UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPE FERNANDES DOS SANTOS BRASIL DE MATOS

UMA ARQUITETURA BASEADA EM DEW COMPUTING E EM PROCESSAMENTO MULTI-LINGUAGEM PARA MELHORIA DO DESEMPENHO COMPUTACIONAL DE APLICAÇÕES MÓVEIS

FORTALEZA

2024

FILIPE FERNANDES DOS SANTOS BRASIL DE MATOS

UMA ARQUITETURA BASEADA EM DEW COMPUTING E EM PROCESSAMENTO MULTI-LINGUAGEM PARA MELHORIA DO DESEMPENHO COMPUTACIONAL DE APLICAÇÕES MÓVEIS

Tese de doutorado apresentada ao Curso de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará (UFC), como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Fernando A. M. Trinta

Coorientador: Prof. Dr. Paulo A. Leal Rego

FORTALEZA

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M381a Matos, Filipe Fernandes dos Santos Brasil de.
Uma arquitetura baseada em dew computing e em processamento multi-linguagem para melhoria do desempenho computacional de aplicações móveis / Filipe Fernandes dos Santos Brasil de Matos. – 2024.
154 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2024.
Orientação: Prof. Dr. Fernando A. M. Trinta.
Coorientação: Prof. Dr. Paulo A. Leal Rego.

1. Computação em orvalho. 2. Multi-linguagem. 3. Offloading. 4. Computação móvel. 5. MEC. I. Título.
CDD 005

FILIPE FERNANDES DOS SANTOS BRASIL DE MATOS

UMA ARQUITETURA BASEADA EM DEW COMPUTING E EM PROCESSAMENTO MULTI-LINGUAGEM PARA MELHORIA DO DESEMPENHO COMPUTACIONAL DE APLICAÇÕES MÓVEIS

Tese de doutorado apresentada ao Curso de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará (UFC), como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software.

Aprovada em: 19/07/2024

BANCA EXAMINADORA

Prof. Dr. Fernando A. M. Trinta (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo A. Leal Rego (Coorientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. José Neuman de Souza
Universidade Federal do Ceará (UFC)

Prof. Dr. Fernando José Castor de Lima Filho
University of Twente (UT)

Prof. Dr. Carlos André Guimarães Ferraz
Universidade Federal de Pernambuco (UFPE)

Aos meus pais, que sempre acreditaram em mim e me ensinaram o valor do trabalho árduo e da honestidade. Dedico este trabalho a vocês.

AGRADECIMENTOS

A Deus, pelo dom da vida, pela inspiração constante e pela sabedoria concedida em todos os momentos da minha jornada escolar e acadêmica.

Aos meus pais, José Gastão Brasil de Matos Mourão e Francisca Sylvia Fernandes dos Santos Brasil de Matos, por seu amor incondicional, apoio constante e pelos valores que me transmitiram, fundamentais para a realização deste trabalho.

A minha esposa, Allana Ramos Araújo Brasil de Matos, por seu amor, compreensão e incentivo inabalável durante todo este percurso acadêmico. Sua presença foi essencial para que eu pudesse alcançar este objetivo.

Aos meus familiares, por todo o apoio, carinho e palavras de encorajamento que me motivaram a seguir em frente.

Ao meu orientador, Prof. Fernando Antônio Mota Trinta, e ao meu coorientador, Prof. Paulo Antônio Leal Rego, por toda a orientação, paciência e valiosas contribuições que enriqueceram significativamente este trabalho.

Aos membros da banca examinadora, pelo tempo dedicado à avaliação do meu trabalho e pela contribuição significativa para o desenvolvimento e aprimoramento desta pesquisa.

Aos meus colegas e amigos, técnicos e professores do campus de Crateús da Universidade Federal do Ceará, por sua colaboração, suporte e troca de conhecimentos ao longo desta caminhada.

A todos os meus professores, cujo ensinamento e dedicação ao longo dos anos foram fundamentais para minha formação e para a concretização deste projeto.

A todos, minha eterna gratidão.

"Everything's got to end sometime. Otherwise, nothing would ever get started."

(Doctor Who)

RESUMO

Baixa capacidade de processamento e limitações de energia são restrições comuns enfrentadas pela maioria dos dispositivos móveis ao processar tarefas computacionais. Diversas pesquisas indicam o *offloading* computacional como uma técnica para enfrentar esse desafio. Nela, dispositivos computacionalmente e/ou energeticamente limitados transferem tarefas para serem executadas em máquinas com maior capacidade, poupando tempo e recursos. Contudo, estudos anteriores mostraram que a adoção de linguagens de programação ineficientes ao executar tais tarefas e a latência da rede impactam negativamente no desempenho do *offloading* computacional. Esta tese propõe a Arquitetura DADOS (*Dew Architecture for Distribution of Offloading Servers*), que ataca esses dois problemas incorporando a abordagem multi-linguagem e o paradigma *Dew Computing* ao *offloading* computacional. A DADOS ao utilizar a abordagem multi-linguagem, permite a interação entre processos desenvolvidos em diferentes linguagens, o que possibilita a adoção de linguagens mais eficientes nos processos servidores de *offloading*, acelerando a execução de tarefas e economizando recursos do dispositivo. Além disso, ao usar o paradigma *Dew Computing*, que reduz a dependência da rede ao permitir que parte dos serviços e dados remotos sejam processados no dispositivo móvel, a DADOS permite trazer os processos de *offloading* para dentro do dispositivo, mitigando os efeitos negativos da rede. Foram conduzidos experimentos, utilizando dispositivos reais, com o objetivo de validar a versão inicial da arquitetura e avaliar o impacto da abordagem oferecida por ela na execução de tarefas móveis. Os experimentos avaliaram o desempenho de três abordagens (*Local*, *Dew* e *Cloudlet*) em relação a três métricas principais (tempo de resposta, consumo de energia e eficiência) em dois cenários com diferentes níveis de sobrecarga na rede. Os resultados foram promissores para a abordagem proporcionada pela DADOS, a *Dew*. Comparando apenas as abordagens *Dew* e *Local*, observou-se que a primeira foi até 7.2x mais rápida e consumiu até 4.6x menos energia que a última em ambos os cenários. Já entre as abordagens *Dew* e *Cloudlet*, a *Dew* superou a *Cloudlet* em condições específicas: ao lidar com grandes volumes de dados, a *Dew* transmitiu até 2.6x menos dados. Em ambientes com alta sobrecarga na rede, a *Dew* processou tarefas até 4.5x mais rapidamente que a *Cloudlet*.

Palavras-chave: computação em orvalho. multi-linguagem. offloading. computação móvel. MEC.

ABSTRACT

Low processing power and power limitations are typical constraints faced by most mobile devices when processing computational tasks. Several researches indicate computational *offloading* as a technique to face this challenge. In it, computationally and/or energetically limited devices transfer tasks to be executed on machines with greater capacity, saving time and resources. However, previous studies have demonstrated that adopting programming languages is inefficient when performing such tasks, and network latency impacts computational *offloading* performance levels. This thesis proposes the DADOS Architecture (*Dew Architecture for Distribution of Offloading Servers*), which attacks these two problems by incorporating the multi-language approach and the *Dew Computing* paradigm into computational *offloading*. When using the multi-language approach, DADOS allows interaction between processes developed in different languages, which enables the adoption of more efficient languages in *offloading* server processes, speeding up the execution of tasks and saving device resources. Furthermore, by using the *Dew Computing* paradigm, which reduces dependence on the network by allowing part of the remote services and data to be processed on the mobile device, DADOS allows bringing *offloading* processes into the device, mitigating adverse network effects. Experiments were conducted using real devices to validate the initial version of the architecture and evaluate the impact of the approach it offers on the execution of mobile tasks. The experiments evaluated the performance of three approaches (*Local*, *Dew* and *Cloudlet*) against three main metrics (response time, energy consumption, and efficiency) in two scenarios with different levels of overload on the network. The results were promising for the approach provided by DADOS, *Dew*. Comparing only the *Dew* and *Local* approaches, it was observed that the former was up to 7.2x faster and consumed up to 4.6x less energy than the latter in both scenarios. Between the *Dew* and *Cloudlet* approaches, *Dew* outperformed *Cloudlet* in specific conditions: when dealing with large volumes of data, *Dew* transmitted up to 2.6x less data. In environments with high network overhead, *Dew* processed tasks up to 4.5x faster than *Cloudlet*.

Keywords: dew computing. multi-language. offloading. mobile computing. MEC.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo tem como objetivo principal apresentar ao leitor conceitos e informações importantes para familiarizá-lo com a problemática enfrentada e a metodologia usada na elaboração deste trabalho. De início, a Seção 1.1 apresenta definições e informações relevantes para a compreensão deste documento, com ênfase nos desafios do processamento de tarefas em dispositivos móveis e nas limitações do *offloading* computacional, técnica frequentemente recomendada pela literatura para atacar esses desafios. A Seção 1.2 destaca algumas informações que enfatizam a relevância do problema tanto para a indústria quanto para a academia. Em seguida, as Seções 1.4 e 1.5 expõem os objetivos deste trabalho, bem como a metodologia de pesquisa adotada para conduzi-lo. Por fim, a Seção 1.6 descreve a estrutura da monografia.

1.1 Contextualização

É notório o aumento no número de dispositivos móveis, como *smartphones* e *tablets*, no mercado mundial nas últimas décadas (LARICCHIA, 2022). O crescimento nas vendas motivou o surgimento de novos aplicativos e o aprimoramento constante no *hardware* desses aparelhos. Contudo, apesar de todo esse progresso tecnológico, boa parte deles ainda possui relevantes restrições computacionais e, principalmente, energéticas (NGUYEN; DRESSLER, 2020). Esses problemas são de difícil resolução ao nível de *hardware*, por serem consequências naturais de pré-requisitos importantes que garantem a mobilidade e a portabilidade desses dispositivos. Por exemplo, todos os *smartphones* modernos são alimentados energeticamente por baterias, facilitando a locomoção e o manuseio dos dispositivos, contudo, torna a energia um recurso escasso, já que as baterias são fontes finitas de eletricidade.

O *offloading* computacional tem se mostrado uma técnica promissora para mitigar as limitações computacionais e energéticas dos dispositivos móveis. Hoang *et al.* (2013) o definem como um mecanismo de transferência de dados de um dispositivo digital para outro. No contexto da computação móvel, o *offloading* permite que dispositivos mais restritivos possam transferir tarefas para processamento (e/ou dados para armazenamento) a outros equipamentos menos restritivos por meio da rede, sem inviabilizar a computação e a persistência no próprio dispositivo de origem. Conforme Kumar *et al.* (2013), a escolha entre processar tarefas e armazenar dados localmente ou em servidores remotos depende de fatores como a latência da rede, largura de banda e o volume de dados transmitidos. Essa decisão visa, primordialmente,

otimizar o desempenho ou reduzir o consumo de energia do dispositivo. Importante destacar que essa escolha ainda representa um desafio significativo na pesquisa atual, sendo tema central em diversos estudos recentes (ZHU *et al.*, 2023; CHUANG; HUNG, 2023; ALVES *et al.*, 2023).

Nos últimos anos, tem-se observado o surgimento de diversos *frameworks* que visam auxiliar os desenvolvedores na criação de aplicações que aproveitem os benefícios proporcionados pelo *offloading*. A maioria desses *frameworks* envolve apenas processos desenvolvidos com a mesma linguagem de programação, simplificando consideravelmente o funcionamento e o processo de desenvolvimento e manutenção do aplicativo. Contudo, alguns estudos na literatura ressaltam que diferentes linguagens de programação apresentam padrões distintos de consumo de recursos computacionais (NANZ; FURIA, 2015) e, inclusive, de energia (GEORGIOU *et al.*, 2018; PEREIRA *et al.*, 2021) ao processar a mesma tarefa ou tarefas semelhantes. Dessa maneira, o uso exclusivo de uma linguagem durante o *offloading* pode resultar em um processamento mais lento das tarefas, ao mesmo tempo que consome mais recursos computacionais e energéticos da máquina servidora (GEORGIOU; SPINELLIS, 2019).

Diversos estudos já investigaram o impacto do *offloading* computacional em contextos que envolvem o uso de distintas linguagens de programação (GEORGIOU; SPINELLIS, 2019; ARAÚJO *et al.*, 2020). O autor deste trabalho também contribuiu para essa área de pesquisa, tendo publicado artigos relacionados (MATOS *et al.*, 2021a; MATOS *et al.*, 2021b). Tais estudos consistentemente demonstram perspectivas positivas para a prática de *offloading*, sugerindo que a integração de múltiplas linguagens de programação pode resultar em melhorias substanciais tanto em eficiência quanto em economia de energia. Por exemplo, os resultados dos testes realizados em (MATOS *et al.*, 2021a) apontaram que ao adotar uma abordagem multilinguagem durante o *offloading* computacional, é possível reduzir o tempo de processamento da tarefa em até 39 vezes, enquanto reduz o consumo de energia do dispositivo móvel cliente em até 96%.

Apesar dos benefícios proporcionados à computação móvel, o *offloading* computacional ainda apresenta uma série de desafios, destacando-se o consumo consciente de recursos computacionais e energéticos (BILAL *et al.*, 2018; SHAKARAMI *et al.*, 2020), bem como a latência de rede (GU *et al.*, 2018). Os dois primeiros já são problemas críticos considerando apenas o escopo dos dispositivos móveis. Entretanto, alguns estudos têm demonstrado preocupação com o agravamento desse problema também no lado do servidor, especialmente quando envolve máquinas de borda da rede. Quanto ao último, é um desafio que impacta diretamente o

desempenho do *offloading* sendo agravado pela própria mobilidade dos dispositivos (WANG *et al.*, 2019). Com o deslocamento do aparelho móvel para longe do receptor, seu sinal naturalmente enfraquece, resultando em maiores atrasos no envio e na recepção de dados. Essa comunicação mais lenta afeta negativamente o *offloading*, uma vez que demanda um tempo maior para sua conclusão.

A notável evolução do *hardware* em dispositivos de usuários nos últimos anos impulsionou o desenvolvimento de um paradigma emergente na computação, conhecido como *Dew Computing* (WANG, 2015b; SKALA *et al.*, 2015; FISHER; YANG, 2016). Este paradigma visa reduzir a dependência dos dispositivos em relação à conectividade com a Internet e aos serviços oferecidos por servidores remotos (RAY, 2019). Em essência, a *Dew* propõe a duplicação local dos serviços e dados que normalmente são disponibilizados por outros paradigmas computacionais, como a *Cloud* e a *Edge Computing*, garantindo a disponibilidade desses recursos mesmo na ausência de conexão com a Internet. Segundo (WANG, 2015a), duplicar é a ação de gerar uma versão local de um recurso hospedado remotamente, permitindo simplificações ou personalizações e a adoção de tecnologias distintas, ajustando-se às necessidades e capacidades locais. Além disso, quando conectados, os dispositivos habilitados para *Dew Computing* podem ampliar a eficácia dos servidores remotos, por exemplo, ao processar e responder a solicitações de dispositivos próximos, otimizando assim a prestação de serviços (RAY, 2018).

Com base no exposto, observa-se que o problema de como processar tarefas de maneira eficaz poderia ser atacado de duas formas distintas: 1) Utilizar linguagens de programação mais eficientes para processar a tarefa de forma mais rápida e consumindo menos recursos do dispositivo; 2) Manter a mesma linguagem de programação, mas transferir o processamento da tarefa para outro dispositivo por meio do *offloading*. Ambas as opções apresentam questões importantes a serem avaliadas. Por exemplo, enquanto a primeira é limitada pela escassa oferta de linguagens disponíveis para uso no contexto de dispositivos móveis, a segunda só funciona efetivamente quando o dispositivo está conectado a outros dispositivos por meio de enlaces de baixa latência, o que pode ser inviável em alguns ambientes. Acredita-se que a adoção do paradigma *Dew Computing* em conjunto com uma abordagem multi-linguagem pode reduzir a dependência da rede e proporcionar uma computação mais eficaz ao dispositivo móvel, tornando-o apto a processar suas próprias tarefas usando linguagens mais eficientes para esse fim.

1.2 Justificativa

Um estudo da Cisco revelou um aumento significativo de aproximadamente 53 vezes no tráfego de dados da Internet ao longo da última década (CISCO, 2020). Especificamente no caso da computação móvel, o Cisco (2020) destacou um crescimento de 5% no número de usuários e de 8% na quantidade de dispositivos e conexões móveis. O mesmo estudo também apontou um acréscimo anual constante, na faixa de centenas de bilhões, no número de aplicativos e uma forte tendência deles em adotar técnicas de *machine learning* para análise de dados e previsão de eventos, o que exigiria ainda mais poder computacional dos dispositivos móveis.

Em termos de consumo de energia, Sabin (2018) destaca que o tempo de vida da bateria é o principal critério no momento da compra de um novo *smartphone* para 95% das pessoas questionadas por eles. Segundo a mesma reportagem, a bateria é mais importante do que a usabilidade ou, até mesmo, a durabilidade do dispositivo (Figura 1). Essa preferência dos usuários por um consumo eficiente de energia já é uma preocupação para as equipes de desenvolvimento. (MANOTAS *et al.*, 2016; OLIVEIRA *et al.*, 2021) são alguns dos trabalhos que destacam a necessidade de criar soluções que tornem os *softwares* energeticamente mais eficientes.

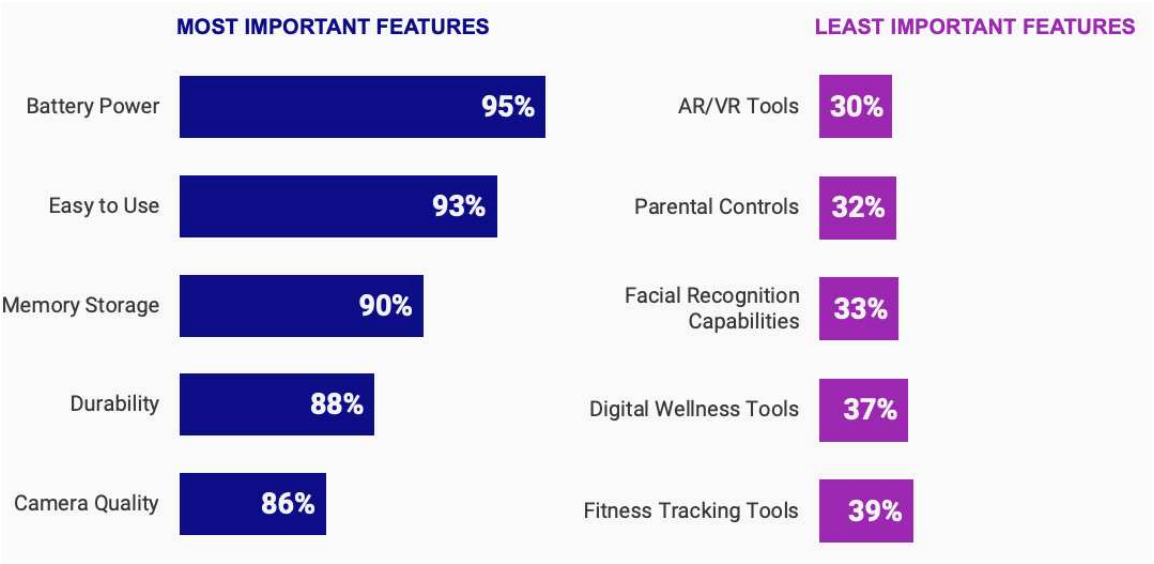


Figura 1 – Percentuais de americanos que indicaram as características mais e menos importantes durante a compra de um *smartphone* (SABIN, 2018)

É notório que o poder computacional dos *smartphones* tem melhorado consideravelmente desde a criação dos primeiros dispositivos. Por exemplo, o Samsung Galaxy Z Fold3[1]

[1] https://insights.samsung.com/2021/08/19/your-phone-is-now-more-powerful-than-your-pc-3/

possui configurações de *hardware* superiores às de muitos computadores pessoais (PCs) atuais. A evolução é tão significativa que aparelhos como esse, quando combinados com plataformas que simulam o ambiente *desktop*, como a Dex[2], permitem a convergência entre celulares e *PCs*. Esse aprimoramento constante de *hardware* já motiva a academia a considerar soluções para os problemas da computação móvel utilizando os próprios dispositivos móveis (SANTOS *et al.*, 2018; HIRSCH *et al.*, 2021; GUO *et al.*, 2021), como o *offloading Device-to-Device* (D2D). Assim, observa-se que os dispositivos móveis, ou pelo menos alguns deles, já possuem recursos suficientes para lidar com tarefas mais complexas.

Apesar do constante aprimoramento do *hardware* dos *smartphones*, eles ainda são considerados pelos pesquisadores como dispositivos com restrições computacionais e, principalmente, energéticas. Isso é um indício de que o problema pode residir no *software* e na maneira como ele consome os recursos desses dispositivos. De fato, alguns estudos têm avaliado o desempenho de diferentes linguagens de programação ao computar tarefas semelhantes no próprio aparelho (NANZ; FURIA, 2015; GEORGIOU *et al.*, 2018; PEREIRA *et al.*, 2021) ou em situações de *offloading* (GEORGIOU; SPINELLIS, 2019; MATOS *et al.*, 2021a) e têm indicado diferenças significativas entre elas. Nesses estudos, linguagens como Java e Swift apresentam baixo custo-benefício quando comparadas às demais, pois, além de processar a maioria das tarefas de forma mais lenta, consomem muita energia e recursos computacionais. Por exemplo, a Figura 2 apresenta um quadro resumo que classifica as linguagens de programação com base na métrica *Energy Delay Product* (EDP), avaliada em três ambientes diferentes: embarcado, *desktop* e servidor (GEORGIOU *et al.*, 2018). A EDP é definida como o produto entre a energia (E) e o tempo ponderado (T^w) necessário para realizar uma determinada tarefa. Quanto maior o valor de w, menor é a influência da energia no resultado da métrica. Nota-se que, em todas as plataformas, Java e Swift apresentam desempenhos inferiores, especialmente em sistemas embarcados.

Como discutido na Seção 1.1, as limitações de processamento e consumo energético em dispositivos móveis permanecem como desafios significativos, atraindo atenção contínua da comunidade científica. As evidências apresentadas sugerem um possível agravamento dessas questões no futuro, ressaltando a importância de encontrar soluções eficazes não apenas para os usuários de *smartphones*, mas também para os desenvolvedores de aplicativos. Diante dos avanços no *hardware* de tais dispositivos, torna-se crucial desenvolver abordagens de *software*

[2] https://insights.samsung.com/2022/02/16/the-beginners-guide-to-samsung-dex-9/

Rank	Embedded			Laptop	Server
	$w = 1$	$w = 2$	$w = 3$	$w = 1, 2, 3$	$w = 1, 2, 3$
1	C	C	C	C	C
2	C++	C++	C++	Go	Go
3	Go	Go	Go	C++	C++
4	Rust	Rust	Rust	JavaScript	C#
5	C#	C#	JavaScript	Rust	JavaScript
6	VB.NET	JavaScript	C#	C#	Rust
7	JavaScript	VB.NET	VB.NET	VB.NET	VB.NET
8	PHP	PHP	PHP	PHP	PHP
9	Ruby	Ruby	Ruby	Ruby	Python
10	Python	Python	Python	Swift	Ruby
11	Perl	Perl	Perl	Python	Swift
12	Java	Java	Java	Perl	Perl
13	Swift	Swift	Swift	Java	Java
14	R	R	R	R	R

Figura 2 – Ranking de linguagens de programação com base na métrica EDP ponderada. Quanto maior o valor de *w*, menor o peso da energia na métrica (GEORGIOU *et al.*, 2018)

que otimizem o processamento de tarefas, seja de forma local ou remota. A adoção de estratégias de computação que integram diversas linguagens de programação emerge como uma solução promissora para superar esses obstáculos.

1.3 Hipótese de Pesquisa

Esta tese explora o potencial de combinar o crescente poder computacional dos dispositivos e as diferenças de desempenho entre as linguagens de programação, com o objetivo de otimizar o processamento das aplicações móveis. Para isso, ela define uma arquitetura baseada no paradigma *Dew Computing* e na interação multi-linguagem entre processos, cuja ideia central é permitir que um dispositivo seja capaz de duplicar serviços desenvolvidos com linguagens distintas daquela adotada no aplicativo móvel. Esses serviços são oferecidos por servidores remotos e disponibilizados ao próprio dispositivo e aos demais membros da rede onde ele se encontra. Dessa maneira, espera-se que o aumento no número de servidores proporcione um melhor balanceamento de carga e minimize o tempo e a energia necessários para computar tarefas submetidas ao processamento remoto. Além disso, espera-se que, por meio do autoatendimento, o dispositivo possa processar suas próprias tarefas de maneira eficiente em termos computacionais e energéticos enquanto economiza largura de banda e contribui para mitigar o problema da latência de rede, mesmo que indiretamente.

Portanto, este trabalho define e trabalha com a seguinte hipótese:

O uso da *Dew Computing* como meio para duplicar serviços implementados por meio de linguagens de programação distintas da adotada no aplicativo cliente torna mais eficiente a computação de aplicativos móveis e melhora indiretamente o desempenho do *offloading* computacional.

1.4 Objetivos

O presente trabalho tem como objetivo geral **"tornar o processamento de tarefas mais eficiente ao combinar serviços multi-linguagem e o paradigma *Dew Computing*"**. A ideia é permitir que os dispositivos móveis, por meio da *Dew Computing*, dupliquem processos servidores de *offloading* desenvolvidos com linguagens de programação mais eficientes e inicialmente hospedados em máquinas remotas alocadas na *Edge*, na *Fog* ou na *Cloud*. Uma vez duplicado, o processo servidor poderá computar tarefas submetidas por aplicativos no próprio dispositivo móvel onde ele se encontra (através do autoatendimento) ou em outros aparelhos (através do *offloading* multi-linguagem) de forma otimizada. Isso significa que, por adotar linguagens de programação mais eficientes em sua concepção, o processo servidor realiza computações mais rápidas, ao mesmo tempo em que consome menos recursos computacionais e/ou energéticos do dispositivo móvel hospedeiro.

Baseado no objetivo geral, foram delineados os seguintes objetivos específicos:

1. Investigar o estado da arte em tópicos como *Dew Computing*, desempenho de aplicações móveis e *offloading* computacional;
2. Avaliar a eficiência computacional dos dispositivos móveis ao processar tarefas usando linguagens de programação variadas, como Go e Rust, tanto localmente quanto em cenários de *offloading* computacional;
3. Avaliar a eficiência computacional dos dispositivos móveis ao realizar o *offloading* computacional por meio do gRPC e Apache Thrift, ferramentas que permitem a interação direta entre processos desenvolvidos com diferentes linguagens de programação;
4. Projetar e desenvolver uma arquitetura, baseada em *Dew Computing* e interação multi-linguagem entre processos, que possibilite ao dispositivo móvel duplicar localmente serviços remotos e oferecê-los a outros dispositivos;
5. Avaliar a arquitetura proposta em um ambiente de *Mobile Cloud Computing* convencional, utilizando métricas como tempo de resposta e sobrecarga de rede.

A adoção da arquitetura proposta neste trabalho apresenta potenciais vantagens que merecem um maior destaque neste momento. Todas essas vantagens estão diretamente relacionadas à capacidade de autoatendimento oferecida pela arquitetura. Ao duplicar o processo servidor no próprio dispositivo móvel, este adquire a habilidade de processar tarefas de maneira mais rápida, ao mesmo tempo em que economiza seus recursos computacionais e energéticos. Isso é possível devido ao desenvolvimento do processo servidor com uma linguagem de programação mais adequada para processar esse tipo específico de tarefa. O autoatendimento também pode proporcionar uma maior economia na largura de banda na rede sem fio, pois a interação entre o aplicativo cliente e o processo servidor duplicado ocorre como uma comunicação local, sem envolver ou consumir recursos da rede externa. Por fim, ao capacitar o próprio dispositivo móvel para atuar como servidor de *offloading*, a arquitetura aumenta a oferta de servidores na rede, o que, aliado a um eficiente algoritmo de balanceamento de carga, pode contribuir para a redução da sobrecarga de trabalho em servidores hospedados na *Edge*, *Fog* e/ou *Cloud*.

Contudo, o uso da arquitetura também apresenta algumas desvantagens. Ao contrário da abordagem tradicional de *offloading*, na qual o programador desenvolve uma única solução válida tanto no lado cliente quanto no lado servidor. Na abordagem desenvolvida neste trabalho, ele poderá ter que desenvolver, no mínimo, duas soluções (uma cliente e outra servidora) e, potencialmente, utilizar diferentes linguagens de programação. Acredita-se que esse desafio pode ser mitigado com o uso de técnicas da Engenharia Orientada a Modelos (MDE) e/ou ferramentas como *Haxe*[3] e *Rhapsody Developer*[4], que, baseadas em uma linguagem padrão ou diagramas da UML (*Unified Modeling Language*), geram códigos em diversas linguagens de programação automaticamente, como Java e C++.

Outro ponto negativo está relacionado à segurança do dispositivo móvel. Ao duplicar o processo servidor remoto por meio do *download* e da execução do binário no dispositivo móvel, o programa pode tornar o dispositivo vulnerável a ataques externos ou, até mesmo, possibilitar que ele próprio ataque o aparelho. Acredita-se que o uso de servidores confiáveis e o *download* de binários utilizando protocolos que garantam a integridade e autenticidade do binário durante a aquisição (como o *Transport Layer Security*[5]), bem como o processamento desses binários em contêineres, podem ajudar a minimizar o problema.

Apesar do reconhecimento dessas limitações, acredita-se que os benefícios propor-

[3] https://haxe.org/
[4] https://www.ibm.com/br-pt/products/uml-tools/details
[5] https://www.rfc-editor.org/rfc/rfc8446

cionados pelo uso da arquitetura para o processamento de aplicações móveis superam esses desafios. Nesse sentido, esses aspectos serão desconsiderados neste trabalho, mas podem ser explorados em pesquisas futuras ou por outros grupos de estudo.

1.5 Metodologia

Com o propósito de atingir os objetivos apresentados na seção anterior, foi adotada a metodologia exibida na Figura 3. Em resumo, o estudo compreendeu seis passos principais: 1) Análise da Área; 2) Revisão da Literatura; 3) Análise do Problema, 4) Experimentos Iniciais; 3) Projeto de Arquitetura e 5) Desenvolvimento e Testes.

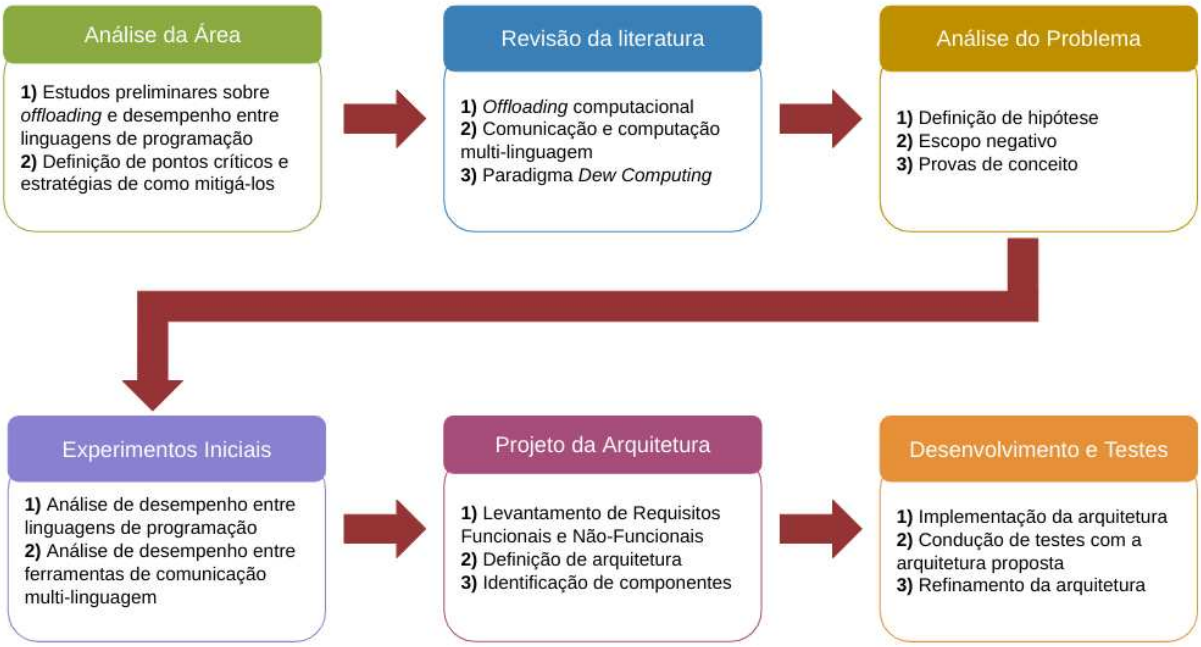


Figura 3 – Etapas da metodologia do trabalho

Na primeira etapa, foi conduzido um estudo preliminar e exploratório visando um embasamento teórico inicial sobre a técnica de *offloading* e sobre o desempenho de linguagens de programação variadas ao computar tarefas iguais ou similares nas principais plataformas disponíveis no mercado (*Desktop*, *Web* e *Mobile*). Para isso, foram utilizados trabalhos do tipo *survey* da área de *offloading*, bem como artigos que compararam o desempenho de linguagens de programação ao processar tarefas localmente ou remotamente (via *offloading* computacional). Em paralelo, também foram identificados os principais desafios de cada área, além de possíveis estratégias e ferramentas candidatas a mitigá-los. Aqui, destaca-se o desafio de como reduzir os efeitos da latência de rede no desempenho do *offloading*, pois foi a partir de pesquisas por

mecanismos de como atacá-lo que surgiu o primeiro contato com o paradigma *Dew Computing*.

A segunda etapa focou, inicialmente, em aprofundar o conhecimento adquirido na etapa anterior. Para isso, os artigos identificados previamente foram cuidadosamente lidos e, durante a leitura, foram encontrados novos trabalhos, os quais julgou-se relevantes para o propósito dessa pesquisa. Esses trabalhos foram lidos e inseridos no repositório da pesquisa. Nessa etapa também foi realizado um estudo minucioso sobre o paradigma *Dew Computing* com foco especial em arquiteturas e soluções já propostas pela literatura para atacar problemas nas mais diversas áreas de conhecimento. O objetivo principal dessa fase foi conhecer e aprofundar o entendimento sobre o paradigma ao identificar as melhores práticas adotadas pelos pesquisadores ao arquitetar sistemas *Dew*, visando adotá-las posteriormente para atacar o problema da latência de rede em um cenário de *offloading* computacional.

A terceira etapa consistiu em, com base em todo conhecimento adquirido nas duas etapas anteriores, formular uma hipótese de pesquisa e listar as principais vantagens ao adotar a arquitetura proposta nesse trabalho (ambas apresentadas na Seção 1.3). Nessa etapa também foram desenvolvidas algumas provas de conceito e conduzidos os primeiros experimentos visando analisar a viabilidade da arquitetura e prospectar os benefícios que ela pode ocasionar tanto em cenários onde o próprio dispositivo móvel cria e processa suas tarefas, mas adotando linguagens distintas para tais ações, como em cenários onde ele delega a computação de suas tarefas através do *offloading* multi-linguagem. Em geral, os resultados obtidos nessa etapa foram promissores para a proposta desse trabalho. Ao mudar a linguagem de programação responsável pela computação de uma tarefa complexa, observou-se que era possível proporcionar uma maior celeridade e uma maior economia de energia no dispositivo móvel.

A quarta etapa pode ser vista como uma extensão da etapa anterior, onde um conjunto de experimentos foi igualmente realizado, contudo seguindo um maior rigor científico. O foco dessa etapa é obter resultados que comprovem, com força estatística, as observações levantadas nos pré-testes realizados na etapa anterior. Em geral, os experimentos foram conduzidos usando dispositivos reais e compararam o desempenho de cinco linguagens de programação servidoras (Java, C++, Python, Go e Rust) e duas ferramentas de comunicação multi-linguagem (gRPC e Apache Thrift) ao processar tarefas de *benchmark* complexas (por exemplo, multiplicar matrizes quadráticas). Também foram realizados testes em ambientes emulados com o intuito de avaliar como a arquitetura se comporta em redes diferentes da WiFi (como redes 3G e 4G) e em cenários onde há uma quantidade relativamente grande de dispositivos móveis clientes (até 24 aparelhos).

Nessa etapa, foram produzidos seis artigos científicos, dos quais quatro já foram publicados (MATOS *et al.*, 2021a; MATOS *et al.*, 2021b; MATOS *et al.*, 2022; MATOS *et al.*, 2023) e dois estão em revisão interna. Em geral, os quatro trabalhos publicados nesta etapa mostraram que modificar a linguagem de programação adotada no desenvolvimento do processo servidor pode reduzir o tempo de resposta e o consumo de energia do dispositivo móvel em cenários de *offloading* (MATOS *et al.*, 2021a; MATOS *et al.*, 2021b) e de autoatendimento (MATOS *et al.*, 2022). Além disso, também observou-se que o uso do autoatendimento pode proporcionar economias significativas na quantidade de dados transmitida através da rede (MATOS *et al.*, 2022) e que o *offloading* multi-linguagem pode ser mais escalável que o *offloading* tradicional mono-linguagem (MATOS *et al.*, 2023).

