, uma vez que a precisão do *timestamp* de recebimento afeta diretamente a sincronização e mensagens gerais, que são unidades de dados do protocolo importantes para o PTP com *timestamp* de transmissão e recebimento descartáveis. Além disso, as mensagens podem ser de cinco tipos:

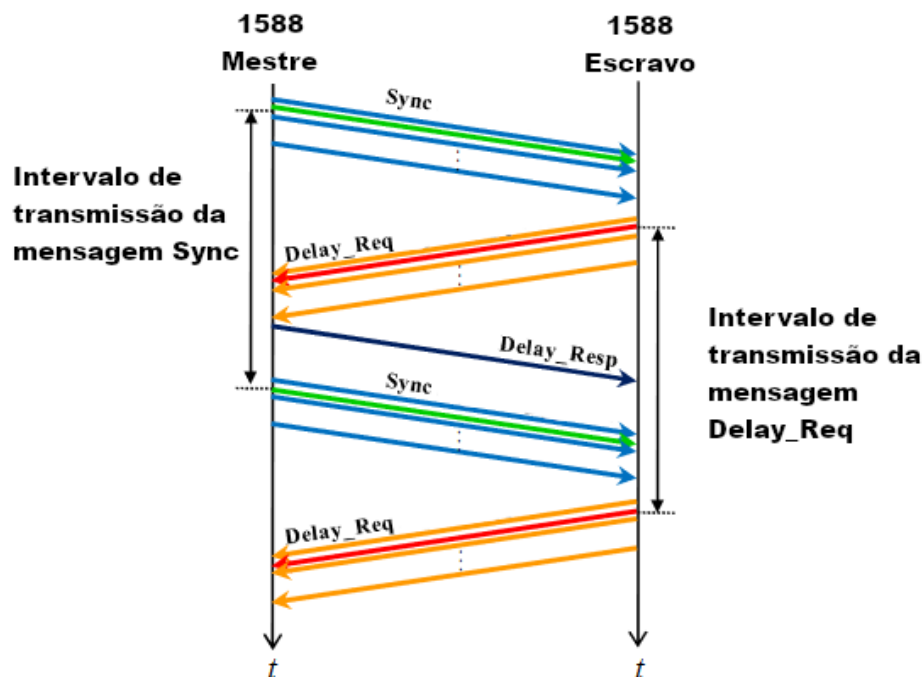
- Mensagens que contém informações relacionadas ao tempo usado para sincronizar os relógios pela rede: *Sync*, *Follow\_Up*, *Delay\_Req* and *Delay\_Res*;
- Mensagens para medir atrasos no meio de comunicação para que possam ser compensados pelo sistema: *Pdelay\_Req*, *Pdelay\_Resp* e *Pdelay\_Resp\_Follow\_Up*;
- Mensagens de anúncio que são utilizadas pelo algoritmo *Best Master Clock* (BMC) para construir a hierarquia de relógios.
- Mensagens de gerenciamento que são utilizadas pelo gerenciamento de rede para monitorar, configurar e manter um sistema PTP.
- Mensagens de sinalização para comunicações não críticas de tempo entre os relógios.

Durante a primeira etapa da sincronização, todos os relógios transmitem suas mensagens e propriedades a todos os outros. Tais dados são utilizados pelo BMC (YU; LI, 2009), que é executado independente em qualquer nó, para determinar automaticamente qual relógio é o mais preciso. Assim, esse melhor relógio se torna o mestre e todos os escravos passam a sincronizar com ele.

Conforme demonstrado na Figura 10, em um período predefinido, o mestre envia

para todos sua hora atual em uma mensagem “*Sync*” e, em seguida, uma correção em uma mensagem de “*Follow\_UP*”. Os escravos conseguem compensar a variação da rede enquanto aguardam um segundo recebimento do par de mensagens “*Sync*” e “*Follow\_UP*”. Como próxima etapa, cada escravo calcula o atraso no recebimento da mensagem do mestre. Para isso, envia uma mensagem “*Delay\_Req*” para ele. Ao receber a mensagem “*Delay\_Res*” com o *timestamp*, o escravo calcula a média dos dois atrasos, mestre-escravo e escravo-mestre, e ajusta a hora do relógio.

Figura 10 – Sincronismo entre mestre e escravo utilizando PTP



Fonte: Adaptado de (MURAKAMI; HORIUCHI, 2010)

Por fim, vale destacar que a revisão do padrão IEEE 1588 do ano de 2008 adiciona um protocolo para equipamento de rede, tais como *switches* e roteadores, que corrige os *timestamps* nas mensagens PTP com o tempo gasto no trânsito pelo equipamento.

#### 2.1.4.4 Comparativo entre os métodos

Face ao exposto, o Quadro 2 mostra um comparativo entre as principais características e resultados dos métodos previamente mencionados.



Quadro 2 – Comparativo entre os métodos de sincronização de relógio

Característica	Método		
	PTP	NTP	GNSS
Abrangência	Poucas sub-redes	Ampla	Ampla
Interface de Comunicação	<i>Ethernet</i>	Internet	Satélite
Precisão	microssegundos	milissegundos	microssegundos
Hardware Específico	Interface de rede adicional	Não	Processador e Receptor
Recursos de Rede	Baixo	Moderado	N/A
Recursos de Processamento	Baixo	Moderado	Moderado

Fonte: Produção do próprio autor.

Dentre os métodos selecionados, o GNSS possui maior precisão, contudo necessita de receptores relativamente caros, antenas corretamente posicionadas e um cabeamento apropriado (KANNISTO et al., 2005). O fato do receptor ter necessidade de visada com o satélite pode ser um problema, uma vez que os dispositivos de medição localizam-se próximos ao nó da infraestrutura que será medida, o qual, geralmente, são localizados dentro de data centers. Segundo (SHARMA, 2005), normalmente é inviável a utilização desse método por razões de custo e esforço de implementação.

Por último, embora o PTP necessite de hardware adicional, ele oferece uma solução mais precisa que o NTP e mais barata que o GNSS. Sua aplicabilidade se restringe a LANs (WEIBEL, 2005), incluindo mas não limitado ao *Ethernet* (SHARMA, 2005) e a sistemas heterogêneos, onde os relógios de diferentes capacidades podem sincronizar uns com os outros utilizando poucos recursos de processamento e rede. Embora a precisão comum do NTP, através da internet, se dê na casa de milissegundos, sua precisão, caso utilize um servidor interno, pode ser na ordem de microssegundos.

### 2.1.5 Medição de latência

Com a evolução da tecnologia, as redes e sistemas de computadores têm se tornado cada vez maiores e mais complexas para abranger o maior número de aplicações possível. Algumas aplicações com interações em tempo real, conforme citado na Seção 1, são sensíveis ao tempo de resposta, isto é, são impossíveis de apresentar um correto funcionamento quando há uma grande latência na rede. Caso latência entre os pontos possa ser medido durante o funcionamento das aplicações é possível utilizar técnicas para compensação, a fim de se obter um desempenho aceitável da aplicação.

Métodos e ferramentas para medições a nível de infraestrutura tem desempenhado um papel imprescindível para garantir um bom desempenho das redes. (CALYAM et al., 2005) define como passivos os métodos de medição que não modificam ou criam tráfego adicional de pacotes na rede e, comumente, obtém informações como taxa de *bits* ou de pacotes, tempo de trânsito de pacotes, etc. Por outro lado, os métodos ativos injetam



pacotes de análise na rede para realizar a coleta de algumas métricas (CALYAM et al., 2005). As métricas coletadas são, em geral: atrasos, perdas e *jitter*.

Nesse contexto, há diversas métricas que avaliam a robustez, largura de banda e efeitos de mudanças de rotas dentro de uma rede. Por sua vez, para avaliar o congestionamento na rede e, conseqüentemente, a latência entre dois pontos, são medidos o atraso de uma via e o atraso de ida e volta, do inglês, respectivamente, *one-way delay* (OWD) e *round-trip time* (RTT).

De uma forma geral, segundo (ALMES; KALIDINDI; ZEKAUSKAS, 1999a) diz-se que OWD é o tempo medido entre o início da transmissão dos dados por um ponto, ou seja, o envio do primeiro *bit* de um pacote, até o recebimento do último *bit* pelo segundo ponto, isto é, o fim da transmissão. Já o RTT pode ser definido como o intervalo de tempo entre o instante que uma solicitação é realizada até o momento que a resposta desta é recebida pelo ponto solicitante (ALMES; KALIDINDI; ZEKAUSKAS, 1999b). Para (WANG; ZHOU; LI, 2004), os métodos para realizar medições podem ser classificados em dois tipos: ativo, quando injetam pacotes de análise da rede e, conseqüentemente, impactam em algumas propriedades do tráfego, e passivo, quando não interferem diretamente na rede. Assim, define-se o RTT e uma das formas de medição do OWD que é baseada em RTT como ativo. Ainda segundo (WANG; ZHOU; LI, 2004), mesmo que as formas passivas de medição do OWD não adicionem diretamente pacotes na rede, é necessário realizar o registro de todos os pacotes de análise enviados e recebidos nos pontos de medição e correlacionar os *timestamps* para realizar o cálculo do atraso. Com isso, mesmo o modelo passivo pode ocasionar impactos no tráfego com a propagação dos dados de medição, dependendo da infraestrutura de rede utilizada.

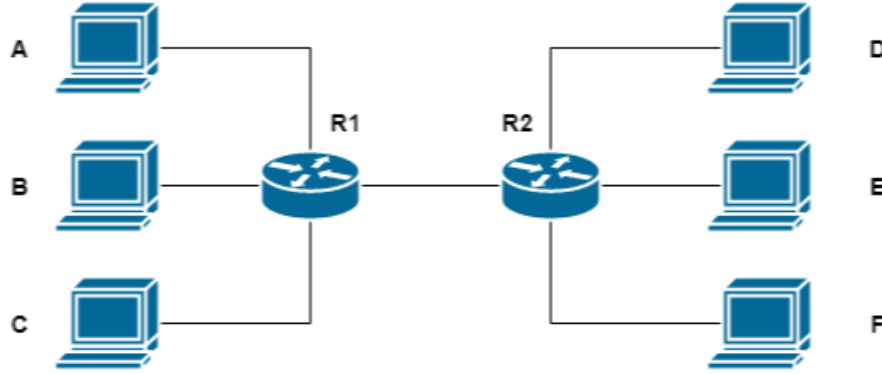
#### 2.1.5.1 Round-Trip Time (RTT)

A medição de RTT é importante para diagnosticar a conectividade da rede, medir o tamanho da janela de congestionamento e tempo limite de retransmissão de uma conexão, além da largura de banda ao longo de um caminho (JAIN; DOVROLIS, 2003). É dependente de uma gama de fatores, como natureza do meio de transmissão, distância entre origem e destino, largura de banda disponível na rede, tamanho do pacote de análise enviado e, embora esteja fortemente ligado à telecomunicação, seu conceito também é utilizado na Internet, nos sistemas de radares e comunicações através de satélites.

O utilitário mais conhecido e adotado para realizar a medição do RTT é o *ping*, que faz uso dos campos *Echo Request* e *Echo Reply* contidos no *Internet Control Message Protocol* (ICMP) (POSTEL, 1981). Seu princípio de funcionamento é simples e direto. Na Figura 11, onde são exibidas duas sub-redes conectadas através de roteadores, um dispositivo A (sub-rede 1) registra o momento do envio de uma solicitação (*Echo\_Request*) para o dispositivo F (sub-rede 2), que responde. Ao receber a resposta (*Echo\_Reply*), o

dispositivo A calcula o tempo decorrido até o recebimento. O caminho percorrido pelo pacote na ida e na volta é, respectivamente,  $\{A-R1-R2-F\}$ ,  $\{F-R2-R1-A\}$ . Caso o dispositivo não receba o *Echo\_Reply* até um determinado tempo limite, pode-se afirmar que não há conectividade entre eles.

Figura 11 – Exemplo de funcionamento da ferramenta *PING*



Fonte: Produção do próprio autor.

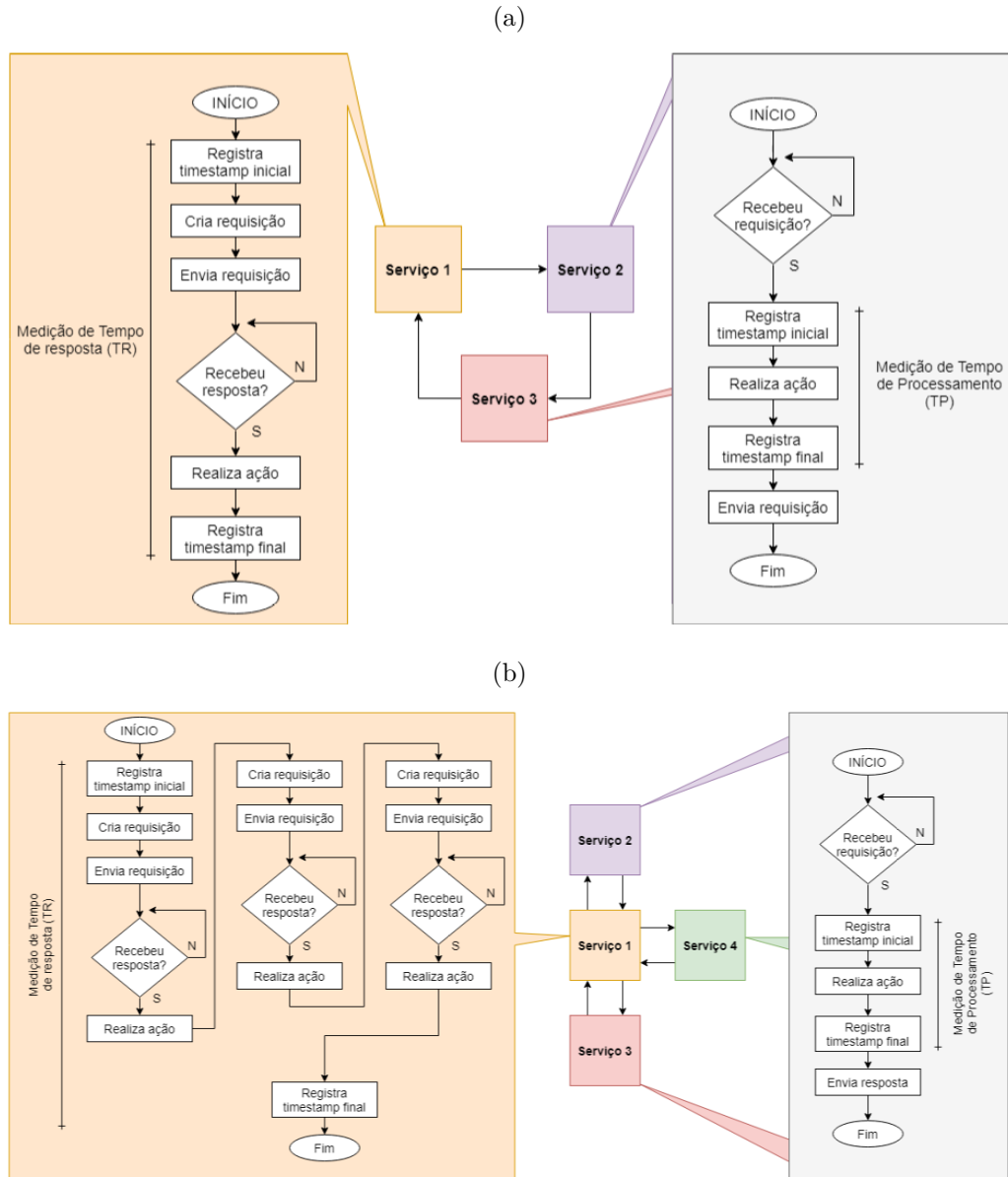
Dessa forma, pode-se simplificar o cálculo do tempo de *round-trip* ( $T_{RT}$ ) como a soma do tempo gasto pelo pacote para atravessar cada trecho do caminho, incluindo o tempo de processamento dentro dos elementos, conforme [Equação 2.1](#):

$$T_{RT} = T_{(A,R_1)} + T_{(R_1,R_2)} + T_{(R_2,F)} + T_{(F,R_2)} + T_{(R_2,R_1)} + T_{(R_1,A)}, \quad (2.1)$$

O conceito RTT pode também ser aplicado dentro do contexto de microsserviços para medição do tempo de resposta e do tempo de comunicação, desde que o serviço inicial seja mesmo que o final. Para medir a comunicação do sistema, há a necessidade de se realizar a instrumentação de todos serviços para conhecimento dos tempos de processamento.

A [Figura 12](#) demonstra exemplos de medição, dentro do contexto de microsserviços, do tempo de comunicação baseado nos tempos de resposta e processamento. Tanto na [Figura 12a](#) quanto na [Figura 12b](#), o Serviço 1 é responsável por registrar os *timestamps* que serão utilizados nos cálculos do tempo de resposta, enquanto os serviços restantes são responsáveis por calcular o tempo de processamento e enviá-los ao Serviço 1.

Figura 12 – Exemplos de utilização de RTT para medições.



Fonte: Produção do próprio autor.

Analogamente ao utilitário *ping*, o serviço inicial do ciclo registra o horário inicial da criação de sua solicitação  $T_{Request}$  e calcula o tempo total decorrido entre todos os serviços da cadeia ao receber sua resposta e registrar o *timestamp* final  $T_{Reply}$ . Sendo assim:

$$TR = T_{Reply} - T_{Request} \quad (2.2)$$

Assim, de posse do tempo de resposta, para encontrar o tempo de comunicação,

subtrai-se os tempos de processamento de cada serviço. A partir da [Equação 2.2](#):

$$TC^{(a)} = TR^{(a)} - TP_{(S1)}^{(a)} - TP_{(S2)}^{(a)} - TP_{(S3)}^{(a)} \quad (2.3)$$

$$TC^{(b)} = TR^{(b)} - TP_{(S1)}^{(b)} - TP_{(S2)}^{(b)} - TP_{(S3)}^{(b)} - TP_{(S4)}^{(b)} \quad (2.4)$$

A partir das equações mostradas, percebe-se que o RTT é útil para tais medições, uma vez que o mesmo método é válido para aplicação em diferentes topologias de microserviços. Entretanto, nesses casos, o tempo de comunicação obtido é tempo total de comunicação, isto é, o tempo gasto na comunicação de cada segmento é desconhecido.

Este tipo de método, para algumas aplicações, pode não ser vantajoso, uma vez que não há a possibilidade de descobrimento do segmento de maior atraso em uma situação de falha. Excetuando-se tal situação, esse método pode ser bem atraente, haja vista que não requer sincronização de relógios.