A quinta etapa compreende a formalização conceitual da arquitetura proposta nesse trabalho. Os bons resultados somados aos bons *feedbacks* dos revisores onde os artigos da etapa anterior foram publicados motivaram a concepção e o amadurecimento teórico da arquitetura. Assim, utilizando-se de alguns dos diagramas sugeridos pela UML (*Unified Modelling Language*) como os de Caso de Uso e de Sequência, buscou-se projetar a arquitetura de forma concisa, porém ainda abstrata. Nessa etapa também foram levantados os principais requisitos funcionais e não-funcionais da arquitetura, bem como foram identificados e definidos seus principais componentes. O presente documento resume e registra todo o progresso obtido até essa etapa e, após realizadas correções e melhorias sugeridas pela banca, serve como artefato de entrada para a sexta e última etapa.

Na etapa final, foi realizada a implementação da arquitetura conceitual definida na fase anterior, juntamente com a execução dos testes finais da tese de doutorado. Durante o processo de implementação e experimentação, a arquitetura passou por ajustes e refinamentos, os quais foram refletidos na documentação gerada ao término da etapa anterior. A arquitetura seguiu o paradigma Cliente-Servidor, onde o Cliente foi desenvolvido como um Serviço Android utilizando a linguagem Java, enquanto o Servidor foi elaborado na forma de microsserviços hospedados em contêineres Docker utilizando a linguagem Python e outras ferramentas avançadas, como o RabbitMQ e o MinIO. Para validar seu desempenho, a arquitetura foi submetida a experimentos em um ambiente real, utilizando uma versão adaptada da aplicação BenchImage (REGO *et al.*, 2017), adaptada para trabalhar com a ferramenta de comunicação multi-linguagem Apache Thrift. Os processos servidores, destinados a atender às demandas de *offloading* multi-linguagem e de autoatendimento, foram implementados utilizando a linguagem de programação Golang. Os

experimentos conduzidos avaliaram o Tempo de Resposta e o Consumo de Rede como métricas. Em geral, os resultados mostraram que, sob determinadas circunstâncias, o autoatendimento melhorar o tempo de resposta em até 4.58 vezes e podem reduzir pela metade o Consumo de Rede comparado a abordagem que usa o *offloading* multi-linguagem.

1.6 Estrutura do Trabalho

O restante deste trabalho está organizado da seguinte forma: O Capítulo 2 aborda os principais conceitos relacionados ao desenvolvimento de aplicativos móveis, ao *offloading* (convencional e multi-linguagem) e ao paradigma *Dew Computing*. O Capítulo 3 apresenta os trabalhos correlatos a esta tese de doutorado, todos alinhados com o objetivo central de aprimorar o desempenho de aplicações móveis adotando *Dew Computing* e linguagens de programação diversas. Os Capítulos 4 e 5 descrevem a arquitetura DADOS. Enquanto o Capítulo 4 foca em detalhar a versão conceitual e idealizada da arquitetura, com ênfase nos seus princípios fundamentais e na estruturação teórica, o Capítulo 5 destaca a versão implementada que, apesar de mínima, é funcional e demonstra de forma concreta os conceitos e princípios abordados na versão idealizada. O Capítulo 6 relata os experimentos conduzidos, descrevendo o ambiente de teste, incluindo as configurações dos equipamentos utilizados, as métricas e a aplicação escolhida. Ele também discute os principais resultados obtidos, seguido por uma análise crítica e resumo das contribuições da arquitetura proposta à literatura. Finalmente, o Capítulo 7 apresenta as conclusões desse trabalho e lista possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo embasa o conhecimento necessário para o entendimento deste trabalho. De início, a Seção 2.1 apresenta conceitos relacionados ao desenvolvimento e ao desempenho de aplicações móveis. Em seguida, a Seção 2.2 resume a grande área de pesquisa relacionada à proposta deste trabalho: *Mobile Cloud Computing*. A Seção 2.3 expõe em detalhes o paradigma computacional *Dew Computing*, identificando as principais definições, características, arquiteturas, componentes e modos de funcionamento. Finalmente, a Seção 2.4 explica os mecanismos do *offloading* tradicional (mono-linguagem) e as particularidades da versão multi-linguagem.

2.1 Desempenho de Aplicações Móveis

O desenvolvimento de aplicações móveis em telefones celulares evoluiu bastante ao longo dos anos. No começo dos anos 90, as plataformas desses aparelhos eram fechadas e rígidas, de tal forma que as primeiras aplicações eram embarcadas diretamente nos dispositivos e era muito difícil (se não impossível) a modificação dos aplicativos já existentes ou o desenvolvimento de novos. Por exemplo, o IBM Simon, considerado por muitos como o primeiro *smartphone* a ser produzido e vendido no mercado[1], possuía aplicações simples e fixas, como calendário, calculadora, agenda de contatos, alarme, além de clientes de e-mails e faxes, que permitiam o envio e o recebimento destes tipos de documentos por meio do dispositivo.

No final da década de 90, à medida que dispositivos móveis como telefones celulares básicos e assistentes digitais pessoais (PDAs) se tornavam mais populares, surgiu a demanda de uma plataforma de desenvolvimento que permitisse a criação de aplicativos para uma variedade crescente de dispositivos e de fabricantes. O J2ME (*Java Platform, Micro Edition*) surgiu para atender essa necessidade. Baseado em uma versão compacta da linguagem de programação Java, o J2ME simplificou bastante o desenvolvimento de aplicativos móveis, pois permitiu que um único código-fonte pudesse ser executado em dispositivos diversos, incluindo sistemas embarcados mais simples como navegadores veiculares, *set-top boxes* e *pagers*, sem a necessidade de reescrevê-lo para cada dispositivo específico.

Ao longo dos anos 2000, os telefones celulares foram se modernizando e adquirindo novas funcionalidades (como acessar à Internet e gravar vídeos), além de recursos avançados de *hardware* (como câmeras e sensores) e *software* (como navegadores e *players*), o que os aproxi-

[1] https://www.bbc.com/news/technology-28802053

maram bastante dos computadores pessoais (*PCs*). Refletindo ao aumento dessa complexidade, surgiram Sistemas Operacionais (SOs), como o Symbian e o Windows Phone, que ao serem incorporados aos aparelhos celulares, não somente padronizavam, mas também simplificavam a interação com a crescente diversidade de recursos disponíveis nos dispositivos. Esses SOs foram essenciais para tornar as plataformas de desenvolvimento de aplicativos móveis abertas e permitissem que terceiros pudessem criar suas próprias aplicações e embarcá-las nos aparelhos.

Desde então, surgiram algumas outras plataformas abertas. Focando especificamente nos *smartphones* e nos *tablets*, as que mais ganharam destaque nesses últimos anos foram a iOS da Apple e a Android da Google. Embora tenha sido criada por uma empresa privada, a plataforma Android adota uma abordagem de licenciamento como software livre e de código aberto. Essa escolha simplifica a personalização e distribuição do Android, tornando-o altamente adaptável às necessidades dos fabricantes de dispositivos e desenvolvedores de aplicativos. Essa flexibilidade tem sido fundamental para consolidar o Android como a plataforma dominante no mercado de dispositivos móveis, atingindo, aproximadamente, o dobro do mercado da segunda colocada, justamente a iOS da Apple (Figura 4).

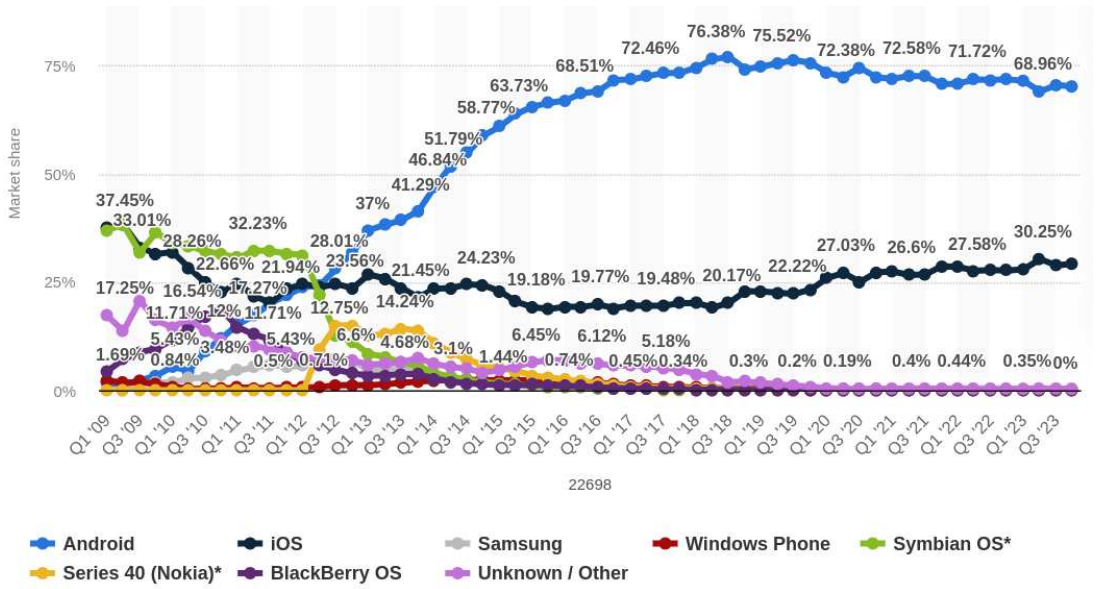


Figura 4 – *Marketshare* dos sistemas operacionais móveis entre 2009 e 2023 (SHERIF, 2024)

Ainda com base na Figura 4, observa-se que as plataformas iOS e Android, combinadas, praticamente comandam todo o mercado de dispositivos móveis na atualidade. Portanto, sempre foi de interesse do mercado criar aplicativos portáveis, ou seja, aplicações desenvolvidas com um único código-fonte, porém aptas a serem executadas nas duas plataformas predominantes pelo menos. Isso motivou o advento de novas abordagens de organização e execução de aplicati-

vos móveis. Se por um lado, a abordagem Web se baseia em tecnologias como HTML, CSS e JavaScript para permitir que programadores criem aplicações acessíveis em qualquer dispositivo móvel por meio de um navegador Web, por outro, o método *Cross*-Plataforma se baseia na escrita de um único código-fonte base que consegue ser executado em múltiplas plataformas.

Durante a evolução dos aplicativos móveis, destacam-se duas questões essenciais: 1) garantir a portabilidade entre diferentes dispositivos/plataformas e 2) assegurar o processamento eficiente das aplicações, visando a execução rápida e a economia de recursos computacionais e energéticos. As primeiras plataformas executavam aplicativos específicos para o dispositivo alvo, o que dificultava a portabilidade. Por outro lado, por serem voltados para uma plataforma particular, era possível explorar melhor os recursos oferecidos por ele. No outro extremo, aplicativos Web e *Cross*-Plataformas são altamente portáveis, mas possuem um desempenho limitado quando comparado ao mecanismo anterior. Os próximos parágrafos discutirão com mais detalhes as diferentes abordagens de como criar aplicações em *smartphones*, com ênfase na plataforma Android, plataforma alvo desse trabalho.

A abordagem mais comum de desenvolvimento de aplicações móveis é a Nativa, onde o programador, adotando uma linguagem de programação suportada pelo Sistema Operacional do dispositivo móvel, cria um aplicativo específico para a plataforma alvo. Por exemplo, no ambiente Android, quando esse mecanismo é utilizado, os aplicativos são escritos em Java ou Kotlin. Em geral, aplicativos nativos apresentam bom desempenho e possuem acesso facilitado aos recursos de *hardware* e *software* que os dispositivos móveis oferecem. Por outro lado, os aplicativos não são portáveis, pois eles são exclusivos da plataforma para o qual foram criados.

O sistema Android ainda oferece um grupo de ferramentas que permite o desenvolvimento e a invocação de métodos ou funções implementadas em linguagens de mais baixo nível, como C/C++ ou Assembly. Através do NDK ou Kit de Desenvolvimento Nativo, o programador consegue reaproveitar códigos ou bibliotecas C/C++ com o intuito de impulsionar ainda mais o desempenho das aplicações nativas. A interação entre as linguagens envolvidas, C/C++ de um lado e Java/Kotlin do outro, acontece por meio da JNI ou Interface Nativa Java, que, em resumo, possibilita códigos Java/Kotlin chamar códigos C/C++ e vice-versa.

A abordagem Nativa força as empresas a dispor duas ou mais versões do mesmo aplicativo, uma para cada plataforma suportada, o que pode ser um problema, pois, no mínimo, dificulta (e encarece) o desenvolvimento e a manutenção dos códigos-fonte gerados. Em razão dessa deficiência, nos últimos anos, tem ganhado força a abordagem *Cross*-Plataforma, onde o

desenvolvedor cria o aplicativo executável em várias plataformas usando um grupo de ferramentas específicas para essa tarefa como, por exemplo, React Native, Flutter, Ionic e Xamarin.

De acordo com (RAJ; TOLETY, 2012), a abordagem *Cross*-Plataforma é subdividida em quatro modelos principais: Web, Híbrido, Interpretado e *Cross*-Compilado. O mecanismo Web consiste em criar aplicativos usando HTML, CSS e JavaScript e utilizar o próprio navegador para exibi-los na tela. Já o Híbrido consiste em criar aplicações parte nativa e parte Web. Nesse mecanismo, a aplicação nativa dispõe de um componente especial (como o *WebView* no Android) que encapsula e executa a aplicação Web. O mecanismo Interpretado, por sua vez, consiste em utilizar um interpretador que interage com os recursos nativos do sistema, enquanto executa o código-fonte da aplicação. Finalmente, o mecanismo *Cross*-Compilado gera múltiplas aplicações nativas tomando como base um código-fonte desenvolvido em uma linguagem padrão.

As vantagens e as desvantagens da abordagem *Cross*-Plataforma dependem do tipo de mecanismo adotado para o desenvolvimento da aplicação móvel. Por exemplo, um aplicativo Web não requer instalação do usuário e as atualizações são automáticas, uma vez que a parte principal da aplicação está hospedada e executa no lado servidor. Em contrapartida, o acesso aos recursos de *hardware* e de *software* do dispositivo móvel é dificultado, devido ao isolamento proporcionado pelo SO. Contudo, todos os mecanismos *Cross*-Plataforma, com exceção do *Cross*-Compilado, possuem uma desvantagem em comum: apresentam um desempenho inferior a abordagem Nativa (HORT *et al.*, 2022). A Tabela 1 resume as principais diferenças entre as abordagens de desenvolvimento de aplicações móveis.

Abordagem de Desenvolvimento	**Linguagens Comuns**	**Instalação e Atualizações**	**Acesso a Plataforma**	**Desempenho do Aplicativo**
Nativa	Java/Kotlin (Android), Objective-C/Swift (iOS) e C/C++	Manual	Direto	Alto
Web	HTML, CSS e JavaScript	Automática	Improvável	Baixo
Híbrida	Java/Kotlin (Android), Objective-C/Swift (iOS) e HTML, CSS e JavaScript	Manual	Indireto	Baixo
Interpretada	JavaScript	Manual	Indireto	Baixo
***Cross*-Compilada**	C#	Manual	Direto	Médio

Tabela 1 – Quadro comparativo entre as abordagens de desenvolvimento de aplicação móvel

Como destacado na Tabela 1, apesar do problema relacionado a heterogeneidade dos dispositivos móveis e a dificuldade em portar códigos entre plataformas, a abordagem Nativa, em geral, é aquela que proporciona o melhor desempenho e que melhor consome os recursos

computacionais e energéticos do dispositivo móvel durante a execução do aplicativo. Como o foco desse trabalho é melhorar a eficiência das aplicações móveis ao computar tarefas complexas, optou-se por adotar aplicações Nativas como base, uma vez que, os ganhos observados nesse tipo de aplicação são facilmente expansíveis para as demais abordagens.

Focando especificamente em aplicações Android, nota-se uma baixa diversidade nas linguagens de programação adotadas pelos programadores enquanto criam seus aplicativos. Por exemplo, Oliveira *et al.* (2017) mencionam que, dentre 109 projetos disponíveis no F-Droid[2], um relevante repositório de aplicativos Android de código-aberto, pouquíssimos deles recorreram a outras linguagens de programação com o intuito de melhorar o desempenho dos seus aplicativos, enquanto a ampla maioria utilizou-se somente da linguagem Java. Como apontado no Capítulo 1, Java não é uma linguagem muito eficiente, uma vez que é lenta e consome muitos recursos do dispositivo móvel. Dessa maneira, julga-se necessário o desenvolvimento de uma estratégia que simplifique o processamento local de tarefas computacionalmente complexas utilizando linguagens mais eficientes do que o Java. A ideia é que esse mecanismo seja complementar ao *offloading* computacional, técnica amplamente indicada para mitigar as restrições dos dispositivos móveis em cenários de *Mobile Cloud/Edge Computing*, o próximo tópico desse Capítulo.

2.2 *Mobile Cloud/Edge, Fog* e *Mist Computing*

Como já comentado no Capítulo 1, a grande maioria dos dispositivos móveis apresentam severas restrições energéticas e/ou computacionais, e tais limitações são consequências naturais de características intrínsecas a este tipo de dispositivo, como a mobilidade e o tamanho reduzido. No caso específico dos *smartphones*, em geral, eles até apresentam melhores configurações de *hardware* e fontes de energia mais duradouras quando comparados a sistemas mais simples como, por exemplo, os *smartwatches*. Porém, os programas que eles executam são também significativamente mais complexos, o que exige bastante esforço computacional e energético para o seu correto processamento. Portanto, embora cada vez mais com melhores recursos, os *smartphones* também são classificados como dispositivos móveis limitados computacionalmente e, em especial, energeticamente.

Nesse aspecto, a nuvem é comumente caracterizada como o oposto dos dispositivos móveis, uma fonte inesgotável de recursos computacionais e alimentada por fontes energéticas robustas, confiáveis, e de alta disponibilidade. A relação entre a Computação em Nuvem e a

[2] https://f-droid.org/

Computação Móvel é tão promissora que, na última década, a comunidade científica propôs um novo paradigma chamado *Mobile Cloud Computing* (MCC) unindo essas duas tecnologias (FERNANDO *et al.*, 2013). Através do MCC, os dispositivos móveis não precisam de recursos poderosos, pois todas as tarefas mais complexas ou arquivos grandes podem ser, respectivamente, processadas ou armazenados na nuvem através de serviços oferecidos por ela mesma (HOANG *et al.*, 2013). A Figura 5 apresenta um ambiente típico de MCC, onde dispositivos móveis, por intermédio de redes de computadores ou de telecomunicações sem fio, se comunicam com servidores hospedados na nuvem com o intuito de mitigar seus problemas e/ou potencializar a experiência de seus usuários

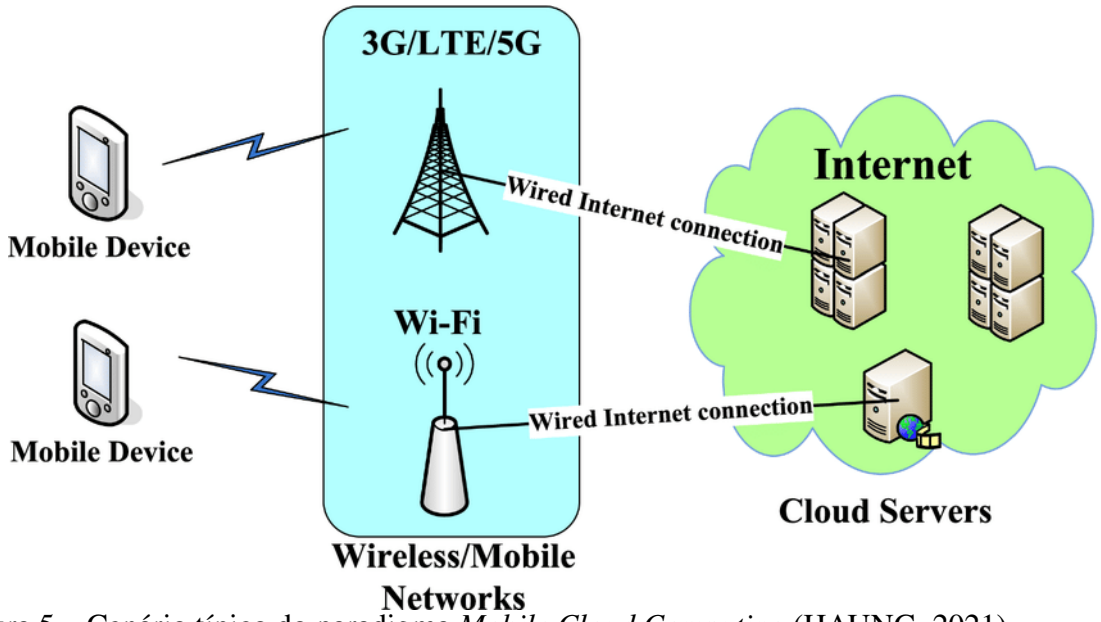


Figura 5 – Cenário típico do paradigma *Mobile Cloud Computing* (HAUNG, 2021)

Embora bastante vantajosa, a relação entre a Computação em Nuvem e a Computação Móvel pode ser desafiadora para alguns tipos de aplicações, especialmente àquelas sensíveis à latência, onde as tarefas devem ser computadas obedecendo a restrições temporais fixas e bem definidas. Por exemplo, aplicações relacionadas à jogos online, sistemas de controle industrial e agrícola, veículos autônomos e monitoramento remoto de enfermos dependem de baixa latência para operar com precisão e eficiência. Contudo, a comunicação com a nuvem é de alta latência (BHATTACHARYA; DE, 2017), devido à distância geográfica entre ela e os terminais, bem como ao congestionamento observado na rede que os interliga (normalmente, a Internet). Dessa forma, para aplicações como essas, a interação com a nuvem é potencialmente problemática.

Assim, nesses últimos anos, tem-se observado uma tentativa de aproximar os poderosos recursos computacionais e de armazenamento, até então presentes somente nas nuvens, dos

usuários finais e seus dispositivos. Através desta aproximação física, busca-se, principalmente, reduzir a latência de comunicação entre eles e, dessa forma, favorecer aplicações como aquelas apresentadas no parágrafo anterior. Com base nessa proposta, surgiram novos paradigmas como *Cloud of Things*, *Cloudlets*, *Fog Computing*, *Edge Computing* e *Mobile Edge Computing* (MEC). A Figura 6 ilustra a relação entre os principais deles. Embora alguns autores tratem estes paradigmas como diferentes, em sua essência, eles representam a mesma ideia: alocar equipamentos computacionais entre a nuvem e os dispositivos móveis com o intuito de melhorar a qualidade dos serviços prestados aos dispositivos móveis. Este trabalho tratará estes paradigmas como sinônimos.

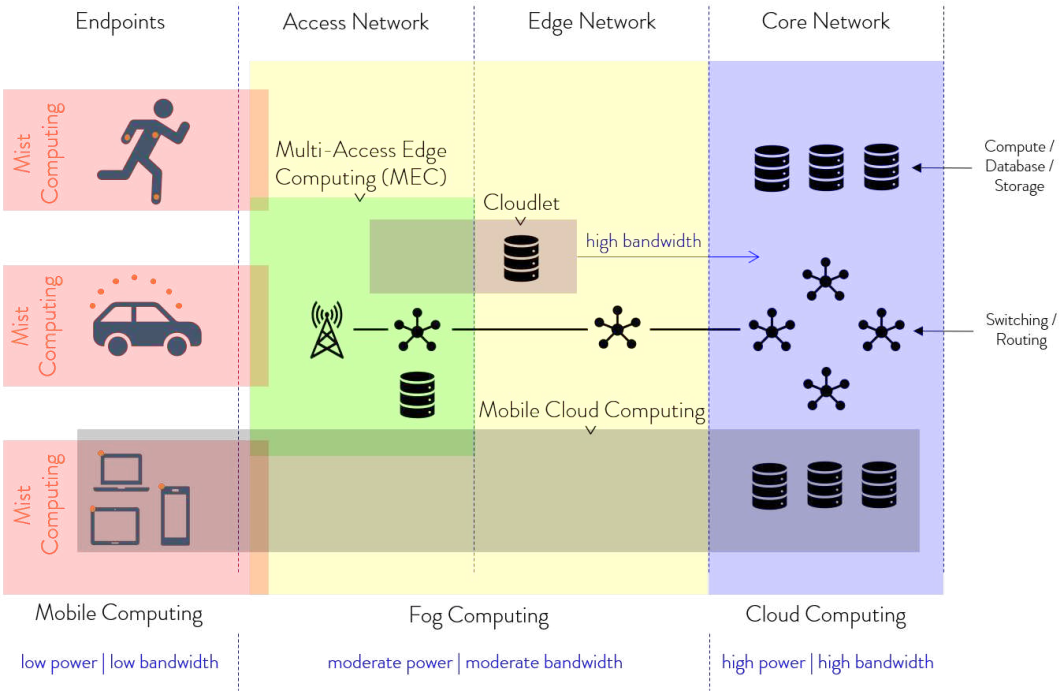


Figura 6 – Estrutura hierárquica entre a nuvem e os dispositivos finais (DREDGE, 2018)

A Figura 6 ainda apresenta um outro paradigma computacional conhecido como *Mist Computing* que, comumente, é aplicada no contexto de Internet das Coisas (IoT). Assim como os paradigmas supracitados, a ideia da *Mist Computing* é reduzir a latência dos serviços IoT prestados, ao aproximar servidores e itens típicos de um ambiente IoT, como sensores e atuadores. Contudo, ao contrário dos demais, na *Mist*, todos os serviços são disponibilizados por dispositivos simples, como computadores de placa única (JUNIOR; KAMIENSKI, 2021). De acordo com (VASCONCELOS *et al.*, 2019), os serviços na *Mist* são processados de forma distribuída e colaborativa, potencialmente motivado pela restrição de *hardware* dos seus componentes. Uma outra importante diferença entre a *Mist Computing* e os demais paradigmas é que seus

dispositivos, embora interligados por meio de uma rede, não necessariamente possuem acesso à Internet e aos serviços disponibilizados pela *Cloud* e demais paradigmas (VASCONCELOS *et al.*, 2019). A variedade de equipamentos na *Mist* também motiva a utilização de tecnologias de comunicação diferentes do WiFi para gerenciar a interação entre seus componentes, como, por exemplo, Zigbee ou LoRa (JUNIOR; KAMIENSKI, 2021).

Perceba que, em todos os paradigmas apresentados anteriormente, a Internet (ou algum tipo de rede) ainda é necessária para o acesso aos recursos e serviços desejados. Contudo, há situações onde não é possível estabelecer conexões com a rede mesmo que de forma temporária. Por exemplo, em ambientes sem infra-estrutura de rede, como em desastres naturais, os dispositivos dos usuários podem não conseguir acesso a Internet ou, em caso de sucesso, podem estabelecer um canal de comunicação de baixa qualidade o que pode comprometer os serviços prestados. Mesmo em ambientes *Mist*, onde a comunicação entre clientes e servidores pode acontecer de maneira direta, a baixa largura de banda pode atrasar a prestação de serviços computacionalmente complexos como os relacionados a MEC.

Motivados por cenários como esses e pela melhoria significativa no *hardware* dos dispositivos de usuário observada nos últimos anos, pesquisadores desenvolveram um novo paradigma chamado *Dew Computing*. Tal paradigma procura criar as condições necessárias para minimizar a dependência dos dispositivos dos usuários em relação à Internet e aos modelos computacionais supracitados. Para isso, a *Dew Computing*, resumidamente, sugere que algumas partes dos recursos oferecidos remotamente sejam duplicados no próprio dispositivo e se mantenham acessíveis ao usuário mesmo quando não houver conexão com a Internet. Mais do que isso, com uma conexão estabelecida com o servidor remoto, seja através da Internet ou não, os dispositivos *Dew* poderiam ajudar os demais paradigmas a prestar melhores serviços.

2.3 *Dew Computing*

O termo *Dew Computing* (Computação em Orvalho) foi definido pela primeira vez pelo Dr. Yingwei Wang em Novembro de 2015 como um paradigma organizacional de *software* que busca potencializar a relação entre os computadores pessoais (*PCs*) e a nuvem (WANG, 2015b). Para (FISHER; YANG, 2016), essa potencialização significa oferecer funcionalidades adicionais e/ou personalizadas durante o acesso convencional (baseado no modelo Requisição-Resposta) aos recursos oferecidos pela nuvem. Tais recursos podem variar desde serviços até objetos Web como páginas e arquivos multimídias, por exemplo.

A definição inicial também menciona as duas características essenciais desse paradigma: a independência e a colaboração entre *PCs* e a nuvem. A independência está relacionada a capacidade dos *PCs* em processar tarefas ou armazenar informação no próprio equipamento, de forma autônoma e descentralizada, sem depender com frequência de conexões de rede estáveis ou de acesso contínuo aos recursos de nuvem. Já a colaboração está relacionada com a capacidade do *PC* em auxiliar a nuvem a prestar o serviço ofertado por ela da melhor forma possível aos usuários. De acordo com (WANG, 2016), esse auxílio pode envolver a sincronização, a correlação ou qualquer outro tipo de interoperação entre os recursos local e remoto, desde que realizado de forma automática.

Para uma melhor compreensão sobre as características supracitadas, considere a aplicação Dropbox[3] como exemplo, mais especificamente, a sua funcionalidade de armazenamento de arquivos. Ao usar programas clientes oferecidos pela empresa, a ferramenta duplica localmente todos os arquivos do usuário armazenados na nuvem. Devido a característica da independência, o usuário pode acessar seus arquivos (e até editá-los) mesmo quando não há Internet disponível. Quando o usuário modifica seus arquivos localmente, a ferramenta automaticamente reflete as mudanças nos arquivos remotos, tão logo exista uma conexão com a nuvem, graças a característica da colaboração.

Antes de continuar, é importante deixar claro a diferença entre duplicar recursos e copiá-los. Wang (2015a), o artigo científico seminal do paradigma *Dew Computing*, resume a diferença entre essas ações em dois aspectos principais: 1) Ao duplicar um recurso, a versão local (hospedada no *PC*) pode ser uma simplificação ou uma personalização da versão original (hospedada na nuvem); 2) Um recurso duplicado pode adotar tecnologias distintas do recurso original. No caso de uma cópia ou réplica, as versões locais e originais dos recursos seriam idênticos e usariam as mesmas tecnologias obrigatoriamente. Baseado nesse raciocínio, pode-se concluir que, no Dropbox, os recursos são duplicados, pois, potencialmente, os programas clientes usam sistemas de arquivos distintos daqueles utilizados na nuvem para refletir a organização dos arquivos do usuário.

Desde a proposta inicial de *Dew Computing*, vários trabalhos tem buscado aperfeiçoá-la. Por exemplo, em (WANG, 2016), o autor modifica a sua primeira versão da definição ao ampliar a gama de dispositivos do usuário e ao enfatizar que a *Dew Computing* é um paradigma organizacional voltado também para o *hardware* e não somente para o *software*. No primeiro

[3] https://www.dropbox.com/

ponto especificamente, o autor altera o termo computador pessoal para computador *on-premise* que, em princípio, significa qualquer dispositivo computacional que não esteja na nuvem como *smartphones*, *notebooks* e, até mesmo, *clusters* de computadores (RINDOS; WANG, 2016).

Em paralelo, outros trabalhos também contribuíram perifericamente para o refinamento da proposta inicial. (SKALA *et al.*, 2015) foi o primeiro a considerar a *Dew Computing* dentro da hierarquia computacional considerando os paradigmas pós-Computação em Nuvem. (RAY, 2018) propôs que os dispositivos *Dew* se organizassem em *clusters* para oferecer diretamente seus recursos uns aos outros com objetivo de ampliar a oferta e melhorar a qualidade dos serviços prestados aos seus usuários. (GUSEV, 2021) recomendou que, mais do que independentes, os dispositivos *Dew* devem possuir um certo nível de autonomia em relação aos demais paradigmas computacionais.

A *Dew Computing*, de maneira análoga a nuvem, pode oferecer um leque diversificado de serviços ou recursos aos seus usuários. (WANG, 2016), inspirado pela maneira como os serviços na Computação em Nuvem são classificados, propõe uma categorização própria para a *Dew Computing* (Tabela 2). Assim, adotando um padrão *X in Dew*, onde X faz referência ao serviço ou recurso oferecido, o autor apresenta classes de serviços como *Web in Dew* (WiD), onde a *Dew*, ao duplicar recursos Web, permite que o usuário navegue por eles, mesmo quando uma conexão com a Internet é inviável. Outro exemplo é o *Infrastructure as Dew* (IaD), onde a *Dew* cria instâncias virtuais para executar aplicações específicas ou sistemas completos nos dispositivos de usuário e se encarrega de mantê-las sincronizadas com versões similares executando na nuvem. Em casos mais simples, ela pode gerenciar apenas configurações/dados relevantes para inicializar tais instâncias. Assim, (WANG, 2016) argumenta que, com a IaD, os usuários conseguiriam resgatar seus aplicativos e sistemas de forma mais simplificada em caso de perda ou dano no seu aparelho físico original, pois bastaria adquirir um novo dispositivo e carregar todas as instâncias da nuvem nele.

Embora seja um paradigma bastante promissor, a *Dew Computing* apresenta alguns desafios relevantes que merecem um maior destaque. Ray (2018) lista e discute alguns deles em seu trabalho. Além daqueles associados a como consumir eficientemente os limitados recursos computacionais e energéticos dos dispositivos de usuários enquanto executam aplicações *Dew*, merecem um maior destaque:

1. **Suporte das plataformas atuais a *Dew*:** O autor defende que o *hardware* e o *software* dos dispositivos de usuários carecem de mecanismos que facilitem o desenvolvimento de

Categoria	Recurso/Serviço Alvo	Função Principal	Aplicações Existentes
***Web in Dew* (WiD)**	Recursos Web	Disponibilizar recursos/sistemas Web de modo *offline* aos usuários	
***Storage in Dew* (STiD)**	Armazenamento	Armazenar cópias de arquivos da nuvem e mantê-las sincronizadas	Dropbox
***Database in Dew* (DBiD)**	Banco de Dados	Criar e manter um *backup* de um banco de dados da nuvem	
***Software in Dew* (SiD)**	Software	Administrar a propriedade do software e configurações pessoais do usuário	Google Play Apple Store
***Platform in Dew* (PiD)**	Pilha de *Software*	Instalar, configurar e executar pilhas de software para desenvolvimento	GitHub
***Infrastructure as Dew* (IaD)**	Dispositivo Físico	Gerenciar instâncias virtuais de executanto em dispositivos de usuário	
***Data in Dew* (DiD)**	Dados e outras aplicações	Aplicativos *Dew* que não estão em nenhuma das categorias acima	Novell Groupwise 7

Tabela 2 – Resumo das categorias de *Dew Computing* (Adaptado de (WANG, 2016))

novas soluções e a execução de aplicações *Dew*. Assim, ele identifica pontos de melhorias nos SOs, nos protocolos de rede e nos modelos de programação, por exemplo;

2. **Baixa produtividade dos dispositivos *Dew*:** Os dispositivos de usuário, ao contrário dos servidores *Edge* e *Cloud*, são computacionalmente limitados, o que dificulta a prestação dos serviços *Dew*. Assim, o autor sugere a criação de técnicas que permitam aos dispositivos *Dew* se agrupar, se organizar e compartilhar seus recursos visando melhorar a qualidade dos serviços prestados aos seus usuários;
3. **Garantir a segurança das bases de dados *Dew*:** A réplica local do repositório de dados mantida pela *Dew Computing* pode ser um elo fraco para a segurança do repositório remoto na nuvem. Um ataque a base de dados *Dew* pode ser propagado para a base de dados da nuvem durante a colaboração automática e deixar todo o sistema vulnerável.

2.3.1 Arquiteturas Dew

Assim como a definição da *Dew Computing* foi aperfeiçoada nesses últimos anos, também se observa uma evolução da arquitetura *Cloud-Dew* proposta inicialmente por (WANG, 2015a) e ilustrada na Figura 7. Assim como na definição inicial, a primeira versão da arquitetura foca no fornecimento dos recursos da nuvem a um único usuário somente, ou seja, não há interações diretas entre dispositivos *Dew*, ou entre eles e outras entidades que não estejam na nuvem. Assim, nessa versão da arquitetura, as requisições do usuário só podem ser atendidas ou pelo próprio dispositivo (auto-atendimento) ou por um servidor hospedado na nuvem.

Analisando a Figura 7, nota-se a presença de dois dispositivos principais: o *Cloud*

Server e o *Local Computer*. O *Cloud Server* é visto como um servidor de nuvem tradicional que administra e disponibiliza recursos Web a outros equipamentos da rede. Já o *Local Computer* é um dispositivo de usuário convencional, porém apto a trabalhar com a *Dew Computing*, o que significa hospedar alguns componentes necessários para implementar os critérios chave do paradigma, no caso, independência e colaboração.

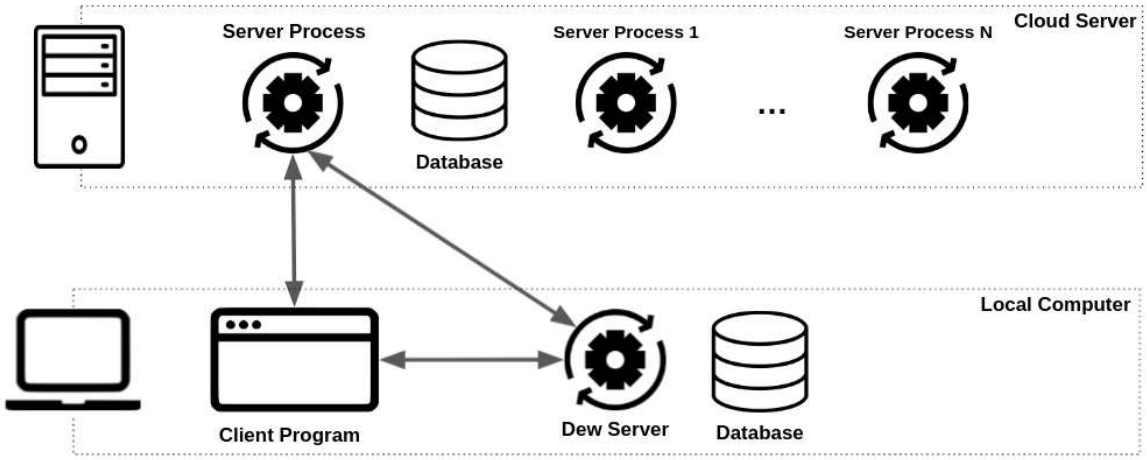


Figura 7 – Arquitetura inicial *Cloud-Dew* proposta por (WANG, 2015a)

Dentre os componentes inseridos no *Local Computer*, o mais importante é o *Dew Server*. Em geral, o *Dew Server* é tratado como um representação local do *Cloud Server*. Assim, quando não há conectividade com a nuvem, o *Dew Server* assume o papel de atender as requisições do *Client Program*. Devido a limitação de *hardware* do *Local Computer*, apenas as partes mais relevantes dos recursos do *Cloud Server* estarão duplicados no *Dew Server* e disponíveis ao usuário para acesso *offline*.

(WANG, 2015a) também destaca alguns recursos importantes para o funcionamento do *Local Computer*. Por exemplo, é essencial a presença de um banco de dados junto ao *Dew Server*, denominado como *Dew DB* pelo autor , para persistir informações produzidas durante a interação com o *Client Program*, além de armazenar temporariamente os dados enquanto não é possível realizar ações colaborativas com o *Cloud Server*. Um outro exemplo, é a presença de um mecanismo que permita ao *Client Program* acessar recursos locais (disponibilizados pelo *Dew Server*) de maneira similar aos recursos remotos (disponibilizados pelo *Cloud Server*). Assim como acontece no mecanismo tradicional, o autor sugeriu a criação de URLs (*Uniform Resource Locator*) especiais para identificar recursos Web e o uso de uma técnica denominada *Local Domain Name System* (LDNS) para direcionar tais requisições diretamente ao *Dew Server* para tratamento.

Analogamente, (FISHER; YANG, 2016) apresentou sua versão da arquitetura *Cloud-Dew* (Figura 8) focada em um único usuário e com interações diretas apenas entre o *Local Computer* e o *Cloud Server*. Segundo os autores, a arquitetura proposta por (WANG, 2015a) possui duas limitações importantes. Em primeiro lugar, falta a liberdade necessária para que cada nuvem possa criar a sua própria solução *Dew*. Além disso, o uso da arquitetura de (WANG, 2015a) pode resultar em subutilização do *hardware* dos *Local Computers*, especialmente em termos de CPU e quando envolve dispositivos de usuários mais robustos. Dessa maneira, a versão proposta em (FISHER; YANG, 2016) tem como objetivo principal mitigar esses problemas.

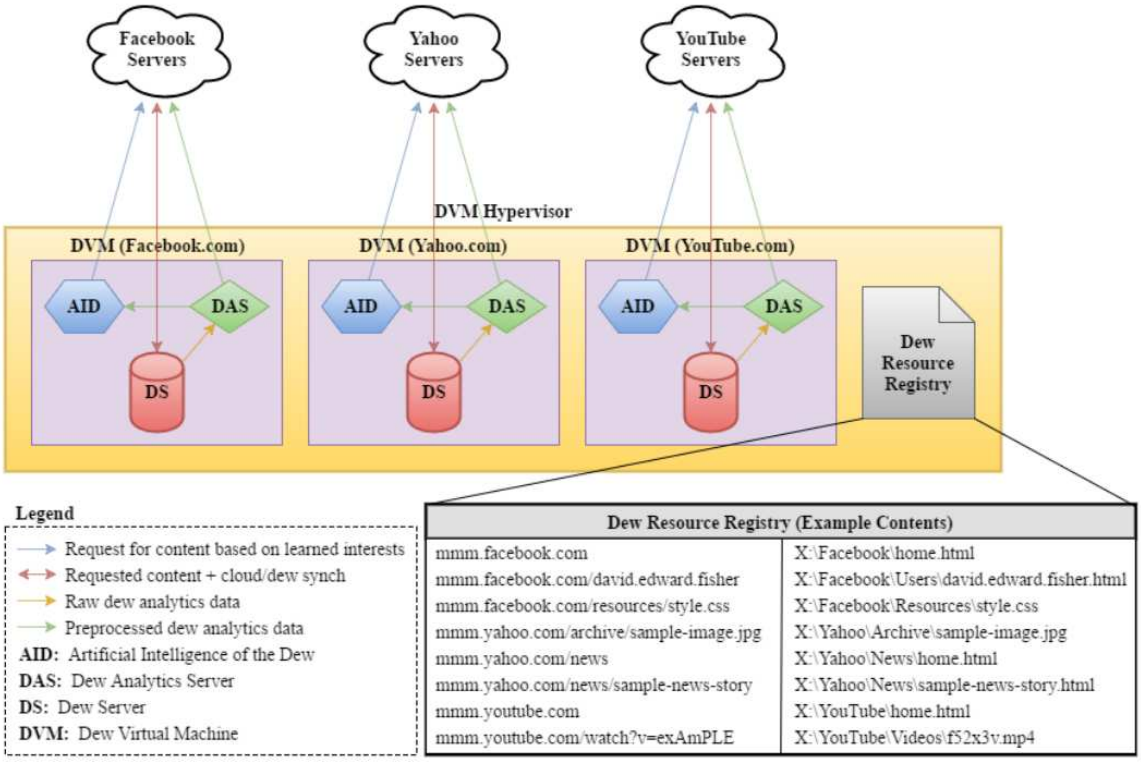


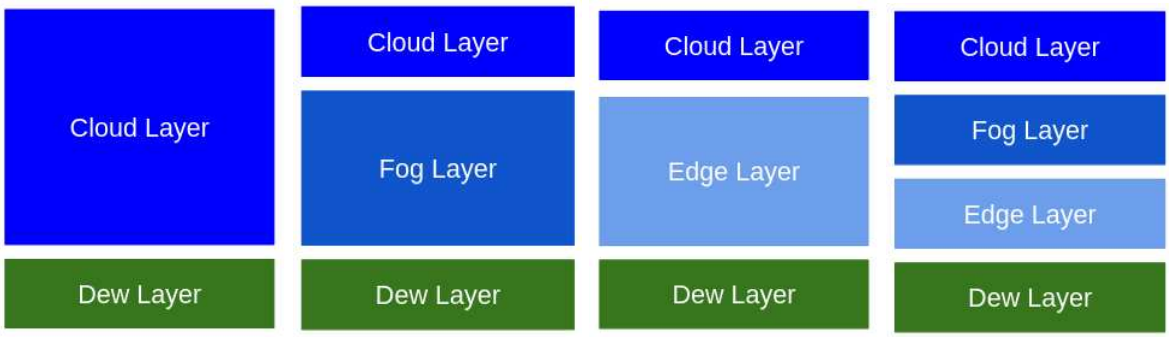
Figura 8 – Arquitetura *Cloud-Dew* proposta por (FISHER; YANG, 2016)

Em relação ao primeiro aspecto, os autores argumentam que, ao adotar um único *Dew Server* para administrar todos os recursos locais no *Local Computer*, a arquitetura impõe o mesmo modelo e as mesmas tecnologias para todas as Nuvens, o que pode dificultar a criação de soluções *Dew*. Os autores então defendem que a melhor maneira de contornar esse problema é por meio de máquinas virtuais (chamadas de *Dew Virtual Machines* ou DVMs). Com tal abordagem, cada solução *Dew* seria hospedada por uma DVM que a isolaria das demais. Assim, cada nuvem teria liberdade suficiente para desenvolver sua própria solução *Dew*.

Já em relação ao segundo aspecto, os autores constatam que, mesmo com as atividades do *Dew Server*, o *Local Computer* pode ainda ser subutilizado, especialmente no caso de

dispositivos com melhores recursos computacionais. Assim, os autores adicionam dois novos componentes à arquitetura original: o *Dew Analytics Servers* (DAS) e o *Artificial Intelligence of the Dew* (AID). Ambos atuam de cooperativamente visando melhorar o serviço prestado pelo *Dew Server*. Enquanto o DAS gerencia os dados produzidos pela interação entre *Client Program* e *Dew Server*, o AID, baseado nesses dados, aplica técnicas de Inteligência Artificial para aprender sobre as preferências e hábitos do usuário e auxiliar na realização de ações proativas, por exemplo, duplicar recursos Web que possam ser de interesse do usuário antes que ele realmente necessite deles.

As duas arquiteturas anteriores focaram somente na interação entre a *Dew* e a nuvem. Porém, conforme já ressaltado anteriormente, tem-se observado o aumento no número de soluções que exploram outros paradigmas como *Fog Computing*, *Edge Computing* e *Cloudlets* por exemplo. O uso desses novos modelos computacionais criou uma nova hierarquia computacional composta por múltiplas camadas. Coube a alguns trabalhos da literatura, como, por exemplo, (SKALA *et al.*, 2015; RAY, 2018; GUSHEV, 2020), posicionar a *Dew Computing* dentro dessa hierarquia. De início, é importante esclarecer o que é a hierarquia computacional atual para esses autores que, em geral, é modelada em três ou quatro níveis. Definitivamente, todos indicam a *Cloud* como a camada do topo e o conjunto de dispositivos do usuário (incluso aparelhos de Internet das Coisas) como a camada da base. Contudo, alguns deles divergem sobre os níveis intermediários, ou seja, na forma como acontece a interação entre a *Dew* e a nuvem. Por exemplo, embora para (RAY, 2018) as camadas *Fog* e *Edge* coexistam, para (SKALA *et al.*, 2015) há apenas a *Fog* e para (GUSHEV, 2020) somente a *Edge*.



(a) (WANG, 2016) (b) (SKALA *et al.*, 2015) (c) (GUSHEV, 2020) (d) (RAY, 2018)

Figura 9 – Comparação entre as arquiteturas propostas pela literatura

Um outro ponto de discordância entre os trabalhos é se os dispositivos computacionais mais simples (como, por exemplo, aqueles relacionados a IoT) fazem parte ou não da *Dew*

Computing. Se por um lado trabalhos como (SKALA *et al.*, 2015) indicam sensores, atuadores e sistemas embarcados como membros da *Dew Computing*, por outro, trabalhos como (GUSHEV, 2020) os colocam em uma camada separada. Essa observação é importante pois, para trabalhos do primeiro grupo, a camada *Dew* é definida como a camada base da hierarquia, diferente dos trabalhos do segundo grupo, onde ela é definida como uma camada intermediária (imediatamente acima da camada IoT).

Finalmente, (RAY, 2018) apresentou uma contribuição interessante sobre a arquitetura da *Dew Computing* ao alterar a forma como os *Local Computers* interagem entre si e a estrutura interna de cada um deles. Adotando um modelo de quatro níveis onde a *Dew Computing* é a camada base, o autor inova ao permitir uma interação direta entre *Dew Servers*. Na verdade, ele vai além e sugere a criação de *clusters* de *Local Computers* para aumentar a oferta de recursos disponíveis aos clientes e melhorar a qualidade dos serviços prestados pelos *Dew Servers* envolvidos. Essa nova forma de relacionamento motivou a adição de novos componentes no *Local Computer* como, por exemplo, servidores dedicados para administrar o funcionamento dos *clusters*. Além disso, outros tipos de servidores foram inseridos no *Local Computer* como, por exemplo, servidores de arquivos e servidores de e-mails.

Observa-se que, durante a evolução das arquiteturas *Dew*, foi proposta uma ampla diversidade de componentes, cada um projetado para utilizar de forma mais eficiente os recursos dos dispositivos dos usuários e amplificar os benefícios proporcionados por esse paradigma. Nas seções seguintes, os componentes mais recorrentes nessas arquiteturas serão discutidos com mais detalhes, destacando suas funcionalidades básicas e as principais relações entre eles.

2.3.1.1 Dew Server

Segundo (WANG, 2015a), o *Dew Server* é um novo tipo de servidor, hospedado no computador do usuário, que, devido a essa condição, oferece seus serviços ao usuário mesmo quando não existe conexão com a Internet disponível. Resumidamente, o *Dew Server* opera duplicando partes dos recursos oferecidos pelos servidores remotos, hospedando-os localmente e disponibilizando-os aos usuários. Este componente representa uma nova camada servidora no modelo Cliente/Servidor convencional da Web, pois ele é capaz de atender os usuários de forma análoga a um servidor tradicional, mas possui uma relação de interdependência com um servidor hospedado na nuvem.

Em geral, um computador de usuário possui recursos computacionais mais limitados

do que um servidor na nuvem. Consequentemente, é desejável que um *Dew Server* seja o mais simples e leve possível para que ele possa, além de ser criado e excluído rapidamente, consumir poucos recursos do dispositivo hospedeiro. Para satisfazer tais requisitos, (RAY, 2018) sugere que cada *Dew Server* tenha uma estrutura bem definida para servir a um propósito específico.

Essencialmente, um *Dew Server* possui duas responsabilidades fundamentais. Primeiro, o *Dew Server* deve colaborar com um servidor na nuvem visando melhorar a qualidade do serviço prestado por ele. Segundo, conforme o propósito da aplicação, o *Dew Server* pode atender a um ou mais usuários simultaneamente. Uma vez que os recursos foram duplicados, os usuários podem acessá-los diretamente através do *Dew Server*, ao invés de interagir com um servidor remoto. Assim, desde que haja rede disponível para interconectar os dispositivos, o *Dew Server* poderia oferecer seus serviços a mais de um usuário. Cabe ao desenvolvedor, durante a fase de projeto, definir a quantidade de usuários que o *Dew Server* é capaz de servir simultaneamente baseado no foco da aplicação.

Ao duplicar um recurso (ou parte dele), o *Dew Server* e o servidor remoto estabelecem um vínculo de colaboração mútua durante a prestação do serviço. Isso significa que eles devem auxiliar um ao outro para juntos melhorar a qualidade do serviço prestado. A ação do *Dew Server* melhora o serviço prestado ao torná-lo mais rápido. Contudo, durante o acesso, o usuário pode alterar o recurso local e torná-lo inconsistente em relação ao remoto, o que pode ser um problema para alguns tipos de aplicações. Para evitar que esse cenário se mantenha, o *Dew Server* deve ser capaz de sincronizar/correlacionar os recursos automaticamente, assim que ele conseguir conexão com o servidor remoto. É desejável que o servidor remoto atue de maneira análoga caso o recurso remoto seja modificado.

(FISHER; YANG, 2016) apresenta um *Dew Server* diferente da versão proposta por (WANG, 2015a) por contar com o auxílio de componentes extras e por ser dedicado a um domínio específico de aplicação. Para os autores, a abordagem de (WANG, 2015a) subutiliza o poder computacional dos dispositivos de usuário, especialmente, em termos de CPU. Assim, eles sugerem a criação de novos componentes, como o *Dew Analytics Servers* (DAS) e o *Artificial Intelligence of the Dew* (AID), para explorar essa limitação enquanto melhoram a qualidade do serviço prestado pelo *Dew Server*.

Para (FISHER; YANG, 2016), o modelo de (WANG, 2015a), onde todos os serviços são disponibilizados por um único *Dew Server*, também é um problema, pois além de dificultar a organização e o gerenciamento do sistema, ele também atrapalha a customização de serviços *Dew*.

Assim, os autores defendem que os *Dew Servers* sejam projetados para domínios específicos e alocados em máquinas virtuais chamadas de *Dew Virtual Machines* (DVMs). Todas as DVMs são gerenciadas pelo *DVM Hypervisor* que também garante o isolamento entre elas. Na abordagem deles, cada máquina virtual hospeda somente um *Dew Server* que, por sua vez, administra apenas uma parte dos serviços relacionados a uma aplicação em específico.

Observe o exemplo mostrado na Figura 8. Nela, nota-se a presença de três aplicações: *Facebook.com*, *Yahoo.com* e *Youtube.com*. Cada aplicação está vinculada a uma nuvem e a uma DVM que hospeda a solução *Dew*, composta pelo *Dew Server* e componentes auxiliares. A adoção dessa abordagem abstrai a complexidade de inicializar e finalizar soluções *Dew*, uma vez que basta informar ao *DVM Hypervisor* a ação desejada. Por exemplo, se o usuário deseja encerrar o serviço *Dew Yahoo.com*, ele só precisa notificar o *DVM Hypervisor* que exclui a DVM associada e finaliza o serviço. Também é muito fácil a customização de soluções *Dew*. Por exemplo, os provedores das aplicações *Yahoo.com* e *Youtube.com* podem usar diferentes tecnologias para implementar a arquitetura padrão proposta e/ou modificá-la através da adição ou exclusão de componentes.

Os autores também sugerem a adoção de técnicas de criptografia como forma de proteger a propriedade intelectual dos provedores. Ainda com base no exemplo apresentado pela Figura 8, cada provedor desenvolve seus próprios códigos para implementar suas soluções *Dew*. Se não realizado o devido tratamento, tais códigos podem ser facilmente acessados por empresas concorrentes que podem replicá-los em seus próprios produtos. Dessa forma, eles propõe que as DVMs ofereçam suporte a criptografia de modo que os recursos sensíveis ao provedor, como códigos-fontes por exemplo, possam ser mantidos encriptados no dispositivo do usuário.

2.3.1.2 Dewsites/Dewlets

Inicialmente, a *Dew Computing* foi projetada para permitir que o usuário navegue parcialmente pela Web em modo *offline*. Para isso, o sistema *Dew* deve armazenar localmente páginas Web (popularmente conhecidas como *Websites*) e permitir que os usuários as acessem através da interface de *loopback* da máquina. Contudo, devido as restrições computacionais das máquinas dos usuários, é aconselhável que essas páginas sejam variantes simplificadas/personalizadas das originais. Para evitar dúvidas conceituais, Wang (2015a) chamou de *Dewsites* as versões manipuladas pelos *Dew Servers*.

Contudo, a *Dew Computing* é aplicável em outros contextos além de páginas Web.

Por exemplo, um servidor é capaz de oferecer serviços de processamento e armazenamento de dados aos dispositivos clientes através da rede. Analogamente, os *Dew Servers* também estão aptos a disponibilizar tais serviços, desde que eles os dupliquem. Para esses casos, Ray (2018) criou o termo *Dewlets* e o definiu como serviços *Dew* estendidos que podem ser alocados em outros equipamentos computacionais com mínimo suporte à *Dew Computing*.

2.3.1.3 Repositório de Dados

Para realizar suas atividades, um *Dew Server* está sempre associado a um repositório de dados. Tal repositório pode ser um banco de dados tradicional (*e.g.*, MySQL, MariaDB ou Redis) ou, simplesmente, um diretório dedicado no sistema de arquivos da máquina. O mais importante é que ele, assim como o *Dew Server*, também deve consumir o mínimo possível dos recursos computacionais do dispositivo de usuário. Assim, cabe ao *Dew Server* utilizá-lo apenas para hospedar aquilo que é essencial ao usuário (WANG, 2015a).

Na versão inicial da arquitetura *Cloud-Dew*, onde um único *Dew Server* é o responsável por hospedar e oferecer vários serviços simultaneamente, Wang (2015a) propõe o uso de repositórios de dados que adotem um mecanismo *plug and play* para que eles possam ser removidos e inseridos facilmente sempre que o *Dew Server* efetuar uma troca de serviço. Já de acordo com a abordagem de (FISHER; YANG, 2016), ao alocar cada *Dew Server* em uma máquina virtual específica, os autores permitem a coexistência de repositórios distintos.

Assim como os demais paradigmas pós-Computação em nuvem, a *Dew Computing* também sofre com problemas como a alta heterogeneidade dos dispositivos e a já mencionada limitação de recursos. Contudo, na *Dew* esses problemas são ainda mais críticos, uma vez que ela trabalha diretamente com variados tipos de dispositivos de usuário (como *smartphones*, *tablets* e *PCs*) e não com máquinas servidoras alocadas nas bordas da rede (como *Cloudlets*) que, embora não tenham tantos recursos computacionais como a nuvem, potencialmente, possuem uma melhor configuração de *hardware* do que os *PCs* e os *smartphones* dos usuários. Assim, é desejável que os repositórios *Dew* sejam versões modificadas dos repositórios nos demais modelos. Tais modificações variam desde o uso de tecnologias mais ágeis e econômicas para o contexto da *Dew* (que podem diferir daquelas adotadas no servidor remoto) até o desenvolvimento de versões enxutas do repositório de dados original. Por exemplo, um repositório de dados *Dew* poderia refletir apenas a parte do repositório de dados remoto que interessa ao *Dew Server* a ele associado e, dessa forma, economizar espaço em disco.

Rindos e Wang (2016) recomendam cuidado ao migrar aplicações para a *Dew Computing*, pois as diferenças entre os paradigmas podem intensificar problemas antigos ou criar novos. Por exemplo, não se pode esperar que a *Dew* proteja os dados nela armazenados com o que há de mais moderno e seguro disponível no mercado como a *Cloud* faz. Assim, usar no lado *Dew* os mesmos modelos e tecnologias utilizadas no repositório do lado *Cloud*, pode resultar no surgimento de vulnerabilidades no sistema. Tais fragilidades podem ser exploradas por usuários mal intencionados para atacar a *Dew* e, indiretamente, a *Cloud*.

2.3.1.4 Componente facilitador de acesso aos recursos Dew

Outro componente importante é aquele que implementa um mecanismo que facilita a forma como o programa cliente pode acessar os recursos locais disponibilizados pelo *Dew Server*. Wang (2015a) criou uma solução baseada no método tradicional de acesso aos recursos na Web, onde, resumidamente, cada recurso é referenciado por um identificador único e um componente auxiliar é responsável por, dado um identificador, indicar onde encontrar o recurso associado a ele. Assim, o autor definiu o conceito de LDNS (*Local Domain Name System*), um serviço DNS de escopo local que traduz URLs especiais (prefixadas por *"mmm."*) em endereços de *Dew Servers*.

Como o foco inicial da *Dew Computing* é o auto-atendimento, basicamente, o LDNS deve retornar o endereço 127.0.0.1, ou seja, o endereço da interface de *loopback* do sistema. Por exemplo, quando o usuário solicita um recurso associado à URL *"mmm.dewpage.com"* através de um programa cliente (um navegador, por exemplo), o LDNS identifica que se trata de um recurso *Dew* devido ao prefixo *"mmm"* e responde o programa cliente com o endereço 127.0.0.1.

Wang (2015a) apresenta três alternativas para realizar o mapeamento proposto pelo LDNS. A primeira delas é modificar o arquivo de zona do próprio DNS para que o sistema, ao receber consultas de endereços prefixados com *"mmm"*, responda com o endereço da interface de *loopback* (127.0.0.1). A segunda consiste em inserir o mapeamento diretamente no arquivo *hosts* no computador do usuário. Por exemplo, ao inserir a linha *"127.0.0.1 mmm.dewpage.com"* no arquivo *hosts*, o usuário informa ao sistema operacional que todas as requisições associadas a URL *"mmm.dewpage.com"* devem ser encaminhadas ao endereço 127.0.0.1. Finalmente, a terceira é alterar o próprio programa cliente para reconhecer URLs especiais e encaminhar todas as requisições diretamente a interface de *loopback*, sem a necessidade de ações extras.

Analogamente a um servidor tradicional, um *Dew Server* pode estar responsável por

gerenciar muitos recursos simultaneamente. Assim, uma vez que uma requisição chega ao *Dew Server*, é necessário em um primeiro momento identificar o recurso solicitado. Especificamente no caso de *scripts* e executáveis, Wang (2015a) recomenda adotar variáveis de ambiente para relacionar URLs e recursos. Por exemplo, o *script* responsável por interagir com o programa cliente enquanto ele navega pela página *"mmm.dewpage.com"* poderia ser referenciado e acessado diretamente através de uma variável de ambiente identificada por *"dewpage"*. Assim, é razoável assumir que esta abordagem simplifica bastante as atividades do *Dew Server*.

Fisher e Yang (2016) propõem um outro método que mapeia diretamente a URL especial e o recurso relacionado a ela. Esse mecanismo exige a presença de um componente que registre e administre todos os mapeamentos conhecidos pelo sistema *Dew*: o *Dew Resource Registry* (DRR). A Figura 8 apresenta um exemplo da DRR que pode ser implementada utilizando qualquer estrutura de dados desde que indexada pela URL, como uma tabela por exemplo. Os autores defendem que, por meio dessa técnica, é mais simples esconder recursos dos usuários, uma vez que basta omitir referências à eles da estrutura. Contudo, eles também comentam que, para funcionar corretamente, o mecanismo necessita que cada *Dew Server* especifique ao DRR todos os mapeamentos conhecidos por ele, ou seja, quais URLs são acessíveis e quais recursos estão relacionados a ela.

2.3.1.5 Outros componentes

Até aqui, foram abordados somente os componentes básicos para o funcionamento da *Dew Computing*. Contudo, principalmente nos últimos anos, tem-se observado transformações profundas na área da Ciência da Computação tal como melhorias significativas no *hardware* dos dispositivos de usuário, bem como a consolidação e o desenvolvimento de novos paradigmas e tecnologias. Em resposta, alguns trabalhos tem apresentado novos componentes que procuram explorar tais mudanças e aperfeiçoar o desempenho da *Dew Computing*. Dentre esses componentes, destacam-se:

1. ***Dew Analytics Servers* (DAS):** Proposto inicialmente por (FISHER; YANG, 2016), trata-se de um componente responsável por registrar informações sobre a interação entre o usuário e o *Dew Server*. Esses dados podem ser tratados e submetidos ao servidor remoto ou podem permanecer no próprio dispositivo do usuário para uso futuro. A ideia é que eles auxiliem na compreensão do comportamento e das preferências do usuário;
2. ***Artificial Intelligence of the Dew* (AID):** Também apresentado por (FISHER; YANG,

2016), é o componente responsável efetivamente aprender as preferências e os hábitos do usuário ao aplicar técnicas de Inteligência de Artificial nos dados coletados pelo DAS. A ideia é que as ações desse componente possam melhorar a qualidade do serviço prestado, por exemplo, ajudando o *Dew Server* a colaborar proativamente com a nuvem;

3. ***User Identity Mapping Table*:** Tabela proposta por (RAY, 2018) que correlaciona identificadores de usuário local e remoto. É uma estrutura essencial para garantir a colaboração automática entre o servidor remoto e o *Dew Server*, em especial quando o usuário usa identificadores distintos para interagir com cada um deles. Sem o devido mapeamento, *Dew Server* e servidor remoto podem não sincronizar recursos devido a falta de equivalência entre identificadores;
4. ***Dew Analyzer*:** Ray (2018) também propõe criar um componente dedicado para coordenar a execução de uma tarefa complexa iniciada a partir da interação entre o usuário e o *Dew Server*. Ao terceirizar a execução desse tipo de tarefa, a ideia é não bloquear o *Dew Server* e permitir que ele continue respondendo rapidamente as requisições mais simples do usuário;
5. ***Co-Server Database*:** Ray (2018) propõe a criação de um repositório de dados auxiliar para trabalhar em conjunto com o *Dew Analyzer*. A ideia é que esse repositório armazene temporariamente as modificações feitas pelo usuário no recurso relacionado. Assim, o repositório de dados *Dew* principal não é bloqueado e continua livre para tratar consultas de leitura emitidas pelo *Dew Server*;
6. ***Dew-IoT Engine*:** Ray e Skala (2021) sugerem a criação de um componente responsável por interconectar as pilhas de *software Dew* e *IoT*. A ideia é possibilitar a comunicação direta entre os dispositivos *IoT* e *Dew*, sem a necessidade da intermediação de servidores da *Edge*, da *Fog* ou da *Cloud*;
7. ***Dew-IoT Middleware*:** Embora o *Dew-IoT Engine* permita a interação direta entre dispositivos *Dew* e *IoT*, eles ainda precisam interagir com servidores hospedados na *Edge*, na *Fog* ou na *Cloud*. Assim Ray e Skala (2021) também propõem o desenvolvimento de um *middleware* responsável por simplificar o relacionamento entre eles.

2.4 *Offloading* Computacional

Dentre os diversos temas de pesquisa relacionados a MCC/MEC, o de *offloading* é um dos tópicos de maior destaque desde a criação do paradigma. De acordo com (DE, 2016), o

offloading permite que um dispositivo, mais limitado em termos de poder computacional e/ou de energia, submeta, através da rede, tarefas ou dados para um ambiente de execução remoto (AER) de maior capacidade de processamento, de armazenamento e/ou de energia. No contexto de MCC/MEC, comumente, o emissor é o próprio dispositivo móvel, enquanto o ambiente de execução remoto é uma máquina servidora ligada à rede elétrica e hospedada em qualquer lugar na hierarquia entre a rede local do dispositivo móvel e a nuvem. Contudo, não é incomum encontrar trabalhos que apresentam propostas alternativas a essa regra como, por exemplo, aqueles que definem outros dispositivos móveis como AER, também conhecido como *offloading Device to Device* ou D2D.

O funcionamento do *offloading* se assemelha bastante ao modelo Cliente/Servidor tradicional, onde uma máquina (cliente) requisita a computação de uma tarefa ou o armazenamento de dados a uma outra máquina (servidor) que deve respondê-la com uma mensagem indicando o resultado da requisição. Porém, ao contrário do modelo Cliente/Servidor, no *offloading* o cliente é capaz de efetuar tais ações sozinho, caso não seja vantajoso para ele executá-las remotamente. Dessa maneira, o cliente, comumente, ganha uma responsabilidade extra de analisar o contexto atual e decidir quando realizar o *offloading* ou não.

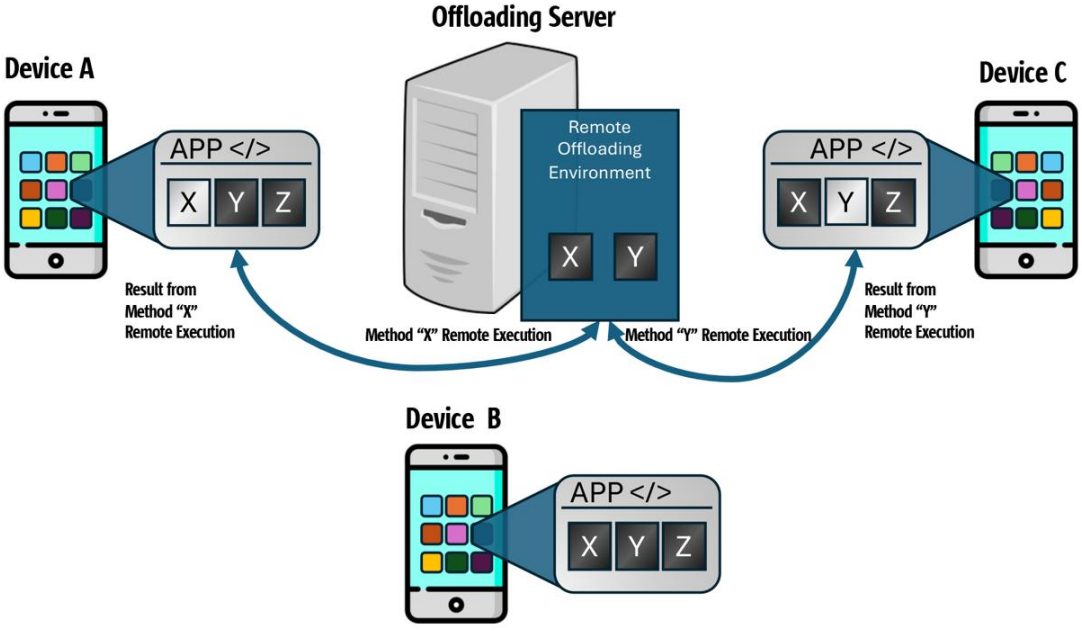


Figura 10 – Cenário exemplo de *offloading* computacional

A Figura 10 apresenta um cenário tradicional de *offloading*. Nele é possível visualizar, além de uma máquina robusta atuando como servidor, três dispositivos móveis (A, B e C) atuando como clientes. Cada dispositivo móvel executa uma aplicação composta por três partes (X, Y

e Z) que, nesse exemplo, serão tratadas como métodos. Durante a computação dos métodos, eventualmente, os dispositivos podem submeter alguns deles para serem processados pelo servidor central através do *offloading* computacional. No exemplo, o dispositivo A requisita ao servidor remoto que ele processe o método X (via *offloading*). Após finalizada a computação do método X, o servidor retorna o resultado para o dispositivo A. Analogamente, o dispositivo C realiza o mesmo procedimento, porém com o método Y. Contudo, observa-se que o dispositivo B, em nenhum momento, realizou qualquer tipo de *offloading*, embora esteja executando a mesma aplicação. Nesse caso, todos os métodos foram executados localmente pelo próprio dispositivo B. Isso é possível de acontecer quando o dispositivo, baseado em critérios como estado da conexão, largura de banda disponível e nível de energia do dispositivo, determina que o *offloading* do método não é vantajoso para ele naquele momento. Tal processo de decisão será discutido com mais detalhes nas próximas seções. Por exemplo, com base na Figura 10, o dispositivo B pode não ter enviado nenhuma tarefa para o servidor remoto devido a uma sobrecarga na rede (e uma consequente redução na largura de banda disponível) ocasionada pelo *offloading* dos outros dois dispositivos (A e C).

Segundo (FERNANDO *et al.*, 2013), há dois tipos principais de *offloading*: de dados e de processamento. O objetivo principal do *offloading* de dados é usar o dispositivo remoto (com maior capacidade de armazenamento) como um repositório de dados. Assim, o dispositivo mais limitado envia dados para serem persistidos em outros dispositivos e, quando necessário, os resgata através da rede. Já o *offloading* de processamento (ou computacional) ocorre quando o dispositivo mais restritivo envia uma tarefa para ser computada por outro dispositivo menos limitado computacionalmente e/ou energeticamente. Vale destacar que a proposta desse trabalho é voltada para o *offloading* computacional, por isso o restante dessa seção focará somente nesse tipo de *offloading*.

Além dos tipos, existem outras questões importantes que devem ser avaliadas durante a realização do *offloading*. A Figura 11 apresenta uma taxonomia sugerida por (REGO, 2016) que resume as principais questões a serem analisadas, bem como as opções para respondê-las. Por conta da abrangência dessa taxonomia, ela será adotada como referência para um maior detalhamento sobre o funcionamento do *offloading* computacional nas próximas seções.

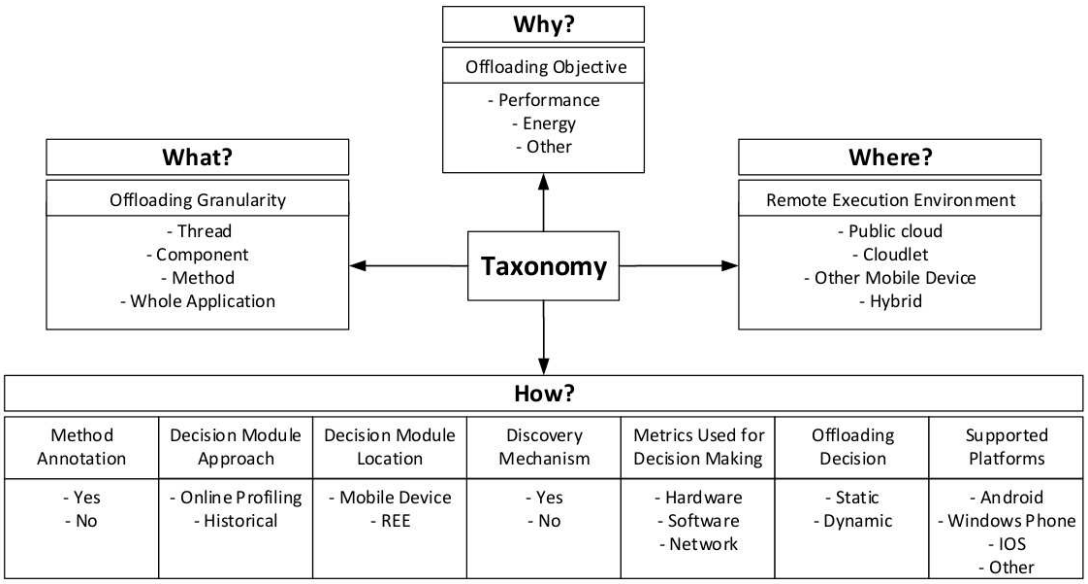


Figura 11 – Taxonomia para soluções de *offloading* proposta por (REGO, 2016)

2.4.1 Por que realizar o offloading computacional?

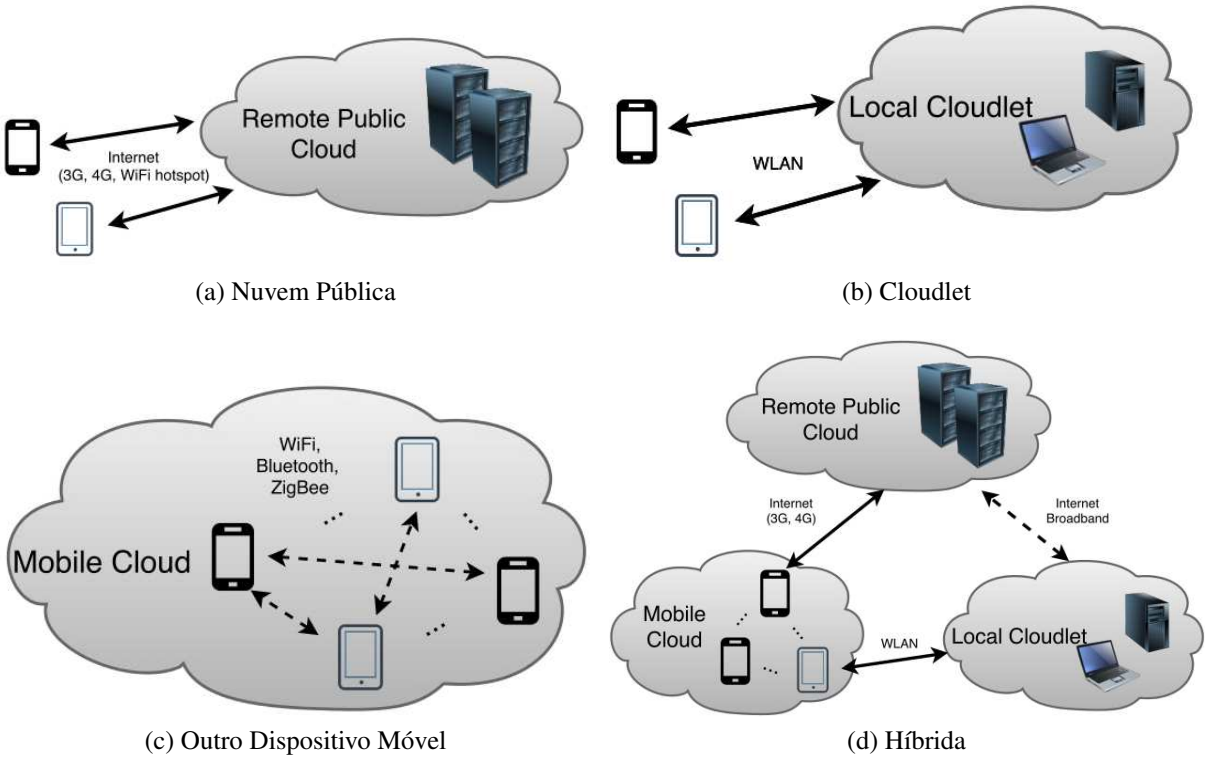
Em geral, o *offloading* objetiva aprimorar o desempenho dos programas, otimizar o consumo de recursos ou ambos, conforme discutido em (KUMAR *et al.*, 2013). No primeiro caso, o dispositivo realiza o *offloading* visando reduzir o tempo de processamento de uma determinada tarefa, enquanto no segundo caso, o objetivo é, ao realizá-lo, fazer com que o dispositivo consuma menos energia e/ou recursos computacionais (como armazenamento, por exemplo). O terceiro, obviamente, busca atingir os dois objetivos anteriores simultaneamente, o que nem sempre é possível.