#### 2.1.5.2 One-Way Delay (OWD)

O OWD, conforme já mencionado, consiste do tempo decorrido desde o envio do primeiro *bit* do pacote na rede até a recepção do último *bit*. Há duas formas principais de se obter uma medida de OWD: passiva e ativa ([WANG; ZHOU; LI, 2004](#)). A forma de medição ativa, ou híbrida, é considerada um *trade-off* entre precisão e praticidade, uma vez que aumenta-se a facilidade de uso do método em troca da diminuição na exatidão da medida. Essa alternativa, além de não necessitar do sincronismo entre os relógios, faz uso do método RTT, discutido na [Seção 2.1.5.1](#), considerando que há uma perfeita simetria nos caminhos de ida e volta do dispositivo solicitante até o destinatário. Dessa forma, estima-se o tempo de *one-way* ( $T_{OW}$ ), isto é, o OWD como:

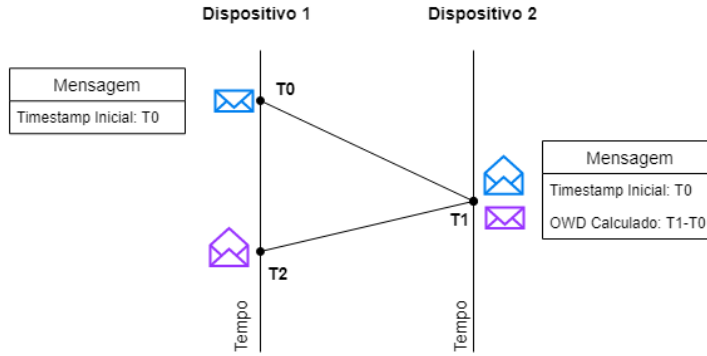
$$T_{OW} = \frac{T_{RT}}{2} \quad (2.5)$$

Já o método de medição passivo consiste na adição de uma identidade única e um *timestamp* do momento do envio para cada pacote. Assim, o pacote é encaminhado para o destino onde serão extraídas as informações mencionadas e o OWD é calculado com base no *timestamp* do momento em que o recebimento foi finalizado. Conforme pode ser visto na [Figura 13](#), o remetente escreve o *timestamp* local em um campo da mensagem e realiza o envio. Assim que a mensagem chega ao destinatário, este realiza a leitura e calcula a diferença subtraindo o seu *timestamp* local pelo *timestamp* do remetente ([LEE, 2015](#)). Dessa forma, generaliza-se o cálculo do OWD conforme a equação [Equação 2.6](#):

$$TC_{(X,Y)} = Timestamp_{(Y)} - Timestamp_{(X)} \quad (2.6)$$

Após o cálculo, o resultado é escrito na mensagem e devolvida à sua origem. Neste ponto, é importante destacar a necessidade da realização de uma sincronização dos relógios na origem e no destino, discutida na [Seção 3.1.2](#), caso não compartilhem do mesmo relógio.

Figura 13 – Medição passiva de OWD



Fonte: Produção do próprio autor.

Dentro do contexto de microserviço, diferentemente da medição do RTT, este método exclui a necessidade de que o início e o fim da cadeia de execução seja o mesmo ponto. Além disso, como a medição de latência é individualizada entre cada serviço, ele permite uma análise mais detalhada de desempenho da rede, sendo possível identificar o trecho onde há congestionamento e, conseqüentemente, maior influência nos tempos de comunicação e resposta. Outra vantagem a ser destacada é a independência para o cálculo do tempo de comunicação, isto é, não há mais necessidade de se estimar o tempo de resposta total e subtrair pelo tempo de processamento de cada serviço. Assim, pela medição de OWD, pode-se calcular diretamente o tempo total de comunicação citado no exemplo da [Figura 12](#), conforme mostra as equações [Equação 2.7](#) e [Equação 2.8](#):

$$TC^{(a)} = TC_{(S1,S2)}^{(a)} + TC_{(S2,S3)}^{(a)} + TC_{(S3,S1)}^{(a)} \quad (2.7)$$

$$TC^{(b)} = TC_{(S1,S2)}^{(b)} + TC_{(S2,S1)}^{(b)} + TC_{(S1,S3)}^{(b)} + TC_{(S3,S1)}^{(b)} + TC_{(S1,S4)}^{(b)} + TC_{(S4,S1)}^{(b)} \quad (2.8)$$

onde  $TC$  é descrito pela [Equação 2.6](#) já mencionada.

Por fim, vale mencionar que, embora necessite de uma sincronização de relógios, o OWD não possui obrigatoriedade do relógio do sistema ser fiel ao relógio do mundo real. Sendo assim, os relógios estando sincronizados entre si, já é suficiente para realizar a medição.

## 2.2 Estado da Arte

Atualmente, garantir o perfeito funcionamento dos microsserviços alocados nos sistemas distribuídos e das redes de computadores tem sido um assunto bem discutido juntamente com os métodos de medição. Em (CORREIA et al., 2018) é dito que acompanhar o desempenho dos microsserviços em ambientes de produção dinâmicos e flexíveis pode ser um grande desafio. E, para tentar contornar tal situação, foi desenvolvido um instrumento que utiliza o *tracing* para realizar a modelagem de microsserviços, numa tentativa de prever a distribuição do tempo de resposta e a zona ótima de operação de um serviço. Entretanto, para esse trabalho, o tempo de resposta foi considerado como o tempo que o sistema leva para processar uma requisição, excluindo o tempo de comunicação. Portanto, o presente trabalho se difere de (CORREIA et al., 2018) por medir o tempo de comunicação e incluí-lo no tempo de resposta.

Já no trabalho (CINQUE et al., 2018), é apresentado um estudo explorativo de monitoramento não intrusivo de sistemas de microsserviços, utilizando software desenvolvido pelos próprios autores. A medição do tempo de resposta é baseada em técnicas de *sniffing*, onde as mensagens trocadas entre os microsserviços na rede são capturadas e analisadas, gerando um registro de rastreamento. Em outro trabalho do mesmo autor, (CINQUE; CORTE; PECCHIA, 2019), embora a metodologia seja a mesma, o método de comunicação usado é através de *broker*, que é o mesmo utilizado nesta dissertação. Em ambos trabalhos, o tempo de resposta é obtido pela diferença do *timestamp* na hora da chegada de uma requisição e da sua resposta, contudo não foi implementado nenhuma forma de medição de tempo de comunicação. Portanto, isso representa a principal diferença entre os trabalhos citados e o presente trabalho.

Outra forma de utilização do *tracing* para obtenção do tempo de resposta pode ser vista em (KANUPARTHY et al., 2016), onde é apresentado o *YTrace*. Tal ferramenta foi desenvolvida pela empresa *Yahoo* para diagnosticar problemas de desempenho nas camadas de aplicação e rede, em seus sistemas que afetam a experiência do usuário. Nesse trabalho, é reforçado um dos objetivos do *tracing*: capturar a causalidade entre todos os serviços durante a execução de uma requisição. O *tracing* é realizado tanto de forma síncrona, relacionada à sessão do usuário, quanto assíncrona, realizada em dispositivos que não podem ser modificados para instrumentação, por exemplo, dispositivos de rede. Outro ponto relevante do trabalho é a consideração do tempo de resposta como a composição do tempo de processamento, das requisições de todos os serviços, adicionado do tempo de comunicação entre elas. Já a medição do tempo de comunicação é realizada de forma a abranger o dispositivo do usuário, a Internet, a rede do data center e o servidor de distribuição de conteúdos. É realizada de diversas maneiras, que vão desde instrumentação na sessão do usuário até medições na camada de transporte, conforme o dispositivo e ambiente. Por fim, vale destacar que a ferramenta desenvolvida é focada para uso interno

da própria empresa, tendo em vista suas diversas aplicações Web, sendo assim, o acesso a ela não está disponível. Isso impossibilita um levantamento de diferenças entre o presente trabalho e a ferramenta citada.

Há também trabalhos focados apenas em realizar a medição do tempo de comunicação, como por exemplo (FERRARI et al., 2018b) e (FERRARI et al., 2018a), onde, no contexto de IoT para a Indústria (IIoT), são propostas, respectivamente, duas metodologias: uma para estimativa experimental e caracterização do atraso na transferência de dados de padrões de protocolos de mensagens, usados em aplicações em escala mundial, e outra para investigar experimentalmente o atraso na transferência de dados entre máquinas, com arquitetura unificada de comunicações de plataforma aberta nativa, e plataformas em nuvem. Ambos trabalhos são baseados em arquitetura de *brokers* e utilizam o método de medição de OWD entre seus elementos. Entretanto, no primeiro trabalho, o OWD também é utilizado para estimar o RTT. Embora o método RTT não necessite de sincronização dos relógios, por estar realizando a medição do OWD, no primeiro trabalho, o autor destaca a importância dos relógios estarem iguais. Devido à necessidade da estabilidade do horário de referência UTC, é utilizada a sincronização por GNSS, mais especificamente pelo sistema de GPS, que é transportada para os dispositivos utilizando o NTP. Nesse caso, a arquitetura que disponibiliza a sincronização é composta por dois servidores NTPs equipados com receptores GPS. Por sua vez, de forma análoga, o NTP foi usado para realizar a sincronização dos relógios, via rede local, de um servidor com receptor GPS no segundo trabalho. Outro ponto de destaque é que, nos dois trabalhos, a medição do OWD é feita utilizando o serviço de troca de mensagens, isto é, a medição é realizada a nível da camada de aplicação do sistema, em vez de ser realizada na camada de comunicação. Isso acaba tendo um impacto na aplicação, consequentemente, aumentando o seu tempo de processamento, caso haja relação de causalidade do resultado da medição. Por último, é válido evidenciar que, pelos trabalhos possuírem cunho estatístico, não houve o desenvolvimento de uma interface que possibilitasse a leitura dos dados coletados. Ao contrário, todas as informações eram salvas em um arquivo de *logging*, o que difere deste presente trabalho, uma vez que a proposta é realizar a medição nas camadas de comunicação e exibi-las ao seu usuário.

Uma abordagem dentro do contexto IoT com aplicação na saúde, mas que também está focada na medição do tempo de comunicação é encontrada em (GOVINDAN; AZAD, 2015). Nesse trabalho, cujo objetivo é assegurar suficientemente a latência e a probabilidade de entrega de conteúdo em um sistema de assistência médica sem fio, é proposto a utilização de dispositivos *gateways* para comunicação entre os nós finais e o *broker*, dentro de uma rede de sensores sem fio. O trabalho busca estimar o tempo quando um nó envia um determinado conteúdo e o tempo no qual o conteúdo é, de fato, recebido pelo usuário. O sistema trabalha com conteúdo não persistente, isto é, o *publisher* esquece do conteúdo da mensagem assim que publicar o conteúdo para o *broker* e o *broker* só realiza o encaminhamento do conteúdo

a determinado nó caso solicitado. Com isso, há duas situações que podem ocorrer: o conteúdo presente e não presente no *broker* no momento da solicitação. Para ambos os casos, o cálculo do tempo de comunicação é realizada utilizando-se a metodologia de medição do RTT, ou seja, é registrado o tempo decorrido entre o início da conexão até a recepção da resposta contendo confirmação de publicação entre dois dispositivos, no caso de publicante, ou a confirmação de entrega dos pacotes, no caso do consumidor. Por fim, como o tempo de processamento dentro da aplicação não foi levado em consideração pelos autores, o conceito de medição de tempo de resposta se restringe ao tempo de comunicação, exclusivamente. Além disso, pelos dados extraídos das medições serem utilizados como parâmetros para o cálculo de probabilidade de entrega de conteúdo, juntamente para a política de garantia, não há qualquer forma de coleta dos resultados por parte do usuário final. Diferentemente de (GOVINDAN; AZAD, 2015), o presente trabalho considera o tempo de resposta como o tempo de processamento somado ao tempo de comunicação, além de exibi-lo em uma interface.

### 2.2.1 Considerações

Hoje, uma infraestrutura de rede eficiente fornece uma base vital para o sucesso das aplicações. Isso pode ser comprovado visto que cada vez mais há serviços críticos sendo desenvolvidos com seu funcionamento distribuído e em rede. As aplicações precisam ser bem gerenciadas, o que justifica a necessidade de um monitoramento completo, pois qualquer degradação de serviço pode afetar diretamente seu funcionamento. No entanto, o gerenciamento e o monitoramento precisam lidar com certas particularidades, especialmente no que tange a realizações de medições. Para o monitoramento do tempo de processamento, há padrões e ferramentas de *tracing* que já são bastante utilizadas, todavia para o tempo de comunicação cada grupo interessado descreve a sua metodologia. Dessa forma, dificulta-se o desenvolvimento de uma ferramenta capaz de realizar tal medição.

De fato, há muitos trabalhos focados em realizar medições de tempo de processamento utilizando *tracing*, como também há muitos outros com o objetivo de realizar medições de latência, seja por meio da medição de OWD ou RTT. Contudo, durante as revisões bibliográficas realizadas, não foram encontrados trabalhos que utilizaram ferramentas específicas para a medição do tempo de comunicação entre serviços de sistemas distribuídos. Além disso, não foram encontrados também trabalhos cujo objetivo fosse a exibição detalhada para o usuário de um tempo de resposta, composto de suas partes individuais.

Portanto, para realizar o monitoramento uma aplicação, de forma completa e precisa, é necessário que se tenham ferramentas apropriadas que permitam realizar e exibir ambas medições ao seu operador. Com isso, as análises de desempenho poderão ser feitas de maneira mais eficaz, o que facilitará a identificação mais precisa de algum problema.



## 3 Proposta de Solução

Este capítulo apresenta mais detalhes sobre a solução proposta neste trabalho. Além dos recursos necessários, apresentados ao fim do capítulo, é apresentada a metodologia proposta para o desenvolvimento do mecanismo de medição do tempo de comunicação em sistemas distribuídos baseados em microsserviços e para a exibição do tempo de resposta completo na ferramenta de *tracing*. Ressalta-se, aqui, que todas as decisões referentes à metodologia serão mostradas e discutidas no capítulo seguinte, na [Seção 4.1](#).

### 3.1 Metodologia

A metodologia aqui proposta permite que um sistema distribuído baseado em microsserviço, realize a medição do tempo de comunicação, juntamente do tempo de processamento, e exponha tais medições dentro da mesma ferramenta de *tracing*. Todo o processo pode ser dividido em cinco etapas: (1) [Definição preliminar](#) (2) [Sincronização de relógios](#) (3) [Modificação do ambiente e ferramentas de \*tracing\*](#) (4) [Avaliação das medições](#) (5) [Mitigação de imprevistos na ferramenta de \*tracing\*](#).

#### 3.1.1 Definição preliminar

Como há diversas alternativas que podem ser implementadas em cada etapa, faz-se necessário, como primeiro passo, realizar certas definições, mostradas nas seções a seguir, a fim de traçar um caminho a ser seguido. Tais definições são de fundamental importância, uma vez que as próximas etapas dependerão diretamente delas.

##### 3.1.1.1 Método de medição a ser adotado

Neste passo, deve-se definir qual será o tipo de medição do tempo de comunicação a ser adotado. Como principais métodos que já foram bem estudados, pode-se citar a medição de OWD, RTT e a híbrida, a qual utiliza um tipo para determinar o outro.

Cada método possui suas vantagens e desvantagens e, com isso, dois possíveis fatores determinantes, não exclusivos, para a escolha podem ser:

- Tipo predominante de cadeia de execução da aplicação, isto é, após análise, determinar se somente um tipo de aplicação será executada. Por exemplo, as aplicações que sejam do tipo onde o serviço inicial e o final seja o mesmo, pode-se optar pela medição de RTT.

- Método mais aderente à generalização das aplicações, isto é, caso o tipo predominante de cadeia de execução não possa ser determinado, deve-se optar pelo método que seja capaz de realizar as medições independente da arquitetura.

### 3.1.1.2 Camada da arquitetura na qual a medição será implementada

Desde o advento dos sistemas distribuídos, a tarefa de projetar e entender os sistemas tolerantes a falhas vem sendo discutida frequentemente. Em (NAYAK; JONE; DAS, 1994), o autor divide, conceitualmente, um sistema computacional em várias camadas de abstração, como hardware, sistema operacional e aplicação, onde o nível mais baixo de abstração, na hierarquia do sistema, é a camada de hardware e o nível mais alto é a aplicação. Nesse sistema, uma camada mais alta de abstração sempre recebe serviços das camadas mais baixas.

Tal divisão serviu de exemplo para diversos trabalhos e também como base para inserção de mais uma camada: a *middleware*. O termo *middleware* se refere à camada de software entre o sistema operacional, incluindo os protocolos de comunicação, e a aplicação distribuída. Seguindo o exemplo, (BAKAR; ATAN; YAAKOB, 2011) divide sua arquitetura em três níveis: aplicação, *middleware* e sensores. Por sua vez, conforme mostrado na Figura 1, (QUEIROZ, 2016) realiza a divisão em quatro camadas, sendo as três já mencionadas anteriormente, com o acréscimo da camada de comunicação.

Diante disso, nota-se a relevância da definição da camada onde serão realizadas as medições do tempo de comunicação. Esse procedimento deve levar em conta a arquitetura do sistema, a sobrecarga que será causada, o nível de abstração das camadas inferiores, entre outros fatores.

A medição do tempo de processamento a partir de ferramentas de *tracing*, comumente, é realizada na camada de aplicação. Por sua vez, a medição do tempo de comunicação pode ser feita tanto na camada de comunicação, *middleware* ou aplicação.

Um ponto que merece destaque é que muitas vezes o responsável pelo desenvolvimento da aplicação não é, necessariamente, o responsável pelo gerenciamento e manutenção da infraestruturas de rede. Assim, informações sobre latência acabam não sendo relevantes ao desenvolvedor e podem gerar um *overhead* desnecessário no sistema. Por isso, optar por incluir a medição de tempo de comunicação na camada de aplicação pode não ser vantajoso. Uma solução para essa situação é realizar a implementação nas camadas mais inferiores.

### 3.1.1.3 Aspectos da medição em relação ao elemento coordenador das trocas de mensagens

Após definir a camada de atuação, deve-se realizar também definições e configurações deste elemento. Pois, o seu mecanismo pode permitir o funcionamento de diversas

maneiras: armazenamento, enfileiramento ou descarte da mensagem quando o destino não está disponível, envio da confirmação de recebimento da mensagem nele, aguardo de confirmação do recebimento da mensagem pelo destinatário e etc.

Deve-se atentar a este procedimento, uma vez que mal configurado, o elemento poderá inserir uma carga extra na rede, bem como influenciar negativamente na medição. Para uma descrição mais clara, considere uma situação onde se deseja medir o RTT entre dois serviços e o coordenador das trocas de mensagem esteja configurado para sempre confirmar o recebimento de mensagens. Uma vez que um serviço está aguardando somente a resposta do outro serviço, ao receber a mensagem de confirmação vinda do coordenador em vez da resposta solicitada, pode haver um enfileiramento de mensagens e, conseqüentemente, influenciar no tempo de chegada da resposta.

Por fim, vale mencionar que, em geral, os elementos coordenadores das trocas de mensagens numa rede não tem suporte para registro do horário de entrega, nem confirmação direta de entrega da mensagem ao destinatário. Sendo assim, caso o elemento coordenador escolhido não tenha esse suporte, essa ação deve ser realizada por uma aplicação externa.

#### 3.1.1.4 Ferramenta de *tracing* a ser utilizada

A última definição a ser realizada deve ser a ferramenta de *tracing*. Esta deve ser a mais completa e flexível em relação aos seus recursos para que possa ser adaptável e compatível com as definições realizadas até então. A ferramenta de *tracing* em questão deve ter o foco na identificação do fluxo de serviços e na medição do tempo de processamento, na camada de aplicação, entretanto deve ser adaptável o suficiente para que o tempo de comunicação possa ser inserido, tendo assim, o tempo de resposta completo.

Dessa forma, o principal requisito para as ferramentas de *tracing* é que o seu código fonte seja aberto a modificações. Em segundo lugar, outro requisito importante é o suporte ao padrão *OpenTracing*, uma vez que a adoção de tal padrão beneficia o sistema na interoperabilidade entre softwares, caso necessário, e na comprovação de um correto funcionamento.

### 3.1.2 Sincronização de relógios

Uma vez que a sincronização dos relógios é um componente vital para o correto funcionamento de diversas aplicações e ferramentas, esse procedimento deve ser realizado para garantir que todos os dispositivos do sistema estejam em conformidade em relação a um mesmo horário. Nesta etapa, pode-se utilizar a sincronização a um relógio físico, não abordada neste trabalho, e a lógica, a qual existem várias metodologias comumente aceitáveis e utilizadas.

A tarefa de sincronização é um relevante desafio, uma vez que o cliente deve receber

informações atualizadas a uma taxa confiável para compensar as variações esperadas no relógio local. Para isso, deve-se ter um conjunto de fontes de referência, como relógios atômicos ou vários GNSS, sincronizados, direta ou indiretamente, de acordo com os padrões nacionais que devem fornecer continuamente a hora local baseada no UTC.

É importante salientar que, atualmente, há diversos softwares que fornecem a sincronização baseada em GNSS ou relógios atômicos, através do NTP pela Internet, não sendo necessária a implementação de um servidor dedicado para isso. Entretanto, em (MILLS, 1991), afirma-se que a sincronização de tempo precisa e confiável em um ambiente de rede só pode ser alcançada através de um projeto integrado.

Por fim, nesta etapa, deve-se verificar a precisão requerida de acordo com a definição realizada na [Seção 3.1.1](#) e, com isso, optar pelo método mais aderente à necessidade.

### 3.1.3 Modificação do ambiente e ferramentas de *tracing*

Muitos problemas em sistemas distribuídos ocorrem devido a falhas na rede ou hardware, *bugs* em software ou até mesmo uma configuração errada. A causa nem sempre é evidente ou aparece imediatamente. Daí, vem a necessidade de se utilizar ferramentas de *tracing*. Entretanto, muitas dessas ferramentas não coletam todas as informações relevantes sobre um determinado evento.

Embora tenha havido um grande progresso no desenvolvimento dessas ferramentas, elas ainda não cobrem a totalidade de informações necessárias para diagnosticar um problema e, geralmente, atacam somente uma propriedade do sistema. Caso se faça uso de tal utilitário para capturar um evento ao qual a ferramenta não está preparada, ela poderá apresentar poucas informações, agrupamentos indesejados ou até mesmo exposição incorreta das informações.

Dessa forma, esta etapa deve ser realizada de forma a adicionar mais instrumentação ao sistema, isto é, permitir que tanto as ferramentas quanto o ambiente estejam habilitados a realizar as medições necessárias, além das que já são realizadas. Esse procedimento é de suma importância, tendo em vista a correta adequação para tal fim.

Para cumprir esse objetivo, deve-se estudar e analisar os pontos que serão modificados no código fonte das ferramentas e na infraestrutura, levando em consideração as definições realizadas na [Seção 3.1.1](#).

Para as modificações em relação às ferramentas, o ponto de interesse deve ser justamente as funções que realizam a coleta e transporte das informações para o núcleo do programa. Uma vez que as ferramentas não apresentam suporte nativo às informações de tempo de comunicação, essas devem ser inseridas através das modificações. Ao avaliar o código-fonte e o funcionamento da ferramenta, deve-se buscar saber se é a própria ferramenta que realiza o cálculo das medições de processamento, baseado nos *timestamps*

recebidos, para que as alterações sejam realizadas a fim de possibilitar a coleta dos *timestamps* e a criação do *span* de comunicação. Assim, os mesmos padrões e métodos utilizados na medição do tempo de processamento poderão ser usados para calcular o tempo de comunicação, tendo apenas como diferença o *span* com início e fim em serviços diferentes.

Já no que diz respeito à arquitetura do sistema distribuído, deve-se buscar os pontos onde as comunicações são realizadas, isto é, onde as aplicações fazem o envio e o recebimento das mensagens trocadas na rede, para que nesses pontos possa ser adicionado algum método de medição. Adicionalmente, pode-se considerar que o tempo gasto com as conversões necessárias estão agregadas ao tempo de comunicação.

Além disso, a relação de causalidade registrada pela ferramenta, em outras palavras o contexto dos *traces*, deve ser sempre transportada nos metadados da mensagem. Para que, com as modificações, se consiga criar relações de causalidade entre os serviços anteriores e as medições.

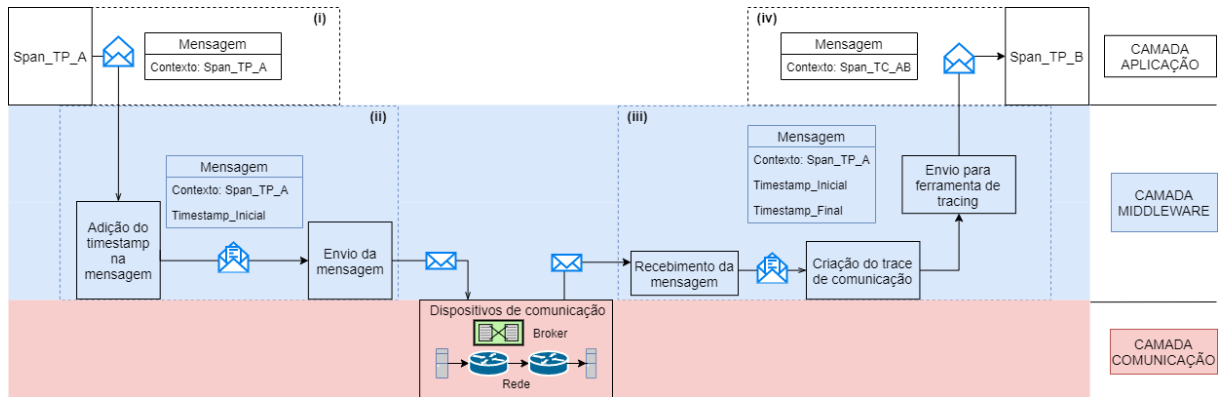
Com isso, torna-se possível a adição dos seguintes passos nas funções de envio e recebimento mencionadas anteriormente.

- Envio da mensagem:
  - Registro do *timestamp* inicial;
  - Encapsulamento do *timestamp* na mensagem a ser enviada;
- Recebimento da mensagem:
  - Registro do *timestamp* final;
  - Extração do contexto dos *traces* contida na mensagem;
  - Criação de um novo *span* com os *timestamps* inicial e final;
  - Exportação do *span* de comunicação criado para a ferramenta de *tracing*;
  - Injeção do novo *trace* como contexto na mensagem;
  - Disponibilização da mensagem com novo contexto para usuário.

A Figura 14 mostra uma visão geral da medição do tempo de resposta de um sistema após a aplicação desta metodologia. Em (i) o Serviço A realiza medição de tempo de processamento na camada de aplicação com a criação de um *Span* (*Span\_TP\_A*), e o injeta na mensagem, que por ser o primeiro é considerado o pai do evento seguinte; (ii) a função de envio, na camada *middleware*, ao ser chamada pelo Serviço A, registra o *timestamp* inicial e a adiciona no corpo da mensagem, junto com o contexto e faz o envio da mensagem pela rede; (iii) ainda na camada *middleware*, a biblioteca de comunicação do serviço B recebe a mensagem, extrai as informações, registra o *timestamp* final e antes de disponibilizar a mensagem é criado e exportado para a ferramenta um novo *Span* (*Span\_TC\_AB*), tendo as informações dos *timestamps* inicial e final para cálculo do tempo de comunicação e o *Span\_TP\_A* como pai. Em seguida, o contexto original (*Span\_TP\_A*)

da mensagem é substituído pelo novo ( $Span\_TC\_AB$ ) e a mensagem é entregue ao Serviço B; (iv) o Serviço B, na camada de aplicação, faz uma medição do tempo de processamento ( $Span\_TP\_B$ ), tendo como *span*-pai o contexto extraído da mensagem recebida, isto é  $Span\_TC\_AB$ .

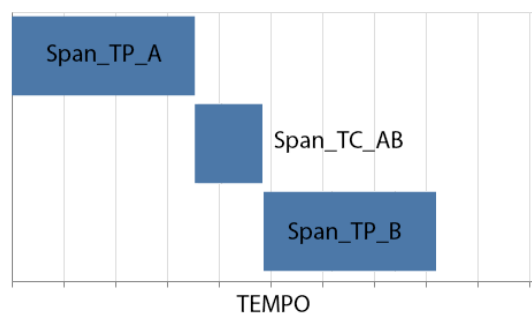
Figura 14 – Visão geral da medição do tempo de resposta de um sistema após a aplicação desta metodologia



Fonte: Produção do próprio autor.

Dessa forma, após as trocas de mensagens, a própria infraestrutura deverá estar apta para determinar o tempo total gasto na comunicação e será possível extrair da ferramenta de *tracing* o tempo de resposta completo, composto das duas propriedades: processamento e comunicação. Como pode ser visto na Figura 15, que é um ilustrativo de como é um *trace* após a aplicação da metodologia, os *spans*  $Span\_TP\_A$  e  $Span\_TP\_B$  são, respectivamente, os tempos de processamento dos serviços A e B. E o  $Span\_TC\_AB$  é o *span* de comunicação entre ambos serviços, tendo o  $Span\_TP\_A$  como *span*-pai por ser a origem da comunicação.

Figura 15 – Exemplo de *trace* após a aplicação desta metodologia



Fonte: Produção do próprio autor.

### 3.1.4 Avaliação das medições

Após realizadas as modificações, é muito importante verificar se a medição condiz com a realidade do sistema antes de inseri-la em alguma aplicação. Verificar se uma medição está sendo feita de maneira correta em uma aplicação já implementada é muito difícil, haja vista que não há parâmetros anteriormente coletados que indiquem ausência de erros. Com isso, deve-se criar formas para avaliar os resultados das modificações antes de qualquer utilização.

Uma abordagem promissora é a criação de serviços de depuração que monitorem e simulem o completo funcionamento de uma aplicação. Nestes serviços, deve haver diversas redundâncias tanto nos registros de *logging* quanto nas medições, com o intuito de fornecer a maior quantidade de informações e possibilitar a comparação de resultados preliminares. Deve-se buscar simplificar as funções geradoras de informações para que os serviços sejam leves e apresentem impactos reduzidos no sistema. Além disso, alguns pontos devem ser observados:

- Simulação de tarefa para medição do tempo de processamento;
- Propagação do *tracing* pela rede;
- Medição do tempo de comunicação;
- Coerência da medição;
- Relações de causalidade mantidas.

Para o penúltimo ponto mencionado acima, coerência da medição, é válido mencionar que deve ser observado eventuais problemas que possam alterar a medição, por exemplo, problema de sincronismo entre os relógios dos servidores. Além disso, pode ser realizado um estudo quantitativo do tamanho da mensagem enviada em relação ao tempo gasto no trânsito dela, para se ter uma estimativa que possa ser utilizada em comparações das mensagens. Assim, pode-se estimar a faixa de tempo em que uma mensagem de determinado tamanho leva trafegando entre dois pontos na rede e avaliar se o tempo medido possui uma discrepância muito grande em relação à estimativa.

### 3.1.5 Mitigação de imprevistos na ferramenta de *tracing*

Por fim, como última etapa, a mitigação de imprevistos deve ser realizada quando não houver mais possibilidade de correções, isto é, caso o sistema apresentar o correto funcionamento e medições, conjuntamente com a manifestação constante de alguma anomalia. Isso pode se dar por diversos fatores, tais como: custo de processamento em alguma biblioteca, atraso adicional ao usar determinado protocolo e etc. Assim, este procedimento busca criar uma possível alternativa que reduza ou solucione o problema em questão e que não é nativamente suportada pelos serviços e bibliotecas já em uso.

## 4 Implementação, Experimentos e Resultados

Para poder validar a proposta deste trabalho, este capítulo descreve como a metodologia foi aplicada e quais resultados foram obtidos. Primeiramente, são apresentados recursos, e, em seguida, os aspectos gerais da implementação, ou seja, são detalhadas as decisões tomadas em cada etapa e como foram implementadas. Na seção seguinte, é apresentada e discutida a aplicação da metodologia em duas situações.

### 4.1 Implementação

Inicialmente, vale lembrar que a implementação deste trabalho está focado em apresentar uma solução funcional para o Espaço Inteligente Programável montado no Laboratório de Visão Computacional e Robótica da UFES, que, conforme já foi explicado, é um sistema distribuído baseado em microsserviços com a comunicação através de *broker*. Com isso, toda a implementação aqui descrita está restrita às diretrizes do projeto do espaço inteligente, isto é, utilizam-se linguagens de programação, bibliotecas, ferramentas e padrões já estabelecidos.

#### 4.1.1 Recursos

Para que todas as etapas pudessem ser desenvolvidas, resultando no presente trabalho, foi necessária a utilização de recursos de hardware e software.

- Para realizar a sincronização dos relógios, foi utilizado o software *Chrony* (CURNOW, 2020), para sistemas operacionais Linux;
- O sistema distribuído baseado em microsserviços utilizado foi o Espaço Inteligente Programável do Laboratório de Visão Computacional e Robótica da UFES composto por:
  - Servidor Modelo *Dell PowerEdge T410* com processador *Intel Xeon E5504* @2.00GHz e 4 GB de RAM;
  - *Desktop Dell Optiplex* com processador *Intel i5-3570* @3.40GHz e 16 GB de RAM;
  - *Desktop* com processador *Intel i7-6850K* @3.60GHz, 128 GB de RAM e *GPU GeForce GTX 1070*;
  - Câmeras IP, Modelo *BlackFly PointGrey BLFY-PGE-09S2*;
  - Código fonte baseado em visão computacional já implementado (QUEIROZ, 2016);



- Software orquestrador de serviços: *Kubernetes* (GOOGLE, 2020);
- Para todo o desenvolvimento, foram utilizadas as linguagens *Python* e *C++* e o Ambiente de Desenvolvimento Integrado (IDE) utilizado foi o *Visual Studio Code* (MICROSOFT, 2020), sempre na versão mais atual;
- Para realização do *tracing*, foi utilizado a ferramenta *Zipkin* (ZIPKIN, 2020);
- De forma a cumprir os requisitos necessários para desenvolvimento de serviços para o PIS, foi utilizado o software *Docker* (DOCKER INC., 2020);

Concluindo, vale comentar que para a realização da revisão bibliográfica, o acesso a periódicos esteve assegurado via portal de periódicos da CAPES, que pôde ser acessada na rede interna da UFES e remotamente via *proxy*.

#### 4.1.2 Definição preliminar

Uma vez que este trabalho teve como ambiente de atuação o PIS, algumas características estavam previamente definidas, contudo outras de mesma importância ainda necessitaram de definições, conforme descritas a seguir.

##### 4.1.2.1 Método de medição a ser adotado

Seguindo a metodologia proposta, após a realização de uma análise quantitativa das aplicações desenvolvidas para o PIS, pode-se perceber que em nenhuma o primeiro serviço era também o último na cadeia de funcionamento. Dessa forma, o método de medição RTT e híbrido foram descartados e, conseqüentemente, optou-se pela medição de OWD. Tal escolha demonstrou-se apropriada uma vez que a disposição da maioria dos microsserviços eram em cadeia contínua. Além disso, a medição de OWD mostrou-se o método mais aderente à generalização, pois cada medição era realizada em um trecho entre dois serviços, independente se havia retorno entre eles ou não. Nessa última situação, caso fosse necessário a obtenção do RTT, bastaria ser realizada a soma de ambos OWD, de ida e retorno.

Um fator extra levado em consideração foi que, caso um tópico tivesse vários consumidores, a medição do RTT seria muito mais complexa em relação à OWD. Diferentemente do RTT, no OWD a medição é realizada pelo serviço de destino. Sendo assim, bastaria o envio do *timestamp* inicial, encapsulada na mensagem, para diversos consumidores que já seria suficiente para realização do cálculo da medição da comunicação entre cada serviço de destino e o publicante, pois cada serviço consumidor registraria o seu *timestamp* final, no momento do recebimento da mensagem, e realizaria o cálculo individualmente.

#### 4.1.2.2 Camada da arquitetura na qual a medição será implementada

Como no PIS foi convencionada a divisão em quatro camadas e que toda a estrutura necessária para troca de mensagens estaria localizada nas camadas de *middleware* e comunicação, essas foram escolhidas como local para implementação das medições referentes à comunicação. A medição de tempo de processamento, não presente no escopo deste trabalho, é realizada na camada de aplicação.

Nas camadas escolhidas, estão algumas bibliotecas necessárias para interlocução entre camadas e definições de comunicação herdadas pela utilização do PIS, por exemplo, tipo de protocolo de comunicação para infraestrutura e *broker* com suporte ao protocolo escolhido que, nesse caso, são, respectivamente o *Advanced Message Queuing Protocol* (AMQP) (O'HARA, 2020) e o *RabbitMQ* (RABBIT TECHNOLOGIES, 2020).

Algumas das razões pela escolha do protocolo AMQP foram a sua interoperabilidade e a quantidade de implementações do protocolo. Usualmente, as implementações desse protocolo são associadas a um *broker*, que nesse caso, o escolhido foi o *RabbitMQ*, devido, entre outros motivos, a sua capacidade de operação com diversos outros protocolos (QUEIROZ, 2016).