Rego (2016) ainda cita alguns motivos menos usuais como melhorar a colaboração, reduzir os custos monetários e possibilitar a computação de aplicações completas ou de partes delas que não seria possível devido a carência de recursos no dispositivo origem. Contudo, vale ressaltar que a grande a maioria dos trabalhos relacionados ao *offloading* computacional focam em tentar atingir um dos três objetivos apresentados no parágrafo anterior.

2.4.2 Onde processar o offloading computacional?

Como destacado anteriormente, o *offloading* computacional se baseia no envio de tarefas para processamento remoto visando atingir algum dos objetivos destacados previamente. Essa questão trata do local onde a computação remota será realizada. A Figura 12 apresenta as opções citadas em (REGO, 2016): 1) Nuvem Pública, 2) *Cloudlet*, 3) Outro Dispositivo Móvel e 4) Híbrida, que mescla duas ou mais das abordagens anteriores. Os parágrafos a seguir tratarão

com mais detalhes as três primeiras opções de destino.



(a) Nuvem Pública (b) Cloudlet

(c) Outro Dispositivo Móvel (d) Híbrida

Figura 12 – Opções de onde processar o *offloading* computacional (REGO, 2016)

A Nuvem Pública é um tipo específico de nuvem caracterizada pelo fornecimento, pago ou gratuito, de serviços à terceiros. Em outras palavras, nesse tipo de nuvem, a empresa provedora monta, mantém e disponibiliza uma infraestrutura ao público geral através da Internet. No contexto do *offloading* computacional, os servidores que compõe essa infraestrutura são utilizados para processar tarefas submetidas pelos dispositivos clientes através de redes celulares (3G, 4G e 5G) ou, mais comumente, redes de computadores (WiFi).

O conceito de *Cloudlet* foi concebido por (SATYANARAYANAN *et al.*, 2009) e definido como um equipamento computacional tradicional, alocado em áreas públicas (como cafés e restaurantes) e próximas aos dispositivos do usuário, cujo o objetivo principal é mitigar a alta latência observada na comunicação com a Nuvem Pública. No contexto do *offloading* computacional, eles são os responsáveis por atender parte das requisições realizadas pelos demais dispositivos que compartilham com ele a mesma rede sem-fio local (WLAN).

Uma terceira opção é utilizar um outro dispositivo móvel, com melhores recursos computacionais e energéticos, para atender o *offloading* computacional de um dispositivo móvel cliente. Neste contexto, é possível realizar o *offloading* entre dispositivos móveis não apenas por

meio de redes sem fio estruturadas, mas também por meio de redes não estruturadas construídas por meio de tecnologias *peer-to-peer* (P2P), como Bluetooth e WiFi Direct. Rego (2016) afirma que o objetivo principal dessa abordagem é permitir que os dispositivos móveis se agrupem e compartilhem seus recursos visando tornar mais célere o processamento de tarefas energia e reduzir o consumo de energia.

2.4.3 O que submeter durante o offloading computacional?

Essa é uma questão complexa com um número significativo de opções para atendê-la. Primeiro, deve-se considerar a granularidade daquilo que se deseja enviar no *offloading*. Nesse aspecto, o dispositivo cliente pode submeter desde métodos até uma aplicação completa para execução remota (LIU *et al.*, 2015). Com base naquilo que é submetido, o *offloading* pode ser classificado como total (a aplicação completa é enviada) ou parcial (parte da aplicação é enviada). Gu *et al.* (2018) apresentam uma lista extensa de partes de uma aplicação que podem ser transmitidas em *offloading*, além dos métodos, também podem ser enviados módulos, objetos, *threads*, dentre outros.

Em seguida, deve-se avaliar se a aplicação completa ou a parte relevante para o *offloading* já está disponível no lado servidor. Caso negativo, o recurso precisará ser enviado para o servidor antes ou junto dos dados de entrada necessários para a computação. Por exemplo, no *framework* CAOS desenvolvido por (GOMES *et al.*, 2017), inicialmente, o servidor desconhece qualquer código de qualquer aplicação cliente. Assim, cabe ao aplicativo Android, informá-lo sempre que necessário, antes de realizar qualquer requisição de *offloading*. Caso o servidor já possua o código necessário para computar aquilo que foi submetido, somente os parâmetros de entrada são necessários.

É importante ressaltar que nem tudo é passível de ser enviado para processamento remoto, mesmo quando a rede é favorável para a sua realização. Nesse caso, tarefas que envolvem recursos disponíveis apenas no dispositivo móvel (por exemplo, consultas a componentes como câmeras e sensores) não podem ser computadas remotamente, apenas localmente.

2.4.4 Quando realizar o offloading computacional?

Como já destacado anteriormente, nem sempre o *offloading* é vantajoso para o dispositivo cliente que inicia a operação. As vezes, o processamento remoto de uma tarefa pode ser mais custoso que a computação local em termos de tempo de processamento, de consumo de

recursos (energéticos e/ou computacionais) e, até mesmo, financeiros. Contudo, na maioria dos casos, os pesquisadores usam o consumo energético e, principalmente, o tempo de processamento como critérios base para decidir quando o *offloading* é vantajoso e, por isso, deve ser realizado. Essa seção abordará essa importante questão no processo de *offloading* adotando o tempo de processamento como critério chave para a decisão.

O *offloading* é realizado através de conexões de rede estabelecidas entre os equipamentos envolvidos. Assim, ele pode ser realizado por meio de redes celulares (como 5G), redes de computadores (como *WiFi*), ou conexões diretas entre os dispositivos (como *Bluetooth*). Por se tratar de um meio compartilhado (sem falar de possíveis interferências externas), as redes podem apresentar diferentes níveis de congestionamento e, por consequência, tempos variáveis para transmitir a mesma quantidade de dados. Esta alta variabilidade afeta diretamente o desempenho do *offloading* e faz da condição atual da rede um dos fatores mais relevantes para decidir se *offloading* deve ou não ocorrer (KUMAR *et al.*, 2013). Caso o *offloading* computacional não aconteça, o processamento é feito no próprio dispositivo.

Em seu modelo analítico, Kumar *et al.* (2013) propõem uma fórmula que compara o tempo de processar a tarefa localmente com o tempo necessário para computar a mesma tarefa remotamente (Fórmula 2.1). A quantidade de computação necessária para processar a tarefa é representada por W e medida em Milhões de Instruções (MI), enquanto o poder computacional dos dispositivos locais e remotos são representados pelas variáveis P_m e P_c, respectivamente, e medidos em Milhões de Instruções Por Segundo (MIPS). Na Fórmula 2.1, o tempo necessário para processar a tarefa localmente e remotamente é obtido através nos seguintes operadores $\frac{W}{P_m}$ e $\frac{W}{P_c}$, respectivamente.

$$\frac{W}{P_m} > \frac{D_u}{T_u} + \frac{W}{P_c} + \frac{D_d}{T_d} \tag{2.1}$$

Contudo, no caso do processamento via *offloading*, também se faz necessário considerar os tempos necessários para enviar a requisição ao servidor remoto e receber a resposta com o resultado. Para obter tais valores, o autor se baseia na quantidade de *bytes* enviados (D_u) e recebidos (D_d) pelo dispositivo móvel cliente durante o *offloading*, assim como as taxas de *upload* (T_u) e *download* (T_d) da rede medidas em *bytes*/segundo. O tempo dispendido para enviar a requisição é obtido da vazão de *upload* ($\frac{D_u}{T_u}$), enquanto o tempo para receber a resposta é adquirido da vazão de *download* ($\frac{D_d}{T_d}$).

Analisando atentamente a Fórmula 2.1 é possível perceber que, em geral, o *offloading* computacional só é vantajoso para o dispositivo móvel quando o valor de W é muito alto (ou seja, trata-se de uma tarefa que requer muito esforço computacional para ser processada) e o envio da requisição e recebimento da resposta do *offloading* são rápidos (ou seja, se $\frac{D_u}{T_u} + \frac{D_d}{T_d}$ é baixo). A transferência de dados através da rede pode ser rápida por dois motivos principais: ou se poucos dados são transmitidos (D_u e D_d baixos), ou se a rede apresenta uma alta vazão (T_u e T_d altos). Embora seja bastante complicado obter com precisão os valores dessas variáveis, essas condições são mais facilmente atendidas quando se infere, por meio do uso de modelos matemáticos e amostras, que a rede apresenta baixos níveis de congestionamento e interferência externa.

2.4.5 Como realizar o offloading computacional?

Como pode ser observado na Figura 11, tal questão abrange diversos aspectos, desde a maneira como o dispositivo móvel cliente descobre máquinas servidoras aptas a atender às suas requisições de *offloading* até a maneira como o desenvolvedor indica quais partes da aplicação são passíveis de *offloading*. Devido a isso, é difícil encontrar soluções que adotem as mesmas estratégias para todas as categorias relacionadas a essa pergunta. Essa seção, cobrirá os mecanismos mais relevantes para a realização do *offloading* computacional.

Antes de realizar qualquer tipo de *offloading*, normalmente, o dispositivo móvel cliente precisa encontrar quais servidores estão aptos a receber o *offloading* dele. Assim, esse procedimento, comumente, é implementado através de mecanismos de descoberta. Em geral, o mecanismo de descoberta pode ser automatizado ou não. No primeiro caso, o dispositivo cliente consegue, através do envio de mensagens *broadcast* ou *multicast*, achar as máquinas servidoras desejadas de forma independente. No segundo caso, o dispositivo cliente conhece antecipadamente o endereço, IP ou URL (*Uniform Resource Locator*), do servidor alvo. Dessa maneira, ele depende dessa informação para encontrar o servidor alvo.

Quando o *offloading* computacional é do tipo parcial, ou seja, quando apenas algumas partes da aplicação são passíveis de submissão para processamento remoto, o desenvolvedor necessita, de alguma forma, indicar quais partes podem ou não ser transmitidas no código-fonte. A maneira mais comum é utilizar anotações (por exemplo, `@Offloadable` no *framework* CAOS (GOMES *et al.*, 2017)). Nesse caso, antes de executar a parte, o *software* a identifica como candidata ao *offloading* e decide se ele deve ou não ser realizado. A outra maneira é utilizar um

software que, dinamicamente, com base na elaboração de perfis de processamento e consumo de recursos, decide quais partes podem ser submetidas em *offloading*.

Como destacado anteriormente, nem sempre a realização do *offloading* é vantajoso para o dispositivo móvel cliente. Algumas vezes, a ação de enviar uma tarefa para processamento pode ser mais custosa do que a computação da mesma tarefa localmente. Por isso, antes de cada submissão, o é necessário decidir se é um bom momento para realizar o *offloading* ou não. Nesse caso, a decisão pode ser estática ou dinâmica e está diretamente relacionada ao momento em que ela é tomada. Enquanto na estática, antes da execução, o próprio programador indica quais partes podem ser enviadas e para onde enviá-las, na dinâmica o próprio sistema decide em tempo de execução com base no contexto do ambiente.

Existem outros aspectos relacionados a tomada de decisão. Um deles está associado as métricas que a norteiam. Comumente, as métricas são distribuídas em três categorias que incluem *Hardware* (por exemplo, consumo de CPU e Memória), *Software* (por exemplo, quantidade de dados a serem transmitidos e complexidade da tarefa e Rede (por exemplo, latência e força do sinal WiFi). As métricas de *Software* e *Hardware*, inclusive, podem contemplar características do dispositivo móvel cliente e/ou do servidor alvo. Um outro aspecto é sobre como os valores dessa métrica são utilizados para tomar a decisão. Uma delas cria um perfil utilizando dados coletados do ambiente em tempo de execução, enquanto a outra usa dados históricos (não obrigatoriamente obtidos em tempo de execução). Por fim, o último ponto é sobre quem toma a decisão. O mais comum é o dispositivo móvel decidir sozinho. Contudo, existem estudos onde a computação das partes mais complexas dessa decisão são delegadas ao servidor remoto, cabendo ao dispositivo móvel somente seguir aquilo que foi definido por ele.

2.4.6 Offloading Multi-Linguagem

Tradicionalmente, pesquisas na área de *offloading* adotam soluções que foram implementadas utilizando uma única linguagem de programação. Tal abordagem se mostra bastante vantajosa, especialmente, durante as fases de desenvolvimento e de manutenção do *software*, uma vez que a modelagem e a programação do aplicativo, em geral, se tornam mais complexas quando são usadas duas ou mais linguagens simultaneamente no mesmo projeto. Embora, como destacado por (KOCHHAR *et al.*, 2016; LI *et al.*, 2021), o reuso de código e a sinergia entre determinadas linguagens podem até agilizar o desenvolvimento e melhorar a produtividade.

Contudo, uma das maiores vantagens em se adotar várias linguagens de programação

aparece durante a execução do *software*. Seja localmente (NANZ; FURIA, 2015; GEORGIOU *et al.*, 2018; PEREIRA *et al.*, 2021), seja em um cenário de *offloading* (GEORGIOU; SPINELLIS, 2019; ARAÚJO *et al.*, 2020), estudos tem mostrado que cada linguagem possui um desempenho computacional e um padrão de consumo de recursos energéticos próprio, que podem variar de acordo com o tipo da tarefa e/ou a plataforma onde ela será computada. Por exemplo, os resultados de (GEORGIOU *et al.*, 2018) mostraram que, ao executar algoritmos distintos (*InsertionSort* e *SelectionSort*) para uma mesma tarefa (ordenar de dados) em uma mesma plataforma (Embarcada), diferentes linguagens podem obter os menores valores de EDP (JavaScript e Golang, respectivamente). Analogamente, os resultados também mostraram que para o mesmo algoritmo (*SelectionSort*) é possível obter três diferentes linguagens como os menores valores de EDP para cada plataforma avaliada: Golang (Embarcada), JavaScript (*Desktop*) e C# (*Web*).

Motivado por esses achados, Matos *et al.* (2021a) definiram o conceito de *offloading* multi-linguagem. Esse mecanismo, assim como o *offloading* tradicional, permite que dispositivos com restrições computacionais ou energéticas submetam parte de seu processamento (no caso do *offloading* multi-linguagem, métodos) à máquinas servidoras remotas com melhores recursos. Contudo, o que diferencia o *offloading* multi-linguagem das técnicas tradicionais é a possibilidade de que o cliente converse com um processo remoto desenvolvido em outra linguagem de programação e, dessa maneira, aproveite as vantagens oferecidas pela linguagem servidora. Por exemplo, em (MATOS *et al.*, 2021a), em uma das aplicações utilizadas nos experimentos, o processo cliente (desenvolvido em Android) enviou matrizes para serem multiplicadas por processos servidores desenvolvidos em C++, Go, Java ou Python.

Outra particularidade importante que distinguem as abordagens de *offloading* tradicional e multi-linguagem é quanto ao mecanismo de comunicação inter-processos (IPC). Por conta de problemas como a interoperabilidade *cross-language* (CHISNALL, 2013), que dificulta a comunicação direta entre processos desenvolvidos com linguagens de programação distintas, o *offloading* multi-linguagem exige o uso de um mecanismo que padronize as mensagens trocadas entre os processos envolvidos e permita a efetiva interação entre eles. Por exemplo, Matos *et al.* (2021a) usaram o *framework* gRPC[4] como mecanismo para intermediar a comunicação entre os processos envolvidos, ao passo que Matos *et al.* (2022) adotaram o Apache Thrift[5].

Em termos arquiteturais, o *offloading* multi-linguagem é também baseado no modelo Cliente/Servidor, onde o cliente é o dispositivo móvel e o servidor é um equipamento com

[4] https://grpc.io/
[5] https://thrift.apache.org/

maior poder computacional e/ou capacidade energética. Assim, servidores podem ser desde outros dispositivos móveis na própria rede local do usuário até poderosas máquinas servidoras hospedadas na nuvem. De maneira análoga a abordagem tradicional, os aparelhos clientes que utilizam o *offloading* multi-linguagem também devem observar questões como quando realizá-lo e onde executá-lo, por exemplo.

Embora os estudos realizados (a maioria deles conduzidos pelo próprio autor dessa proposta) tenham apontado que a adoção de outras linguagens de programação durante o *offloading* computacional proporcionam melhorias consideráveis na computação da tarefa alvo, a abordagem multi-linguagem possui um problema intrínseco e relevante aos desenvolvedores de aplicações móveis: a necessidade de programar o mesmo algoritmo em, no mínimo, duas linguagens distintas (a do lado cliente e a do lado servidor). Nesse aspecto, a abordagem tradicional é a mais vantajosa, pois exige o desenvolvimento da solução em uma única linguagem de programação e delegando ao sistema a responsabilidade de implantá-la no lado servidor sem a ação direta do programador. Contudo, vale lembrar que tal problema pode ser mitigado com a adoção de técnicas da Engenharia Orientada a Modelos e/ou ferramentas como a *Haxe* e a *Rhapsody Developer*.

2.5 Considerações Finais

Esse capítulo apresentou conceitos relevantes ao desenvolvimento de aplicações móveis, bem como ao desempenho dos dispositivos móveis ao processar tarefas. Em seguida, ele realizou uma rápida explanação sobre o paradigma de *Mobile Cloud Computing*, onde foi apresentada a motivação do seu surgimento, suas vantagens e desvantagens e a relação entre ele e os novos paradigmas que surgiram em resposta as suas limitações, como *Cloudlets*, *Fog Computing* e *Mobile Edge Computing* e, especialmente, a *Dew Computing*. Finalmente, o capítulo definiu tópicos importantes relacionados a um mecanismo bastante convencional da área de MCC: a técnica de *offloading* computacional, tanto em sua abordagem tradicional, como multi-linguagem.

A Seção 2.1 indicou que os aplicativos móveis enfrentam um claro problema de eficiência ao computar tarefas, mesmo quando se tratam de aplicativos Nativos, que são conhecidos por apresentarem melhor desempenho. A Seção também destaca o domínio da plataforma Android no mercado atual dos *smartphones*. Devido a essas observações, esta tese de doutorado procura melhorar a eficiência da computação de tarefas produzidas por aplicações Nativas em

smartphones Android. A ideia é que, se a solução proposta nessa tese melhorar o desempenho da aplicação Nativa, potencialmente ela também melhorará o desempenho dos demais tipos. Ao optar pela plataforma Android também espera-se simplificar os testes a serem realizados.

A Seção 2.4 discutiu as principais características relacionadas a técnica de *offloading* e também expôs o conceito e os benefícios do *offloading* multi-linguagem, uma variação da versão convencional da técnica que possibilita a realização do *offloading* entre processos desenvolvidos com diferentes linguagens de programação. Esta tese de doutorado visa explorar as vantagens que a interação multi-linguagem pode oferecer para superar o desafio da eficiência na execução de tarefas em dispositivos móveis. Porém, a Seção indica que, mesmo na versão multi-linguagem, o desempenho do *offloading* é fortemente afetado pela qualidade da rede. Sem uma conexão estável e de baixa latência, o *offloading* torna-se inadequado para o problema da eficiência.

Além disso, foi apresentado na Seção 2.2 que a *Dew Computing* criou uma estratégia muito eficiente para reduzir a latência da rede, que consiste na duplicação de servidores. Esta tese adota diretamente essa estratégia, permitindo a duplicação de processos servidores que antes estavam hospedados em computadores da borda no próprio dispositivo móvel, habilitando-o a atender as requisições dos seus próprios aplicativos. A ideia é que a interação entre processos servidores duplicados e aplicativos clientes, potencialmente desenvolvidos com linguagens de programação distintas, aconteçam dentro do próprio dispositivo móvel, minimizando ao máximo o número de ações na rede externa e reduzindo a dependência sobre ela.

Por conta da importância do paradigma *Dew Computing* para a concepção desse trabalho, a Seção 2.3 mergulhou e examinou minuciosamente as principais arquiteturas e componentes estabelecidos pela literatura nos últimos anos. Dentre as diversas arquiteturas apresentadas, esta tese se baseia no modelo proposto por (WANG, 2016), constituída apenas de duas camadas. No entanto, algumas alterações foram necessárias para adaptá-la ao contexto da tese, como, por exemplo, modificar a camada superior de *Cloud* para a *Edge*, visando tornar o procedimento de duplicação mais ágil. Quanto aos componentes, foram usados apenas os componentes básicos *Dew Server*, Repositório de Dados e *Dewlets*, mas renomeados para melhor se adequar ao contexto da tese. Não foram utilizados componentes auxiliares ou facilitadores, pois a ideia foi construir uma versão simples, mas funcional da arquitetura e que possa ser melhorada enquanto a pesquisa evolui. A arquitetura proposta será discutida com maior profundidade no Capítulo 4

O próximo capítulo apresentará os trabalhos relacionados que, ou avaliaram o desempenho computacional e, em alguns deles, energético de diferentes linguagens de programação

durante o processamento de tarefas nas principais plataformas disponíveis hoje no mercado: embarcado, web e/ou *desktop*. Tais trabalhos abrangem o processamento de tarefas tanto no escopo local (quando o próprio dispositivo computa sua própria tarefa), como remoto (quando o dispositivo submete a tarefa para ser computada por um outro). Também serão apresentados alguns trabalhos que, através da *Dew Computing*, criaram recursos para atacar problemas ou potencializar soluções já existentes em diversas áreas do conhecimento.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos correlatos a este trabalho de doutorado. Antes de iniciá-lo, é relevante lembrar que o principal objetivo desta tese de doutorado é modelar e desenvolver uma solução baseada em *Dew Computing* que visa, diretamente, melhorar o desempenho de aplicações móveis (explorando o poder de determinadas linguagens de programação ao computar tarefas específicas) e, indiretamente, atacar problemas relacionados ao *offloading* computacional.

Assim, com base nesse objetivo, os artigos foram separados em três grupos: Aqueles que comparam o desempenho dos aplicativos ao processar tarefas localmente (Seção 3.1) ou durante a realização do *offloading* computacional (Seção 3.2) usando diversas linguagens de programação e aqueles que modelaram soluções *Dew* para atacar problemas das mais diversas áreas do conhecimento (Seção 3.3).

3.1 Desempenho de Aplicações *Multi-Language*

Alguns trabalhos da literatura tem avaliado o desempenho das linguagens de programação ao processar *benchmarks* ou tarefas relacionadas a aplicações reais ou problemas clássicos da computação, como ordenar vetores ou manipular *strings*. A maioria desses estudos é voltada para plataformas Web e/ou *Desktop*, enquanto poucos foram realizados em ambientes embarcados. Essa seção apresenta um breve resumo sobre tais pesquisas, com um enfoque especial nos principais achados dos experimentos conduzidos por delas. Dessa forma, espera-se identificar oportunidades a serem exploradas, questões de pesquisa em aberto, bem como desafios a serem evitados ao adotar linguagens de programação específicas para computar tarefas iguais ou semelhantes as avaliadas nesses trabalhos.

NANZ; FURIA (2015) compararam o desempenho de oito linguagens de programação (C, Go, Java, C#, F#, Haskell, Python e Ruby) ao processar tarefas em uma máquina tradicional de usuário. Para isso, os autores adotaram métricas como tempo de processamento, tamanho do executável gerado, consumo de memória RAM, entre outras. As linguagens foram escolhidas com base no paradigma e na posição delas no ranking das páginas *Rosetta Code*[1] e *TIOBE*[2], usualmente adotados como referência para comparar o nível de popularidade de diferentes linguagens de programação. Já as tarefas foram retiradas exclusivamente da página *Rosetta*

[1] http://www.rosettacode.org/wiki/Rosetta_Code
[2] https://www.tiobe.com/tiobe-index/

Code com base em uma classificação formulada pelos próprios autores. Os resultados indicaram que, se por um lado, as linguagens compiladas (C e Go) executaram tarefas mais rapidamente e consumiram menos memória RAM, por outro, elas produziram os maiores executáveis e códigos-fonte menos concisos. Linguagens de *script* (Python e Ruby) e funcionais (F# e Haskell) apresentaram um comportamento inverso, ou seja, obtiveram um desempenho ruim ao computar tarefas, contudo geraram os menores códigos-fonte e executáveis. As linguagens orientadas a objeto (Java e C#), em geral, obtiveram um desempenho intermediário em todas as métricas avaliadas.

Analogamente, COUTO *et al.* (2017) avaliaram o desempenho de dez linguagens de programação (C, C#, Fortran, Go, Java, JRuby, Lua, OCaml, Perl e Racket) em termos de tempo de processamento e consumo de energia ao processar tarefas em um computador *Desktop*. Tanto as linguagens, como as tarefas foram escolhidas com base nos códigos-fonte disponibilizados na página *The Computer Language Benchmark Game*[3], um projeto que compara o desempenho de linguagens de programação distintas ao executar um conjunto de *benchmarks* sugeridos pelo projeto, mas implementados pela comunidade. Foram selecionadas tarefas dos mais variados tipos como aquelas com o uso intensivo de CPU e de memória RAM. Os resultados mostraram, em primeiro lugar, que nem sempre a linguagem mais rápida é aquela que consome menos energia. Em segundo lugar, eles também mostraram que além de serem as mais lentas, as linguagens interpretadas (Lua, JRuby e Perl) também são aquelas que mais consomem energia. Finalmente, embora tenham obtido bons resultados no ranking geral, linguagens como Java e C# são significativamente mais lentas e consomem mais energia do que as melhores linguagens em cada tarefa, em especial, quando ela faz um uso intensivo de CPU.

PEREIRA *et al.* (2017) aprofundaram o estudo realizado em (COUTO *et al.*, 2017) ao considerar 17 novas linguagens (como Rust, Haskell, JavaScript e Swift) e uma nova métrica durante a análise de desempenho: o consumo de pico de memória RAM. Avaliar essa métrica é relevante, pois indica quais linguagens consomem mais memória RAM, o que pode resultar em escassez desse recurso e afetar o desempenho geral do dispositivo. Foram mantidas as mesmas tarefas e a mesma máquina *Desktop* para a realização dos experimentos. Os resultados reforçam a conclusão de (COUTO *et al.*, 2017), que não existe uma relação direta entre consumo de energia e tempo de processamento, bem como que linguagens compiladas processam tarefas mais rapidamente (e também consomem menos energia), ao passo que as interpretadas apresentam

[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

desempenhos ruins nessas métricas. Mais do que isso, os resultados expandem essas conclusões ao apontar um comportamento semelhante ao comparar as métricas de consumo de memória e energia, embora exista uma correlação mínima entre elas. Finalmente, os autores montaram um ranking com as linguagens de programação mais indicadas para atender dois ou três critérios simultaneamente. Aqui, destaca-se que linguagens como C/C++, Go e Rust são mais indicadas que Java e C# quando se pretende poupar energia, memória e tempo de processamento.

OLIVEIRA *et al.* (2017) compararam aplicações Nativas (somente código Java), Web (HTML5, CSS3 e JavaScript) e Nativas com NDK (Java e C/C++) em termos de consumo energético e tempo de processamento de tarefas. Para isso, os autores adotaram programas de *benchmark* disponíveis nas páginas *The Computer Language Benchmark Game* e *Rosetta Code*, além de aplicações de código-aberto reais disponíveis no F-Droid[4], uma plataforma de distribuição de software livre e de código aberto para dispositivos Android. No primeiro caso, foram selecionados programas desenvolvidos com linguagens relevantes para o estudo: Java, JavaScript ou C/C++. Já no segundo caso, as aplicações, originalmente desenvolvidas apenas em Java, foram adaptadas para o formato híbrido. Em geral, as modificações consistiram em substituir partes Java de computação intensiva de cada aplicação por códigos equivalentes desenvolvidos em C/C++ ou JavaScript. Os resultados obtidos indicaram que a adoção de linguagens não-Java pode economizar energia dos dispositivos móveis. Ao executar os programas de *benchmark*, observou-se que aqueles desenvolvidos em JavaScript apresentaram um menor consumo de energia, ao passo que os programas JavaScript e C/C++ obtiveram os menores tempos de processamento da tarefa. Os testes com as aplicações reais indicaram que a abordagem híbrida é mais vantajosa do que a nativa.

GEORGIOU *et al.* (2018) avaliaram o desempenho de 14 linguagens de programação (por exemplo, C, VB.NET, PHP e R) ao processar tarefas em plataformas embarcada, web e *desktop*. As linguagens foram escolhidas com base no nível de popularidade delas (atual ou potencial) na TIOBE, enquanto as tarefas foram selecionadas da página *Rosetta Code*. As tarefas compreendem ações típicas utilizadas em algoritmos computacionais como a realização de operações aritméticas, compressão de dados, manipulação de arquivos, ordenação de dados, entre outras. Como métrica, os autores adotaram a *Energy Delay Product* (EDP) que resume as métricas de consumo de energia e tempo de processamento. Em geral, os resultados foram bastante favoráveis as linguagens compiladas, independente da plataforma avaliada. C, C++ e

[4] https://www.f-droid.org/

Go apresentaram os menores valores de EDP. Java, por sua vez, obteve um dos piores resultados também em todas as plataformas. Avaliando os resultados de cada tarefa separadamente, observa-se alguns casos excepcionais como, por exemplo, o JavaScript que obteve os melhores resultados ao computar expressão regular em todas as plataformas.

PEREIRA *et al.* (2021) estenderam os trabalhos de (COUTO *et al.*, 2017) e (PEREIRA *et al.*, 2017), ao considerar novas tarefas em sua análise (obtidas do *Rosetta Code*) e ao avaliar o nível de correlação entre as métricas adotadas nos estudos (tempo de processamento, consumo de energia e de memória) através da realização de testes estatísticos. Foram adicionadas nove tarefas ao total, todas elas associadas a problemas clássicos da computação como a ordenação de vetores, o problema das N rainhas e o cálculo da série de Fibonacci. Os testes foram executados na mesma máquina dos demais trabalhos. Em geral, as linguagens mantiveram seus desempenhos ao computar as novas tarefas, exceto Java, onde observou-se um perda de desempenho significativa entre os resultados antigos e novos (respectivamente, a quinta e a décima linguagem que mais economizou energia). Os autores acreditam que o uso excessivo de coleções Java (como *List* e *Set*) nos novos algoritmos foi o responsável por essa perda de desempenho. Finalmente, os testes estatísticos indicaram correlação (1) muito forte entre tempo de processamento e consumo de energia, (2) de moderada para forte entre consumo de memória e energia e (3) de fraca para moderada entre tempo de processamento e consumo de memória.

KOEDIJK; OPRESCU (2022) compararam oito linguagens de programação (C, C++, JavaScript, Python, Ruby, PHP, Java e C#) ao processar sete tarefas (todas oriundas da página *The Computer Language Benchmarks Game*) em GPUs (*Graphics Processing Units*). Os autores optaram por diversificar as tarefas escolhidas, com uso intensivo de CPU (como a *Spectral Norm*), de memória (como a *Fasta*) e disco (*Reverse Complement*). Adicionalmente, esse trabalho, além confrontar linguagens, também verificou o desempenho de diferentes algoritmos, desenvolvidos com a mesma linguagem, para computar uma dada tarefa. Ao comparar as linguagens, assim como os demais trabalhos, os autores constataram que as linguagens compiladas (C e C++) foram as que mais economizaram energia e ressaltaram os bons resultados apresentados pela JavaScript quando comparada as linguagens interpretadas e parcialmente interpretadas. Ao comparar os programas, os autores obtiveram resultados (com força estatística) que laços *while* consomem mais energia que laços *for* e que, portanto, devem ser evitados. Assim, observa-se que as variações de desempenho podem acontecer tanto entre linguagens, como entre programas em uma mesma linguagem.

BUGDEN; ALAHMAR (2022) avaliaram o desempenho e a segurança de seis linguagens de programação (C, C++, Go, Java, Python e Rust) ao processar dois algoritmos: ordenação de inteiros com o *BubbleSort* e estimativa do valor do π pelo método de Monte Carlo. No contexto específico do artigo, o conceito de segurança foi definido como a capacidade dos componentes da linguagem em evitar erros em tempo de execução ou apresentar vulnerabilidades que poderiam ser exploradas por usuários mal-intencionados. Os resultados destacaram que Rust demonstrou o menor tempo de CPU, enquanto C exibiu o menor consumo de RAM em ambas as aplicações. Java e Python foram as linguagens que apresentaram um maior consumo de RAM em ambas as aplicações. Go e C++ obtiveram um desempenho intermediário em ambas as métricas e aplicações. Rust foi apontada como uma das linguagens mais seguras, especialmente em ambientes concorrentes, devido à preocupação excessiva em evitar condições de corrida durante o desenvolvimento do código.

Finalmente, LION *et al.* (2022) propuseram uma análise quantitativa e abrangente do desempenho de quatro linguagens de programação (Java, Go, JavaScript e Python) em diversas aplicações, contemplando distintos modelos de consumo de computação, memória, rede, disco e paralelismo. Utilizando o C++ como referência, os resultados destacaram que JavaScript e Python são consideravelmente mais lentos que o C++, enquanto Java e Go mostraram ser competitivas em relação a linguagem C++, chegando, em algumas aplicações, a superá-la. Adicionalmente, JavaScript e Python apresentaram limitações em termos de paralelismo, experimentando perda de desempenho quando várias *threads* foram empregadas em suas soluções. Em contrapartida, Go e Java exibiram boa escalabilidade em ambientes com múltiplos núcleos. Quanto ao consumo de RAM, todas as linguagens consumiram pelo menos o dobro em comparação ao C++, sendo mais notáveis os casos de JavaScript e Java, que se destacaram como as que mais demandaram RAM entre todas as avaliadas.

3.2 *Offloading* Computacional

Vários trabalhos na literatura tem estudado o desempenho da técnica do *offloading* computacional em um contexto de MCC. Alguns deles criaram *frameworks* que ajudam os programadores a adotar o *offloading* durante o desenvolvimento de aplicativos móveis (CUERVO *et al.*, 2010; KOSTA *et al.*, 2012; COSTA *et al.*, 2015; GOMES *et al.*, 2017) e permitem que um dispositivo móvel cliente submeta tarefas para serem computadas por máquinas servidoras hospedadas na nuvem ou, até mesmo, nas bordas da Rede. Alguns outros, como (SANTOS *et*

al., 2018; GUO *et al.*, 2021), propuseram mecanismos que aprimoram o intitulado *offloading Device-to-Device* ou D2D, onde as tarefas submetidas por dispositivos móveis são processadas por outros dispositivos móveis, porém com melhores recursos computacionais e energéticos. No entanto, poucos deles avaliaram o desempenho de *offloading* quando envolveram processos desenvolvidos em diferentes linguagens de programação.

CHAMAS *et al.* (2017) avaliaram o consumo de energia durante a realização do *offloading* computacional entre uma aplicação Android e um processo servidor Java através dos seguintes mecanismos de comunicação: REST, SOAP, Socket e gRPC. Para isso, os autores utilizaram um aplicativo que ordena inteiros, pontos flutuantes ou objetos usando famosos algoritmos da literatura: (*Bubble Sort*, *Heap Sort* ou *Selection Sort*). Os resultados obtidos foram relevantes pois mostraram que o *offloading* computacional, mesmo quando realizado através de mecanismos comumente associados a comunicação multi-linguagem (REST, SOAP ou gRPC), proporciona uma maior economia de energia ao dispositivo móvel do que a computação local da tarefa, em especial, quando envolve algoritmos de alta complexidade e/ou parâmetros de entrada grandes. Por exemplo, quando realizado através do gRPC, o *offloading* reduziu o consumo de energia do dispositivo móvel em até 5 vezes quando comparado a abordagem local.