#### 4.1.2.3 Aspectos da medição em relação ao elemento coordenador das trocas de mensagens

As definições e configurações do elemento coordenador das trocas de mensagens, nesse caso, o *broker*, em sua totalidade, foram definidas no momento da criação do PIS. Sendo assim, anteriormente ao desenvolvimento do presente trabalho.

Um *broker* é um elemento intermediário na comunicação que tem como função o recebimento e encaminhamento de mensagens entre os clientes, coordenando, assim, as trocas de mensagens de um sistema. Com a sua utilização, é possível monitorar e centralizar o fluxo de mensagens para um único ponto. Dessa forma, do ponto de vista da aplicação, somente o seu endereço é necessário para que possa ocorrer a comunicação.

As configurações utilizadas neste trabalho estão listadas no [Quadro 3](#):

Quadro 3 – Configurações pré-definidas do broker

Configuração	Estado
Confirmação de entrega ao consumidor	Não
Confirmação de entrega ao publicante	Não
Retransmissão de mensagem	Não
Armazenamento de mensagem	Não
Tamanho da fila de mensagens	64 mensagens
Tempo de vida da mensagem	1 segundo
Consumidor exclusivo	Não

Fonte: Produção do próprio autor.

Além disso, foi necessário também fazer a definição que todo o tempo gasto, seja processamento ou comunicação, no *broker* seria considerado como tempo de comunicação, isto é, estaria embutido no OWD. Uma vez que o resultado final de tais procedimentos internos do *broker* é justamente o encaminhamento de mensagens, tomou-se tal decisão visando simplificar a medição, pois não é de interesse do PIS um detalhamento de quanto tempo uma mensagem gasta para chegar e sair do *broker* e sim de quanto tempo gastou de um serviço origem a um serviço destino.

#### 4.1.2.4 Ferramenta de *tracing* a ser utilizada

Assim como no caso anterior, a ferramenta de *tracing* também foi definida em trabalhos realizados no PIS antes deste. Sendo assim, como pode ser visto em (PICORETI et al., 2018), a ferramenta escolhida foi o *Zipkin*.

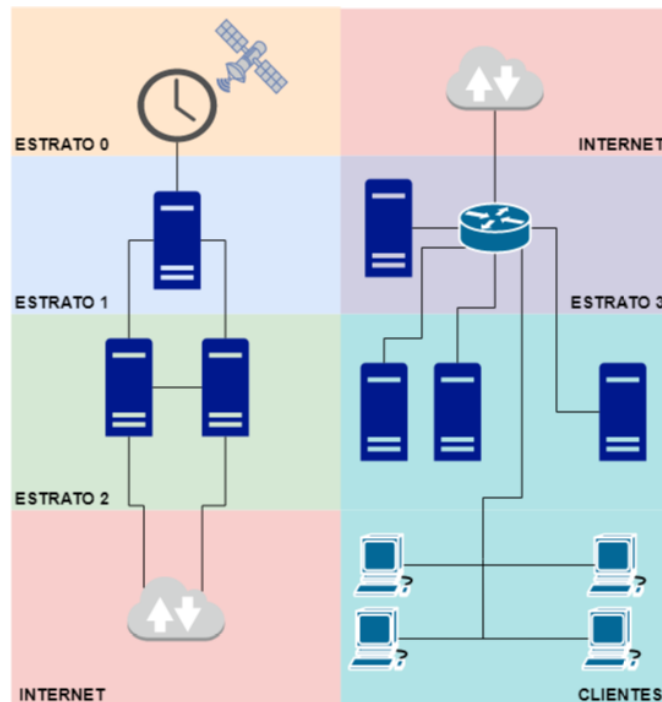
O Zipkin (ZIPKIN, 2020), conforme destacado em Seção 2.1.3.1, é uma ferramenta para realização de *tracing*, focada em sistema distribuído baseado em microsserviços. Esta ferramenta recebe os dados de *tracing* do sistema, trata as informações e as exibe para o usuário em forma de gráficos e diagramas. Sua escolha foi devido ao seu suporte no transporte dos dados, já que trabalha com mais protocolos que as outras ferramentas.

### 4.1.3 Sincronização de relógios

Uma vez que o método de medição de tempo escolhido para este trabalho foi o OWD, a sincronização de relógios tornou-se imprescindível para o correto funcionamento. Estando em concordância com as necessidades do PIS, juntamente com os recursos disponíveis, optou-se pela utilização do protocolo NTP através da Internet. Essa solução é adotada em muitas instituições ao redor do mundo, devido à sua facilidade e custo de implementação.

A estrutura utilizada neste trabalho é semelhante à mencionada na Seção 2.1.4.2 e pode ser vista na figura a seguir. Conforme mostra a Figura 16, primeiramente, decidiu-se que somente um servidor pertencente ao laboratório, localizado no estrato três, seria conectado e sincronizado com servidores públicos NTP do estrato dois, através da internet. Em seguida, todos os dispositivos, denominados clientes na figura, foram configurados para realizar a sincronização com o servidor local, localizado no estrato três, através da rede interna. Tal procedimento foi adotado visando garantir o funcionamento do sistema, mesmo que o servidor local perca a sincronização com os servidores NTP. Uma vez que todos os computadores da rede interna estejam sincronizados entre si, a sincronização com os servidores externos não possuem necessidade de serem precisas. Além disso, esse sincronismo interno possibilita a precisão resultante na ordem de microssegundos.

Figura 16 – Estrutura de servidores utilizada para sincronização dos relógios



Fonte: Produção do próprio autor.

Para implementar o procedimento mencionado acima, utilizou-se o software *Chrony* em todos os dispositivos do laboratório. Esse é um aplicativo que roda em segundo plano no computador, monitora o relógio e o status de conectividade com o servidor. Caso o relógio local necessite de ajustes, o programa vai ajustando o horário suavemente, sem causar traumas programáticos que podem ocorrer caso o relógio seja simplesmente reajustado para um novo horário.

A configuração é realizada através de um arquivo de configuração onde são especificados o servidor, ou servidores, e algumas diretrizes relacionadas à frequência e precisão da sincronização, que servem tanto para o dispositivo atuar como servidor ou cliente. No caso deste trabalho, a configuração realizada foi somente na segunda linha do arquivo, onde foi alterada para se referir ao servidor local. O restante da configuração foi mantida com seus valores padrões.

Por fim, vale mencionar que, assim como o *Chrony*, há diversos outros softwares que desempenham as mesmas funções, entretanto os fatores que determinaram a escolha foram:

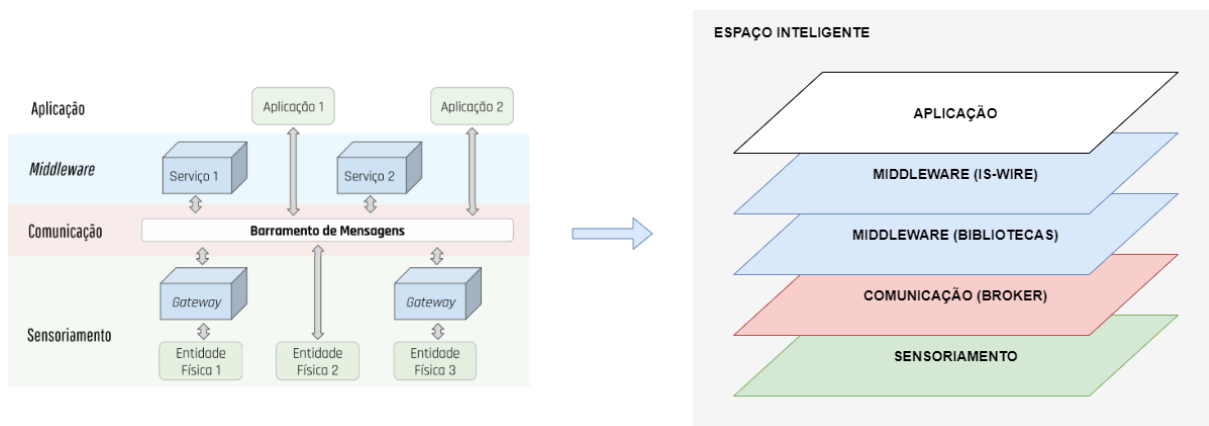
- O *Chrony* consegue sincronização com servidores de horário mais rápido;
- É o mais recomendado para dispositivos que não funcionam constantemente;
- Compensa flutuações nas frequências do relógio, por exemplo, quando um computador hiberna ou entra no modo de suspensão;

- Realiza ajuste para atrasos e latência da rede.
- Após a sincronização da hora inicial, o *Chrony* modifica diretamente o relógio. Isso garante intervalos de tempo estáveis e consistentes para serviços e aplicativos do sistema.
- Há possibilidade de funcionamento mesmo sem uma conexão de rede. Nesse caso, o dispositivo pode ser atualizado manualmente.

#### 4.1.4 Modificação do ambiente e ferramentas de *tracing*

Para que as medições pudessem ser realizadas, foi necessário executar o procedimento de habilitação do PIS e das ferramentas de *tracing*. Para isso, realizou-se um estudo aprofundado sobre o desenvolvimento e funcionamento das camadas de *middleware* e comunicação do ambiente. De posse do código fonte do PIS, juntamente das suas aplicações, percebeu-se que a camada *middleware* original é composta por bibliotecas que fornecem diversas ferramentas e recursos para uma aplicação, juntamente de outra biblioteca que realiza a abstração destas. Desta forma, visualizou-se a possibilidade de divisão da camada de *middleware* original em duas subcamadas, uma contendo todas as bibliotecas e outra contendo a biblioteca de abstração do restante das camadas inferiores, conforme [Figura 17](#).

Figura 17 – Arquitetura em camadas do PIS



Fonte: Produção do próprio autor.

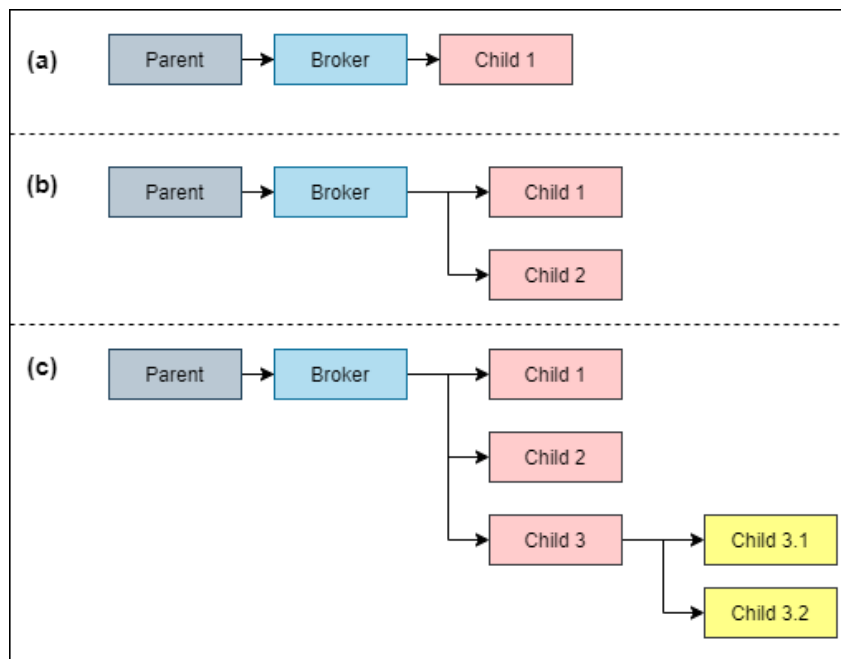
A subcamada "*middleware* (bibliotecas)" é onde estão localizadas as bibliotecas que servem de base para utilização das ferramentas, como *broker* e *Zipkin*. Já a subcamada superior à camada *middleware*, aqui chamada de "*middleware (is-wire)*", é responsável por fazer a abstração da "*middleware* (bibliotecas)" para a camada de aplicação. No desenvolvimento de aplicações para o PIS, é somente a biblioteca *is-wire* que precisa ser utilizada por um desenvolvedor, todas as bibliotecas da camada inferior, necessárias para desempenhar qualquer atividade, estão embutidas e abstraídas nesta.

Assim, com esse entendimento, pode-se buscar alternativas para se alcançar os objetivos do presente trabalho. Foram desenvolvidas três soluções que atacassem o problema.

#### 4.1.4.1 Solução 1

Inicialmente, foi criada uma aplicação extra para realizar a medição por meio do recebimento de *timestamps*, via protocolo *Hyper Text Transfer Protocol* (HTTP). A aplicação, desenvolvida na linguagem *NodeJS*, implementa um servidor *Representational State Transfer* (REST) que fica aguardando a chegada das mensagens no formato *JavaScript Object Notation* (JSON). Para a criação, optou-se por utilizar o padrão *OpenTracing* e adaptá-lo com inserção de um novo campo. Assim, ao receber uma mensagem, a ferramenta verificava os campos *SpanParent* e *timestamp* e os adotava como referencial para todas as mensagens seguintes que chegassem com o mesmo *SpanParent*. Em seguida, era calculada a diferença de *timestamps* das mensagens com o referencial e o resultado era exibido numa interface gráfica na *web*. Três testes de funcionamento foram realizados, conforme a [Figura 18](#) e seus resultados são mostrados na [Figura 19](#).

Figura 18 – Testes de funcionamento da Solução 1



Fonte: Produção do próprio autor.

Na [Figura 19a](#), é exibido na interface gráfica o resultado de todos os três *traces* realizados nesse experimento. Em seguida, [Figura 19b](#), mostra o tempo de comunicação medido entre um *publisher* e um *consumer*, [Figura 18a](#). Por sua vez, [Figura 19c](#) mediu-se o tempo de trânsito da mensagem para dois consumidores, [Figura 18b](#). E, por fim, na [Figura 19d](#), a medição foi realizada em dois momentos: no primeiro, três consumidores e, no segundo, dois consumidores, sendo que o publicante era anteriormente um consumidor, [Figura 18c](#).

Figura 19 – Serviço de medição de comunicação de *traces* por recebimento de timestamps

## Communication Traces

Span Name: Parent.ID.1 - Timestamp: 1557530377248	Span Name: Parent.ID.1 - Timestamp: 1557530377248
Span Name: Child.ID.1.1 - Timestamp: 1557530377301 - Duration: 53 ms	Span Name: Child.ID.1.1 - Timestamp: 1557530377301 - Duration: 53 ms
Span Name: Parent.ID.2 - Timestamp: 1557530485491	(b)
Span Name: Child.ID.2.1 - Timestamp: 1557530485527 - Duration: 36 ms	Span Name: Parent.ID.2 - Timestamp: 1557530485491
Span Name: Child.ID.2.2 - Timestamp: 1557530485819 - Duration: 328 ms	Span Name: Child.ID.2.1 - Timestamp: 1557530485527 - Duration: 36 ms
Span Name: Parent.ID.3 - Timestamp: 1557530831159	Span Name: Child.ID.2.2 - Timestamp: 1557530485819 - Duration: 328 ms
Span Name: Child.ID.3.1 - Timestamp: 1557530831228 - Duration: 69 ms	(c)
Span Name: Child.ID.3.2 - Timestamp: 1557530831243 - Duration: 84 ms	Span Name: Parent.ID.3 - Timestamp: 1557530831159
Span Name: Child.ID.3.3 - Timestamp: 1557530831204 - Duration: 45 ms	Span Name: Child.ID.3.1 - Timestamp: 1557530831228 - Duration: 69 ms
Span Name: Child.ID.3.2 - Timestamp: 1557530831256	Span Name: Child.ID.3.2 - Timestamp: 1557530831243 - Duration: 84 ms
Span Name: Child.ID.3.2.1 - Timestamp: 1557530831326 - Duration: 70 ms	Span Name: Child.ID.3.3 - Timestamp: 1557530831204 - Duration: 45 ms
Span Name: Child.ID.3.2.2 - Timestamp: 1557530831347 - Duration: 91 ms	Span Name: Child.ID.3.2 - Timestamp: 1557530831256
(a)	Span Name: Child.ID.3.2.1 - Timestamp: 1557530831326 - Duration: 70 ms
	Span Name: Child.ID.3.2.2 - Timestamp: 1557530831347 - Duration: 91 ms
	(d)

Fonte: Produção do próprio autor.

Embora a aplicação tenha cumprido seu propósito de medição, o segundo objetivo deste trabalho, exibir o tempo de resposta completo, não foi alcançado. Infelizmente, não foi possível fazer a extração dos contextos utilizados para realizar o *tracing* na ferramenta *Zipkin*, escolhida para medição do tempo de processamento. Para realizar a extração do contexto seria necessária a criação de uma nova biblioteca extra que implementasse os padrões utilizados pelo *Zipkin*, o que seria um desvio considerável do escopo do presente trabalho. Além disso, como pode-se perceber pela Figura 19, não é possível determinar de forma intuitiva o caminho percorrido durante a comunicação. Por fim, o terceiro fator, além de outros menos relevantes, que determinou a descontinuação da solução foi que, para realizar o rastreamento da comunicação, toda a instrumentação da aplicação tinha que ser feita diretamente no código do serviço, na camada de aplicação. A consequência disso é um *overhead* no processamento, além do fato de que informações sobre tempo de comunicação muitas vezes são desconhecidas e até irrelevantes para desenvolvedores.

### 4.1.4.2 Solução 2

Então, uma segunda solução, que consistia na modificação das bibliotecas responsáveis pela implementação do protocolo AMQP e das ferramentas de *tracing* na camada inferior do *middleware*, foi desenvolvida seguindo a metodologia proposta. Como primeiro passo desta solução e pelo fato do PIS ser um ambiente com suporte a diversas linguagens de programação, foi necessário realizar uma análise qualitativa em todas as aplicações desenvolvidas até o momento. Chegou-se à conclusão de que seria necessária a modificação dessas bibliotecas, nas linguagens C++ e *Python*. Desta forma, realizou-se um profundo estudo sobre o funcionamento das bibliotecas de comunicação com o *broker*, a fim de identificar a parte correta do código onde seria implementada a medição.