GEORGIOU; SPINELLIS (2019) avaliaram o tempo total e a energia dispendida com a comunicação entre processos desenvolvidos em uma das seguintes linguagens: Go, JavaScript, Java, Python, PHP, Ruby e C#. Cada processo assumiu o papel de cliente ou de servidor de uma aplicação Requisição-Resposta bastante simples, nela, o cliente invoca a execução remota de um procedimento via RPC, REST ou gRPC e o servidor responde ao cliente com uma mensagem *"Hello-World"*. Os testes avaliaram o *offloading* quando realizado entre máquinas de arquitetura Intel e ARM conectados entre si através de uma rede cabeada. Os resultados indicaram Go e JavaScript como as melhores linguagens. Por exemplo, através do gRPC, Go reduziu em 20% o tempo de processamento da tarefa e cerca de quatro vezes a energia do dispositivo quando comparada à Java. Já o JavaScript, por sua vez, apresentou os melhores resultados quando os testes envolveram máquinas Intel independente do mecanismo de invocação remota adotada.

ARAÚJO *et al.* (2020) avaliaram o tempo de processamento da tarefa de duas aplicações (uma de multiplicação de matrizes e outra de processamento de imagens) entre dois processos, um cliente (desenvolvido em Kotlin) e outro servidor (desenvolvido em C++, Python, Kotlin, Ruby, Go ou Java), que se comunicavam entre si via gRPC. Os autores também avaliaram o desempenho do *offloading* computacional ao variar o Sistema Operacional (SO) da entidade

servidora. Dessa forma, foram considerados os SOs Linux Debian, Android KitKat ou Android Vanilla. Os resultados indicaram que a adoção da comunicação *multi-language* foi vantajosa para o *offloading*, em especial quando envolveu servidores C++ ou Python. Por exemplo, executando sobre o Linux Debian, o servidor C++ reduziu em cerca de 35% e 40% o tempo de computação das matrizes quando comparado aos processamentos local e remoto com Java nesse mesmo sistema operacional, respectivamente.

Além desses artigos disponíveis na literatura, o próprio autor deste trabalho desenvolveu alguns trabalhos que indicaram ganhos significativos no *offloading* computacional quando realizado através de uma abordagem multi-linguagem, ou seja, quando realizado entre processos desenvolvidos com linguagens de programação distintas (MATOS *et al.*, 2021a), (MATOS *et al.*, 2021a), (MATOS *et al.*, 2022) e (MATOS *et al.*, 2023). Esses trabalhos serviram de base para a criação da arquitetura proposta nessa tese de doutorado, por conta disso, eles serão melhores detalhados no Capítulo 4

3.3 *Dew Computing*

Os trabalhos dessa seção apresentam soluções que adotam o paradigma *Dew Computing* como mecanismo para atacar problemas ou potencializar a ação de sistemas nas mais diversas áreas, como cidades inteligentes e *e-health*. Porém, para o propósito dessa tese de doutorado, mais importante que os resultados obtidos ou as problemáticas atacadas, é estudar as arquiteturas das soluções propostas para, com base nelas, definir o modelo do sistema proposto nesse documento. Assim, nos próximos parágrafos, os principais trabalhos serão resumidos com foco na arquitetura da solução sugerida e no seu funcionamento.

WANG (2020) propôs o *Dewblock*, uma rede *Blockchain* baseado na arquitetura *Cloud-Dew* (WANG, 2015a), onde a *Dew* foca em computar operações e manter no dispositivo cliente apenas os blocos que interessam ao usuário, enquanto a parte mais significativa da *Blockchain* é mantida na *Cloud*. Arquiteturalmente, o *Dewblock* é composto por *Cloud Servers* e *Dew Servers*. *Cloud Servers* são servidores, hospedados em nuvens públicas ou privadas, responsáveis por minerar novos blocos e, principalmente, manter a estrutura completa da *Blockchain*. Os *Dew Servers* são entidades de *software*, hospedadas nos dispositivos dos usuários, capazes de realizar várias ações sozinhos como processar e validar transações, mas não conseguem computar tarefas que envolvam a *Blockchain* completa, pois eles não a possuem. Assim, eventualmente, *Dew* e *Cloud Servers* devem interagir entre si para computar determinadas ações da rede *Blockchain*.

Tal interação é gerenciada por um protocolo próprio chamado *Pair Connection Protocol*.

GARROCHO; OLIVEIRA (2020) conduziram um estudo teórico e prático sobre o uso de *smartwatches* como *Dew Devices*. Inicialmente, os autores investigam o poder computacional e energético dos *smartwatches* mais populares entre 2015 e 2017 e concluem que, além de serem subutilizados em termos de processamento e armazenamento, a bateria de um *smartwatch* pode durar mais se reduzir o número de operações com a rede e aumentar o processamento local no dispositivo. Em seguida, os autores arquitetaram um sistema em quatro camadas: Física, *Dew*, *Fog* e *Cloud* (Figura 13). A camada Física é constituída por terminais de borda de rede, próximos ao usuário como computadores pessoais, *notebooks*, *smartphones* e dispositivos IoT. A camada *Dew* é composta exclusivamente por *smartwatches*, chamados de DeWatch, e aparelhos pareados à eles, como *smartphones*. Em geral, a *Dew* processa tarefas e armazena dados submetidos pelos dispositivos da camada Física. As camadas *Fog* e *Cloud* são formadas por servidores alocados próximos aos dispositivos de usuário ou na nuvem, respectivamente.

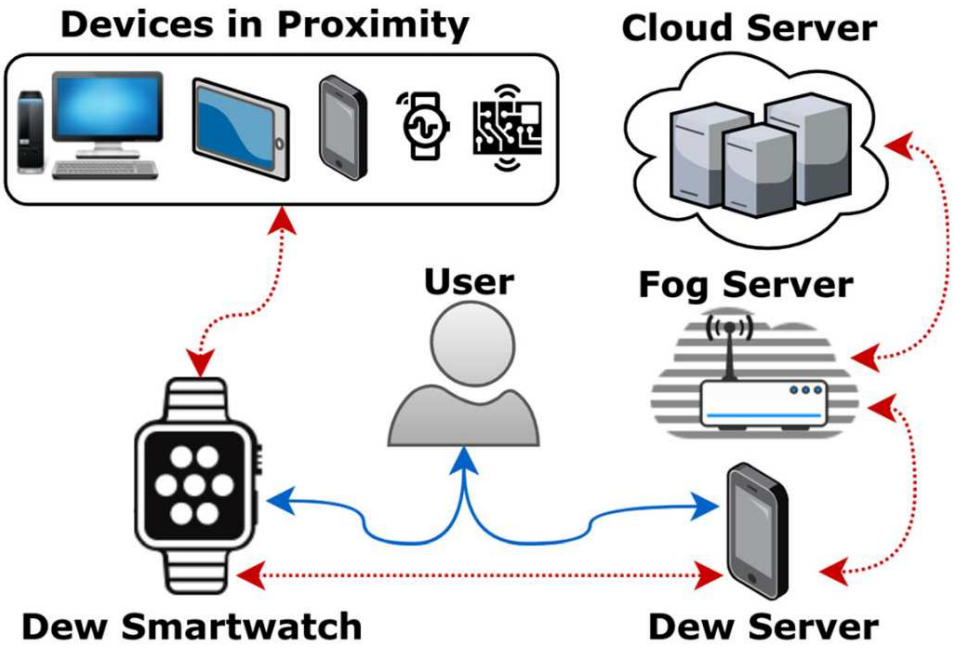


Figura 13 – Arquitetura *Dew* proposta por (GARROCHO; OLIVEIRA, 2020)

SUWANSRIKHAM *et al.* (2020) propuseram um *framework* capaz de persistir partes de arquivos em múltiplos provedores de nuvem e/ou um *Dew Server* hospedado em uma máquina na rede local. Arquiteturalmente, o *framework* é formado por três camadas: Usuário, *Dew* e *Cloud*. A camada de Usuário é constituída pelos dispositivos de usuário que podem compartilhar ou ser o alvo do compartilhamento de um arquivo. A camada *Dew* é composta por uma máquina central que gerencia fragmentos de arquivos e usuários. Finalmente, a camada *Cloud* é composta

por servidores hospedados na nuvem que fornecem serviços de armazenamento para os arquivos submetidos pelos dispositivos de usuário. A camada *Dew* possui duas responsabilidades principais, ambas essenciais quando a *Cloud* está inacessível. Primeiramente, ela replica fragmentos de arquivos, armazenando-os localmente e disponibilizando-os quando solicitados pelos usuários interessados. Em segundo lugar, com base em uma lista de usuários mantida pelo servidor central *Dew*, encaminha requisições de fragmentos de arquivos aos próprios proprietários, possibilitando assim uma interação direta (por meio de canais seguros) entre o proprietário do arquivo e o usuário alvo do compartilhamento.

GUBEROVIC *et al.* (2021) apresentaram uma arquitetura baseada em *Dew Computing* para simplificar e potencializar sistemas de aprendizagem federada, um paradigma de aprendizado de máquina distribuído em que os modelos são treinados localmente em computadores, mantendo a informação no próprio equipamento e compartilhando somente os pesos do modelo agregado, preservando a privacidade dos dados coletados. A arquitetura é constituída por duas camadas: *Dew* e *Cloud*. A *Dew* compreende os dispositivos de usuário federados, onde cada dispositivo treina o seu próprio modelo com as informações locais do seu usuário. Após o treinamento, cada modelo é enviado para um servidor de agregação hospedado na *Cloud*. Tal servidor agrega os modelos recebidos e retorna o resultante aos dispositivos na camada *Dew* que atualizam seus modelos locais. É importante esclarecer que o servidor de agregação alocado na camada *Cloud* foi apenas uma referência. Na verdade, os autores não definem exatamente onde ele deve ser alocado, somente destacam que o local deve ser uma camada superior a *Dew*. Assim, camadas como a *Fog* ou a *Edge* também são candidatas a hospedar o servidor de agregação. O dispositivo *Dew* é capaz de produzir o conhecimento local e usá-lo, mesmo sem conexão com o servidor de agregação, ao passo que o servidor de agregação ainda pode gerar o conhecimento global da federação, mesmo quando alguns dispositivos de usuário não informam o conhecimento local deles.

HIRSCH *et al.* (2021) apresentaram um sistema voltado para cidades inteligentes onde os dispositivos móveis (*smartphones* e *tablets*) dos passageiros de um ônibus se voluntariam para computar imagens geradas por uma câmera frontal ao automóvel e identificar buracos na via. Arquiteturalmente, o sistema é formado por dois tipos de componentes: N dispositivos móveis aptos a receber tarefas para processamento e um computador *on-chip-pc*, denominado *proxy*, apto a receber tarefas (como identificar objetos em imagens) e distribuí-las entre os dispositivos móveis seguindo algum critério ou heurística. Para receber tarefas, os dispositivos móveis precisam

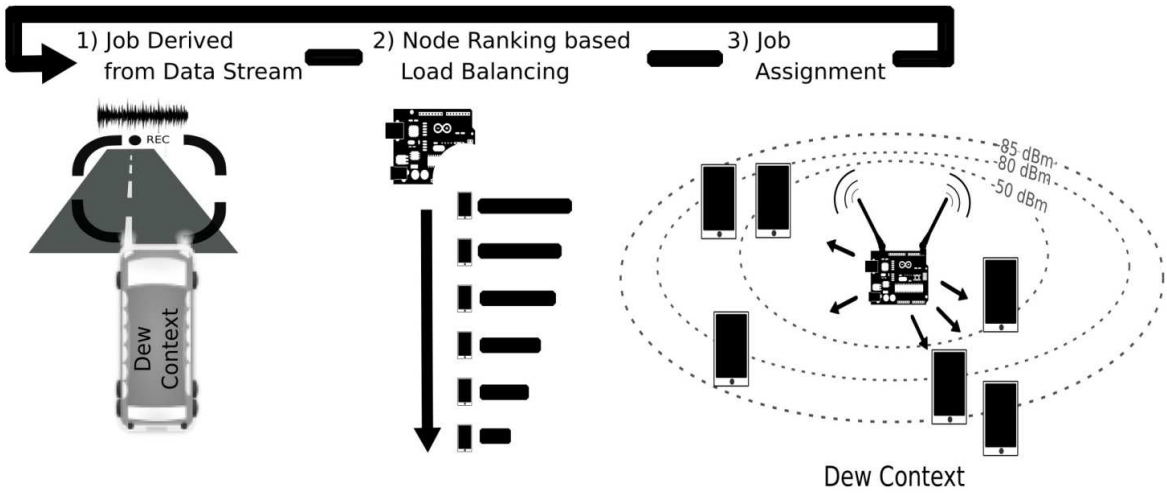


Figura 14 – Arquitetura *Dew* proposta por (HIRSCH *et al.*, 2021)

manifestar interesse através de um registro junto ao *proxy*. Uma vez cadastrado, cada aparelho deve enviar ao *proxy*, periodicamente, dados sobre o seu consumo de recursos computacionais e/ou energéticos. Todos esses dados são úteis para os algoritmos de escalonamento de tarefas: o *Ahead Enhanced Simple Energy Aware Scheduler* (AhESEAS) e o *Computation-Communication Throughput and Energy Contribution* (ComTECAC). Em essência, o escalonamento é feito com base em dados como a capacidade máxima da CPU, o último nível de bateria reportado e a capacidade máxima de rede do dispositivo móvel. A Figura 14 apresenta uma visão geral sobre funcionamento da solução de (HIRSCH *et al.*, 2021), onde o veículo captura informações sobre o ambiente e produz tarefas (Passo 1) que são inicialmente repassadas ao computador *on-chip-pc* que cria um ranking dos dispositivos móveis (Passo 2) e distribui as tarefas para esses dispositivos com base em algum algoritmo de escalonamento (Passo 3).

SINGH *et al.* (2021) propuseram um modelo arquitetural que usa a *Dew Computing* para identificar falsos alarmes do IDS (Sistema de Detecção de Intrusão) e uma técnica de Inteligência Artificial (IA) para tornar mais precisa a classificação dos ataques. O sistema é modelado em três camadas IDS, *Dew* e *Cloud*. A camada IDS é composta por máquinas que observam a rede e identificam possíveis ataques em progresso. A camada *Dew* é formada por dispositivos como *PCs* e *notebooks*. Por fim, a camada *Cloud* compreende um grupo de servidores, com grande capacidade de processamento e armazenamento, hospedados na nuvem. A cada classificação realizada, a IDS submete sua decisão e os dados coletados para a camada *Dew*. A *Dew*, inicialmente, pré-processa os dados e extrai as informações necessárias que servirão de entrada para o algoritmo de IA que revisará a decisão da IDS. Em seguida, a *Dew* decide onde o algoritmo será processado: localmente ou remotamente (na *Cloud*). Uma vez

decidido, o algoritmo ML é computado e os resultados são repassados a um Sistema de Controle que decide se é um falso alarme. Caso negativo, o Sistema de Controle notifica a IDS de que realmente se trata de um ataque.

MEDHI *et al.* (2022) propuseram um sistema, chamado *DC-Health*, que integra *Dew Computing* e IoT com o objetivo de potencializar aplicações voltadas para a assistência médica. O sistema é arquitetado em três camadas: *Dew*, *Edge* e *Cloud* (Figura 15). A camada *Dew* é composta por dispositivos simples, como *smartphones* ou *PCs*, que recebem dados sobre a saúde de cada paciente (oriundos de sensores que coletam informações sobre os pacientes e o ambiente hospitalar), pré-processa-os e analisa-os com o intuito de identificar ou prever possíveis anormalidades. As camadas *Edge* e *Cloud* possuem servidores robustos que oferecem serviços e suporte as ações da camada *Dew* quando a rede está disponível. Internamente, cada dispositivo *Dew* é organizado em módulos que realizam atividades específicas. Por exemplo, além do *Dew Database* e do *Dew Server* que armazena e processa os dados recebidos dos sensores, respectivamente, ele também conta com módulos de interação com sensores (*Communication Interface*) e Edge/Cloud (*Edge/Cloud Gateway Interface*) e com funções de gerenciamento (*Device Handler* e *Manager*).

SINGH *et al.* (2023) modelam uma arquitetura *Dew-Cloud* com o intuito de melhorar a segurança das informações geradas, manipuladas e persistidas em um ambiente de *Internet of Medical Things* (IoMT). A ideia é que através da técnica de Aprendizado Federado a arquitetura *Dew-Cloud* sugerida detecte a ocorrência de ataques em um ambiente IoMT. O sistema é modelado em três camadas: *IoMT*, *Dew* e *Cloud*. A camada *IoMT* coleta dados do ambiente e dos pacientes por meio de sensores e os transmite para a camada *Dew* através de tecnologias de comunicação como Zigbee ou Bluetooth. A camada *Dew* pré-processa e agrega as informações oriundas da camada *IoMT* para, posteriormente, enviá-las, via WiFi, à camada *Cloud*. A camada *Cloud* recebe os dados agregados, os compila e produz os prontuários médicos eletrônicos (EMR). Ela também disponibiliza as APIs necessárias que garantem a interação do sistema com os aplicativos de pacientes, médicos, cuidadores e administradores. Os modelos locais e global gerados pelo Aprendizado Federado nas camadas *Dew* e *Cloud*, respectivamente, servem como base para identificar possíveis agentes maliciosos no sistema. Se um modelo local diferir muito dos demais e/ou do modelo global, é sinal que o dispositivo relacionado está sofrendo algum tipo de ataque.

BERA *et al.* (2023) criaram um sistema para prever a produtividade dos solos em

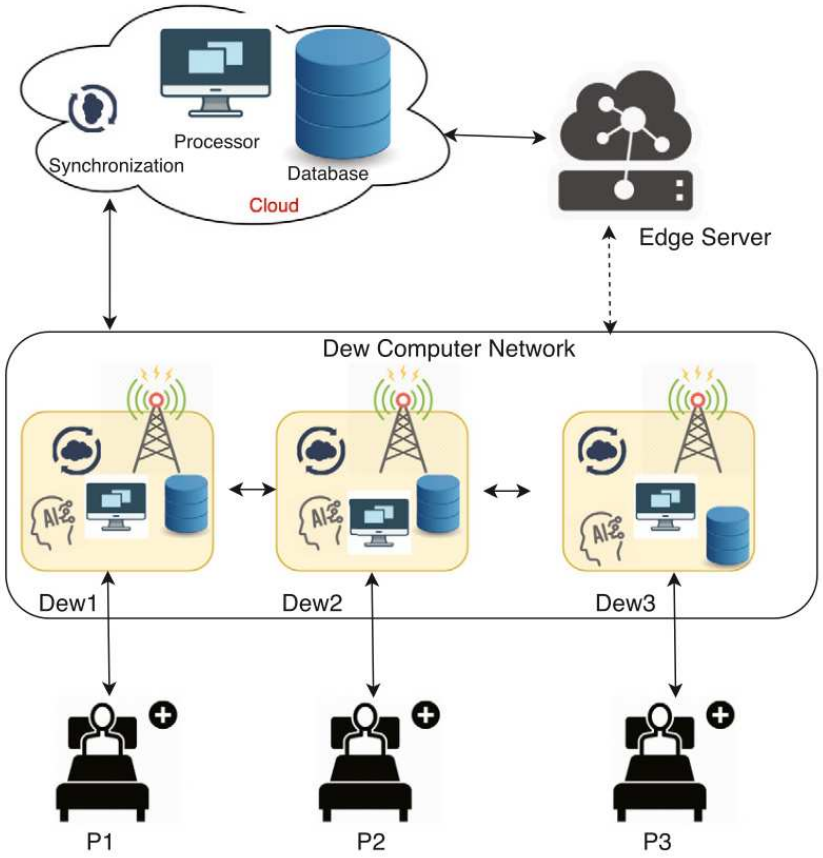


Figura 15 – Arquitetura *Dew* proposta por (MEDHI *et al.*, 2022)

um ambiente de *Internet of Agricultural Things* (IoAT). O sistema é arquitetado em quatro camadas: *IoT*, *Dew*, *Edge* e *Cloud* (Figura 16). A camada *Dew* é composta por dispositivos móveis ou qualquer outro dispositivo computacional com uma boa capacidade de processamento e armazenamento. O papel da *Dew* é acumular dados coletados pelos sensores da camada *IoT*, pré-processá-los e enviá-los para os servidores da *Edge* que executa algoritmos de Aprendizagem de Máquina, tentando prever a produtividade da colheita e recomendar as culturas mais adequadas para o solo em análise. É importante destacar que a comunicação entre as camadas *Dew* e *Edge* ocorre exclusivamente quando há uma conexão de rede disponível entre elas. Na ausência de tal conexão, a camada *Dew* armazena os dados recebidos até que uma conexão seja estabelecida. A camada *Cloud* tem como papel principal persistir os dados do ambiente e resultados da camada *Edge* e podem, eventualmente, realizar análises adicionais nos dados da região para melhorar previsão da produtividade agrícola.

YANNIBELLI *et al.* (2023) sugerem um mecanismo que permite recriar, de forma simplificada, as condições iniciais de um experimento. O objetivo é, recriando tais condições, permitir que os experimentos sejam refeitos/repetidos de maneira mais justa. O mecanismo é arquitetado em apenas duas camadas: *Dew* e *Edge*. A camada *Dew* é composta pelos *smartphones*,

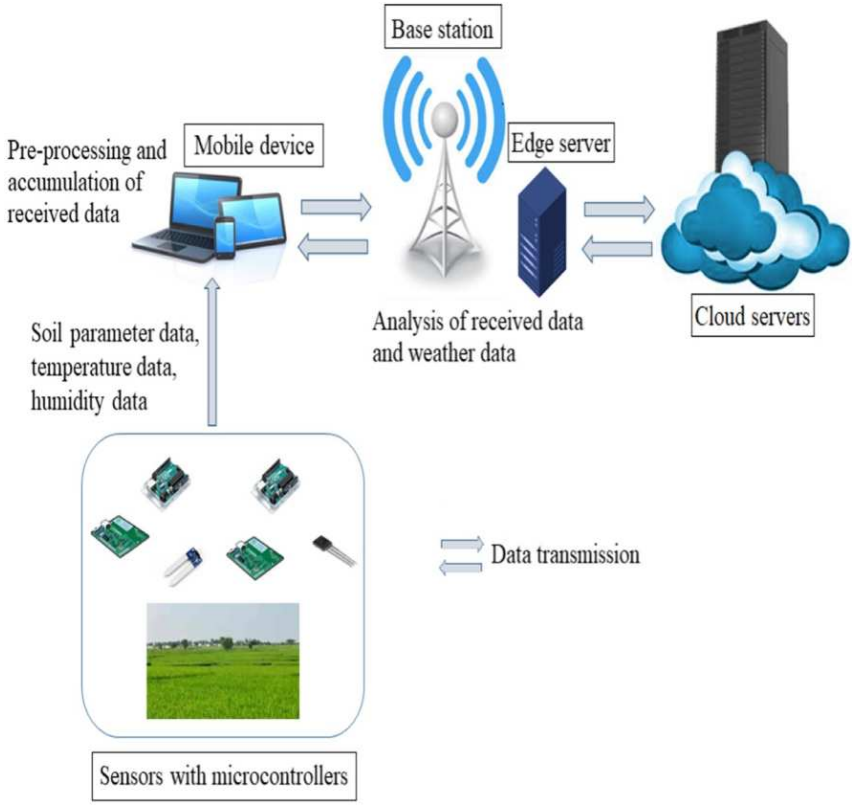


Figura 16 – Arquitetura *Dew* proposta por (BERA *et al.*, 2023)

ao passo que um computador central (um *notebook*, por exemplo) age como um servidor da *Edge*. A ideia é executar um algoritmo evolucionário na camada *Edge* para gerar a menor quantidade de ações necessárias para que todos os *smartphones* envolvidos no teste alcancem o nível de bateria desejado no menor tempo possível. Os autores utilizam a *Dew* para simplificar a (re-)configuração dos *smartphones* antes do início dos experimentos.

3.4 Considerações Finais

Esse capítulo apresentou os principais trabalhos correlatos a essa tese de doutorado. Os trabalhos foram segmentados em três grupos: 1) Aqueles que compararam o desempenho de várias linguagens de programação ao processar tarefas em ambientes *desktop*, embarcado e web; 2) Aqueles que avaliaram o desempenho do *offloading* computacional quando realizado entre processos desenvolvidos com diferentes linguagens; 3) Aqueles que conceberam soluções baseadas em *Dew Computing* para enfrentar desafios recorrentes encontrados em áreas como *e-health*, *blockchain* e cidades inteligentes.

Em geral, os resultados dos dois primeiros grupos mostram que Java, a principal linguagem para o desenvolvimento de aplicativos Android, apresenta resultados insatisfatórios

em termos de tempo de processamento e/ou consumo de recursos, sendo superada por outras linguagens de programação, especialmente as compiladas, como C/C++, Go e Rust. Isso motiva criar uma solução que permita computar tarefas utilizando linguagens alternativas à Java na plataforma Android. Contudo, os resultados também indicam não haver uma linguagem única que garanta eficiência computacional ao processar todos os tipos de tarefas e que, mesmo adotando aquelas mais adequadas, o desempenho depende fortemente do algoritmo escolhido. Assim, a solução deve oferecer recursos que permitam a execução de tarefas com diferentes linguagens e/ou algoritmos dentro da plataforma Android, independente da linguagem usada.

Os artigos do terceiro grupo apresentam modelos que possibilitam uma maior independência dos dispositivos de usuário através da *Dew Computing*, mas sem impedir a colaboração entre os dispositivos *Dew* e os equipamentos dos demais paradigmas computacionais (tais como *Edge*, *Fog* ou *Cloud*). Por meio da *Dew Computing*, um processo servidor pode ser duplicado no próprio aparelho do usuário e pode continuar a atender o aplicativo cliente mesmo sob condições adversas, como quando não há rede disponível. Isso motiva o desenvolvimento de uma solução que trate o *offloading* de uma tarefa como um serviço a ser prestado por processos passíveis de duplicação por meio da *Dew Computing*. Para aumentar a eficiência, a solução deve suportar que tais processos possam ser programados com linguagens diferentes daquelas usadas no cliente (no caso do Android, geralmente Java).

Os estudos conduzidos ao longo deste doutorado, que motivaram a concepção da arquitetura foco desta tese, diferem do primeiro grupo de artigos ao comparar o desempenho de diferentes linguagens de programação, mas em um contexto de *offloading* computacional, no qual o processamento da tarefa é realizado por um computador remoto e não localmente no próprio dispositivo gerador da tarefa. Ao compará-los com os trabalhos do segundo grupo, percebe-se uma maior variedade de contribuições. Além de considerar uma nova linguagem servidora (Rust) e uma nova ferramenta de comunicação multi-linguagem (Apache Thrift), esses estudos também avaliaram o desempenho do *offloading* sob diferentes aspectos, como o uso de canais seguros, múltiplos clientes simultâneos e a influência da rede. A Tabela 3 compara os trabalhos do segundo grupo com aqueles produzidos pelo autor desta tese, incluindo esta própria.

A contribuição mais significativa deste trabalho é a proposta de trazer o processo servidor, desenvolvido com uma linguagem mais eficiente, para o dispositivo móvel, adotando a estratégia da *Dew Computing*. Ao concluir este capítulo, observa-se que essa estratégia ainda não havia sido explorada na literatura, nem por estudos que compararam o desempenho com-

putacional de linguagens de programação, nem por trabalhos que utilizaram a *Dew Computing* para atacar desafios em aberto. Os resultados apresentados no Capítulo 6 indicam que adotar essa estratégia pode não só reduzir a dependência de servidores remotos e da própria rede, mas também melhorar a eficiência do processamento de tarefas em aplicativos móveis. Além disso, o uso dessa abordagem também pode acarretar em outras potenciais vantagens, como melhorias escalabilidade e na privacidade e segurança das informações processadas.

Trabalho	**Linguagens de Processamento**	**Ferramenta de Comunicação**	**Métricas Avaliadas**	**Aplicações Utilizadas**
(CHAMAS *et al.*, 2017)	**Cliente e Servidora:** Java	REST, SOAP, gRPC e Socket	Energia Consumida (Joule)	Ordenar números e objetos
(GEORGIOU; SPINELLIS, 2019)	**Cliente e Servidora:** Go, Java, Python, Ruby, C#, JavaScript e PHP	REST, RPC e gRPC	Energia Consumida (Joule), Tempo de Resposta (s), Tamanho Máximo de RAM (MB), Número de Chamadas de Sistema, Número de Falhas de Página e Número de Trocas de Contexto	*Toy Application* (Request-Response)
(ARAÚJO *et al.*, 2020)	**Cliente:** Kotlin **Servidora:** C++, Python, Java, Go e Ruby	gRPC	Tempo de Resposta (s)	Multiplicar Matrizes e Aplicar Filtros em Imagens
(MATOS *et al.*, 2021a)	**Cliente:** Java **Servidora:** C++, Python, Java e Go	gRPC	Tempo de Resposta (ms), Tempo de Rede (ms) e Energia Consumida (uAh)	Multiplicar Matrizes e Aplicar Filtros em Imagens
(MATOS *et al.*, 2021b)	**Cliente:** Java **Servidora:** Java e Go	gRPC	Tempo de Resposta (ms), Consumo de Rede (KB) e Energia Consumida (uAh)	Aplicar Filtros em Imagens
(MATOS *et al.*, 2022)	**Cliente:** Java **Servidor:** Rust	Apache Thrift	Tempo de Resposta (s), Consumo de Rede (MB) e Energia Consumida (Joule)	Fasta, K-Nucleotide e Spectral-Norm
(MATOS *et al.*, 2023)	**Cliente:** Java **Servidor:** Java e Go	gRPC	Tempo de Resposta (s) e Tempo de Rede (ms)	Multiplicar Matrizes
Essa Tese	**Cliente:** Java **Servidor:** Go	Apache Thrift	Tempo de Resposta (s) e Consumo de Rede (KB)	Aplicar Filtros em Imagens

Tabela 3 – Comparação entre artigos da literatura que investigaram o desempenho de diversas linguagens em cenários de *offloading* e os trabalhos desenvolvidos durante a elaboração desta tese de doutorado, incluindo esta própria pesquisa.

4 ARQUITETURA DADOS (*DEW ARCHITECTURE FOR DISTRIBUTION OF OFFLOADING SERVERS*)

Os tópicos abordados nos capítulos anteriores indicam uma oportunidade de melhoria no desempenho de aplicações móveis por meio do uso de técnicas de *ofllloading*, suporte a múltiplas linguagens e *Dew Computing*. Em face a estas oportunidades, este capítulo apresenta a DADOS (*Dew Architecture for Distribution of Offloading Servers*), uma arquitetura com o propósito de aprimorar o desempenho de aplicações móveis ao possibilitar o processamento de tarefas computacionalmente complexas diretamente no dispositivo móvel por meio de processos servidores desenvolvidos com linguagens de programação mais eficientes. Em resumo, a DADOS transforma os *smartphones* dos usuários em dispositivos *Dew*, capazes de replicar processos que, até então, eram executados apenas em servidores na nuvem, na *Fog* ou na *Edge*, contribuindo, assim, com os serviços de *offloading* oferecidos por esses paradigmas.

Por meio do atendimento de suas próprias requisições e/ou das requisições de outros aparelhos, a arquitetura reduz a carga de trabalho submetida aos servidores remotos hospedados na *Fog* ou na *Edge*. Esta característica oferece uma contribuição extra ao potencialmente aliviar a sobrecarga na rede, uma vez que ele utiliza recursos do próprio dispositivo e não envolve qualquer ação na rede externa durante a comunicação entre o aplicativo cliente e o processo servidor já duplicado.

A partir dos estudos feitos na literatura e previamente apresentados nos capítulos anteriores, este capítulo é estruturado de modo a apresentar o processo de concepção da DADOS. A Seção 4.1 descreve um cenário onde a arquitetura pode ser utilizada e destaca quais vantagens ela pode proporcionar. A Seção 4.2 resume o itinerário de pesquisas realizadas durante este doutorado, que motivou a concepção desta nova arquitetura baseada na integração de três abordagens (*offloading* computacional, processamento multi-linguagem e *Dew Computing*) até então tratadas separadamente pela literatura. A Seção 4.3 aborda a modelagem da arquitetura, desde a definição dos principais requisitos até a descrição dos componentes e da interação entre eles durante o funcionamento da arquitetura. Lembrando que a modelagem descrita na Seção 4.3 é a versão idealizada da DADOS. A versão inicial, mínima e concreta que foi utilizada nos experimentos será descrita no próximo capítulo.

4.1 Cenário Motivador

No processo de concepção de uma infraestrutura como a arquitetura proposta, é fundamental se estabelecer os cenários da aplicação a ser suportada. Neste sentido, esta seção apresenta um cenário de uma empresa de desenvolvimento de aplicações chamada *miCasa*, cujo foco é automação residencial. Um dos sistemas propostos pela empresa envolve usar os microfones e as câmeras dos próprios dispositivos móveis para capturar informações do ambiente, deduzir as ações desejadas pelo morador e realizá-las. Por exemplo, quando o morador fala "Ligar a TV!" e aponta a câmera do seu *smartphone* em direção ao aparelho na sala de estar, ele expressa sua vontade (ligar a televisão da sala de estar) e, uma vez reconhecido o contexto, será prontamente realizada. O objetivo é tornar a interação com a solução de casa inteligente da empresa mais intuitiva e fácil.

A implementação do sistema proposto inclui executar um conjunto de tarefas complexas, como reconhecer a voz do morador e interpretar os comandos emitidos por ele. Essas tarefas utilizam algoritmos sofisticados de Inteligência Artificial que envolvem cálculos intensivos e exigem um poder computacional e energético considerável dos dispositivos que as executam. Ciente dessa demanda, a *miCasa* opta por produzir um serviço especializado para executar essas tarefas rapidamente, otimizando o consumo de recursos dos dispositivos envolvidos. Tal escolha, pode exigir que o serviço seja implementado em uma linguagem diferente daquela adotada no aplicativo, visando uma execução mais eficiente da tarefa computacional.

Contudo, o desenvolvimento desse serviço especializado pode ser bastante desafiador. Primeiramente, a escolha da linguagem de programação mais adequada não é trivial, pois não existe uma única opção válida para todas as tarefas. Além disso, mesmo escolhendo a linguagem mais adequada para um determinada tarefa, o desempenho do serviço pode variar dependendo do contexto onde o ele está (por exemplo, da plataforma que o hospeda) e do algoritmo que ele executa. Esse último aspecto destaca a importância da *expertise* do desenvolvedor para garantir a execução eficiente do serviço.

A *miCasa* tem então duas opções para o desenvolvimento do serviço especializado: 1) Solicitar ao seu time de desenvolvimento; ou 2) Usar um repositório com serviços prontos. Essa segunda opção, inclusive, permite o consumo dos serviços através das abordagens *offloading* e/ou *Dew Computing* para permitir que eles possam ser executados remotamente ou mesmo replicados nos dispositivos móveis, respectivamente. O time de desenvolvedores da empresa possui amplo conhecimento em desenvolvimento Java para Android, mas tem experiência mínima

com outras linguagens de programação. Assim, os gestores da *miCasa* fazem uma análise para decidir qual o melhor caminho seguir.

Na primeira opção, o serviço seria implementado em uma versão única com uma linguagem e algoritmo específicos, possivelmente com desempenho fraco devido à falta de experiência da equipe da *miCasa* com a linguagem. Além disso, a versão pode ser inapropriada para certos contextos. Por exemplo, uma versão ágil na execução de tarefas, mas que consome muitos recursos, pode ser problemática quando o dispositivo precisa poupar tais recursos.

Na segunda opção, há diferentes versões do serviço implementadas com linguagens e algoritmos variados, possivelmente criadas por especialistas, oferecendo um serviço de qualidade superior. Cada versão tem um desempenho particular adequado para diferentes contextos. Por exemplo, uma versão pode ter baixo tempo de resposta, mas consumir muitos recursos, enquanto outra economiza recursos, mas é mais lenta. Com alta disponibilidade de recursos, a primeira versão é ideal. Contudo, em caso de escassez, a segunda é preferível.

Com base nessa avaliação, a empresa opta pela segunda opção. A Figura 17 apresenta uma visão de alto nível de como ela funcionaria, segmentado em quatros passos. Na figura, os serviços especializados são chamados de módulos, e o repositório de serviços é denominado repositório de módulos multi-linguagem.