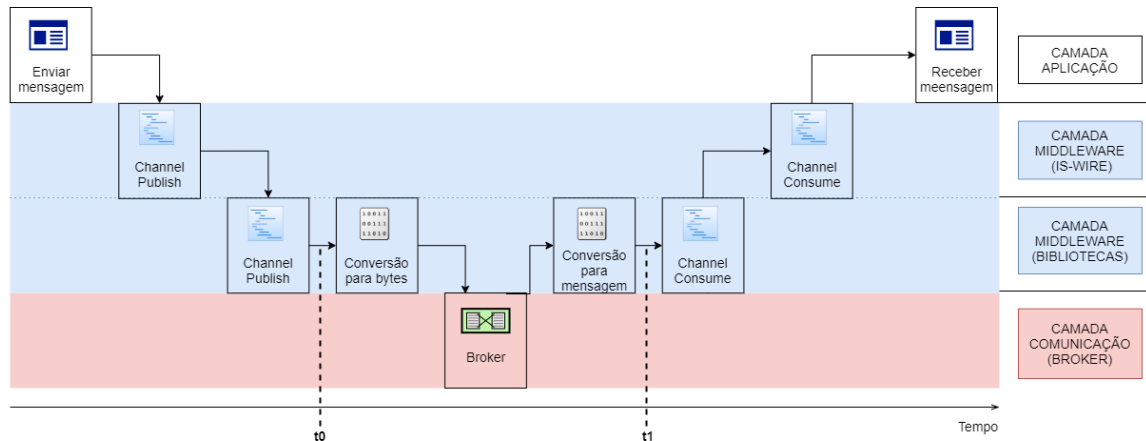
A primeira descoberta com esse estudo foi que ambas bibliotecas possuem estruturas de códigos bem similares, já que seguem um padrão de implementação. Devido a isso, neste



trabalho, ambas foram tratadas como se fossem uma única biblioteca genérica. Além desta, a principal descoberta foi que há duas funções atribuídas à classe "*Channel*" para realizar o envio e o recebimento das mensagens, depois das manipulações necessárias, através do *broker*. Para o envio, a biblioteca captura a informação a ser enviada, a encapsula em uma mensagem junto de dados necessários para o *broker*, realiza a conversão da mensagem em *bytes* e inicia a transmissão dos *bytes* para o *broker* na rede. Para o recebimento, ocorre o caminho inverso, isto é, a biblioteca recebe os *bytes* anteriormente enviados, faz a conversão de *bytes* para a mensagem no padrão AMQP, extrai o conteúdo relevante fornecido pelo *broker* e, por fim, disponibiliza a informação enviada para a aplicação. Com isso, escolheu-se esses como os melhores pontos para se realizar a medição.

Como o método de medição do OWD consiste em registrar um primeiro *timestamp*, no momento do envio do primeiro bit da mensagem, e um segundo, no momento em que toda a mensagem é recebida, ficou convencionado que o primeiro momento de tempo ( $t_0$ ) seria registrado no início da conversão para bytes da mensagem a ser enviada e o segundo momento ( $t_1$ ) seria ao final da conversão de *bytes* para a mensagem. Assim, os tempos de conversão seriam considerados como parcelas do tempo de comunicação. Desta forma, o primeiro objetivo do trabalho foi alcançado. Tais convenções, bem como o detalhamento das camadas estão demonstradas na [Figura 20](#).

Figura 20 – Modificação da camada *middleware* de bibliotecas para medição de tempo de comunicação



Fonte: Produção do próprio autor.

De posse da medição do tempo de comunicação, foram estudadas formas de se agregar tal informação à medição do tempo de processamento na ferramenta de *tracing* *Zipkin*. Essa ferramenta, bem como as extensões necessárias para seu uso são de código aberto, o que ajudou a determinar os melhores pontos para realizar as modificações a fim de habilitar o suporte para exibição do tempo de comunicação junto do tempo de processamento.



Ao avaliar o código-fonte e o funcionamento da ferramenta, percebeu-se que era o próprio *Zipkin* que realizava o cálculo das medições de processamento, baseado nos *timestamps* recebidos, isto é, a instrumentação capturava o *timestamp* inicial e final de um determinado bloco de instruções e enviava para o núcleo da ferramenta, que exibia na através de uma interface *web*.

Sendo assim, conforme consta na metodologia, o código-fonte do *Zipkin* foi alterado para permitir a criação de um novo *span* com as informações do *timestamp* inicial e final de serviços separados, provenientes da biblioteca de comunicação, e exportar para o seu núcleo.

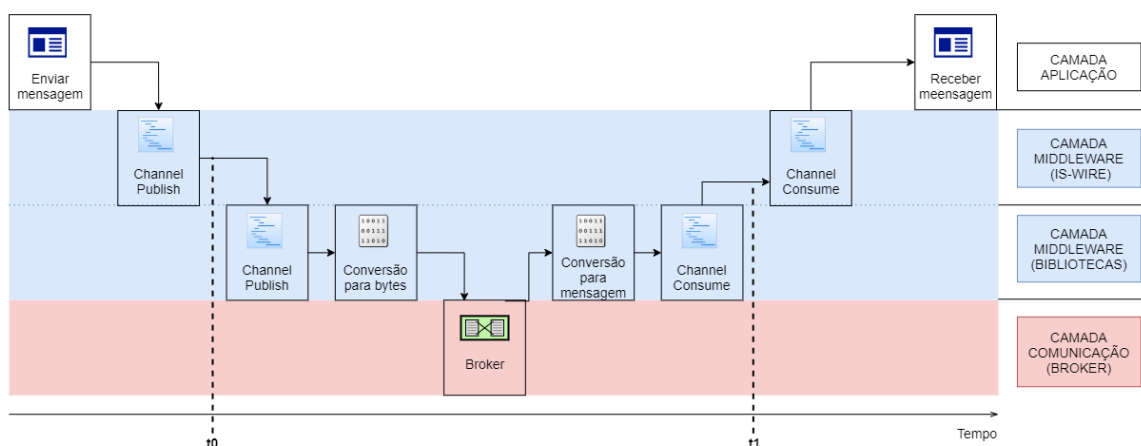
Dessa forma, foi possível visualizar, através da interface web da ferramenta, os tempos de processamento e comunicação. Tendo assim, ambos objetivos deste trabalho alcançados.

#### 4.1.4.3 Solução 3

Após alcançar os dois objetivos deste trabalho, buscou-se uma nova solução embutisse a medição na própria estrutura do PIS para facilitar sua implementação e aumentasse sua precisão, isto é, que adicionasse as parcelas da comunicação que antes não estavam sendo devidamente contabilizadas. Para isso, dessa vez, estudou-se a implementação na subcamada superior da camada *middleware*, uma vez que para a aplicação alcançar a função da subcamada inferior, primeiramente, a chamada da função deveria ser realizada na subcamada superior.

Os pontos escolhidos para inserção da medição foram os mesmos da solução anterior, com a diferença de ter sido adicionada na na camada superior do *middleware*, ou seja, a *is-wire*, em vez da camada inferior do *middleware*, conforme demonstra [Figura 21](#).

Figura 21 – Modificação da camada *middleware is-wire* para medição de tempo de comunicação



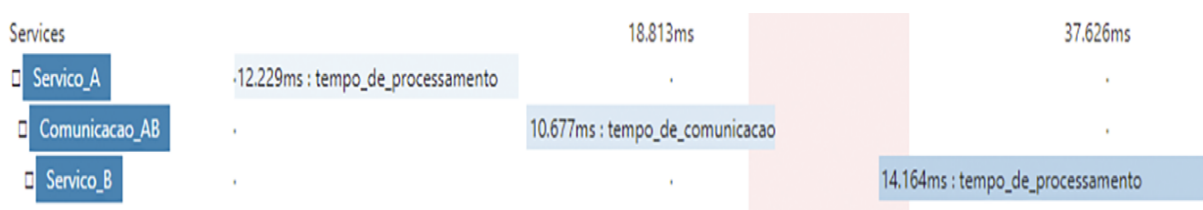
### 4.1.5 Avaliação das medições

Conforme consta na metodologia, após a implementação da solução foi necessário fazer uma avaliação prévia das medições para verificar se esta estava sendo realizada de forma coerente. Para isso, foram criados três serviços, nas duas linguagens abordadas, para serem utilizados em algumas disposições, sendo eles: Serviço A que gera as mensagens de tamanho determinado e as envia pela rede; Serviço B que consome e faz o encaminhamento das mensagens para o próximo serviço; e, por último, o Serviço C que somente consome a mensagem. Em todas as trocas de mensagens o tempo gasto na comunicação foi medido e registrado para posterior comparação. Além disso, foram determinados diferentes tamanhos de mensagens. Assim, pôde-se estimar uma faixa para o valor medido, buscando verificar a coerência das medições em relação ao tamanho das mensagens.

O foco principal dessa etapa foi garantir que a medição estivesse em conformidade com a correta ocorrência de eventos e possível erros na técnica de medição. Dessa forma, vale ressaltar que os resultados obtidos nesta etapa foram referentes à validação das medições e que os resultados dos experimentos realizados com esses serviços serão descritos em seção posterior.

Ao longo da execução desta etapa, pode-se perceber que, embora a medição do tempo de trânsito dos pacotes estivesse sendo feita corretamente e de forma coerente, havia certas lacunas nas medições que correspondem a partes extras do código que estavam sendo contabilizadas nos cálculos, conforme mostra a parte destacada de rosa na [Figura 22](#). Ao analisar o funcionamento do código, identificou-se que esse comportamento acontecia na etapa no momento em que o *span* era exportado, através do protocolo HTTP, para o *Zipkin* e que só ocorria na biblioteca cuja linguagem de programação era C++. Diferentemente da biblioteca em *Python* que realiza o envio do *span* para ferramenta e segue o fluxo do programa, a implementação base do protocolo AMQP da biblioteca em C++ causa esse atraso adicional ao aguardar, do *Zipkin*, a confirmação de recebimento da requisição ou até estourar um tempo limite, com isso, além de calcular o OWD entre dois pontos, também era contabilizado erroneamente o RTT, ou o tempo limite, entre o serviço e a ferramenta de *tracing*.

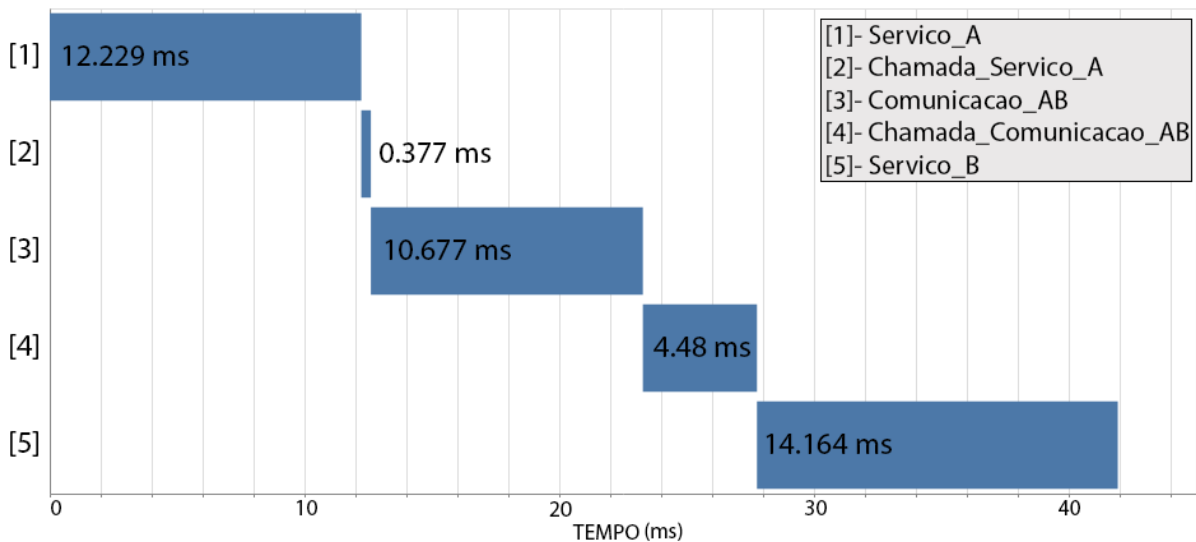
Figura 22 – Lacunas após as medições do tempo de comunicação



Fonte: Produção do próprio autor.

Para uma melhor visualização do problema supracitado, um programa extra também foi desenvolvido com o intuito de gerar uma nova exibição gráfica com detalhamento de cada etapa ocorrida. Dessa maneira, as lacunas nas medições ficaram evidentes e, inclusive, puderam ser medidas. A [Figura 23](#) exibe a aplicação desse programa, obtendo o detalhamento, na situação relatada pela [Figura 22](#).

Figura 23 – Programa criado para detalhamento do tracing



Fonte: Produção do próprio autor.

Como pode ser visto na [Figura 23](#), o *trace* da [Figura 22](#) foi fielmente reproduzido e cada espaço entre os *spans* devidamente mensurado. Para a situação em questão, o tempo de chamada das funções responsáveis pelo início da comunicação no Serviço A foi de 0.377 milissegundos, enquanto a chamada no Serviço B referente ao fim da comunicação teve um tempo de 4.48 milissegundos. Assim, com utilização dessa ferramenta, os problemas desse tipo ficaram mais evidentes.

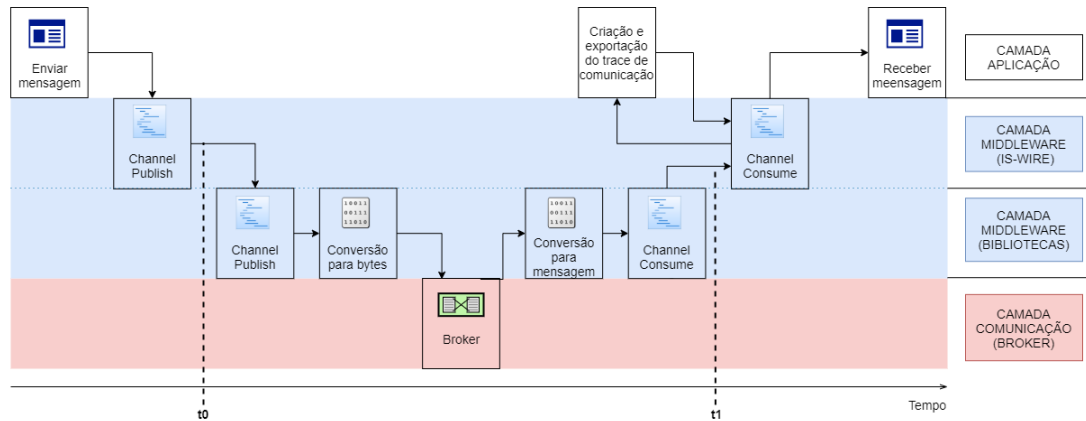
#### 4.1.6 Mitigação de imprevistos na ferramenta de tracing

Após a avaliação da medição, buscou-se resolver a situação imprevista em relação a biblioteca C++. Inicialmente, uma vez que a proposta era não realizar modificações na biblioteca de terceiros, descartou-se a possibilidade de modificação do código fonte para remover a espera da confirmação da requisição e foi idealizado o desenvolvimento de uma solução alternativa utilizando a camada de aplicação.

Foi criado, então, um serviço semelhante ao mencionado na Solução 1 que, neste caso, consistia em um servidor *socket-udp* aguardando por mensagens no formato JSON. Ao receber tais mensagens, o serviço extraía as informações da mensagem, criava e exportava para a ferramenta *Zipkin*, utilizando a biblioteca em *Python*, o *span* correspondente aos

*timestamps* recebidos e, por fim, devolvia o contexto gerado ao cliente *socket-udp*, como pode ser visto na Figura 24. É válido mencionar que a adoção de tal solução teve como foco a redução do tempo gasto no referido passo, pois era sabido que a espera de uma resposta por parte do cliente não poderia ser removida em qualquer alternativa proposta, haja vista que o cliente necessita receber o contexto do tempo de comunicação para entregá-lo à aplicação.

Figura 24 – Adição do serviço extra para publicação dos *traces* de comunicação



Fonte: Produção do próprio autor.

Com a implementação da alternativa proposta, percebeu-se uma redução de até 20 vezes do atraso da biblioteca em C++ identificado através da ferramenta criada. Tal estimativa foi realizada utilizando-se sempre o mesmo tamanho da mensagem em *bytes* e os serviços descritos na Seção 3.1.4, ou seja, foram mantidos todos os elementos presentes na situação problemática e adicionado somente a alternativa proposta. Dessa forma, a Figura 25 e a Figura 26 exibem um *trace* da mesma cadeia de serviços, respectivamente, sem e com a utilização do serviço de publicação com o intuito de realizar um comparativo do rastreamento realizado.

Figura 25 – Exibição do *trace* sem a utilização do serviço de publicação

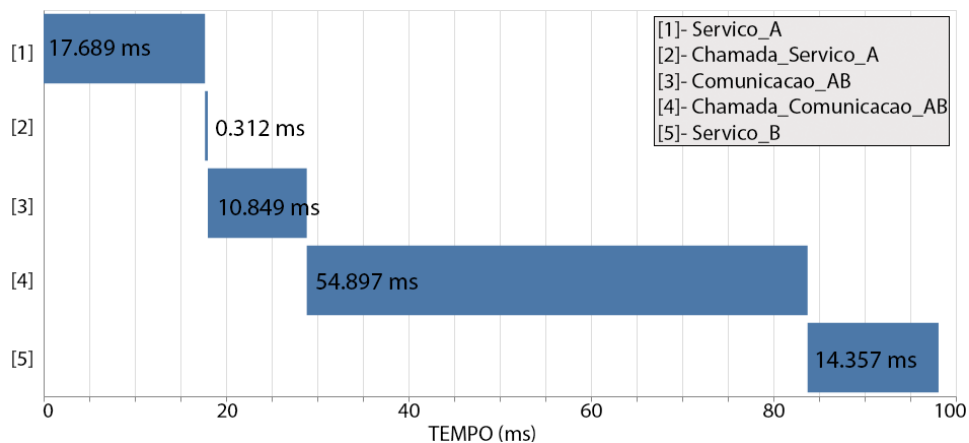
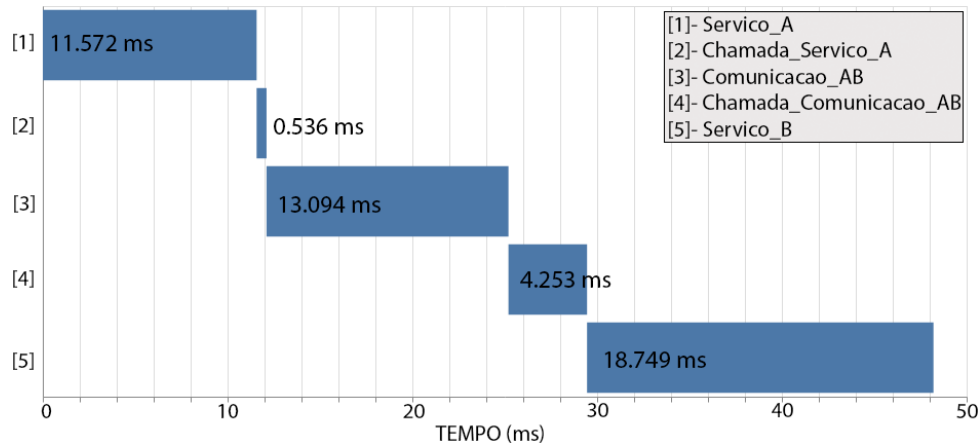


Figura 26 – Exibição do *trace* com a utilização do serviço de publicação

Como pode se concluir do comparativo, neste caso, inicialmente, sem o serviço extra, o tempo gasto para a realização da publicação do *trace* foi de 54.897ms, enquanto a utilização do serviço extra o tempo medido foi de 4.253ms, o que demonstra uma redução de, aproximadamente, 13 vezes e a eficiência da alternativa proposta para mitigar esse imprevisto.