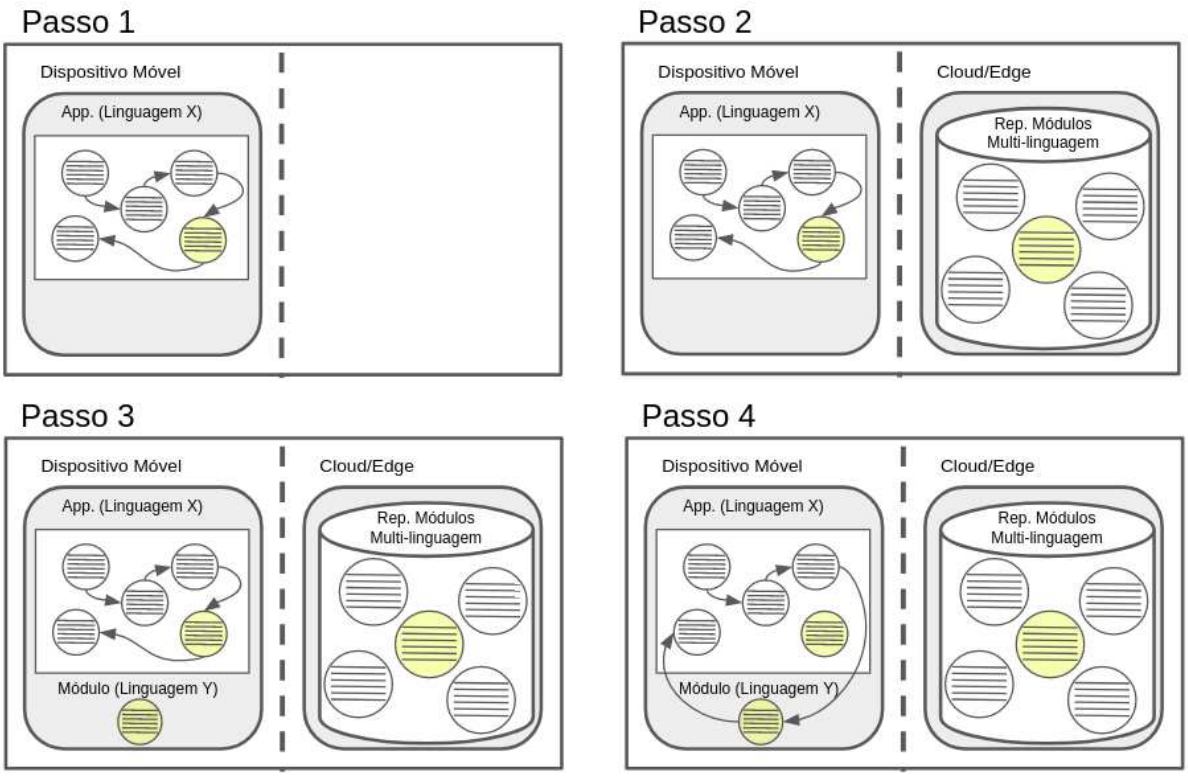


Figura 17 – Visão geral sobre o funcionamento do cenário motivador

No Passo 1, o dispositivo móvel do usuário executa o aplicativo de automação residencial desenvolvido em Java. Internamente, tal aplicativo é composto por cinco partes menores (representadas por círculos) que são executadas seguindo a sequência definida pelas setas. Uma dessas partes (destacada em amarelo) faz o reconhecimento de objetos em uma imagem capturada pela câmera do dispositivo móvel. Por se tratar de uma tarefa computacionalmente complexa, é aconselhável processá-la de forma célere e, se possível, consumindo poucos recursos do dispositivo móvel. Por meio da abordagem de *offloading* computacional, um servidor remoto hospedado na nuvem ou mesmo em um dispositivo de borda, oferece um repositório de módulos para procedimentos comuns, que podem ser escritos em diversas linguagens, como ilustrado no Passo 2. Estes módulos podem ser desenvolvidos por comunidades ou empresas especializadas de acordo com políticas de incentivo à distribuição destes artefatos de software.

Os módulos hospedados neste repositório podem ser duplicados pelos dispositivos móveis de acordo com a necessidade da aplicação, conforme ilustrado no Passo 3. Importante frisar que tais módulos podem ser escritos em linguagens diversas, e duplicados para execução no dispositivo de acordo com sua compatibilidade. Estes módulos então atuariam como serviços locais ao dispositivo, fazendo com que a invocação de seus serviços não demandasse mais chamadas remotas, evitando com isso gargalos de troca de dados entre dispositivos distribuídos (Passo 4).

O cenário proposto então potencializa que comunidades de desenvolvedores ou empresas especializadas possam interagir pela demanda de serviços recorrentes em determinados tipos de aplicações, num modelo bastante popular hoje de loja de aplicações (*marketplace*). Por meio desta abordagem, a equipe da empresa *miCasa* pode criar seu aplicativo para interagir e consumir tais serviços, ao invés de criar uma solução própria ou propor a concepção de uma. Em qualquer das abordagens, a *miCasa* teria à sua disposição não apenas várias soluções em uma mesma linguagem, como também em linguagens de programação diversas.

As opções poderiam ser ranqueadas de acordo com diferentes critérios como custo do serviço, desempenho, consumo energético ou de recursos, dentre outros fatores. Caberia à equipe de desenvolvimento a escolha dos módulos mais apropriados para o contexto do projeto a ser desenvolvido.

Os modelos de negócio possíveis incluem abordagens tipo *pay-per-use*, no qual os clientes são tarifados com base no nível de consumo do serviço, de modo semelhante ao que já acontece com ambientes *Serverless*. Neste sentido, desenvolvedores que ofertam serviços

seriam responsáveis unicamente pela criação dos módulos, enquanto provedores de nuvem ou infraestruturas *edge* garantiriam requisitos de escalabilidade e disponibilidade para potenciais clientes, mediante partilha de ganhos na comercialização dos módulos.

De acordo com o cenário apresentado, a proposição de uma infraestrutura de suporte traria benefícios para aplicações móveis, desenvolvedores e operadores de suporte, com impactos positivos no uso do aplicativo proposto pela empresa *miCasa*. Dentre alguns benefícios, incluem-se:

1. **Otimização do Processamento Local:** A utilização de linguagens de programação de alta eficiência melhoria o processamento de tarefas nos próprios dispositivos móveis, elevando o desempenho geral;
2. **Minimização da Latência e Independência da Rede:** Ao possibilitar que os dispositivos móveis processem tarefas localmente (auto-atendimento), a dependência da conectividade de rede e os efeitos da latência seriam reduzidos;
3. **Escalabilidade Aprimorada:** A capacidade de aumentar o número de servidores que suportam o *offloading* multi-linguagem de tarefas computacionais de outros dispositivos na rede contribui para uma melhor escalabilidade do sistema.

No contexto do cenário apresentado, a arquitetura DADOS se propõe a atender parte dos requisitos que compõem as funcionalidades apresentadas. De forma sucinta, a arquitetura proposta nesta tese de doutorado objetiva dar suporte ao repositório de aplicações descrito anteriormente, bem como à possibilidade de aplicações interagirem com este repositório para duplicar módulos no contexto local ao dispositivo, seguindo o modelo de *Dew Computing*. Estes módulos podem ser escritos em linguagens diversas, e são disponibilizados para clientes para integração com suas aplicações escritas em linguagens comuns ao desenvolvimento de *apps* móveis, como Java.

Porém, ressalta-se também que no cenário descrito, existem requisitos que agregam funcionalidades, mas não são essenciais para viabilizar de sua realização. No contexto da arquitetura proposta nesta tese de doutorado, 1) a criação de rankings para aplicações, 2) mecanismos de incentivo ou modelos de negócio para remuneração de desenvolvedores e 3) ferramentas de tomada de decisão para orquestrar a duplicação eficiente dos processos estão fora do escopo do trabalho. Além disso, o correto suporte à escalabilidade de um repositório de aplicações é obtido por meio de soluções complementares como balanceadores de carga.

A partir do cenário motivador, foram realizados experimentos de validação para

aspectos considerados importantes para a viabilização de uma plataforma de suporte ao desenvolvimento de aplicações móveis como descrito anteriormente. Ao todo, quatro experimentos preliminares foram conduzidos e que indicaram a viabilidade e os benefícios em juntar as técnicas do *offloading* computacional e da comunicação multi-linguagem e o paradigma *Dew Computing*. Estes experimentos são descritos nas subseções a seguir bem como os principais resultados obtidos em cada um deles.

4.2 Rumo à integração entre *offloading*, suporte a multi-linguagens e *Dew Computing*

A partir do cenário motivador, foram realizados experimentos de validação para aspectos considerados importantes para a viabilização de uma plataforma de suporte ao desenvolvimento de aplicações móveis como descrito anteriormente. Ao todo, quatro experimentos preliminares foram conduzidos e que indicaram a viabilidade e os benefícios em juntar as técnicas do *offloading* computacional e da comunicação multi-linguagem e o paradigma *Dew Computing.* Estes experimentos são descritos nas subseções a seguir bem como os principais resultados obtidos em cada um deles.

4.2.1 Etapa 1: Concepção do Offloading Multi-Linguagem

As pesquisas iniciais conduzidas nesta tese identificaram uma lacuna na literatura relacionada ao *offloading* computacional quando realizado entre processos desenvolvidos com diferentes linguagens de programação. Assim, esta etapa da pesquisa teve como objetivo avaliar a viabilidade desse tipo de *offloading*, denominado multi-linguagem, ao investigar a seguinte hipótese de pesquisa: "O *offloading* multi-linguagem potencializa o desempenho dos aplicativos móveis mais do que o *offloading* convencional (mono-linguagem)". Se comprovada esta hipótese, a pesquisa proporcionaria um impacto positivo na área de computação móvel, pois mostraria, com base em evidências empíricas, que adotar estratégias distintas nos lados cliente e servidor poderia melhorar o desempenho do *offloading* computacional e, consequentemente, dos aplicativos móveis.

Os experimentos avaliaram, como métricas, o tempo de processamento de tarefas e o consumo de energia do dispositivo móvel. O ambiente de teste consistiu em um *smartphone* (cliente do *offloading*) conectado através de uma rede sem fio local e dedicada a um *laptop* (servidor do *offloading*). O *laptop* hospedou quatro processos servidores desenvolvidos em

linguagens puramente interpretada (Python), parcialmente compilada (Java) e totalmente compilada (C++ e Go). Já o *smartphone* cliente executou versões modificadas de dois aplicativos Android Java já propostos na literatura (REGO *et al.*, 2017). O *MatrizGRPC* computava tarefas de multiplicação de matrizes quadráticas de dimensões variadas, enquanto o *BenchImageGRPC* computava tarefas para aplicar filtros de cinza em uma mesma imagem, mas com diferentes resoluções. A comunicação entre aplicativos cliente e processos servidores foi realizada usando o canais gRPC inseguros. Assumiu-se a computação no próprio dispositivo móvel e o *offloading* para o servidor Java como *baselines*. A Tabela 4 resume a configuração dos experimentos dessa etapa.

Objetivo(s)	Avaliar a viabilidade e os impactos (positivos e negativos) de aplicar a comunicação multi-linguagem em conjunto com o *offloading* computacional na computação de aplicativos móveis.
Sistema (dispositivos utilizados)	Smartphone com processador Qualcomm Snapdragon 600 (1.9GHz, Quad Core), 2GB de RAM e Android 5.0.2, Notebook com processador Intel Core i5-5200U (2.20GHz, Quad Core), 8GB de RAM e Ubuntu 19.04 64 bits, Um roteador Netgear WGR612 para construir uma rede sem fio exclusiva de 2,4 GHz entre os dispositivos. PowerMonitor conectado ao smartphone para coletar dados de consumo de energia do mesmo.
Fatores/ Níveis	Aplicativos (MatrizGRPC e BenchImageGRPC), Tipo de Processamento (Local ou Remoto), Linguagens de Níveis de Programação (Go 1.14.6, Java Openjdk 8, Python 2.7.16 e C++ 5.1.0), Dimensão Matrix de MatrixGRPC (400x400, 700x700 e 1000x1000) e resolução de imagem do BenchImageGPRC (0,3 MP, 4 MP e 8 MP).
Iterações	Cada experimento foi realizado 33 vezes para cada combinação de Fatores/Níveis.
Métricas avaliadas	Tempo total de processamento, Tempo gasto em operações de rede (envio e recebimento de dados) e Consumo de energia do dispositivo móvel.

Tabela 4 – Detalhes sobre a configuração dos experimentos realizados na Etapa 1

Os resultados mostraram que o *offloading*, quando direcionado a servidores desenvolvidos com linguagens compiladas (Go e C++), reduziu em até 38% o tempo de processamento da tarefa e 25% o consumo de energia do dispositivo móvel comparado ao mesmo procedimento, porém destinado a um processo servidor Java. A Tabela 5 apresenta uma parte dos resultados obtidos nessa etapa. Os resultados também mostraram que, no mínimo, 50% do tempo de processamento remoto da tarefa foi gasto somente com operações de rede, independente da linguagem servidora e da aplicação considerada. Em alguns casos, esse percentual chegou a 97%, o que comprova a forte influência da rede no desempenho do *offloading* multi-linguagem. Isso indica que, apesar dos bons resultados observados, eles poderiam ter sido ainda melhores se, de alguma forma, o tempo dispendido com a rede pudesse ser mitigado. Finalmente, julgou-se os resultados obtidos como, no mínimo, encorajadores para a continuidade da pesquisa.

Mais detalhes sobre o experimento dessa etapa e outros resultados podem ser obtidos em (MATOS *et al.*, 2021a).

Tamanho da Matriz	400x400		700x700		1000x1000	
Método	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**
Local (Android)	6626	23908.68	101224	383591.64	336696	1173827.79
Remoto (Go)	1450	6578.56	4018	19273.02	8598	41568.77
Remoto (C++)	1549	6938.14	4286	19980.85	8642	42606.12
Remoto (Java)	1651	7435.55	5680	23848.72	13948	55925.93
Remoto (Python)	2123	8347.34	7971	28277.73	21740	67494.83

Tabela 5 – Parte dos resultados obtidos na Etapa 1

4.2.2 Etapa 2: Avaliação do Offloading Multi-Linguagem através de Canais Seguros

A partir dos resultados promissores encontrados no estudo anterior, foi iniciada uma nova etapa da pesquisa com a realização de novos experimentos para avaliar o desempenho do *offloading* multi-linguagem, agora utilizando canais seguros de comunicação. Assim, foram formuladas duas questões de pesquisa para orientar os trabalhos: 1) "A utilização de conexões seguras inviabiliza o uso de tecnologias multi-linguagem no *offloading* computacional?"; 2) "Qual é o real impacto da rede no desempenho do *offloading* multi-linguagem?". Além de avaliar o desempenho do *offloading* multi-linguagem em um novo contexto, onde a transmissão segura de mensagens é essencial, as perguntas de pesquisa visavam aprofundar as discussões iniciadas na etapa anterior ao investigar um pouco mais o nível de influência da rede no *offloading* multi-linguagem.

Os experimentos avaliaram as mesmas métricas da etapa anterior (tempo de processamento e consumo de energia), acrescidas da quantidade de dados transmitida durante o *offloading*. O ambiente de teste foi montado utilizando a mesma topologia e os mesmos dispositivos da etapa anterior. No entanto, foram realizadas algumas mudanças em termos de software. Em primeiro lugar, os processos servidores foram limitados apenas a Go e Java. O aplicativo cliente adotado foi uma versão modificada do *BenchImageGRPC*, um dos aplicativos utilizados na etapa anterior, porém agora projetado para trabalhar com canais gRPC seguros, ou seja, conexões estabelecidas através do protocolo TLS (*Transport Layer Security*), em sua versão 1.2, que é suportada nativamente pelo gRPC. A comunicação foi realizada através de canais inseguros (*baseline*) e seguros, montados com autenticação somente do cliente (TLS padrão) ou com autenticação mútua de cliente e servidor (mTLS). A Tabela 6 resume a configuração dos experimentos desta etapa.

Os resultados indicaram que mesmo adotando conexões gRPC seguras, o *offloading* multi-linguagem reduziu o tempo de resposta da aplicação em até 87% e economizou a energia do dispositivo móvel cliente em até 90%, enquanto provocou uma sobrecarga ínfima na rede

Objetivo(s)	Avaliar o desempenho do *offloading* multi-linguagem quando realizado com conexões seguras e identificar estratégias para mitigar o impacto negativo da rede em seu desempenho.
Sistema (dispositivos utilizados)	Smartphone com processador Qualcomm Snapdragon 600 (1.9GHz, Quad Core), 2GB de RAM e Android 5.0.2, Notebook com processador Intel Core i5-5200U (2.20GHz, Quad Core), 8GB de RAM e Ubuntu 19.04 64 bits, Um roteador Netgear WGR612 para criar uma rede sem fio exclusiva de 2,4 GHz entre os dispositivos. PowerMonitor conectado ao smartphone para coletar dados de consumo de energia do mesmo. Wireshark no notebook para coletar dados de consumo de rede entre notebook e smartphone
Fatores/ Níveis	Aplicativo (BenchImageGRPC), Tipo de Processamento (Local, Remoto Inseguro, Remoto Seguro TLSv1.2 e Remoto Seguro mTLSv1.2), Linguagens de Programação (Go 1.14.6 e Java Openjdk 8) e Resolução de imagem do BenchImageGPRC (1 MP, 4 MP e 8 MP).
Iterações	Cada experimento foi realizado 50 vezes para cada combinação de Fatores/Níveis.
Métricas avaliadas	Tempo total de processamento, Tempo gasto em operações de rede (envio e recebimento de dados), Consumo de rede durante o descarregamento e Consumo de energia do dispositivo móvel.

Tabela 6 – Detalhes sobre a configuração dos experimentos realizados na Etapa 2

(no máximo, 1%). Também foi percebido que, adotar transmissões seguras, representou um acréscimo mínimo de 51% no tempo de resposta do processamento remoto e de 49% no consumo de energia do dispositivo móvel. A Tabela 7 exibe uma parte dos resultados obtidos nessa etapa. Como esperado, a rede continuou a consumir um parte considerável do tempo de processamento remoto da tarefa, entre 60% e 75%. Esse fato corrobora com a recomendação no estudo anterior que a rede é um fator chave no desempenho do *offloading* multi-linguagem. A partir de então, o foco da pesquisa voltou-se para buscar soluções que mitigassem os efeitos negativos da rede no *offloading* multi-linguagem.

Resolução da Imagem	**1MP**		**4MP**		**8MP**	
Método	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**	**Tempo Cons (em ms)**	**Energia Cons (em uAh)**
Local(Android)	2712	345.46	10590	1221.70	20469	2542.25
Remoto(Inseguro)	344	45.72	942	119.50	2106	313.71
Remoto(TLSv1.2)	579	74.04	1175	145.36	2398	354.60
Remoto(mTLSv1.2)	666	92.20	1307	159.29	2559	378.08

Tabela 7 – Parte dos resultados obtidos na Etapa 2

Mais detalhes sobre o experimento dessa etapa e outros resultados podem ser obtidos em (MATOS *et al.*, 2021b).

4.2.3 Etapa 3: Estudo sobre a Escalabilidade do Offloading Multi-Linguagem

Em paralelo a etapa anterior, foi desenvolvida uma pesquisa que avaliou o desempenho do *offloading* multi-linguagem quando realizado em redes celulares (especificamente, HSDPA e LTE) e analisou o nível de escalabilidade da técnica nesses tipos de rede. Para isso foram formuladas duas hipóteses de pesquisa. A primeira conjectura que “o bom desempenho

em redes WiFi observado na Etapa 1 seria mantido em redes celulares", enquanto a segunda afirma que "que a solução multi-linguagem apresentaria uma melhor escalabilidade que a solução mono-linguagem tradicional, independente da rede adotada." Se confirmadas tais hipóteses, além de mostrar que a solução também é eficiente em outros tipos de redes, também poderia indicar o *offloading* multi-linguagem como uma estratégia promissora para o desenvolvimento de processos servidores que computam tarefas complexas e/ou precisam lidar com um grande número de usuários.

Os experimentos consideraram, em especial, o tempo de resposta da tarefa como métrica de desempenho. Foram montados, com o auxílio da ferramenta androidTestBed (BARBOSA; REGO, 2019), dois ambientes de testes. O primeiro era composto por dois contêineres *Docker*, um atuando como cliente e outro como servidor, hospedados em duas máquinas servidoras distintas. O segundo seguia esse mesmo modelo, exceto pelo número de contêineres *Docker* clientes que variaram entre 2, 4, 8, 12, 16, 20 ou 24. Por intermédio do emulador, os ambientes foram configurados para refletir redes HSDPA (3G), LTE (4G) ou FULL, uma rede fictícia onde não há limites de velocidade de *download*, nem de *upload*. Adotou-se o mesmo aplicativo Android Java utilizado na Etapa 1 (*MatrixGRPC*) e as mesmas linguagens servidoras da Etapa 2 (Java e Go). A interação entre clientes e servidores foi realizado através de canais inseguros gRPC. Mais uma vez, assumiu-se a computação no próprio dispositivo móvel e o *offloading* para o servidor Java como *baselines*. A Tabela 8 apresenta um resumo da configuração dos experimentos realizados nesse trabalho.

Objetivo(s)	Avaliar o desempenho do *offloading* multi-linguagem quando realizado em redes celulares e estudar o nível de escalabilidade da técnica nesses tipos de ambientes.
Sistema (dispositivos utilizados)	2 Servidores com Processador (2.40GHz, 24-Core), 32GB de RAM e Ubuntu 18.04 64 bits conectado à mesma LAN para hospedar clientes e servidor (exclusivamente). Cada dispositivo cliente (emulado) foi configurado com 512 MBytes de RAM. Ao mesmo tempo, o contêiner servidor foi limitado somente pelos recursos computacionais disponíveis na máquina servidora.
Fatores/ Níveis	Aplicação (MatrixGRPC), Tipo de Processamento (Local ou Remoto), Linguagens de Níveis de Programação (Go 1.14.6 e Java Openjdk 8), Dimensão Matrix de MatrixGRPC (400x400 e 1000x1000), Número de dispositivos clientes (1, 2, 4, 8, 12, 16, 20 e 24), Tipo de Rede (HSDPA, LTE e FULL)
Iterações	Cada experimento foi executado 40 vezes para cada combinação de Fatores/Níveis.
Métricas avaliadas	Tempo de resposta da tarefa (em milissegundos), Tempo de processamento no lado servidor (em milissegundos) e Tempo de Rede (em milissegundos).

Tabela 8 – Detalhes sobre a configuração dos experimentos realizados na Etapa 3

Os resultados indicaram que o *offloading* multi-linguagem obteve um bom desempenho também em redes HSDPA e LTE. Por exemplo, mesmo em uma rede de baixa largura

de banda, como a HSDPA, o *offloading* de matrizes 1000x1000 para um servidor Go foi, aproximadamente, 34% mais rápido que o processamento local. Os resultados também mostraram que o sistema também possui um bom nível de escalabilidade quando utiliza o *offloading* multi-linguagem. Por exemplo, para matrizes 1000x1000, o tempo de resposta médio do *offloading* para o servidor Go foi menor que o tempo de processamento local, independente da rede adotada e do número de dispositivos clientes simultâneos (no máximo, 24 aparelhos). Mais uma vez, observou-se uma influência significativa da rede no tempo de resposta do *offloading*, especialmente quando envolveu um processo servidor Go, onde o tempo dispendido com a rede foi de, no mínimo, 97%. A Figura 18 apresenta uma parte dos resultados obtidos nesse trabalho.

(a) 400x400

(b) 1000x1000

Figura 18 – Parte dos resultados obtidos na Etapa 3

Mais detalhes sobre o experimento dessa etapa e outros resultados podem ser obtidos em (MATOS *et al.*, 2023).

4.2.4 Etapa 4: União do Offloading Multi-Linguagem com a Dew Computing

Por fim, a última etapa consistiu em uma avaliação preliminar sobre a viabilidade da arquitetura proposta neste trabalho. As Etapas 1 e 2 indicaram que a rede é um fator crítico para o desempenho do *offloading* multi-linguagem, enquanto a Etapa 3 revelou que a técnica permite "fazer mais com menos", uma vez que ela consegue atender um número maior de usuários simultâneos consumindo a mesma quantidade de recursos. Inspirados por essas observações, iniciou-se a busca por uma solução alternativa ao *offloading* que reduzisse a dependência da

rede, mesmo que isso significasse utilizar os recursos computacionais do próprio dispositivo móvel. Uma estratégia promissora que parecia atender esses requisitos era a ideia de duplicação sugerida pela *Dew Computing*.

Nesta etapa, o foco foi projetar um serviço Android chamado Serviço de Offloading Multi-Linguagem (SML), que tem a capacidade de duplicar um processo servidor remoto ao baixar seu binário e executá-lo no próprio *smartphone*. Uma vez duplicado, ele estaria apto a atender às solicitações do aplicativo do usuário por meio da interface de *loopback* do sistema e sem envolver operações na rede externa. Além do projeto, foram realizados experimentos para avaliar se a adoção do SML poderia melhorar o desempenho de um aplicativo móvel. De início, foram estabelecidas as seguintes hipóteses de pesquisa: 1) "A solução com o SML é a mais eficiente em termos de tempo de resposta da tarefa e/ou o consumo de energia do dispositivo móvel"; 2) "A adoção do SML economiza largura de banda da rede, sem prejudicar a qualidade do serviço oferecido pelo processo servidor". A confirmação dessas hipóteses sugeriria que o uso do SML não somente economizaria recursos de rede, mas também proporcionaria computações mais rápidas e eficientes em termos de energia, mesmo em comparação com a solução de *offloading* multi-linguagem.

Os experimentos consideraram métricas de desempenho como o tempo de resposta da tarefa, o consumo de energia do dispositivo móvel e o consumo de largura de banda de cada solução. O ambiente de teste consistiu em um *smartphone* cliente conectado através de uma rede sem fio local e dedicada a um *laptop* servidor. O *laptop* hospedou um processo servidor desenvolvido em Rust e todos os componentes necessários que permitissem o *download* do binário do processo durante a duplicação. O *smartphone* executou, além do SML, um novo aplicativo Android Java, o CLBGames, desenvolvido para esses experimentos e baseado em *benchmarks* disponíveis na página *The Computer Language Benchmarks Game*[1], um conhecido projeto Web que avalia o desempenho de diferentes linguagens. Os experimentos também usaram um novo mecanismo de comunicação multi-linguagem para intermediar a interação entre clientes e servidores, o Apache Thrift. A Tabela 9 apresenta um resumo da configuração dos experimentos realizados nessa etapa.

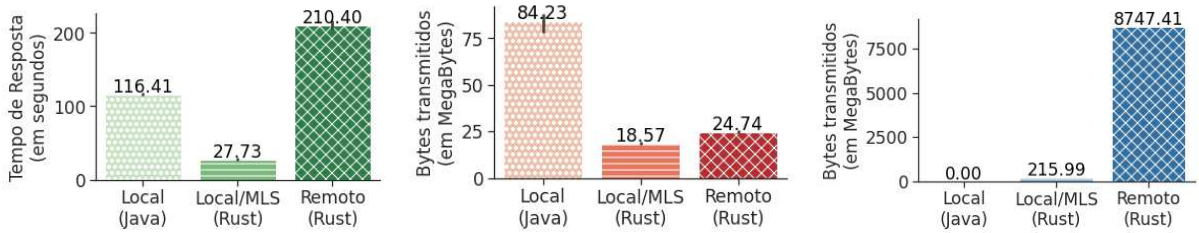
Os resultados obtidos mostraram que duplicar o processo servidor no próprio dispositivo móvel e permitir a ele o auto-atendimento, pode ser benéfico para o processamento da aplicação móvel e, indiretamente, melhorar o desempenho do *offloading* de outras aplicações. O

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Objetivo(s)	Desenvolver uma versão seminal da arquitetura proposta neste trabalho e avaliar seu impacto no desempenho do aplicativo móvel em comparação com o *offloading* multi-linguagem.
Sistema (dispositivos utilizados)	Smartphone com Processador Qualcomm Snapdragon 410 MSM8916 (1.2GHz, Quad Core), 2GB de RAM, Android 7.1.1, Notebook com Processador Intel Core i5-5200U (2.20GHz, Quad Core), 8GB de RAM e Ubuntu 19.04 64 bits, Roteador Netgear WGR612 para criar uma rede sem fio exclusiva de 2,4 GHz entre os dispositivos. PowerMonitor conectado ao smartphone para coletar dados de consumo de energia do mesmo. Wireshark no notebook para coletar dados de consumo de rede entre notebook e smartphone
Fatores/ Níveis	Aplicação (CLBGame), Método(Linguagem) (Local(Java), Remoto(Rust) e Local/SML(Rust)) e Entradas dos *benchmarks* (20x 1000000 *Fasta*, 20x 3000 *Spectral Norm* e 20x fasta100000 *K-Nucleotide*).
Iterações	Cada experimento foi executado 40 vezes para cada combinação de Fatores/Níveis.
Métricas avaliadas	Tempo de resposta da tarefa (em segundos), Consumo de energia do dispositivo móvel (em joules) e Consumo de Rede (em megabytes).

Tabela 9 – Detalhes sobre a configuração dos experimentos realizados na Etapa 4

estudo mostrou que a abordagem do auto-atendimento reduz o tempo de resposta da tarefa em até 87% e o consumo de energia do dispositivo móvel em até 25% quando o *offloading* envolve o envio de um número de dados significativo através da rede. Em paralelo, também observou-se que, para esse mesmo tipo de tarefa, o auto-atendimento reduziu o tráfego na rede em até 97%, o que representa um menor consumo da largura de banda na rede. Com mais banda disponível, outras aplicações ou dispositivos terão mais recursos para transmitir dados durante o *offloading*. Sem falar que, o auto-atendimento também reduz a carga de trabalho do servidor central, o que pode contribuir para aumentar a escalabilidade do sistema. A Figura 19 apresenta uma parte dos resultados obtidos nesse trabalho.



(a) Tempo de Resposta (Fasta) (b) Consumo de Energia (Fasta) (c) Consumo de Rede (Fasta)

Figura 19 – Parte dos resultados obtidos na Etapa 4

Mais detalhes sobre o experimento dessa etapa e outros resultados podem ser obtidos em (MATOS *et al.*, 2022).

4.3 Modelagem da Arquitetura Conceitual

Esta seção detalha a arquitetura projetada por este trabalho. A ideia é proporcionar ao leitor uma visão aprofundada do funcionamento da arquitetura, tanto internamente (ou seja, como ocorrem as interações entre os componentes) quanto externamente (ou seja, como ela se relaciona com os desenvolvedores e aplicativos móveis). Para facilitar a compreensão desta Seção, ela foi dividida em duas subseções. A primeira concentra-se na definição dos requisitos fundamentais que orientaram a modelagem da arquitetura conceitual da DADOS. Assim, a Seção 4.3.1 aborda os atores e dispositivos que interagem com a DADOS, os artefatos que ela deve manipular e as ações que ela deve suportar e/ou executar. A segunda discute a arquitetura conceitual da DADOS e os seus componentes. Assim, a Seção 4.3.2 apresenta a organização interna da arquitetura, destacando os principais componentes que integram os lados servidor e cliente, bem como as relações projetadas entre eles.

4.3.1 Requisitos para Suporte à Integração Offloading Multi-linguagem e Dew Computing

Como forma de permitir a união entre *offloading* computacional multi-linguagem e *Dew Computing*, esta tese de doutorado propõe a existência de uma arquitetura de software que suporte tal união. Esta arquitetura foi batizada como Arquitetura *Dew* para Distribuição de Servidores de *Offloading* (do inglês, *Dew Architecture for Distribution of Offloading Servers* – DADOS). A Figura 20 ilustra os elementos presentes no cenário de suporte pretendido pela arquitetura proposta neste trabalho. Nela, é possível identificar os artefatos manipulados, os atores/dispositivos envolvidos e as principais ações que cada um pode realizar na arquitetura. Estes elementos são melhor detalhados a seguir.

4.3.1.1 Atores/Dispositivos

Basicamente, a DADOS é destinada a dois tipos de atores: 1) Desenvolvedores de Aplicativos e 2) Desenvolvedores de Serviços. Um Desenvolvedor de Aplicativo é aquele que, como o time de desenvolvimento da *miCasa*, possui ampla experiência na programação de aplicativos móveis, mas tem pouco ou nenhum domínio em linguagens e tecnologias incomuns no desenvolvimento móvel. Já um Desenvolvedor de Serviço é aquele que possui um amplo conhecimento em linguagens e tecnologias incomuns no desenvolvimento móvel, como as comunidades de programadores especialistas em uma dada linguagem ou tecnologia.

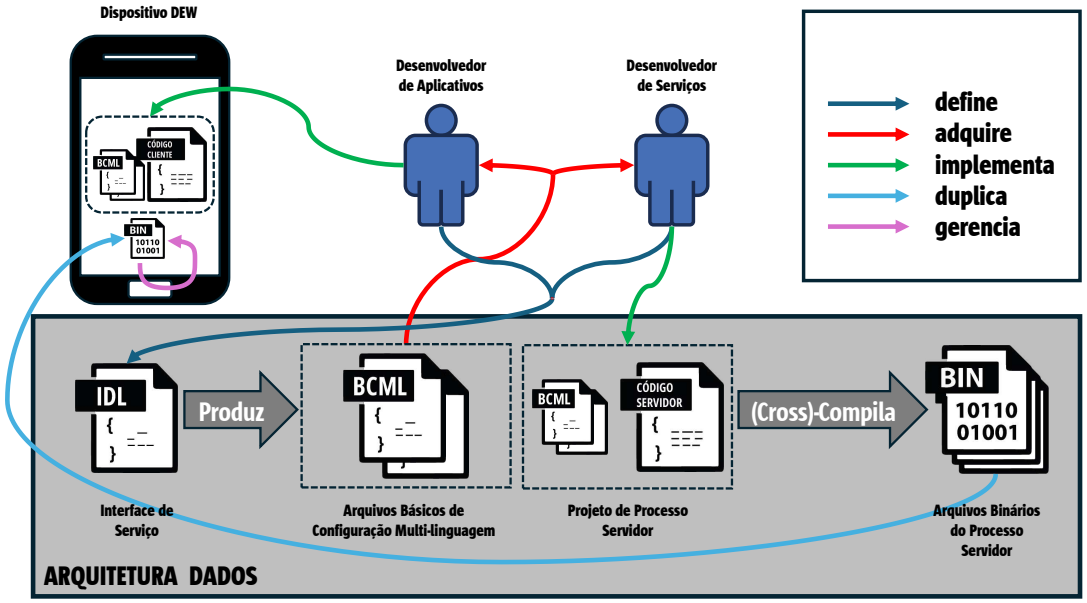


Figura 20 – Visão geral sobre o funcionamento do cenário motivador

Além deles, outros sistemas também interagem com a DADOS, visando complementar as funcionalidades dela, no caso, os Dispositivos *Dew* (representado na Figura 20 por um *smartphone*). Estes dispositivos são sistemas computacionais aptos a duplicar processos servidores remotos. Tais sistemas podem ser desde *smartphones* e *tablets* antigos até máquinas servidoras atuais e robustas. O que difere esses dispositivos dos convencionais é que eles possuem um conjunto de componentes de software que os permitem, de forma automatizada, integrá-las ao paradigma *Dew Computing*.

No caso particular da arquitetura aqui proposta, como ela foi projetada para o contexto do *offloading* computacional, a ideia central é que os Dispositivos *Dew*, ao duplicar processos servidores remotos inicialmente hospedados máquinas na *Edge*/*Fog*, possam se tornar independentes e, ao mesmo tempo, colaborar com os serviços de *offloading* prestados pela *Edge*/*Fog*. Ao duplicar um processo servidor, o Dispositivo *Dew* é capaz de atender suas próprios requisições, o que, de certa forma, o torna independente da *Edge*/*Fog*. Semelhantemente, ele também é capaz atender as requisições de outros dispositivos, o que potencialmente reduz a carga de trabalho da *Edge*/*Fog*, contribuindo com os serviços de *offloading* prestados por ela.

4.3.1.2 Artefatos

A DADOS manipula quatro tipos básicos de artefatos, a saber, 1) interfaces de serviço; 2) arquivos básicos de comunicação multi-linguagem; 3) projetos de processos servidores

e 4) binários de processos servidores.

Uma **interface de serviço** é um arquivo de texto, descrito com uma linguagem específica, que estabelece um grupo de métodos passíveis de serem chamados por meio de chamadas de procedimentos remoto (RPCs). Cada método representa uma tarefa que o processo servidor, uma vez implementado, é capaz de atender. Cada método é definido com um identificador único no serviço e seus parâmetros de entrada e tipos de retorno. O objetivo central da interface é formalizar a comunicação entre dois processos e guiar os desenvolvedores na integração do processo servidor e do aplicativo cliente. Assume-se que as interfaces de serviços sejam criadas a partir de ferramentas externas à arquitetura (como editores de texto ou ambientes integrados de desenvolvimento) e que sejam sempre claras e bem formatadas de acordo com a gramática da linguagem utilizada para descrevê-la. De fato, a interface de serviço segue a abordagem clássica de linguagens de definição de interfaces (do inglês, *Interface Definition Language* – IDL) para permitir a interoperabilidade entre processos de diferentes plataformas, i.e., diferentes linguagens de programação ou diferentes arquiteturas de execução.

Já um **arquivo básico de comunicação multi-linguagem** contém, em essência, os recursos necessários que possibilitam a interação direta entre o processo servidor e o aplicativo cliente, independente da linguagem de programação utilizada no desenvolvimento. Dentre os recursos, estão *stubs* e mensagens. Em um sistema distribuído, um *stub* é um componente de software utilizado para simplificar a comunicação remota entre clientes e servidores. Ele atua como um representante local da entidade remota e abstrai do desenvolvedor detalhes sobre a comunicação, como a formatação e a serialização das informações transmitidas e recebidas. Já uma mensagem é uma unidade de comunicação trocada entre os processos, possibilitando a troca de dados e a solicitação da execução de ações entre eles.

Um **projeto de processo servidor** é um grupo de arquivos que implementa as funcionalidades de um processo servidor em uma dada linguagem de programação. Esse grupo é formado pelo arquivo básico de comunicação multi-linguagem e pelo código-fonte que implementa o comportamento do processo servidor. O código-fonte pode ser um arquivo único ou a composição de diversos arquivos, dependendo da forma como o Desenvolvedor de Serviço estrutura cada projeto. Essa tese assume que um projeto de processo servidor sempre implementa corretamente o comportamento esperado do serviço e não executa nenhuma outra ação além daquelas especificadas na interface de serviço.