## 4.2 Experimentos

Após a descrição da metodologia utilizada e seus aspectos de implementação, esta seção apresenta, para fins de validação da metodologia, os resultados obtidos com a aplicação da Solução 3, proposta neste trabalho, em duas situações: a primeira, utilizando serviços criados com o intuito de realizar uma depuração e avaliação da solução e, a segunda, utilizando uma aplicação real desenvolvida anteriormente à concepção deste trabalho. A título de detalhamento, as nomenclaturas utilizadas nos experimentos ficaram convencionadas como mostra o [Quadro 4](#) abaixo:

Quadro 4 – Nomenclaturas convencionadas para os experimentos

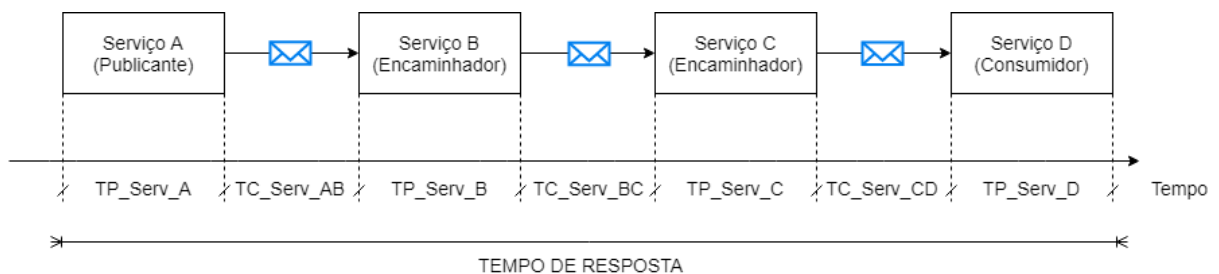
<i>Trace</i>	<i>Span</i>	Descrição
Serv_x	TP_Serv_x	Medição referente ao processamento do serviço x
	TB_Serv_x	Tempo gasto pela biblioteca do serviço x da chamada da função até o início da comunicação
CommTracer	TC_{ID_Consumidor}	Medição de OWD, ou seja, comunicação entre o serviço anterior e o consumidor descrito pela identificação única fornecida pelo <i>broker</i>
	TB_tc_{ID_Consumidor}	Tempo gasto pela biblioteca do consumidor para exportar o TC_{ID_Consumidor} para ferramenta de <i>tracing</i> ;

Fonte: Produção do próprio autor.

## 4.3 Aplicação: Serviços de Depuração

Para a utilização da solução proposta nesta situação, foram utilizados quatro serviços de depuração conectados através do *broker*. Os serviços tiveram versões nas duas linguagens de programação, *Python* e C++, e foram todos implementados no PIS, cuja arquitetura é distribuída em diversos computadores e baseada em microsserviços. Além disso, todos os tempos de processamento foram emulados através de uma função causadora de atraso aleatório no serviço e medidos utilizando a ferramenta *Zipkin*. Nesses moldes, ao todo foram realizados quatro experimentos: todos os serviços em C++, todos os serviços em *Python*, serviço B com linguagem diferente e múltiplos consumidores. Nesses experimentos, o foco foi avaliar os aspectos comportamentais entre as linguagens, bem como o correto funcionamento das bibliotecas. Portanto, todos os serviços foram desenvolvidos identicamente, utilizando inclusive os mesmos parâmetros, tendo como diferença somente a linguagem de programação. Uma vez que cada experimento gerou um grande número de *traces* e seria inviável fazer a avaliação individual de cada amostra, convencionou-se o uso da média e desvio padrão de 60 amostras para fins de demonstração do resultado. Por fim, os resultados obtidos de cada um dos experimentos estão descritos a seguir e o esquemático geral do funcionamento dos três primeiros experimentos está representado na Figura 27.

Figura 27 – Esquemático do funcionamento dos três primeiros experimentos

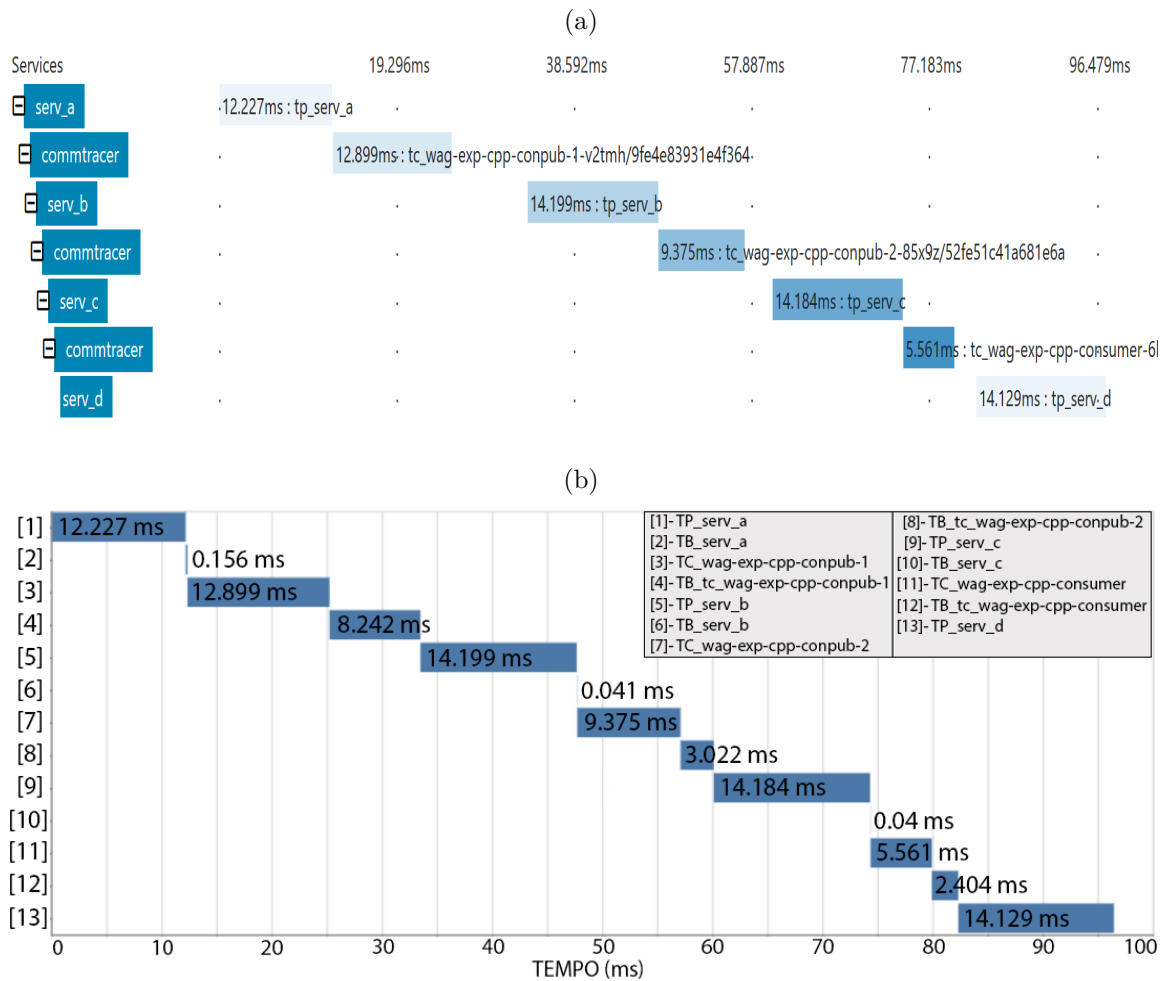


Fonte: Produção do próprio autor.

Detalhando a Figura 27, o serviço A emula um tempo de processamento, gera mensagens com conteúdos aleatórios e tamanho fixo e as envia para o Serviço B, que recebe, emula um novo tempo de processamento e encaminha a mensagem recebida para o Serviço C. O serviço C realiza o mesmo procedimento do serviço B, enviando a mensagem para o serviço D. Por fim, este recebe a mensagem, emula um último tempo de processamento e finaliza a cadeia de execução. Todas as etapas, seja processamento ou comunicação, são devidamente mensuradas e exportadas para a ferramenta *Zipkin*. Com as medições realizadas, é esperado que, no *Zipkin*, seja visualizado tanto o tempo de processamento, quanto o tempo de comunicação com suas devidas lacunas, enquanto na ferramenta de exibição detalhada seja visto o tempo medido em cada uma das etapas do tempo de resposta.

O primeiro experimento consistiu no desenvolvimento de todos os serviços utilizando a linguagem C++. Com o intuito apenas de demonstração do resultado gráfico final, na [Figura 28a](#), tem-se a interface gráfica da ferramenta *Zipkin* para um exemplo de amostra do resultado obtido com o rastreamento. Já na [Figura 28b](#), tem-se o mesmo tempo de resposta sendo detalhado pelo programa criado para avaliação das medições.

Figura 28 – Exemplo de uma amostra do resultado do primeiro experimento



Fonte: Produção do próprio autor.

Pela [Figura 28](#), como pode ser observado, o tempo simulado para o processamento da aplicação no Serviço A, isto é, TP\_Serv\_A medido foi de 12.227 ms e junto a este tem um espaço quase imperceptível de 0.156 ms, referente ao TB\_Serv\_A. O tempo de comunicação, TC\_wag-exp-cpp-conpub-1, é o próximo item da [Figura 28a](#) com 12.899 ms, seguido de uma lacuna de 8.242 ms adicionada pelo TB\_tc\_wag-exp-cpp-conpub-1. Da mesma forma, a simulação de processamento no serviço B, TP\_Serv\_B, e o tempo de chamada função pela biblioteca do serviço B, TB\_Serv\_B, foram, respectivamente, 14.199 ms e 0.041 ms. Por fim, as medições se repetem de forma análoga para os serviços seguintes.

Desta forma, a título de avaliação, o primeiro experimento foi executado repetidamente até se obter 60 amostras para realizar o cálculo da média e do desvio padrão das medições dos tempos de processamento e comunicação dos serviços. O resultado desse cálculo está exibido no [Quadro 5](#).

Quadro 5 – Médias das medições dos serviços no primeiro experimento

Serviço	Média (ms)	Desvio padrão (ms)
TP_Serv_A	12.140	0.253
TB_Serv_A	0.167	0.040
TC_Serv_AB	14.009	18.462
TB_tc_Serv_AB	5.477	8.862
TP_Serv_B	14.135	0.068
TB_Serv_B	0.026	0.008
TC_Serv_BC	4.950	0.922
TB_tc_Serv_BC	4.100	8.934
TP_Serv_C	14.175	0.378
TB_Serv_C	0.025	0.005
TC_Serv_CD	6.071	2.068
TB_tc_Serv_CD	8.186	15.164
TP_Serv_D	14.184	0.395

Fonte: Produção do próprio autor.

Para o segundo experimento, a camada de *middleware* utilizada no desenvolvimento dos serviços foi programada na linguagem *Python*. Aqui, o experimento também foi executado 60 vezes, visando a coleta de um número de amostras suficiente para o cálculo da média e do desvio padrão. O resultado do cálculo encontra-se no [Quadro 6](#).

Quadro 6 – Médias das medições dos serviços no segundo experimento

Serviço	Média (ms)	Desvio padrão (ms)
TP_Serv_A	15.962	2.569
TB_Serv_A	0.134	0.032
TC_Serv_AB	11.789	25.691
TB_tc_Serv_AB	0.597	0.110
TP_Serv_B	17.066	3.963
TB_Serv_B	0.200	0.0462
TC_Serv_BC	12.112	21.542
TB_tc_Serv_BC	0.590	0.112
TP_Serv_C	18.146	2.976
TB_Serv_C	0.209	0.049
TC_Serv_CD	5.856	11.096
TB_tc_Serv_CD	0.582	0.117
TP_Serv_D	17.336	2.738

Fonte: Produção do próprio autor.



No penúltimo experimento, considerou-se a avaliação da interoperabilidade entre as bibliotecas de linguagens diferentes. Para isso, definiu-se que todos os serviços seriam de uma mesma linguagem, enquanto somente o serviço B seria de linguagem diferente. A escolha desse serviço teve como fundamento que o mesmo realiza as duas funções básicas, consumo e publicação, dos outros serviços. Sendo assim, a análise desse serviço em relação aos outros seria mais completa.

Além disso, foi decidido que este experimento seria realizado em duas etapas: a primeira, onde todos os serviços utilizariam a biblioteca em C++ e o serviço B utilizaria a biblioteca em *Python* e a segunda, onde todos os serviços estariam na linguagem *Python* e o serviço B na linguagem C++. Dessa maneira, 60 amostras de cada etapa foram coletadas para o cálculo da média e desvio padrão das medições e estão mostradas no [Quadro 7](#).

Quadro 7 – Médias das medições dos serviços no terceiro experimento

Serviço	Média (ms)	Desvio padrão (ms)	Linguagem
<b>(a) Primeira Etapa</b>			
TP_Serv_A	12.161	0.068	C++
TB_Serv_A	0.18	0.078	C++
TC_Serv_AB	11.366	13.443	C++/Python
TB_tc_Serv_AB	0.662	0.056	Python
TP_Serv_B	18.342	2.887	Python
TB_Serv_B	0.209	0.035	Python
TC_Serv_BC	10.107	12.894	Python/C++
TB_tc_Serv_BC	2.66	1.096	C++
TP_Serv_C	14.153	0.032	C++
TB_Serv_C	0.036	0.013	C++
TC_Serv_CD	8.866	2.12	C++
TB_tc_Serv_CD	2.885	3.291	C++
TP_Serv_D	14.125	0.035	C++
<b>(b) Segunda Etapa</b>			
TP_Serv_A	15.929	3.22	Python
TB_Serv_A	0.13	0.024	Python
TC_Serv_AB	12.346	22.76	Python/C++
TB_tc_Serv_AB	2.977	2.725	C++
TP_Serv_B	14.123	0.024	C++
TB_Serv_B	0.028	0.01	C++
TC_Serv_BC	8.561	14.268	C++/Python
TB_tc_Serv_BC	0.639	0.083	Python
TP_Serv_C	17.439	3.187	Python
TB_Serv_C	0.216	0.032	Python
TC_Serv_CD	4.235	4.684	Python
TB_tc_Serv_CD	0.641	0.083	Python
TP_Serv_D	16.982	2.978	Python

Fonte: Produção do próprio autor.

De posse dos resultados obtidos, algumas conclusões relevantes puderam ser alcançadas. Nota-se, pela observância de todos os *spans* na [Figura 28](#), que as medições ocorreram seguindo uma ordem cronológica correta em cadeia, onde um *span* só teve início após o término do anterior. Esse comportamento é atribuído à correta sincronização dos relógios entre os computadores do PIS. Além disso, embora o tamanho da mensagem fosse fixo entre os serviços, constatou-se, como já era esperado, que os tempos de comunicação tiveram uma variação entre eles, o que é considerado normal nas aplicações. Os tempos de comunicação medidos estão dentro de uma faixa aceitável de diferença, o que comprova o funcionamento correto da ferramenta, enquanto a diferença pode ser devida a fatores da própria rede.

No que tange à interoperabilidade entre as bibliotecas, nenhum problema foi notado. O que poderia ter acontecido era uma incompatibilidade de *timestamps*, devido à representação diferente das estruturas de dados nas duas linguagens, entretanto ao forçar a utilização para um mesmo tipo, visando trabalhar com *timestamps* em microssegundos, esse problema não ocorreu.

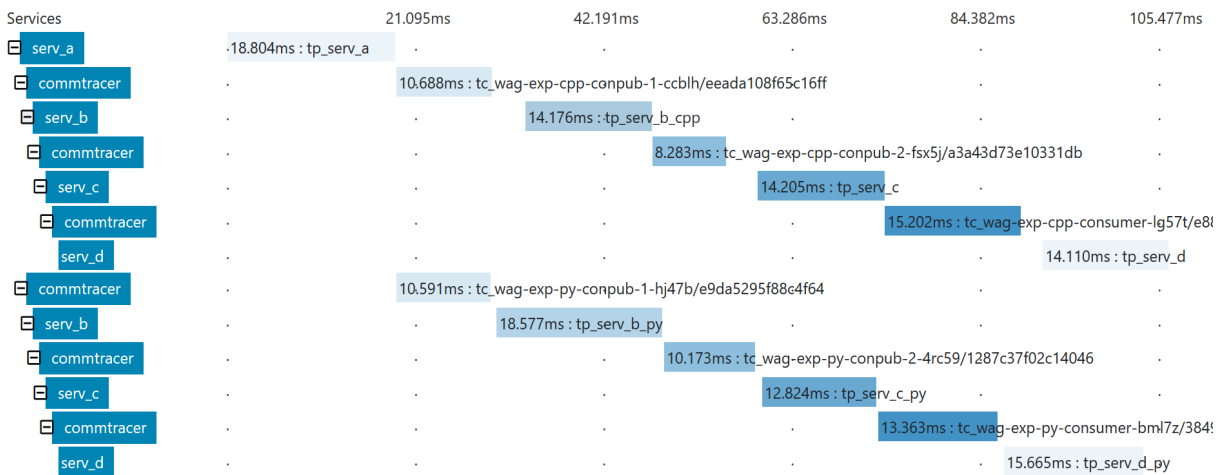
Por fim, comparando-se o [Quadro 5](#) e [Quadro 6](#), pode-se perceber um melhor desempenho da versão *Python* sobre a versão C++ na medição do tempo de comunicação, uma vez que aquela tem um impacto menor na medição. Provavelmente, isso se dá devido à implementação das formas de comunicação utilizada pela própria biblioteca. Isso fica evidente na comparação dos campos `TB_tc_Serv` nos quadros mencionados. No experimento onde todos os serviços estavam em *Python*, foi apresentado um pequeno desvio padrão, ao passo que em C++ o desvio padrão é bem superior, indicando uma alta variação no tempo gasto pela biblioteca para realizar a exportação do *trace* para a ferramenta. Analisando ainda os mesmos campos, pode-se chegar na mesma comprovação observando os tempos gastos na biblioteca, após a medição do tempo de comunicação, onde o *overhead* inserido pela biblioteca em C++ é superior em todos os serviços, ao contrário da biblioteca em *Python*. Uma outra forma de se visualizar isso é analisar as duas etapas do [Quadro 7](#). No [Quadro 7b](#), onde a maioria dos serviços são em C++, o único serviço que apresenta um *overhead* do tempo de comunicação reduzido é a versão em *Python*. Por sua vez, na [Quadro 7a](#), onde grande parte dos serviços são em *Python*, o *overhead* do tempo de comunicação superior é atribuído à biblioteca em C++.