Um **binário de processo servidor** é um arquivo executável em uma dada plataforma,

resultado da (*cross*-)compilação de um projeto do processo servidor. A *cross*-compilação, em particular, refere-se ao processo de compilação do código em uma máquina para que ele seja executado em uma plataforma diferente. O binário produzido é fundamental para o funcionamento do servidor na plataforma de destino, garantindo que todos os serviços e o processo que ele representa sejam executados conforme projetado. Na proposta da DADOS, a plataforma é vista como a composição de uma arquitetura de hardware e o sistema operacional que a utiliza. Por exemplo, *smartphones* são normalmente construídos seguindo uma determinada versão da arquitetura ARM (por exemplo, ARMv7) e executando um sistema operacional (por exemplo, Android). Por outro lado, notebooks são construídos seguindo uma arquitetura Intel (por exemplo, x86-64) e executando um sistema operacional (por exemplo, Linux). Portanto, nesse exemplo, existiriam dois binários de processo servidor, um apto a ser executado no *smartphone* (ARMv7 + Android) e outro no notebook (x86-64 + Linux).

No caso particular da DADOS, o fato de envolver serviços escritos em linguagens distintas, impõe o uso de abordagens que viabilizem a interoperabilidade entre processos escritos em diferentes linguagens. Não existe uma imposição sobre qual tecnologia ou abordagem utilizar. Contudo, os experimentos feitos previamente apontaram o gRPC ou mesmo o Apache Thrift, como soluções candidatas para definir os serviços disponibilizados suportados. A utilização de tais soluções, poupa os desenvolvedores de criar suas próprias ferramentas, evitando que lidem com questões complexas relacionadas à garantia da interoperabilidade entre linguagens. Além disso, tais ferramentas também dispõem de um mecanismo que permite criar, de forma automatizada, *stubs* e mensagens padronizadas e agnósticas às linguagens. Por fim, estas soluções são amplamente suportadas por uma grande variedade de linguagens, e já consolidadas na comunidade de desenvolvedores em todo o mundo.

4.3.1.3 Ações

A partir dos conceitos apresentados, a arquitetura proposta nesta tese permite as seguintes ações: 1) definir interfaces de serviço; 2) produzir arquivos básicos de comunicação multi-linguagem; 3) adquirir arquivos básicos de comunicação multi-linguagem; 4) implementar projetos de processes servidores; 5) produzir binários de processos servidores; 6) duplicar processo servidor remoto e 7) gerenciar o processo servidor duplicado.

Ambos os atores estão aptos a **definir uma interface de serviço** a ser oferecido (Desenvolvedor de Serviço) ou a ser consumido (Desenvolvedor de Aplicativo). Baseando-se

em uma interface de serviço registrada, a arquitetura é capaz de **produzir arquivos básicos de comunicação multi-linguagem** para a linguagem de programação desejada pelo ator. Uma vez gerados, tais arquivos são disponibilizados pela DADOS mediante a requisições. Assim, os atores podem também **adquirir os arquivos básicos de comunicação multi-linguagem** produzidos para a linguagem alvo do aplicativo cliente (Java, por exemplo) e do processo servidor (Go, por exemplo).

De posse dos arquivos básicos de comunicação multi-linguagem, o Desenvolvedor de Serviço pode então **implementar o projeto de processo servidor**. Tal procedimento, basicamente, consiste em implementar o comportamento do processo ao desenvolver o código-fonte servidor, integrando-o aos arquivos básicos de comunicação multi-linguagem. Uma vez produzido, o Desenvolvedor de Serviço deve registrar o projeto de processo servidor junto à arquitetura. A arquitetura, por sua vez, pode **gerar os binários do processo servidor** através da (*cross-*)compilação do projeto recém-registrado. Uma vez que os binários estão disponíveis, o Dispositivo *Dew* pode **duplicar o processo servidor remoto** ao fazer o *download* do binário compatível com a sua plataforma para execução futura. É importante destacar que, para consumir os serviços do processo duplicado, o Desenvolvedor de Aplicativo deve, enquanto implementa o código-fonte cliente, integrar chamadas ao serviço utilizando os arquivos básicos de comunicação multi-linguagem.

Após a duplicação, o Dispositivo *Dew* deve **gerir o ciclo de vida do processo servidor duplicado**. Isso implica que, da maneira mais autônoma possível, o dispositivo deve determinar os melhores momentos para executar ou parar um processo servidor, além de remover o binário do processo servidor baixado anteriormente quando necessário. Esse gerenciamento eficiente assegura uma melhor administração dos recursos do dispositivo móvel, como o espaço interno, permitindo que binários sejam removidos quando houver necessidade de alocar recursos para outros processos do Dispositivo *Dew*. Além disso, o dispositivo também deve ser capaz de interromper um serviço *Dew* caso este consuma excessivamente memória, garantindo assim uma melhor otimização do desempenho e a sustentabilidade dos recursos do dispositivo.

4.3.2 Arquitetura Conceitual e os seus Componentes

Após uma visão geral sobre a DADOS, o próximo passo é projetá-la, mesmo que de uma maneira ainda muito abstrata. Conforme (SOMMERVILLE, 2011), o projeto de arquitetura é um processo no qual o sistema em desenvolvimento é planejado de forma a satisfazer os

requisitos previamente levantados. O objetivo geral é descrever os componentes arquiteturais e os relacionamentos entre eles. Pressman e Maxim (2016) também ressalta a importância de apresentar as propriedades acessíveis pelos usuários externos, sejam pessoas ou outros sistemas computacionais.

O projeto de arquitetura pode ser simplificado com a adoção de alguns padrões ou estilos arquiteturais já existentes. Conforme (SOMMERVILLE, 2011), um padrão de arquitetura descreve, de forma abstrata e estilizada, modelos de sistemas já testados em diferentes ambientes e sistemas. Portanto, o objetivo é registrar e difundir as boas práticas de organização de um sistema, destacando os pontos fortes e fracos de cada padrão, bem como identificando em que momentos um determinado padrão é mais indicado do que outro.

Conforme (PRESSMAN; MAXIM, 2016), o primeiro passo para projetar uma arquitetura é estabelecer o contexto no qual ela irá atuar. Para isso, o autor sugere a apresentação, por meio de um Diagrama de Contexto Arquitetural (ACD), das entidades externas que se relacionarão com o sistema em desenvolvimento e das interfaces que possibilitarão essa comunicação. Com base naquilo que foi apresentado na Seção 4.3, o contexto da arquitetura foi definido usando um ACD ilustrado na Figura 21. Essa arquitetura foi criada com base no modelo tradicional de Cliente/Servidor de duas camadas, onde a arquitetura proposta neste trabalho (o Sistema-Alvo) atua como servidor, enquanto os atores e os sistemas subordinados agem como clientes.

Figura 21 – Diagrama de Contexto Arquitetural do Sistema-Alvo

A Figura 21 mostra dois atores (Desenvolvedor de Serviço e Desenvolvedor de Aplicativo) e dois sistemas subordinados (Dispositivo *Dew* e Servidor *Edge*). Conforme (PRESSMAN; MAXIM, 2016), um ator é uma entidade (seja uma pessoa ou dispositivo) que produz ou consome os dados necessários para o funcionamento do sistema. Ambos os desenvolvedores interagem com a arquitetura, gerenciando interfaces de serviços e obtendo arquivos básicos que facilitam a comunicação em várias linguagens. Além disso, o Desenvolvedor de Serviço também pode gerenciar projetos. Por outro lado, um sistema subordinado é aquele dispositivo

computacional que a arquitetura utiliza, fornecendo dados ou processamento para completar sua funcionalidade (PRESSMAN; MAXIM, 2016). Os Servidores *Edge* e os Dispositivos *Dew* obtêm os binários persistidos pela arquitetura e os executam localmente para fornecer seus serviços de processamento de tarefas, seja localmente, seja em resposta a solicitações de dispositivos remotos, aplicando uma abordagem multi-linguagem.

A Figura 21 também mostra as interfaces que permitem a interação entre a arquitetura e cada ator ou sistema subordinado, representadas como círculos azuis na imagem. Tais interfaces devem seguir algum padrão arquitetural que possibilite integrar o Sistema-Alvo aos sistemas subordinados e atores de forma facilitada e segura, desde que possibilitem: 1) Ao Desenvolvedor de Aplicativo e ao Desenvolvedor de Serviço gerenciar interfaces de serviço e requisitar arquivos básicos, com o objetivo de simplificar a comunicação multi-linguagem entre o aplicativo cliente e o processo servidor; 2) Ao Desenvolvedor de Serviço gerenciar projetos de processos servidores desenvolvidos em diferentes linguagens de programação; 3) Ao Servidor *Edge* e ao Dispositivo *Dew* adquirir binários de processos servidores.

A representação da Figura 21 também descreve, de forma lógica, a arquitetura proposta neste trabalho, sem entrar em detalhes sobre sua estrutura física, ou seja, quais máquinas receberão quais softwares, onde elas estão alocadas e o tipo de conexão que as interliga. Segundo Tanenbaum e Steen (2007), essa descrição é conhecida como arquitetura de sistema e também precisa ser apresentada. Em geral, o Sistema-Alvo compreenderá uma ou mais máquinas servidoras alocadas nas bordas da rede (*Edge*), enquanto atores e sistemas subordinados são terminais que estão na mesma rede local do Sistema-Alvo. A exceção é o Servidor *Edge*, que, como o nome indica, também faz parte da *Edge*. Os terminais dos atores são máquinas tradicionais, como *desktops* ou *notebooks*, enquanto os sistemas subordinados são dispositivos com um conjunto de componentes de software que permitem a duplicação de processos servidores. Esses componentes serão abordados com mais detalhes no futuro.

Em termos de conectividade, presume-se que a comunicação entre os dispositivos na *Edge*, ou seja, entre o Sistema-Alvo e o Servidor *Edge*, ocorra por meio de redes cabeadas, o que torna a interação entre eles mais rápida e menos suscetível a problemas comuns em redes sem fio. Por outro lado, a comunicação entre o Dispositivo *Dew* e o Sistema-Alvo ocorre através de redes sem fio, uma vez que os dispositivos que podem funcionar como Dispositivos *Dew* são móveis, como *smartphones*, *smartwatches* e *notebooks*. Por fim, a comunicação entre os atores e o Sistema-Alvo pode acontecer tanto via cabo quanto via rede sem fio, dependendo do tipo de

dispositivo utilizado pelo ator para interagir com o Sistema-Alvo.

Uma vez especificado o contexto externo da arquitetura, o próximo passo é estruturá-la internamente, identificando seus componentes e os relacionamentos entre eles (Figura 22). Para facilitar a compreensão do modelo proposto, o mesmo esquema de cores da Figura 21 é mantido para destacar quais componentes estão ligados aos atores (em verde) e aos sistemas subordinados (em roxo), assim como os mesmos formatos dos retângulos para indicar quais componentes estão relacionados ao Sistema-Alvo (com borda arredondada) e aos atores/sistemas subordinados (com borda reta). Em linhas gerais, os componentes relacionados aos atores lidam com interfaces de serviços, arquivos básicos para a comunicação multi-linguagem e projetos de processos servidores, enquanto os sistemas subordinados trabalham exclusivamente com os binários dos processos servidores, que servirão de base para o procedimento de duplicação.

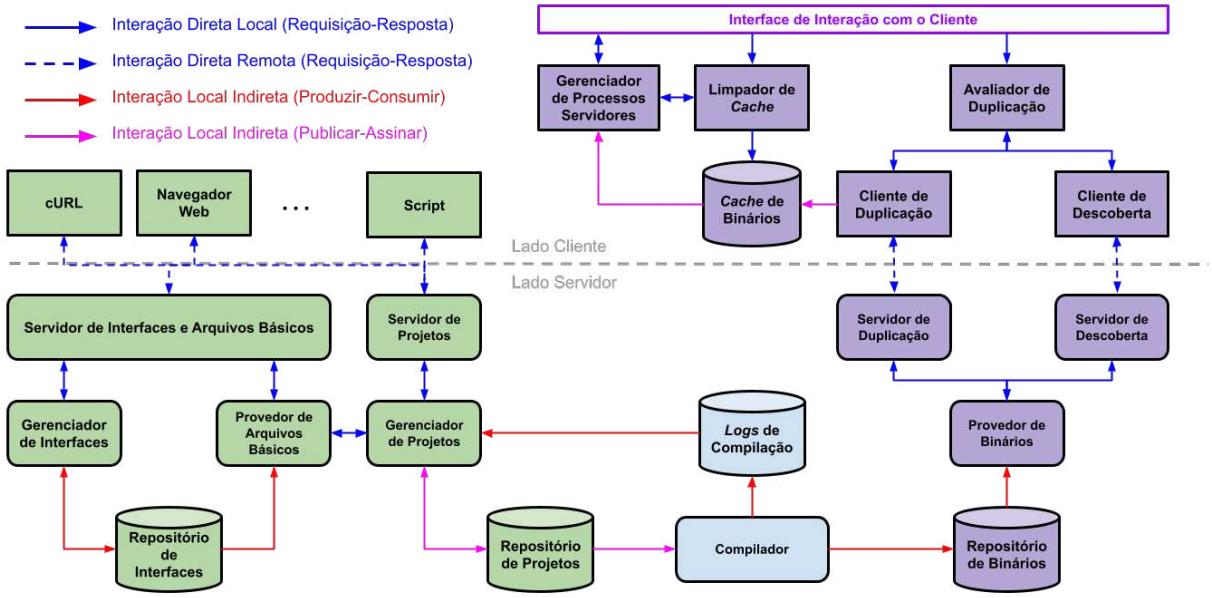


Figura 22 – Organização interna da arquitetura conceitual

A arquitetura foi projetada assumindo os repositórios de Interfaces, de Projetos e de Binários como componentes centrais, todos ilustrados na Figura 22. Cada um desses repositórios persiste um recurso relevante para o funcionamento da arquitetura, como no caso do Repositório de Interfaces, que persiste as interfaces de serviço. Cada uma dessas interfaces define um conjunto de métodos que, essencialmente, representam tarefas candidatas ao *offloading* computacional. Com base nessas interfaces, a arquitetura produz os arquivos básicos que facilitam a interação multi-linguagem entre o aplicativo cliente e o processo servidor.

O Repositório de Projetos armazena projetos de processos servidores desenvolvidos

com base nos arquivos básicos mencionados anteriormente. Cada projeto deve ser desenvolvido usando apenas uma linguagem de programação, empregando os arquivos de comunicação multi-linguagem. Finalmente, o Repositório de Binários armazena os binários gerados com a (*cross*-)compilação dos projetos salvos no Repositório de Projetos. O Repositório de Binários deve ser organizado visando agilizar a pesquisa por binários específicos, levando em consideração a arquitetura do Dispositivo *Dew* e a linguagem do processo servidor desejada.

Antes de apresentar os demais componentes da arquitetura com mais detalhes, é importante diferenciar os conceitos de Gerenciadores e Provedores. Ambos são componentes de *software*, mas diferem entre si na maneira como interagem com o repositório associado. Neste trabalho, um Gerenciador é um componente capaz de modificar o repositório vinculado a ele. Em outras palavras, além de requisitar itens, um Gerenciador também pode adicionar novos itens, bem como alterar e remover itens antigos. Por outro lado, o Provedor realiza apenas operações de leitura. Portanto, ele não permite inserir, editar ou excluir itens do repositório, apenas permite pesquisar por eles e obtê-los.

Dito isso, em essência, a arquitetura pode ser segmentada em serviços de Interfaces e Arquivos Básicos, Projetos, Duplicação e Descoberta, todos acessíveis externamente. Os dois últimos serviços são exclusivos dos sistemas subordinados, enquanto os dois primeiros são reservados aos atores, embora o Serviço de Projetos seja restrito apenas ao Desenvolvedor de Serviço. A arquitetura também oferece um quinto serviço, o Serviço de Compilação, que não é acessível por usuários externos. Esse serviço não está diretamente relacionado nem aos atores nem aos sistemas subordinados, sendo portanto destacado em azul. Na verdade, ele atua de forma autônoma, servindo como um elo entre o Serviço de Duplicação/Descoberta e o Serviço de Projetos. O Serviço de Compilação será explicado mais detalhadamente no próximo capítulo.

O Serviço de Interfaces e Arquivos Básicos é responsável por gerenciar as interfaces de serviço armazenadas no Repositório de Interfaces e por fornecer os arquivos básicos necessários para a comunicação multi-linguagem. Por sua vez, o Serviço de Projetos administra os projetos de processos servidores submetidos à arquitetura, que são armazenados no Repositório de Projetos. O Serviço de Compilação realiza a (*cross*-)compilação desses projetos, produz os binários resultantes e os salva no Repositório de Binários. Finalmente, os Serviços de Descoberta e de Duplicação também estão associados ao Repositório de Binários e, por meio deles, um sistema subordinado descobre a localização do binário do serviço desejado e realiza a duplicação por meio do *download* e execução do binário.

No lado do cliente, os componentes também variam de acordo com o tipo de usuário que interage com a arquitetura. Os atores devem utilizar clientes compatíveis com o mecanismo de comunicação definido pelas interfaces de interação com o Servidor de Interfaces e Arquivos Básicos e Servidor de Projetos no lado servidor. Portanto, eles podem usar desde um navegador web convencional até criar seus próprios *scripts* para enviar as requisições. Por outro lado, os sistemas subordinados requerem uma organização mais complexa no lado do cliente e, devido a isso, merecem uma explicação mais detalhada.

O lado cliente nos sistemas subordinados, similar ao lado servidor, também possui um repositório para o armazenamento de binários. Contudo, devido a restrições computacionais, esse repositório atua como uma *cache*, ou seja, deve persistir os binários temporariamente. A *Cache* de Binários desempenha um papel central na organização interna do cliente, uma vez que os demais componentes trabalham, direta ou indiretamente, para gerenciá-la. Por exemplo, enquanto o Cliente de Duplicação insere arquivos binários na *Cache*, cabe ao Limpador de *Cache* eliminar binários desnecessários e liberar espaço para o armazenamento de novos binários.

Outros dois componentes que requerem uma explicação mais detalhada são o Avaliador de Duplicação e o Gerenciador de Processos Servidores. Enquanto o primeiro é responsável por analisar os dados do ambiente e determinar o momento mais adequado para duplicar um processo remoto, o segundo assume a responsabilidade de iniciar e controlar o funcionamento dos processos servidores locais. Portanto, além de executar os binários armazenados na *Cache* de Binários, o Gerenciador de Processos Servidores também desempenha a função de encerrar os processos quando eles não são mais necessários e registrar as portas em que eles estão operando. Essas informações são cruciais para orientar o aplicativo cliente sobre onde direcionar suas solicitações de processamento durante o autoatendimento.

O último componente do lado cliente é a Interface de Interação com o Cliente. É por meio dessa interface que o aplicativo configura a parte do cliente da arquitetura. A ideia é que, ao instanciar o lado do cliente, duas políticas sejam informadas como entrada, uma para o Limpador de *Cache* e outra para o Avaliador de Duplicação. Equipados com essas políticas, esses componentes atuam de forma autônoma para determinar quando excluir arquivos da *Cache* de Binários e quando iniciar o processo de duplicação de um servidor, respectivamente.

A interface também deve oferecer um mecanismo que permita a comunicação direta entre o aplicativo cliente e o Gerenciador de Processos Servidores. Espera-se que, por meio dessa interação, o aplicativo cliente possa obter informações sobre os servidores locais (por

exemplo, a porta em que estão operando e seu estado) e também solicitar o encerramento deles conforme necessário.

Para concluir, é importante esclarecer os mecanismos de comunicação estabelecidos entre os componentes de cada serviço. Em geral, a comunicação ocorre diretamente entre os componentes, seguindo um mecanismo de Requisição-Resposta. Contudo, também pode ocorrer por meio de um repositório, utilizando os mecanismos de Produzir-Consumir ou Publicar-Assinar (COULOURIS *et al.*, 2011). Por exemplo, dentro do Serviço de Interfaces e Arquivos Básicos, o Gerenciador de Interfaces "produz" as interfaces de serviço que serão consumidas posteriormente pelo Provedor de Arquivos Básicos. De maneira semelhante, o Compilador "produz" os binários que serão consumidos futuramente pelo Provedor de Binários. Por outro lado, o Gerenciador de Projetos "publica" projetos no Repositório de Projetos, que serão automaticamente compilados pelo Compilador, que atua como um assinante. O Compilador também "produz" no repositório auxiliar "*Logs* de Compilação", que contêm os resultados das (*cross*-)compilações, e esses *logs* serão consumidos pelo Repositório de Projetos no futuro, quando este fornecer informações sobre o estado do projeto.

No lado do cliente, também é possível observar a interação direta entre componentes, além de ocorrer indiretamente através de um mecanismo de Publicar-Assinar. A interação direta ocorre através do mecanismo Requisição-Resposta entre os componentes Avaliador de Duplicação, Cliente de Duplicação e Cliente de Descoberta. Nesse cenário, o Avaliador de Duplicação, ao determinar o momento adequado para duplicar um processo servidor, se comunica com o Cliente de Descoberta para localizar um servidor que suporte a duplicação e, em seguida, aciona o Cliente de Duplicação para realizar o *download* do binário. Por outro lado, a interação por meio do mecanismo Publicar-Assinar ocorre entre os componentes Cliente de Duplicação e Gerenciador de Processos Servidores. Cada binário publicado pelo Cliente de Duplicação é iniciado pelo Gerenciador de Processos Servidores, que atua como um assinante.

5 IMPLEMENTAÇÃO DE REFERÊNCIA DA ARQUITETURA DADOS

Após uma extensa explanação sobre a modelagem e a versão conceitual da arquitetura, a presente seção detalha a implementação particular dos componentes da DADOS, utilizada para realização de experimentos e testes da arquitetura. Aqui, serão discutidos os padrões e as tecnologias utilizados para produzir a versão seminal da arquitetura implementada, assim como os motivos que nortearam as decisões de projeto tomadas durante o desenvolvimento, sempre que necessário. A Figura 23 ilustra a versão atual da arquitetura e destaca as tecnologias empregadas na criação de cada componente. É importante notar que a versão implementada difere um pouco da versão proposta e discutida no Capítulo 4. As mudanças foram feitas com o objetivo de gerar uma versão mínima, mas funcional, da arquitetura conceitual, com a intenção de utilizá-la nos experimentos de prova de conceito que serão descritos no Capítulo 6. Porém ressalta-se que esta abordagem não compromete a validação da arquitetura, pois a essência dos componentes principais e suas interações foram mantidas, garantindo que os resultados dos experimentos possam ser extrapolados para a arquitetura completa.

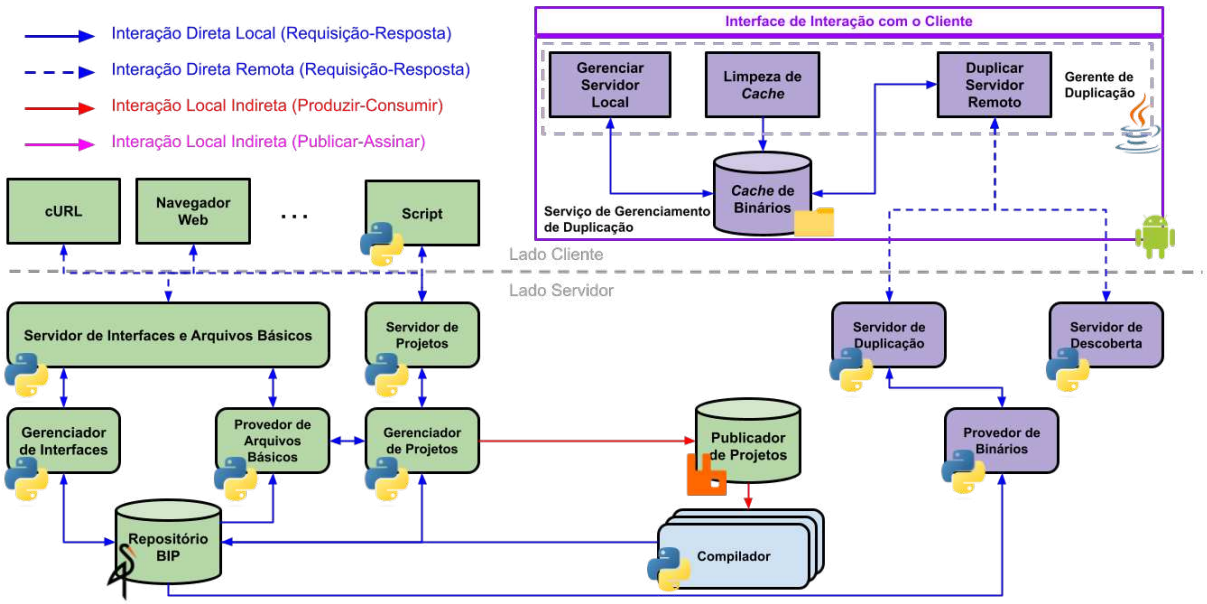


Figura 23 – Organização interna da arquitetura implementada

Com o objetivo de promover uma organização mais clara, o restante desta seção está dividido em quatro partes distintas. Inicialmente, serão abordadas as estratégias, as decisões e as tecnologias adotadas durante a implementação do lado do servidor da arquitetura (Seção 5.1). Em seguida, na segunda parte, esses mesmos aspectos serão discutidos, porém direcionados ao contexto do lado do cliente da arquitetura (Seção 5.2). As Seções 5.3 e 5.4 encerram o capítulo

apresentando, respectivamente, uma descrição detalhada sobre o funcionamento da arquitetura implementada e as considerações finais sobre ela.

5.1 Lado Servidor

A implementação do Lado Servidor da arquitetura foi baseada fortemente na abordagem de microsserviços. Segundo (NEWMAN, 2021), microsserviços representam um estilo arquitetural de software onde a aplicação é segmentada em serviços pequenos, independentes e altamente especializados. Cada microsserviço realiza uma função específica e autocontida da aplicação completa e disponibiliza uma interface de comunicação para que ele possa interagir com outros microsserviços que realizam outras funções da aplicação completa. Juntos, todos os microsserviços fazem a aplicação funcionar. Dentre as vantagens normalmente atribuídas aos microsserviços estão a alta escalabilidade e a alta tolerância a falhas (NEWMAN, 2021).

Nesse contexto, o Lado Servidor foi dividido em sete microsserviços, todos encapsulados por contêineres *Docker*[1] que interagem entre si através de uma rede virtual criada pelo próprio *Docker* e disponível apenas dentro da máquina servidora que os hospeda. A escolha de encapsular os microsserviços em contêineres *Docker* foi motivada, principalmente, por questões de desempenho. Por se tratar de uma virtualização leve (KANE; MATTHIAS, 2018), espera-se proporcionar uma inicialização rápida do Lado Servidor da arquitetura. A escolha do *Docker* foi fundamentada por ele ser uma das ferramentas de virtualização por contêineres mais populares na atualidade, senão a mais popular. Para simplificar e otimizar o gerenciamento dos microsserviços, adotou-se a ferramenta *Docker Compose* para, de maneira automatizada, configurar e levantar todos os contêineres do Lado Servidor e a rede privada que os conecta.

Antes de discorrer sobre os microsserviços que constituem a arquitetura, é essencial apresentar dois componentes centrais para o seu funcionamento: o Repositório BIP e o Publicador de Projetos. O Repositório BIP (acrônimo de Binários, Interfaces e Projetos) unifica os Repositórios de Binários, de Interfaces e de Projetos exibidos na Figura 22 e armazena binários de processos, interfaces de serviços e projetos de processos servidores. O Publicador de Projetos atua como um componente auxiliar que permite a interação indireta Publicar-Assinar entre o Gerenciador de Projetos e o Compilador exibida na Figura 22. Essa interação é fundamental para que projetos de processos servidores sejam convertidos em binários de processos por meio da (*cross*-)compilação. Esta interação será melhor explicada em seções futuras.

[1] https://www.docker.com/

A arquitetura implementada trata todos os seus recursos de interesse (isto é, interfaces, projetos e binários) como objetos. Portanto, o Repositório BIP assumiu um papel de servidor de armazenamento de objetos. Nesse contexto, optou-se pela ferramenta MinIO[2]. Contudo, o fator mais relevante foi a sua compatibilidade com a Amazon S3, uma das principais soluções de persistência em nuvem da atualidade. Assim, planeja-se simplificar a integração entre *Edge* e *Cloud*, promovendo uma colaboração mútua entre elas. Na arquitetura, o Repositório BIP é representado como um contêiner *Docker* criado com base na imagem *minio/minio* e configurado com 0.50-1.00 de CPU e 512MB-1GB de Memória Principal. Esses valores foram escolhidos com o objetivo de garantir que o serviço funcione de maneira básica, sem considerar aspectos de desempenho. O contêiner também recebe um IP fixo, não acessível externamente e bem conhecido pelos demais serviços.

O Publicador de Projetos atua como intermediário que permite a interação indireta Publicar-Assinar entre o Gerenciador de Projetos e o Compilador, conforme representado na Figura 22. Optou-se por representá-lo como um *Broker RabbitMQ* devido à familiaridade do autor com a ferramenta e, principalmente, porque o *RabbitMQ* disponibiliza um módulo Python que facilita a interação com a ferramenta. O Publicador de Projetos também é representado como um contêiner *Docker* de IP fixo e bem conhecido pelos serviços que interagem com ele. Assim como o Repositório BIP, ele não é acessível por usuários ou programas externos ao Lado Servidor. O contêiner foi configurado com 0.25-0.50 de CPU e 128-512MB de Memória Principal e foi construído com base na imagem *rabbitmq:3.11-alpine*. Novamente, as configurações de CPU e Memória Principal foram estabelecidas com base em considerações pessoais do autor.

Além dos contêineres mencionados, outros cinco compõem a arquitetura. Cada um representa um microsserviço específico e essencial para o funcionamento da arquitetura: 1) Serviço de Interfaces e Arquivos Básicos; 2) Serviço de Projetos; 3) Serviço de Compilação; 4) Serviço de Duplicação e 5) Serviço de Descoberta. Todos foram programados pelo autor desta tese em Python 3.8.16 e, quando necessário, foram utilizados módulos de terceiros para implementar funcionalidades específicas dos microsserviços. Optou-se pelo Python por ser uma linguagem familiar ao autor e por possibilitar um rápido desenvolvimento. Devido à sua importância, tais serviços serão explicados com mais detalhes nas seções seguintes.

[2] https://min.io/, um sistema de armazenamento de objetos leve, distribuído, escalável e de alto desempenho (MAKRIS *et al.*, 2022), como o Repositório BIP

5.1.1 Serviço de Interfaces e Arquivos Básicos

O Serviço de Interfaces e Arquivos Básicos permite que Desenvolvedores de Aplicativos e Desenvolvedores de Serviços gerenciem suas interfaces de serviço e obtenham os arquivos básicos necessários para comunicação multi-linguagem. Ele engloba os seguintes componentes da Figura 23: 1) Servidor de Interfaces e Arquivos Básicos, 2) Gerenciador de Interfaces e 3) Provedor de Arquivos Básicos. O Serviço de Interfaces e Arquivos Básicos também possui uma relação direta com o Repositório BIP, onde são persistidas as interfaces de serviços submetidas à arquitetura. Essa relação acontece através de um mecanismo de Requisição-Resposta (Figura 23), onde o Gerenciador de Interfaces e o Provedor de Arquivos requisitam ao Repositório BIP a realização de uma determinada operação sobre uma interface de serviço e recebem uma resposta que indica se a operação foi processada com sucesso ou não.

Em linhas gerais, o Servidor de Interfaces e Arquivos Básicos atua como um servidor HTTP que aguarda requisições REST dos Desenvolvedores de Serviços e dos Desenvolvedores de Aplicativos em uma porta padrão, a 50060. Dependendo do propósito da requisição, ela é direcionada ou ao Gerenciador de Interfaces ou ao Provedor de Arquivos Básicos, os quais devem interagir com o Repositório BIP para processá-la. Dentre as operações possíveis, o Gerenciador de Interfaces consegue solicitar, inserir, atualizar ou excluir interfaces de serviço, enquanto o Provedor de Arquivos Básicos está limitado a solicitar apenas uma interface de serviço e, com base nela, produzir o arquivo básico de comunicação multi-linguagem correspondente. Caso ocorra algum erro durante o atendimento da requisição, o Servidor de Interfaces e Arquivos Básicos retorna ao Desenvolvedor uma mensagem indicando o motivo do problema.

O Servidor de Interface e Arquivos Básicos foi desenvolvido com o auxílio do Flask[3], um *framework* Web Python que simplifica e agiliza a programação de um servidor HTTP. Este servidor oferece dois *endpoints*, */ifaces* e */mlfiles*, para atender às requisições sobre interfaces de serviço e arquivos básicos de comunicação multi-linguagem, respectivamente. As requisições direcionadas a */ifaces* são encaminhadas ao Gerenciador de Interfaces, enquanto as requisições destinadas a */mlfiles* são direcionadas ao Provedor de Arquivos Básicos. Porém, antes de qualquer encaminhamento, o Servidor de Interface e Arquivos Básicos realiza uma avaliação geral da requisição em busca de inconsistências. Por exemplo, ele verifica se o método HTTP usado na requisição é suportado pelo *endpoint* de destino e se a requisição possui todos os parâmetros necessários ao seu processamento. Se alguma incoerência for detectada, o sistema

[3] https://flask.palletsprojects.com/en/3.0.x/

retorna imediatamente uma mensagem de erro ao Desenvolvedor e interrompe o atendimento da requisição.

GET /ifaces	
Entrada	Identificador do serviço (*serviceId*) e da ferramenta multi-linguagem do serviço (*frameworkId*).
Saída	Sucesso → Código 200 + Arquivo com a Interface do Serviço Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!* ou *Service interface not found!*)
POST /ifaces	
Entrada	Identificador do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*) e o arquivo da interface do serviço a ser inserida.
Saída	Sucesso → Código 200 + Mensagem (*Service interface inserted sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!* ou *Service interface already exists!*)
PUT /ifaces	
Entrada	Identificador do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*) e o arquivo da interface do serviço a ser atualizada.
Saída	Sucesso → Código 200 + Mensagem (*Service interface updated sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!* ou *Service interface not found!*)
DELETE /ifaces	
Entrada	Identificador do serviço (*serviceId*) e da ferramenta multi-linguagem do serviço (*frameworkId*).
Saída	Sucesso → Código 200 + Mensagem (*Service interface deleted sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!* ou *Service interface not found!*)
GET /mfiles	
Entrada	Identificador do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*) e da linguagem do aplicativo cliente (Desenvolvedor de Aplicativo) ou do processo servidor (Desenvolvedor de Serviço).
Saída	Sucesso → Código 200 + Arquivos Básicos de Comunicação Multi-Linguagem (comprimido) Falha → Código 400 + *Input parameters mismatch!*, *HTTP method unsuported!*, *Unknown framework!*, *Unknown server language!* ou *Service interface not found!*)

Tabela 10 – Interfaces REST de interação com o Serviço de Interfaces e Arquivos Básicos

O Gerenciador de Interfaces é capaz de executar operações de aquisição, inserção, atualização e exclusão de uma interface de serviço. Por isso, o *endpoint /ifaces* suporta os métodos HTTP: GET, POST, PUT e DELETE (Tabela 10). O método GET é utilizado para obter uma interface de serviço, dado o identificador do serviço e o *framework* multi-linguagem base para o seu desenvolvimento. O método DELETE é utilizado para excluir uma determinada interface de serviço e recebe os mesmos parâmetros de entrada do método GET. Os métodos POST e PUT servem, respectivamente, para inserir e atualizar uma interface de serviço. Para isso, eles recebem como entrada, além do identificador do serviço alvo e do *framework* multi-linguagem, a própria interface de serviço que será inserida ou atualizada. O Gerenciador de Interfaces complementa o trabalho do Servidor de Interface e Arquivos Básicos ao realizar uma análise mais profunda sobre os parâmetros de entrada da requisição antes de processar a operação junto ao Repositório BIP. É responsabilidade dele, por exemplo, verificar se o identificador do

framework multi-linguagem informado é suportado pela arquitetura.

O Provedor de Arquivos Básicos é capaz de executar apenas operações de aquisição dos arquivos básicos de comunicação multi-linguagem. Portanto, o *endpoint /mlfiles* suporta apenas o método GET do HTTP (Tabela 10). Esse método recebe como entrada os identificadores do serviço, do *framework* multi-linguagem e da linguagem de programação do aplicativo cliente (caso seja um Desenvolvedor de Aplicativo) ou do processo servidor (caso seja um Desenvolvedor de Serviço) e retorna, como saída, um arquivo comprimido contendo todos os arquivos básicos de comunicação multi-linguagem implementados na linguagem indicada na entrada. Esses arquivos são gerados com a ajuda do compilador do *framework* multi-linguagem informado como entrada. O Provedor de Arquivos Básicos também realiza algumas verificações nos parâmetros de entrada antes de gerar os arquivos básicos de comunicação multi-linguagem, como verificar se a linguagem de programação é suportada pelo *framework* e se o serviço informado realmente existe no Repositório BIP.

A comunicação entre o Gerenciador de Interfaces e o Provedor de Arquivos Básicos com o Repositório BIP é totalmente realizada por meio do módulo Python Minio. Esse módulo possibilita a realização de operações CRUD (*Create-Request-Update-Delete*) relacionadas às interfaces de serviço. É importante destacar, no entanto, que no caso do Provedor de Arquivos Básicos, apenas a operação *Request* é realizável. Qualquer resultado da operação, falha ou sucesso, é informado ao Servidor de Interface e Arquivos Básicos para, em seguida, ser transmitido ao Desenvolvedor de Aplicativo ou de Serviço em resposta.