No último experimento, foi avaliada a capacidade da ferramenta em medir múltiplos tempos de comunicação, tendo uma mesma origem. Pela estrutura de comunicação do PIS ser baseada no método *publish-subscribe* através de um *broker*, é possível que haja a situação de uma mensagem ser publicada para um tópico e ser consumida por mais de um serviço, daí torna-se necessário executar mais esse experimento.

Com isso, para essa realização foram utilizados sete serviços: um publicante que, assim como os experimentos passados, emula um tempo de processamento, gera e publica

uma mensagem; e, para cada linguagem, há dois serviços de encaminhamento de mensagem e um consumidor final, que recebe a mensagem e emula o tempo de processamento antes do fim. Optou-se por utilizar a mesma cadeia de serviços dos experimentos passados em cada linguagem para validar mais uma vez, além da medição, a interoperabilidade e comparar os dois resultados. A Figura 29 exibe uma amostra coletada desse experimento.

Figura 29 – Resultado do quarto experimento



Fonte: Produção do próprio autor.

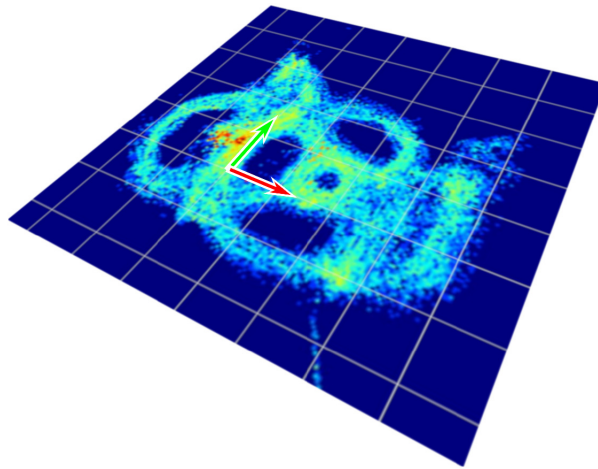
Como se pode observar, além do supracitado sobre a interoperabilidade, nesse caso, também não ocorre inconsistências em relação às linguagens serem diferentes. Por sua vez, em relação às medições, pode-se ver que foi possível realizar a medição para dois consumidores, assim, estima-se que tal comportamento deverá ser o mesmo para N consumidores, uma vez que estejam nas mesmas condições. Isso se dá pela escolha do método de medição. Como foi escolhido a medição de OWD, o *timestamp* é inserido na mensagem e propagado para todos os serviços que consumirem determinado tópico, assim, quando a mensagem chega ao final do caminho é possível realizar a medição em todos.

Além disso, pela Figura 29, pode-se notar que o tempo de processamento do serviço A é o *span*-pai dos *spans* dos consumidores e de toda cadeia seguinte, demonstrando que a metodologia desenvolvida neste trabalho não altera o registro do fluxo de execução capturado pela ferramenta *Zipkin*, mantém a relação de causalidade e mede o tempo de resposta independente da quantidade de serviços para duas aplicações distintas.

## 4.4 Aplicação: Mapa de ocupação

A execução da solução proposta nesta aplicação foi a última etapa considerada para validação deste trabalho. Um mapa de ocupação, Figura 30, é uma representação gráfica colorida de uma matriz que mostra em quais pontos de um ambiente houve maior presença do objeto rastreado.

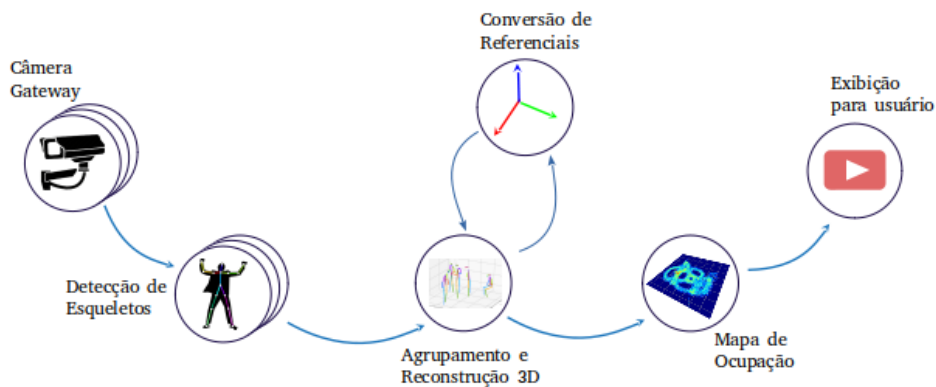
Figura 30 – Exemplo de mapa de ocupação



Fonte: Produção do próprio autor.

No caso do mapa de ocupação do Laboratório de Visão Computacional e Robótica, é feito o rastreamento de pessoas, ou seja, o mapa apresentará cores mais avermelhadas onde as pessoas permanecerem por mais tempo ou passarem por mais vezes e, para alcançar esse objetivo, essa aplicação é formada por diversos serviços em cadeia, conforme mostra [Figura 31](#).

Figura 31 – Representação de todos os serviços necessários para gerar o mapa de ocupação



Fonte: Produção do próprio autor.

Pela [Figura 31](#), pode-se ver que o primeiro serviço é responsável por capturar imagens em uma taxa de quadros definida e enviá-las para um tópico que será consumido pelo próximo serviço, detecção de esqueletos. O serviço de detecção de esqueletos consiste na detecção, extração das juntas e esqueletos de indivíduos em imagens. Em seguida, realiza o envio para um tópico que é consumido pelo serviço de agrupamento das detecções de esqueletos em múltiplas câmeras com sobreposição e reconstrução 3D. Após a realização do agrupamento e da reconstrução 3D dos esqueletos, o serviço responsável pela geração

do mapa de ocupação consome a mensagem contendo os dados necessários para realizar a identificação das posições de cada indivíduo e renderiza, através de uma interface gráfica, as informações no mapa do ambiente.

Essa aplicação, isto é, o conjunto de serviços, foi desenvolvida e finalizada antes da concepção deste trabalho, sendo assim, não houve qualquer modificação em relação aos códigos-fonte. O único ponto modificado em todos os serviços foi a camada *middleware*, a qual foi atualizada para utilizar a versão contendo as medições. Também é válido ressaltar que tal aplicação é composta por serviços que utilizam a linguagem *Python* e outros que utilizam *C++*. Em relação ao colhimento do resultado, foi definido que seria realizada uma média de 1000 amostras de cada serviço para demonstração dos resultados, apresentadas no [Quadro 8](#), uma vez que o PIS é composto por quatro câmeras idênticas e o resultado do rastreamento seria o mesmo para todas, tanto em relação ao fluxo de execução quanto em relação à quantidade de *spans*, variando somente a duração desses. Por fim, o tempo de resposta foi convencionado e medido desde o momento em que uma imagem é capturada até o momento em que a informação é renderizada e exibida ao usuário final. Uma amostra coletada, à título de ilustração, pode ser conferida na [Figura 32](#).

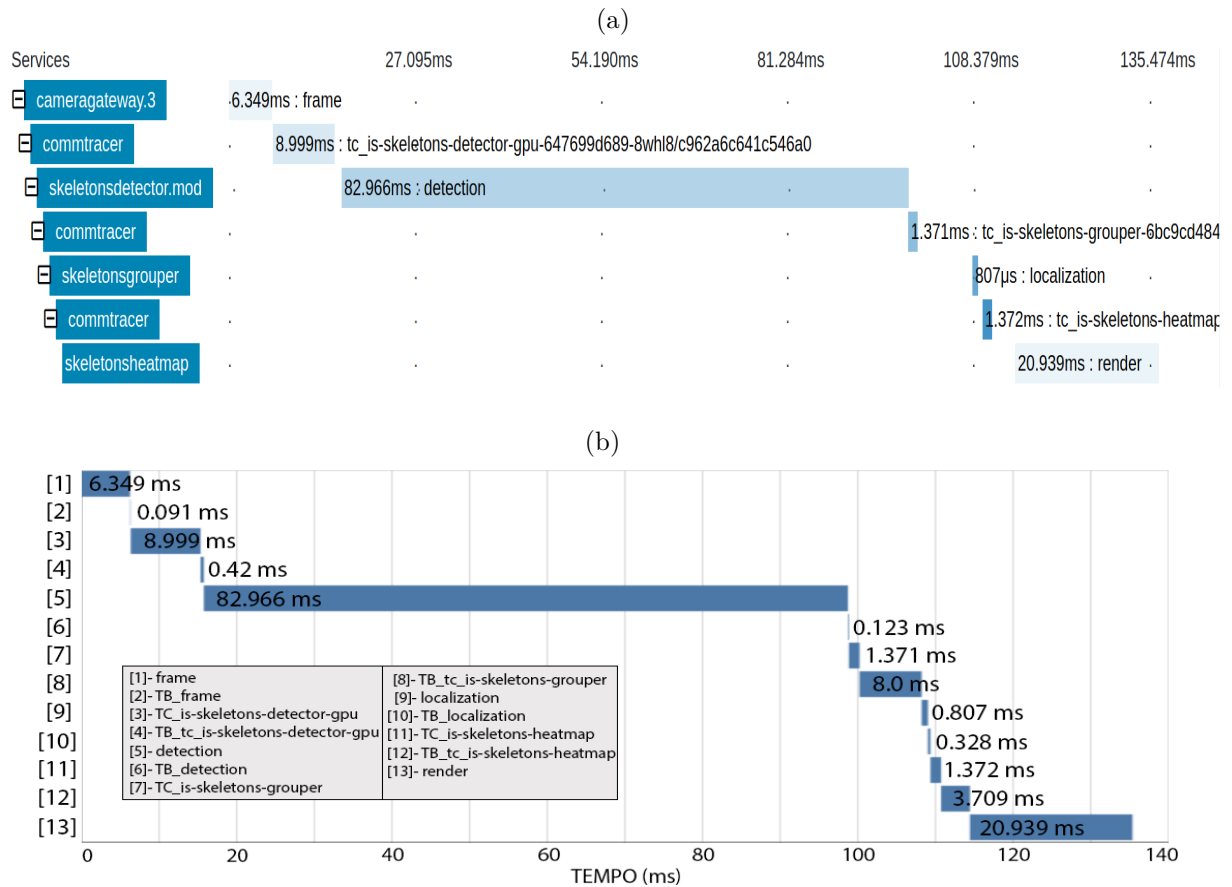
Quadro 8 – Médias das medições dos serviços na aplicação mapa de ocupação

Serviço	Média (ms)	Desvio padrão (ms)	Linguagem
Frame	24.874	14.51	Python
TB_frame	0.393	0.017	Python
TC_is-skeletons-detector-gpu	25.782	14.476	Python
TB_tc_is-skeletons-detector-gpu	0.522	0.185	Python
Detection	59.866	23.195	Python
TB_detection	0.403	0.118	Python
TC_is-skeletons-grouper	16.166	14.335	Python/C++
TB_tc_is-skeletons-grouper	5.527	2.526	C++
Localization	4.448	2.027	C++
TB_localization	0.401	0.049	C++
TC_is-skeletons-heatmap	4.961	2.833	C++/Python
TB_tc_is-skeletons-heatmap	0.516	0.187	Python
Render	43.016	28.851	Python

Fonte: Produção do próprio autor.

Como ilustrado na [Figura 32](#) e comparando-se com o [Quadro 8](#), o serviço de captura de imagem exibe um primeiro *span* com duração média de 24.874 ms. Esse *span* é referente ao tempo que o serviço gasta do momento inicial da captura de um quadro até o seu envio para o serviço seguinte. Em seguida, o tempo de trânsito até o serviço de detecção de esqueletos foi em média 27.782 ms, com um *overhead* médio na comunicação de 0.522 ms. Esse mesmo serviço teve um tempo de médio de processamento de 59.866 ms, o que significa que o serviço recebeu a imagem, realizou as detecções e extraiu as informações nessa janela de tempo. Na sequência, foi realizado o envio para o serviço

Figura 32 – Rastreamento realizado em todos os serviços da aplicação mapa de ocupação



Fonte: Produção do próprio autor.

de agrupamento das detecções de esqueletos que durou em média 16.166 ms. Depois, foi realizado o agrupamento com um tempo médio de 4.448 ms e encaminhado com duração média de 4.961 ms para o último serviço. Nas duas últimas comunicações, as bibliotecas de comunicação apresentaram, respectivamente, 5.527 ms e 0.516 ms de *overhead* médio. Por fim, este último serviço realizou o processamento necessário até a exibição ao usuário com um tempo médio de 43.016 ms.

Com a utilização deste trabalho na aplicação do mapa de ocupação, pode-se notar, primeiramente, que a ordem cronológica dos eventos foi exibida coerentemente, o que significa que a sincronização dos relógios foi realizada de forma correta. Em segundo lugar, diferentemente da aplicação em serviços de depuração, nesse caso, o tamanho da mensagem variou em todas as etapas do fluxo de execução, com isso, não foi possível verificar precisamente se o tempo gasto na comunicação para um trecho estava maior ou menor, contudo, ao analisar outros *traces*, pode-se concluir que a medição estava sendo feita corretamente. O tamanho da mensagem também justifica o pequeno tempo de comunicação entre os serviços. Como já era esperado, não houve problemas no rastreo devido ao serviços serem de linguagens diferentes. Além disso, é perceptível o melhor

desempenho da implementação da biblioteca de comunicação em *Python*, justificado pela comparação dos *overheads* das duas bibliotecas apresentados nas medições.

Também percebeu-se com esta aplicação que, dependendo do tamanho da mensagem, implementação das bibliotecas, das ferramentas utilizadas e da aplicação, a própria medição pode prejudicar o desempenho do sistema, embora, nesse caso específico, o prejuízo não tenha sido relevante. Para outras aplicações, deve-se analisar cada caso separadamente a fim de se chegar a uma conclusão coerente a cerca das medições e os impactos das ferramentas. Em algumas situações, como o *overhead* é medido, pode-se considerar a exclusão deste sem impactar diretamente na medição real do tempo de resposta.

Por último, pôde-se concluir que para situações, onde o tempo de comunicação é muito inferior ao *overhead*, a utilização da biblioteca em C++ não é viável, devido à sua implementação, restando como solução a portabilidade da aplicação para outra linguagem.

Por fim, diante de todos os resultados obtidos e embora em algumas situações o impacto da ferramenta possa prejudicar o desempenho da aplicação, ficou evidente o correto funcionamento da proposta de solução para a determinação do tempo de comunicação, conseqüentemente, do tempo de resposta e para a exibição das medições na mesma ferramenta.

## 5 Considerações Finais

### 5.1 Conclusão

O conceito de cidades inteligentes como área urbanizada onde vários setores cooperam para alcançar resultados sustentáveis traz agregado, direta e indiretamente, uma grande quantidade de tecnologias e conceitos. Uma cidade inteligente pode ser estudada através da correspondência com um espaço inteligente programável, onde o espaço físico é equipado com uma rede de sensores que obtém informações sobre o mundo que observa. Dessa forma, algumas aplicações desenvolvidas para esse espaço inteligente buscam soluções para problemas urbanos, enquanto que outras possuem o foco na coleta de informações. Quando a aplicação precisa interagir com o ambiente ou com usuários, pode haver o requisito de funcionamento em tempo real e, com isso, é levantada a questão da sensibilidade ao tempo de resposta, onde, para um perfeito funcionamento, o sistema tem uma janela de tempo para uma execução de todos os serviços em cadeia. O tempo de resposta pode ser observado utilizando ferramentas de *tracing*, quando se trata de processamento, contudo no que diz respeito à comunicação, há uma ausência de ferramentas específicas. Normalmente, cada grupo interessado desenvolve sua solução de medição conforme sua necessidade, sendo assim, a padronização e a criação de uma ferramenta própria fica comprometida. Com isso, acaba ficando claro a necessidade do desenvolvimento de uma maneira capaz de medir a comunicação, especialmente para microsserviços.

Sendo assim, sabendo da importância das medições na depuração de aplicações e da falta de ferramentas que realizam uma medição completa do tempo de resposta para o PIS, este trabalho propôs uma metodologia que realiza a medição da comunicação em um sistema distribuído baseado em microsserviços e agrega tal informação junto do tempo medido no processamento por outros meios, para que o tempo de resposta seja mais fielmente mensurado e exibido na ferramenta de *tracing*. Foram propostas modificações na estrutura interna dos microsserviços e em bibliotecas *opensource*, específicas de comunicação, para viabilizar a utilização da ferramenta de *tracing* na arquitetura de software do espaço inteligente, com o intuito de realizar a medição do tempo de comunicação. Além disso, foram propostos o desenvolvimento de serviços e ferramentas que permitissem a avaliação após a aplicação da metodologia.

Ao aplicar a metodologia tanto nos serviços de depuração quanto em uma aplicação desenvolvida previamente, obteve-se um resultado positivo, onde a medição em todos os casos estava seguindo uma ordem cronológica coerente em cadeia, onde um *span* só teve início após o término do anterior, devido ao correto sincronismo dos relógios dos servidores distribuídos. Além disso, a correta medição pôde ser comprovada observando



que, para uma mensagem de tamanho fixo, ocorreram variações insignificantes no tempo de comunicação, atribuídas a próprios fatores de rede. Obteve-se também, como resultado, êxito na interoperabilidade das bibliotecas, uma vez que a solução foi desenvolvida em duas linguagens diferentes, tendo como foco buscar alcançar a totalidade dos serviços desenvolvidos para o PIS. Embora uma das linguagens tenha o desempenho ligeiramente inferior em relação à outra, ambas alcançaram satisfatoriamente objetivo deste trabalho.

Tendo como exemplo a aplicação que gera o mapa de ocupação de um ambiente, inicialmente, para se determinar o tempo gasto nas comunicações, eram utilizados arquivos contendo registros de tempos de todas as mensagens enviadas e recebidas, que eram acessados após o encerramento das aplicações. Nesse arquivo, além dos registros, era possível ver o fluxo de execução e, assim, calcular manualmente o tempo gasto nas comunicações entre os serviços. Os dados eram extraídos manualmente e não havia interface gráfica que facilitasse a observação. Com o desenvolvimento deste trabalho, as medições passaram a ser realizadas automaticamente, em tempo real durante o funcionamento dos serviços e de forma abstraída da aplicação, isto é, a aplicação não tinha ciência da ocorrência das medições. Além deste, outro ponto relevante é que todo o tempo de resposta passou a ser visto diretamente na interface gráfica da ferramenta de *tracing*, *Zipkin*, que só tem suporte nativamente para a medição do tempo de processamento.

## 5.2 Trabalhos Futuros

Este trabalho, apesar de contribuir expressivamente para o PIS, deixa algumas lacunas que devem ser preenchidas para que se possa obter um aprimoramento da solução desenvolvida. Dessa forma, a seguir, estão elencados algumas sugestões para trabalhos que podem ser realizados futuramente.

### 5.2.1 Implementação em novas linguagens

Como o PIS é um objeto de constante estudo e aprimoramento, sugere-se que a solução proposta seja desenvolvida em outras linguagens suportadas pelo ambiente.

### 5.2.2 Medição através do elemento coordenador da troca de mensagens

Uma sugestão que envolve alteração em software de terceiros ou a criação de *plugins* é a medição através do elemento coordenador da troca de mensagens do sistema. Como o elemento gerenciador de mensagem é utilizado, caso este realize a medição, poderá haver uma garantia uma maior precisão na medição do OWD. Sugere-se, neste caso, que seja criada uma forma de registrar o *timestamp* assim que o envio de uma mensagem é iniciado, no serviço remetente, e do momento em que o consumidor confirmar o recebimento. Assim,

pode-se calcular as medições e exportar ao usuário de forma totalmente abstraída das aplicações e independente das ferramentas de *tracing*.

### 5.2.3 Continuação da Solução 1

Outra sugestão a ser considerada é a continuação da solução apresentada na [Seção 4.1.4.1](#). A solução criada cumpriu o seu propósito de medição, entretanto não conseguia extrair da ferramenta *Zipkin* as informações de contexto dos tempos de processamento, sem ter que criar uma nova biblioteca para isso. Por isso, a exibição do tempo de resposta completa não foi possível de ser alcançada. Assim, sugere-se a criação de uma forma de propagação do contexto dos tempos de processamento juntamente com os *timestamps*, para que sejam exibidos na interface gráfica. Além disso, algumas modificações podem ser feitas na interface gráfica, de forma a deixá-la mais intuitiva para observância do fluxo de execução do programa.

### 5.2.4 Criação da biblioteca de medição OWD

Muitas aplicações não desejam obter o tempo de resposta completo, tendo interesse apenas no tempo de comunicação. Sugere-se, então, que seja criada uma biblioteca, mais facilmente em *Python*, com as funções de envio e recebimento, que insere *timestamps* nas mensagens e as envia pela rede de alguma forma. A criação dessa biblioteca substitui a camada de *middleware* já que o funcionamento é o mesmo dos serviços do PIS.

# Referências

- ALI, A. et al. Technologies and challenges in developing machine-to-machine applications: A survey. In: *Journal of Network and Computer Applications*. [S.l.: s.n.], 2017. v. 83, p. 124 – 139. Citado na página 23.
- ALLAN, D. W.; ASHBY, N.; HODGE, C. C. *The science of timekeeping*. [S.l.]: Hewlett-Packard, 1997. Citado na página 38.
- ALMES, G.; KALIDINDI, S.; ZEKAUSKAS, M. *RFC2679: A one-way delay metric for IPPM*. [S.l.]: RFC Editor, 1999. Citado na página 44.
- ALMES, G.; KALIDINDI, S.; ZEKAUSKAS, M. *RFC2681: A Round-trip Delay Metric for IPPM*. [S.l.]: RFC Editor, 1999. Citado na página 44.
- ALMONFREY, D. et al. A flexible human detection service suitable for intelligent spaces based on a multi-camera network. *International Journal of Distributed Sensor Networks*, v. 14, n. 3, 2018. ISSN 15501477. Citado na página 24.
- ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. [S.l.: s.n.], 2016. p. 44–51. Citado na página 33.
- ANDREEV, S. et al. Understanding the iot connectivity landscape: A contemporary m2m radio technology roadmap. In: *IEEE Communications Magazine*, vol. 53, no. 9. [S.l.: s.n.], 2015. p. 32—40. Citado na página 23.
- BAGCHI, N. *Management Information Systems*. [S.l.]: Vikas, 2010. 396 p. ISBN 8125938524. Citado na página 29.
- BAKAR, A.; ATAN, R.; YAAKOB, R. Middleware framework for integration of heterogeneous hardware and software. 12 2011. Citado na página 54.
- BEYER, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 149192912X. Citado na página 25.
- BURNICKI, M. *Time reference for your network*. 2020. Disponível em: <<https://www.ntp-zeit.de/index-en.htm>>. Citado na página 40.
- BUYYA, R. *High performance cluster computing*. Upper Saddle River, N.J: Prentice Hall PTR, 1999. ISBN 978-0130137845. Citado na página 30.
- CALYAM, P. et al. Active and passive measurements on campus, regional and national network backbone paths. In: *Proceedings. 14th International Conference on Computer Communications and Networks, 2005. ICCCN 2005*. [S.l.: s.n.], 2005. p. 537–542. Citado 2 vezes nas páginas 43 e 44.
- CARMO, A. P. et al. Programmable intelligent spaces for industry 4.0: Indoor visual localization driving attocell networks. v. 688941, p. 1–19, 2019. Citado 2 vezes nas páginas 24 e 26.

- CINQUE, M. et al. An exploratory study on zeroconf monitoring of microservices systems. In: IEEE. *2018 14th European Dependable Computing Conference (EDCC)*. [S.l.], 2018. p. 112–115. Citado na página 49.
- CINQUE, M.; CORTE, R. D.; PECCHIA, A. Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*, IEEE, 2019. Citado na página 49.
- COLOURIS, G. et al. *Distributed Systems: Concepts and Design*. 5ª ed.. ed. [S.l.]: Pearson, 2011. 1080 p. ISBN 0132143011. Citado 2 vezes nas páginas 29 e 30.
- CORREIA, J. et al. Response time characterization of microservice-based systems. In: IEEE. *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. [S.l.], 2018. p. 1–5. Citado na página 49.
- COTTA, W. A. A. et al. Mobilysa - sistema de localização e controle do cão-guia robô lysa para ambientes internos baseado em visão computacional. In: *Anais Estendidos do XXV Simpósio Brasileiro de Sistemas Multimídia e Web*. Porto Alegre, RS, Brasil: SBC, 2019. p. 159–162. ISSN 2596-1683. Disponível em: <[https://sol.sbc.org.br/index.php/webmedia\\_estendido/article/view/8155](https://sol.sbc.org.br/index.php/webmedia_estendido/article/view/8155)>. Citado na página 26.
- CURNOW, R. *chrony: a versatile implementation of the Network Time Protocol (NTP)*. 2020. Disponível em: <<https://chrony.tuxfamily.org/>>. Citado na página 61.
- DANA, P. H.; PENROD, B. M. The role of gps in precise time and frequency dissemination. *GPS World*, v. 1, n. 4, p. 38–43, 1990. Citado na página 38.
- DANTAS, M. *Computação Distribuída de Alto Desempenho: Redes, Clusters e Grids Computacionais*. [S.l.: s.n.], 2005. ISBN 8573232404. Citado 2 vezes nas páginas 30 e 31.
- DOCKER INC. *Docker: Empowering App Development for Developers*. 2020. Disponível em: <<https://www.docker.com>>. Citado na página 62.
- DOD, U. Global positioning system standard positioning service performance standard. *Assistant secretary of defense for command, control, communications, and intelligence*, 2001. Citado na página 38.
- DOLESCHAL, J. et al. Internal timer synchronization for parallel event tracing. In: LASTOVETSKY, A.; KECHADI, T.; DONGARRA, J. (Ed.). *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 202–209. ISBN 978-3-540-87475-1. Citado na página 37.
- DRAGONI, N. et al. Microservices: Yesterday, today, and tomorrow. In: \_\_\_\_\_. *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 2017. p. 195–216. ISBN 978-3-319-67425-4. Disponível em: <[https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)>. Citado na página 32.
- DUNN, M. J.; DISL, D. *Global Positioning System Directorate Systems Engineering & Integration Interface Specification IS-GPS-200*. [S.l.]: March, 2012. Citado na página 39.
- FERNÁNDEZ-HERNÁNDEZ, I. et al. The galileo commercial service: current status and prospects. In: *Proceedings of the European navigation conference*. [S.l.: s.n.], 2014. Citado na página 39.

- FERRARI, P. et al. Evaluation of communication delay in iot applications based on opcua. In: . [S.l.: s.n.], 2018. p. 224–229. Citado na página 50.
- FERRARI, P. et al. Delay estimation of industrial iot applications based on messaging protocols. *IEEE Transactions on Instrumentation and Measurement*, IEEE, v. 67, n. 9, p. 2188–2199, 2018. Citado na página 50.
- FONSECA, R. et al. X-trace: A pervasive network tracing framework. In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, 2007. Disponível em: <<https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework>>. Citado na página 34.
- FOWLER, M.; LEWIS, J. *Microservices: a definition of this new architectural term*. 2014. Disponível em: <<https://martinfowler.com/articles/microservices.html>>. Citado na página 32.
- GARTNET. *Hype Cycle For Smart City Technologies and Solutions*. [S.l.], 2012. Disponível em: <<http://www.gartner.com/resId=2098315>>. Citado na página 21.
- GEIMER, M. et al. A generic and configurable source-code instrumentation component. In: ALLEN, G. et al. (Ed.). *Computational Science – ICCS 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 696–705. ISBN 978-3-642-01973-9. Citado na página 33.
- GOOGLE. *Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications*. 2020. Disponível em: <<https://kubernetes.io>>. Citado na página 62.
- GOVINDAN, K.; AZAD, A. P. End-to-end service assurance in iot mqtt-sn. In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. [S.l.: s.n.], 2015. p. 290–296. Citado 2 vezes nas páginas 50 e 51.
- HOWARTH, D. The physics of the solid-state maser. *IRE Transactions on Component Parts*, v. 6, n. 2, p. 81–93, 1959. Citado na página 39.
- JAEGER. *Jaeger: open source, end-to-end distributed tracing*. 2020. Disponível em: <<https://www.jaegertracing.io/>>. Citado na página 36.
- JAIN, M.; DOVROLIS, C. End-to-end available bandwidth: measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Transactions on Networking*, v. 11, n. 4, p. 537–549, 2003. Citado na página 44.
- JANAPATI, S. *Distributed Logging Architecture for Microservices*. 2017. Disponível em: <<https://dzone.com/articles/distributed-logging-architecturefor-microservices>>. Citado na página 25.
- KANNISTO, J. et al. Software and hardware prototypes of the ieee 1588 precision time protocol on wireless lan. In: *2005 14th IEEE Workshop on Local Metropolitan Area Networks*. [S.l.: s.n.], 2005. p. 6 pp.–6. Citado na página 43.
- KANUPARTHY, P. et al. Ytrace: End-to-end performance diagnosis in large cloud and content providers. *arXiv preprint arXiv:1602.03273*, 2016. Citado na página 49.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <<https://doi.org/10.1145/359545.359563>>. Citado na página 38.

LAW, K. H.; LYNCH, J. P. Smart city: Technologies and challenges. *IT Professional*, IEEE Computer Society, v. 21, n. 6, p. 46–51, nov 2019. ISSN 1941045X. Citado na página 21.

LEE, C. *Latency-based Congestion Detection and Control for Datacenters*. Tese (Doutorado) — KAIST - Korea Advanced Institute of Science and Technology, Daejeon, South Korea, 5 2015. Citado na página 47.

LEE, J.-H.; ANDO, N.; HASHIMOTO, H. Intelligent space for human and mobile robot. In: IEEE. *Advanced Intelligent Mechatronics, 1999. Proceedings. 1999 IEEE/ASME International Conference on*. [S.l.], 1999. p. 784–784. Citado na página 22.

LEE, K. B.; ELDSON, J. Standard for a precision clock synchronization protocol for networked measurement and control systems. In: . [S.l.: s.n.], 2004. Citado na página 41.

LIBERG, O. et al. Cellular internet of things. In: ACADEMIC PRESS. [S.l.], 2018. Citado na página 23.

MICROSOFT. *Visual Studio Code - Code Editing. Redefined*. 2020. Disponível em: <<https://code.visualstudio.com>>. Citado na página 62.

MILLS, D. *Network Time Protocol (NTP)*. *DARPA Network Working Group Report RFC-958*. [S.l.], 1985. Citado na página 39.

MILLS, D. Network time protocol (version 2) specification and implementation; rfc-1119. *Internet Requests for Comments*, n. 1119, 1989. Citado na página 39.

MILLS, D. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, v. 39, n. 10, p. 1482–1493, 1991. Citado na página 56.

MILLS, D. *IEEE 1588 Precision Time Protocol (PTP)*. 2012. Citado na página 41.

MROUE, H. et al. Mac layer-based evaluation of iot technologies: Lora, sigfox and nb-iot. In: *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*. [S.l.: s.n.], 2018. Citado na página 23.

MURAKAMI, T.; HORIUCHI, Y. A master redundancy technique in ieee 1588 synchronization with a link congestion estimation. In: *2010 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. [S.l.: s.n.], 2010. p. 30–35. Citado na página 42.

NAYAK, A. R.; JONE, W.-B.; DAS, S. R. Designing general-purpose fault-tolerant distributed systems-a layered approach. In: IEEE. *Proceedings of 1994 International Conference on Parallel and Distributed Systems*. [S.l.], 1994. p. 360–364. Citado na página 54.

NEWMAN, S. *Building microservices : designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN 9781491950357. Citado na página 32.



- OAKS, O. et al. Performance of gps on-orbit navstar frequency standards and monitor station time references. p. 11, 12 1998. Citado na página 38.
- O'HARA, J. *Advanced Message Queuing Protocol*. 2020. Disponível em: <<https://www.amqp.com>>. Citado na página 63.
- OPENTRACING. *The OpenTracing data model specification*. 2020. Disponível em: <<https://github.com/opentracing/specification/blob/master/specification.md>>. Citado 2 vezes nas páginas 34 e 35.
- OPENTRACING. *OpenTracing: Semantic Conventions*. 2020. Disponível em: <[https://github.com/opentracing/specification/blob/master/semantic\\_conventions.md](https://github.com/opentracing/specification/blob/master/semantic_conventions.md)>. Citado na página 34.
- OPENTRACING.IO. *What is Distributed Tracing?* 2020. Disponível em: <<https://opentracing.io/docs/overview/what-is-tracing/>>. Citado na página 25.
- PAHL, C. et al. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, v. 7, n. 3, p. 677–692, 2019. Citado na página 32.
- PICORETI, R. et al. Multilevel observability in cloud orchestration. 2018. Citado 3 vezes nas páginas 24, 25 e 64.
- POSTEL, J. *User datagram protocol (No. RFC 768)*. [S.l.], 1980. Citado na página 39.
- POSTEL, J. Internet control message protocol darpa internet program protocol specification. *RFC 792*, 1981. Citado na página 44.
- POSTEL, J. et al. Internet protocol. STD 5, RFC 791, September, 1981. Citado na página 39.
- PRIETO, J. et al. Iot approaches for distributed computing. *Wireless Communications and Mobile Computing*, v. 2018, p. 1–2, 03 2018. Citado na página 23.
- QUEIROZ, F. M. de. *Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visao Computacional e IoT*. Vitoria: [s.n.], 2016. 80 p. Citado 4 vezes nas páginas 24, 54, 61 e 63.
- RABBIT TECHNOLOGIES. *RabbitMQ: Messaging that just works*. 2020. Disponível em: <<https://www.rabbitmq.com>>. Citado na página 63.
- RAMPINELLI, M. et al. An intelligent space for mobile robot localization using a multi-camera system. *Sensors (Basel, Switzerland)*, v. 14, n. 8, p. 15039–15064, 2014. ISSN 14248220. Citado na página 24.
- SAMBASIVAN, R. R. et al. Principled workflow-centric tracing of distributed systems. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2016. (SoCC '16), p. 401–414. ISBN 9781450345255. Disponível em: <<https://doi.org/10.1145/2987550.2987568>>. Citado na página 33.
- SCHWARZ, R.; MATTERN, F. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, v. 7, 02 1993. Citado na página 38.
- SHARMA, P. Ieee 1588^\* in network processors for next-generation industrial automation solutions. *Technology@ Intel magazine*, Intel Corp., 2005. Citado na página 43.

- SIGELMAN, B. H. et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. [S.l.], 2010. Disponível em: <<https://research.google.com/archive/papers/dapper-2010-1.pdf>>. Citado na página 33.
- SOURCEGRAPH. *AppDash: application tracing system*. 2020. Disponível em: <<https://github.com/sourcegraph/appdash>>. Citado na página 36.
- TANENBAUM, A. *Sistemas distribuídos: Princípios e Paradigmas*. São Paulo: Pearson Educacion, 2008. ISBN 978-8576051428. Citado 2 vezes nas páginas 22 e 29.
- TECHNOLOGY, T. C. on S. Ieee standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, p. 1–300, 2008. Citado na página 41.
- WANG, J.; ZHOU, M.; LI, Y. Survey on the end-to-end internet delay measurements. In: SPRINGER. *IEEE International Conference on High Speed Networks and Multimedia Communications*. [S.l.], 2004. p. 155–166. Citado 2 vezes nas páginas 44 e 47.
- WEIBEL, H. High precision clock synchronization according to ieee 1588 implementation and performance issues. 01 2005. Citado na página 43.
- WEISER, M. The computer for the 21st century. *Scientific American*, p. 94–104, 1991. Disponível em: <<https://www.lri.fr/mb/Stanford/CS477/papers/Weiser-SciAm.pdf>>. Citado na página 22.
- YANG, Z.; MARSLAND, T. A. Annotated bibliography on global states and times in distributed systems. *SIGOPS Oper. Syst. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 27, n. 3, p. 55–74, jul. 1993. ISSN 0163-5980. Disponível em: <<https://doi.org/10.1145/155870.155878>>. Citado na página 37.
- YU, Z.; LI, Z. Best master clock algorithm of precision clock synchronization protocol. *Dianli Zidonghua Shebei/Electric Power Automation Equipment*, v. 29, p. 74–77, 11 2009. Citado na página 41.
- ZIPKIN. *Zipkin: a distributed tracing system*. 2020. Disponível em: <<https://zipkin.io/>>. Citado 3 vezes nas páginas 36, 62 e 64.