O Serviço de Interfaces e Arquivos Básicos foi desenvolvido com base na imagem *python:3.8.16-slim-bullseye* disponível no *Docker Hub*, acrescida dos módulos Flask e Minio. O contêiner *Docker* que hospeda o serviço foi configurado com recursos de 0.25-0.50 de CPU e 128-512 MB de memória principal. Essa configuração foi utilizada como padrão na maioria dos serviços do Lado Servidor. Também foi estabelecida uma dependência entre o Serviço de Interfaces e Arquivos Básicos e o Repositório BIP, onde o serviço só é inicializado quando o contêiner do Repositório BIP estiver ativo e saudável. Essa dependência é justificada pelo fato de que todas as operações do Serviço de Interfaces e Arquivos Básicos só podem ser executadas quando o Repositório BIP está disponível.

5.1.2 Serviço de Projetos

O Serviço de Projetos permite que Desenvolvedores de Serviços administrem seus projetos de processos servidores e que os Desenvolvedores de Aplicativo busquem informações sobre esses projetos. Ele engloba os seguintes componentes da Figura 23: 1) Servidor de Projetos e 2) Gerenciador de Projetos. O Serviço de Projetos também tem relações diretas com o Repositório BIP, onde são armazenados os projetos de processos servidores submetidos à arquitetura, e com o Publicador de Projetos, onde os eventos de criação ou alteração de um projeto são publicados para serem consumidos pelo Serviço de Compilação, visando à sua (*cross*-)compilação e, consequentemente, à produção dos binários.

Em linhas gerais, o Servidor de Projetos atua como um servidor HTTP que espera requisições REST dos Desenvolvedores de Serviços e dos Desenvolvedores de Aplicativos em uma porta padrão, a 50061. É importante ressaltar que apenas os Desenvolvedores de Serviços podem inserir, modificar e excluir projetos de processos servidores. Os Desenvolvedores de Aplicativos podem apenas solicitar dados sobre os projetos. Todas as requisições são encaminhadas ao Gerenciador de Projetos, que interage com o Repositório BIP para realizar a operação desejada. Dentre as operações possíveis, o Gerenciador de Projetos pode requisitar, inserir, atualizar ou excluir projetos de processos servidores específicos. As operações de atualização e inserção geram a publicação desses eventos no Publicador de Projetos, que posteriormente serão consumidos pelo Serviço de Compilação para gerar os binários apropriados.

O Servidor de Projetos foi desenvolvido com o auxílio do *framework* Flask e disponibiliza um único *endpoint*, */projs*, para tratar as requisições relacionadas aos projetos de processos servidores. Antes de serem repassadas ao Gerenciador de Projetos, todas as solicitações passam por uma verificação inicial para garantir que estejam corretamente formatadas. Essa validação é semelhante à do Servidor de Interface e Arquivos Básicos, onde é analisado se o método HTTP é suportado pelo *endpoint* e se a requisição possui todos os parâmetros necessários para ser tratada. Caso algum problema seja identificado, uma mensagem de erro é retornada ao usuário (Desenvolvedor de Serviço ou de Aplicativo) e a computação da requisição é finalizada.

O Gerenciador de Projetos é capaz de executar operações de aquisição, inserção, atualização e exclusão de um projeto de processo servidor. Por conta disso, o *endpoint /projs* oferece suporte aos métodos HTTP: GET, POST, PUT e DELETE (Tabela 11). Os métodos GET e DELETE recebem quatro identificadores como entrada: 1) serviço, 2) *framework* multilinguagem, 3) projeto do processo servidor e 4) linguagem de programação servidora. Os métodos

POST e PUT recebem os mesmos quatro identificadores, além de um arquivo comprimido contendo o projeto do processo servidor como entrada. O método GET é utilizado para obter o projeto que atende aos parâmetros de entrada informados. Os métodos POST, PUT e DELETE, respectivamente, inserem, atualizam e excluem um determinado projeto no Repositório BIP. O Gerenciador de Projetos faz uma análise mais detalhada sobre os parâmetros de entrada, complementando a avaliação feita pelo Servidor de Projetos. Por exemplo, ele verifica se o *framework* informado é suportado pela arquitetura e se a linguagem é suportada pelo *framework*. Caso algum erro seja identificado, uma mensagem de erro é retornada ao Servidor de Projetos e repassada ao usuário.

GET /projs	
Entrada	Identificadores do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*), do projeto (*projectId*) e da linguagem de programação servidora (*languageId*)
Saída	Sucesso → Código 200 + Arquivo com o Projeto do Processo Servidor (comprimido) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!*, *Unknown server language!* ou *Unsuported format of the project ID!*
POST /projs	
Entrada	Identificadores do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*), do projeto (*projectId*), da linguagem de programação servidora (*languageId*) e o Arquivo comprimido com o projeto do processo servidor a ser inserido
Saída	Sucesso → Código 200 + Mensagem (*Service project inserted sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!*, *Unknown server language!*, *Service project already exists!* ou *Unsuported format of the project ID!*
PUT /projs	
Entrada	Identificadores do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*), do projeto (*projectId*), da linguagem de programação servidora (*languageId*) e o Arquivo comprimido com o projeto do processo servidor a ser atualizado
Saída	Sucesso → Código 200 + Mensagem (*Service project updated sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!*, *Unknown server language!*, *Service interface not found!* ou *Unsuported format of the project ID!*
DELETE /projs	
Entrada	Identificadores do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*), do projeto (*projectId*) e da linguagem de programação servidora (*languageId*)
Saída	Sucesso → Código 200 + Mensagem (*Service project deleted sucessfully!*) Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown framework!*, *Unknown server language! ou Unsuported format of the project ID!*

Tabela 11 – Interface REST de interação com o Serviço de Projetos

Especificamente no caso do POST (inserção) e do PUT (atualização), o Gerenciador de Projetos, depois de finalizar a operação no Repositório BIP, também publica informações sobre o projeto no Publicador de Projetos. Esses dados são necessários para o Serviço de Compilação gerar os binários relacionados ao projeto. As informações publicadas são basicamente as mesmas recebidas como parâmetros de entrada do método, ou seja, os identificadores do projeto, do serviço, do *framework*, da linguagem de programação, além do caminho no Repositório BIP onde foi salvo o projeto do processo servidor.

O Serviço de Projetos foi criado a partir da imagem *python:3.8.16-slim-bullseye*, com adição dos módulos Flask, Pika e Minio. O contêiner *Docker* que hospeda o serviço também adotou a configuração padrão (0.25-0.50 de CPU e 128-512 MB de Memória Principal) e possui uma dependência em relação ao Repositório BIP, semelhante ao Serviço de Interfaces e Arquivos Básicos. Da mesma forma, foi registrada uma dependência com o Publicador de Projetos. Isso significa que o Serviço de Projetos só é iniciado quando os contêineres do Repositório BIP e do Publicador de Projetos estão ativos e em perfeito estado de funcionamento.

5.1.3 Serviço de Compilação

O Serviço de Compilação atua na (*cross*-)compilação de projetos recém-armazenados pelo Serviço de Projetos no Repositório BIP. Esse serviço possui só o componente Compilador da Figura 23. Ele interage diretamente com o Repositório BIP, lendo os projetos dos processos do servidor e salvando os binários produzidos pela (*cross*-)compilação, e também se comunica com o Publicador de Projetos, recebendo os avisos de inserção e atualização de projetos de processos servidores.

Em linhas gerais, o Serviço de Compilação se comporta como um assinante junto ao Publicador de Projetos interessado na ocorrência de eventos sobre projetos de processos servidores. O Serviço de Projetos, após inserir ou modificar um projeto de processo servidor no Repositório BIP com sucesso, publica dados essenciais sobre o projeto no Publicador de Projetos, como o identificador do serviço e a linguagem de programação adotada no projeto. Como assinante, o Serviço de Compilação recebe tais informações indiretamente e, de posse delas, (*cross*-)compila o projeto e produz os binários para as arquiteturas suportadas. Cada binário gerado é salvo no Repositório BIP.

Diferentemente dos demais serviços, o Serviço de Compilação não dispõe de uma interface com o ambiente externo, o que significa que não pode ser acessado diretamente por usuários ou processos fora do Lado Servidor. De fato, ele só trabalha quando o registro de um projeto de processo do servidor é divulgado pelo Serviço de Projetos no Publicador de Projetos. Uma outra diferença importante é que podem existir múltiplas instâncias de Compilador ativas simultaneamente. A ideia é que exista uma instância de Compilador para cada linguagem de programação e *framework* multi-linguagem que a arquitetura suporta. Por exemplo, se a arquitetura suporta as linguagens de programação C++, Go e Rust e os *frameworks* Apache Thrift e gRPC-ProtocolBuffer, então existirão seis instâncias de Compilador coexistindo: C++/Apache Thrift,

Rust/Apache Thrift, Go/Apache Thrift, C++/gRPC-ProtocolBuffer, Rust/gRPC-ProtocolBuffer e Go/gRPC-ProtocolBuffer. Cada instância será responsável por (*cross*-)compilar os projetos desenvolvidos na linguagem e *framework* relacionados a ela.

O Serviço de Compilação foi criado a partir da imagem *python:3.8.16-slim-bullseye*, com adição dos módulos Pika e Minio. Também foram adicionados recursos específicos de cada linguagem e *framework* para tornar possível a (*cross*-)compilação dos projetos recém-registrados. O contêiner *Docker* que hospeda o serviço foi configurado diferentemente dos demais serviços. Dado que as ações executadas pelo Compilador são complexas, tomou-se a decisão de duplicar a configuração padrão, isto é, alocando recursos de 0.50-1.00 de CPU e 512 MB-1 GB de Memória Principal. Da mesma forma que ocorre com o Serviço de Projetos, também foi definido que o Serviço de Compilação só inicia após o Repositório BIP e o Publicador de Projetos terem sido inicializados com êxito.

5.1.4 Serviço de Duplicação

O Serviço de Duplicação permite que o Lado Cliente da arquitetura, embarcado em um dispositivo móvel Android, obtenha os binários dos processos servidores em duplicação. Ele é composto por dois elementos da Figura 23: 1) Servidor de Duplicação e 2) Provedor de Binários. O Serviço de Duplicação também possui uma relação direta com o Repositório BIP, de onde obtém os binários dos processos servidores.

O Serviço de Duplicação funciona como um servidor HTTP aguardando requisições REST geradas pelo Lado Cliente em uma porta padrão, a 50062. Cada requisição recebida, se estiver corretamente formatada, é encaminhada ao Provedor de Binários, que busca o binário do processo servidor alvo no Repositório BIP. Caso seja encontrado, o serviço o envia ao Lado Cliente, que o armazena inicialmente na *Cache* de Binários e, posteriormente, o executa localmente, finalizando o procedimento de duplicação.

O Servidor de Binários foi desenvolvido com o auxílio do *framework* Flask e disponibiliza um único *endpoint*, */bins*, para atender às requisições do Lado Cliente sobre a aquisição de binários dos processos servidores. Antes de serem repassadas ao Provedor de Binários, todas as requisições passam por uma validação inicial para garantir que elas estejam bem formatadas. Uma requisição bem formatada inclui quatro identificadores essenciais para a busca do binário: 1) serviço; 2) *framework* multi-linguagem; 3) arquitetura de *hardware* do sistema subordinado; e 4) linguagem de programação servidora. Caso algum problema seja identificado na requisição,

uma mensagem de erro é retornada ao Lado Cliente e o atendimento da requisição é encerrado.

GET /bins	
Entrada	Identificadores do serviço (*serviceId*), da ferramenta multi-linguagem do serviço (*frameworkId*), do projeto (*projectId*), da linguagem de programação servidora (*languageId*) e da arquitetura do sistema subordinado (*architectureId*)
Saída	Sucesso → Código 200 + Binário do Processo Servidor Falha → Código 400 + Mensagem (*Input parameters mismatch!*, *Unknown server language!*, *Unknown framework!*, *Unknown architecture!* ou *Service binary not found!*)

Tabela 12 – Interface REST de interação com o Serviço de Duplicação

O Provedor de Binários está apto a realizar operações apenas de obtenção do binários dos processos servidores. Por conta disso, o *endpoint /bins* suporta somente o método GET do HTTP (Tabela 12). Nesse método único, recebe-se os identificadores do serviço, da linguagem de programação, do *framework* multi-linguagem e da arquitetura do dispositivo cliente como entrada e retorna-se o binário que atende a todos os esses parâmetros. O Provedor de Binários também faz algumas verificações nos parâmetros, antes de retornar os binários alvos, como avaliar se a linguagem de programação ou o *framework* é suportada pelo arquitetura proposta nesse trabalho.

O Serviço de Duplicação foi criado a partir da imagem *python:3.8.16-slim-bullseye*, com adição dos módulos Flask e Minio. O serviço possui uma forte dependência do Repositório BIP, uma vez que é no BIP que estão salvos os binários dos processos servidores. Devido a isso, foi registrada uma dependência ao Repositório BIP. Isso significa que o Serviço de Duplicação só é iniciado quando o contêiner do Repositório BIP está ativo e em perfeito funcionamento. As configurações de recursos são similares às dos demais serviços, exceto o Serviço de Compilação, que por ser central e computacionalmente complexo, tem uma configuração diferenciada.

5.1.5 Serviço de Descoberta

Finalmente, o Serviço de Descoberta opera de maneira simplificada, tendo como principal função fornecer ao Lado Cliente, no início do processo de duplicação, o endereço IP da máquina que hospeda o Lado Servidor. Esse endereço é essencial para que o Lado Cliente consiga interagir com o Serviço de Duplicação para solicitar o binário do processo em duplicação. Para facilitar a localização do Serviço de Descoberta, optou-se por incorporá-lo a um grupo *Multicast* bem-conhecido (224.1.1.1), escutando em uma porta padrão (50063). Portanto, para estabelecer uma conexão com o Serviço de Descoberta, o Lado Cliente simplesmente envia uma mensagem ao grupo *Multicast* e para a porta 50063, obtendo o endereço IP da máquina servidora

como resposta. Devido à sua atuação simplificada, nenhum módulo de terceiros foi empregado. Também foi adotada a imagem *Docker* base e a configuração padrão dos demais serviços.

5.2 Lado Cliente

O Lado Cliente da Figura 23 também expõe os componentes que possibilitam que atores e sistemas subordinados possam interagir com o Lado Servidor da arquitetura. Entretanto, atores e sistemas subordinados têm maneiras diferentes de interagir com o Lado Servidor. A interação dos atores acontece em tempo de desenvolvimento e envolve gerenciar interfaces de serviços ou projetos de processos servidores e solicitar arquivos básicos de comunicação multi-linguagem produzidos pelo Lado Servidor. Já a interação dos sistemas subordinados acontece em tempo de execução e envolve somente solicitar binários gerados pelo Lado Servidor. Por conta disso, criou-se um cliente voltado para os atores e outro para os sistemas subordinados.

Os atores interagem com o Lado Servidor utilizando qualquer cliente HTTP, desde que o cliente ofereça os mecanismos necessários para que o ator elabore e envie uma requisição no formato esperado pela interface REST disponibilizada pelo Lado Servidor. Optou-se pelo desenvolvimento de um *script* Python que, com o módulo *requests* nativo da própria linguagem, envia as requisições HTTP. Essa abordagem foi adotada visando especialmente a realização de testes automatizados. Espera-se que, com a disponibilidade desses testes, os pesquisadores possam modificar a arquitetura interna do Lado Servidor à vontade e, de forma simples e rápida, verificar se tais alterações não comprometeram o funcionamento do Lado Servidor.

Os sistemas subordinados, por outro lado, exigem um cliente mais sofisticado para interagir com o Lado Servidor, de modo que a aplicação interessada na duplicação do processo intervenha o mínimo possível nessa interação. Como a presente pesquisa adotou *smartphones* Android como sistemas subordinados, optou-se por desenvolver o Lado Cliente como um Serviço nesse tipo de plataforma. A documentação oficial do Android[4] apresenta um Serviço como um componente executável em segundo plano e que funciona mesmo quando o usuário não está interagindo com o aplicativo. Portanto, ao fazer do Lado Cliente um Serviço Android, espera-se proporcionar o máximo de autonomia possível a ele. Nesse contexto, foi proposto o Serviço de Gerenciamento de Duplicação que, uma vez configurado, age de forma quase que independente do aplicativo Android que o instanciou. A Figura 24 apresenta o Diagrama de Classes do Serviço de Gerenciamento de Duplicação.

[4] https://developer.android.com/develop/background-work/services?hl=pt-br

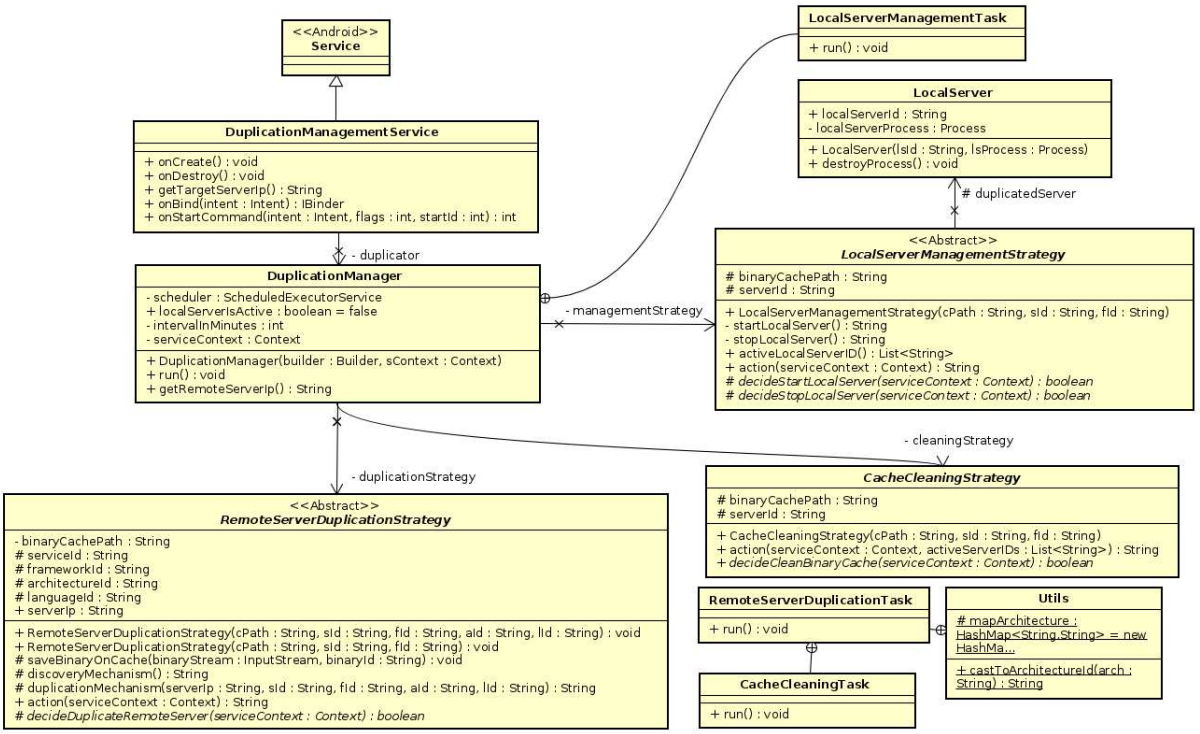


Figura 24 – Diagrama de Classes do Serviço de Gerenciamento de Duplicação

O Serviço de Gerenciamento de Duplicação (*DuplicationManagementService*) possui dois componentes básicos: a *Cache* de Binários e o Gerente de Duplicação. No contexto da *Dew Computing*, esses componentes seriam equivalentes ao *Dew DB* e ao *Dew Server*, ambos apresentados no Capítulo 2. A *Cache* de Binários é uma pasta no sistema de arquivos do próprio *smartphone* onde serão armazenados os binários dos processos servidores duplicados ou em duplicação. É fundamental que o Serviço de Gerenciamento de Duplicação tenha amplos privilégios de leitura, de escrita e, principalmente, de execução dos arquivos persistidos nessa pasta, uma vez que ele deverá salvar e executar os binários dos processos servidores armazenados nela. Já o Gerente de Duplicação (*DuplicationManager*) controla todo o procedimento de duplicação e administra os arquivos da *Cache* de Binários. Em geral, ele realiza três ações principais: 1) Duplicar Servidor Remoto; 2) Administrar Servidor Local e 3) Limpar a *Cache* de Binários.

A atividade de Duplicar Servidor Remoto compreende os passos necessários para salvar o binário do processo servidor remoto na *Cache* de Binários. A primeira ação é decidir quando realizar tal procedimento. Aqui, podem ser considerados os contextos do *smartphone* (por exemplo, o nível de consumo de recursos computacionais ou energéticos) e/ou, especialmente, do ambiente (por exemplo, o nível de latência da rede) para a tomada de decisão. A próxima ação é buscar e identificar a máquina servidora detentora do binário do processo servidor alvo na

rede. Tal busca foi implementada através do envio de mensagens ao grupo *Multicast* do qual o Serviço de Descoberta do Lado Servidor é membro. Com o endereço da máquina servidora, os próximos passos são realizar o *download* do binário alvo, através do envio de uma solicitação ao Serviço de Duplicação do Lado Servidor e, finalmente, salvá-lo na *Cache*. Essa atividade deve ser realizada regularmente, em especial, quando o binário alvo está indisponível localmente.

A atividade de Administrar Servidor Local envolve identificar o melhor momento para iniciar e encerrar um processo servidor local, bem como executar essas ações. É essencial ressaltar que a inicialização de um processo servidor local pressupõe que o binário desse processo já está disponível na *Cache*, o que implica que a atividade de Duplicar Servidor Remoto já foi realizada em algum momento no passado. Considerando aspectos exclusivamente relacionados ao *smartphone*, uma vez que suas ações não dependem do meio externo, a atividade periodicamente determina se aquele instante é adequado para iniciar um novo ou encerrar um processo servidor em execução. Caso positivo, a ação (início ou encerramento) é realizada.

Finalmente, a atividade de Limpar a *Cache* de Binários envolve excluir binários do sistema de arquivos do *smartphone*, com o objetivo de poupar recursos computacionais do dispositivo. Para isso, periodicamente, a atividade é executada e avalia se o binário do processo servidor pode ser excluído da *Cache*. É claro que, para que a ação seja consumada, o binário também não deve estar em uso. A decisão de excluir ou não o binário pode considerar fatores relacionados à rede externa, mas, sobretudo, deve avaliar o contexto atual do *smartphone*, como se o armazenamento dele está completamente cheio ou próximo da saturação.

Atividades	**Passos**	**Métodos**	**Estratégias**
Duplicar Servidor Remoto	Salva o binário do processo servidor remoto na *Cache* de Binários	*saveBinaryOnCache*	*RemoteServerDuplication*
	Descobre o endereço IP da máquina servidora com o binário alvo.	*discoveryMechanism*	
	Obtém o binário alvo junto a máquina servidora recém-descoberta.	*duplicationMechanism*	
	Determina se o processo servidor remoto deve ser duplicado.	*decideDuplicateRemoteServer*	
Administrar Servidor Local	Inicia o processo servidor local.	*startLocalServer*	*LocalServerManagement*
	Encerra o processo servidor local.	*stopLocalServer*	
	Determina se o processo servidor local deve ser inicializado.	*decideStartLocalServer*	
	Determina se o processo servidor local deve ser encerrado.	*decideStopLocalServer*	
Limpar *Cache* de Binários	Determina se o binário do processo servidor deve ser removido.	*decideCleanBinaryCache*	*CacheCleaningStrategy*

Tabela 13 – Resumo sobre os passos e métodos das atividades do Gerente de Duplicação

Cada atividade é executada por uma *thread* específica, instanciada durante a inicialização do Gerente de Duplicação. O comportamento de cada *thread* é determinado dinamicamente durante a configuração do Gerente de Duplicação, por meio de estratégias. A Tabela 13 relaciona os passos de cada atividade aos respectivos métodos de cada estratégia. Em geral, os métodos de decisão são abstratos e devem ser implementados antes de serem usados. Além das estratégias, o Gerente de Duplicação também precisa saber a frequência com que cada atividade será realizada. Assim, durante a configuração do serviço, também deve-se informar os intervalos de tempo em cada *thread* será escalonada para execução. Por padrão, as atividades são realizadas a cada N minutos, exceto a atividade Limpar a *Cache* de Binários que ocorre a cada N/3 minutos.

Algoritmo 1 Método principal de ação da classe *LocalServerManagementStrategy*

Entrada: $serviceContext$
Saída: $result$

```
1:  result ← " "
2:  if decideDuplicateRemoteServer(serviceContext) then
3:      serverIp ← discoveryMechanism()
4:      if serverIp ≠ vazio then
5:          serverName ← duplicationMechanism(serverIp)
6:          if serverName ≠ null then
7:              result ← "{serviceId} was cached succesfully!"
8:          else
9:              result ← "{serviceId} was not founded! A problem occured!"
10:         end if
11:     else
12:         result ← "Remote server process not found!"
13:     end if
14: else
15:     result ← "No duplication!"
16: end if
17: return result
```

A estratégia responsável pela atividade de Duplicar Servidor Remoto do Gerente de Duplicação (Tabela 13) é implementada na classe abstrata *RemoteServerDuplicationStrategy*. A classe possui um construtor que recebe três parâmetros de entrada: 1) o caminho no sistema de arquivos da *Cache* de Binários e os identificadores 2) do serviço associado do aplicativo cliente e 3) do *framework* multi-linguagem base para a comunicação. O comportamento do método que executa a ação da estratégia é detalhado no Algoritmo 1. A estratégia inicia invocando o método *decideDuplicateRemoteServer* para determinar se o processo servidor remoto deve ser duplicado (Linha 2). Para isso, o método recebe como entrada o contexto do dispositivo e retorna um valor

booleano indicando se a duplicação deve ser iniciada ou não. Caso positivo, a estratégia invoca o método *discoveryMechanism* (Linha 3) para encontrar o IP da máquina servidora com o binário alvo. De posse do endereço, a estratégia chama o método *duplicationMechanism* (Linha 5) que realiza o *download* do binário e o salva na *Cache*. As ações dessa atividade serão explicadas com maiores detalhes na Seção 5.3.

A estratégia responsável pela atividade de Administrar Servidor Local do Gerente de Duplicação (Tabela 13) é implementada na classe abstrata *LocalServerManagementStrategy*. A classe tem um construtor que recebe 1) o caminho no sistema de arquivos da *Cache* de Binários e 2) o identificador do serviço associado ao aplicativo cliente como entrada. A estratégia inicia invocando o método *decideStartLocalServer* para determinar se deve iniciar um processo servidor já duplicado, ou seja, cujo binário já está na *Cache*. Caso positivo, ela chama o método *startLocalServer* e inicia o processo. Em seguida, ela realiza uma ação similar, mas para encerrar um processo servidor. Portanto, ela invoca, nessa ordem, os métodos *decideStopLocalServer* e *stopLocalServer*. Ambos os métodos de decisão recebem como entrada o contexto do dispositivo móvel e retornam um *booleano* indicando se o processo servidor deve ser iniciado (*decideStartLocalServer*) ou encerrado (*decideStopLocalServer*).

A estratégia responsável pela atividade de Limpar a *Cache* de Binários do Gerente de Duplicação (Tabela 13) é implementada na classe abstrata *CacheCleaningStrategy*. A classe possui um construtor que recebe 1) o caminho no sistema de arquivos da *Cache* de Binários e 2) o identificador do serviço associado ao aplicativo cliente como entrada. O comportamento do método que executa a ação da estratégia é detalhado no Algoritmo 2. A estratégia inicia avaliando se há algum processo ativo criado com o binário candidato à exclusão (Linha 2). Caso negativo, ela consulta o método *decideCleanBinaryCache* (Linha 3), que, com base no contexto do dispositivo móvel, retorna um *booleano* indicando se o binário deve ser excluído ou não. A estratégia finaliza removendo ou não o binário (Linha 4).

5.3 Descrição detalhada do funcionamento da arquitetura implementada

Com o objetivo de fornecer uma compreensão mais profunda sobre a arquitetura implementada neste trabalho, esta seção apresenta uma explicação sobre as principais interações da arquitetura com seus atores e com um dos sistemas subordinados apresentados anteriormente, o *smartphone* Android. Para isso, adotou-se um exemplo envolvendo três pessoas: Ana, no papel de Desenvolvedora de Serviço; Bob, como o Desenvolvedor de Aplicativo; e Cid, o dono de um

Algoritmo 2 Método principal de ação da classe *CacheCleaningStrategy*

Entrada: *serviceContext*, *Lista de Servidores Locais Ativos*
Saída: *result*

```
1:  result ← " "
2:  if serverId ∄ Lista de Servidores Locais Ativos then
3:     if decideCleanBinaryCache(serviceContext) then
4:        if deleteBinaryFile(serverId) then
5:           result ← "{serverId} was deleted from cache succesfully!"
6:        else
7:           result ← "{serverId} was not deleted! A problem occured!"
8:        end if
9:     else
10:       result ← "{serverId} was not deleted! It is not time to do this!"
11:    end if
12: else
13:    result ← "{serverId} cannot be deleted! Server is running yet!"
14: end if
15: return result
```

smartphone Android e usuário do aplicativo desenvolvido por Bob.

A seção foi dividida em duas partes para facilitar a compreensão do leitor. A primeira parte trata das ações durante a concepção do aplicativo e envolve apenas Ana e Bob. A segunda parte mostra o uso da arquitetura com o aplicativo já criado e envolve apenas Cid. Enquanto a primeira parte aborda ações relacionadas ao registro de interfaces de serviços, obtenção de arquivos básicos, registro de projetos de processo servidor e geração de binários de processo servidor, a segunda se concentra no procedimento de duplicação do processo servidor remoto.

5.3.1 Fase de desenvolvimento

Bob está desenvolvendo um aplicativo para manipular fotos e deseja usar a arquitetura proposta como uma fonte de servidores multi-linguagens para processar tarefas do aplicativo, como aplicar filtros em fotos. Inicialmente, ele cria uma interface que descreva o serviço desejado. Para isso, ele pode usar um formulário web ou um editor de texto convencional para descrever a estrutura da interface obedecendo a sintaxe de uma ferramenta de comunicação multi-linguagem, como o Apache Thrift. Nessa descrição, Bob indica o identificador do serviço (por exemplo, ImageFilterService) e as assinaturas dos métodos, onde cada método representa uma tarefa do aplicativo computável pelo serviço. O Código-Fonte 1 exemplifica uma possível interface de serviço produzida por Bob, onde as Linhas 3 à 5 definem três procedimentos invocáveis pelo processo servidor a ser desenvolvido.

Código-fonte 1 – Exemplo de interface de serviço baseada no Apache Thrift

```
1  // Based on Apache Thrift syntax
2  service ImageFilterService {
3    binary grayscaleFilter(1: binary img)
4    binary cartoonFilter(1: binary img)
5    binary pencilFilter(1: binary img, 2: i32 blurFactor)
6  }
```

Concluída a interface de serviço, Bob deve submetê-la ao Servidor de Interfaces e Arquivos Básicos por meio de uma requisição HTTP POST a fim de registrá-la. A Figura 25 ilustra todo este procedimento feito por Bob através de um Diagrama de Sequência. Ao receber a requisição, o componente deve realizar uma série de verificações, como identificar a ausência de algum parâmetro obrigatório e se a ferramenta de comunicação multi-linguagem é suportada. Em caso de erro, o Servidor de Interfaces e Arquivos Básicos deve retornar a Bob uma mensagem indicando o motivo da falha. Em caso de sucesso, ele deve repassar os dados ao Gerenciador de Interfaces, que deve armazená-los no Repositório BIP. Ao final do procedimento, Bob é notificado sobre o resultado da operação.

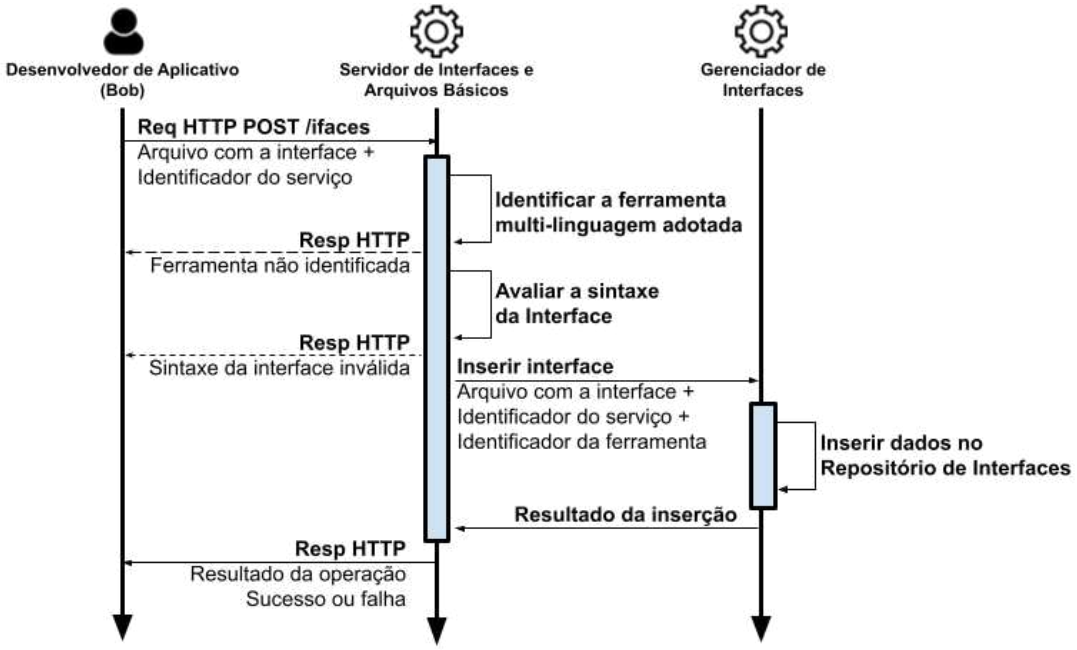


Figura 25 – Ações relacionadas ao registro de uma interface de serviço

Ana está interessada em desenvolver o serviço solicitado por Bob, adotando uma linguagem de programação Y (Go, por exemplo). Inicialmente, ela precisa adquirir os arquivos básicos de comunicação multi-linguagem junto a arquitetura, com o propósito de facilitar a

interação entre o processo servidor que ela está desenvolvendo e o aplicativo cliente de Bob. A Figura 26 também descreve as ações necessárias para que Ana obtenha os arquivos básicos através de um Diagrama de Sequência. Ana interage com o Servidor de Interfaces e Arquivos Básicos através do envio de uma requisição HTTP GET contendo os identificadores do serviço de Bob, da linguagem Y (Go) e da ferramenta multi-linguagem (Apache Thrift). Após algumas verificações (omitidas na Figura), o Servidor de Interfaces e Arquivos Básicos repassa os dados ao Provedor de Arquivos Básicos.

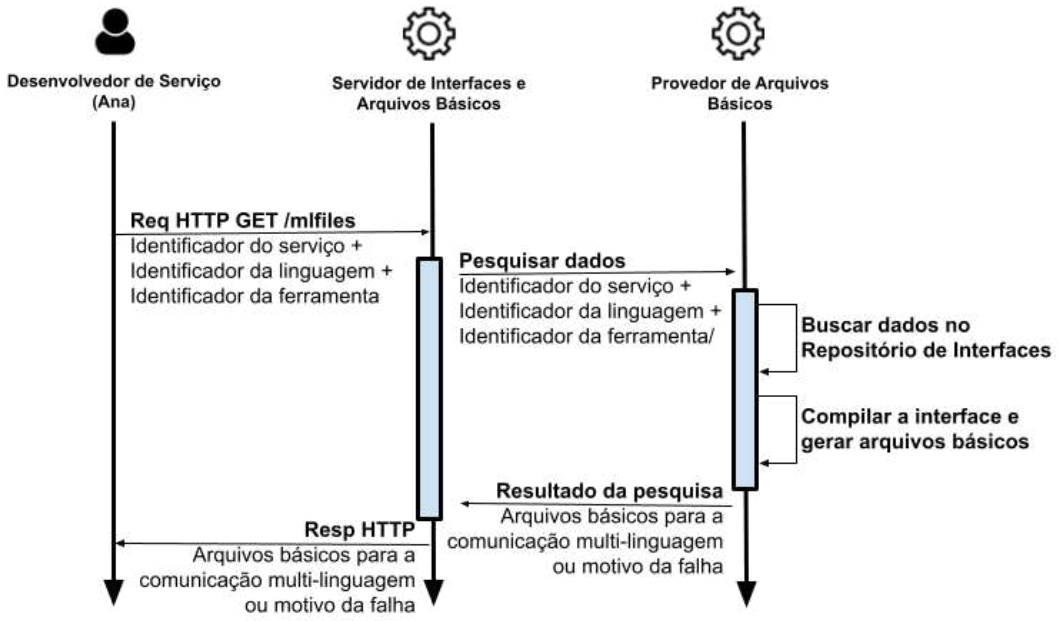


Figura 26 – Ações relacionadas a obtenção dos arquivos básicos de comunicação multi-linguagem

De posse dos dados, o Provedor de Arquivos Básicos consulta o Repositório BIP e resgata a interface de serviço associada ao identificador do serviço de Bob. Com tais informações, o Provedor de Arquivos Básicos cria os arquivos básicos de comunicação multi-linguagem ao submeter a interface do serviço e o identificador da linguagem Y ao compilador da ferramenta multi-linguagem. Em seguida, os arquivos básicos são repassados ao Servidor de Interfaces e Arquivos Básicos, que os devolve para Ana para criação de um projeto e desenvolvimento do processo servidor. Caso ocorra algum tipo de falha durante o procedimento, o Servidor de Interfaces e Arquivos Básicos deve ser notificado e a causa do erro deve ser repassada a Ana. Analogamente, Bob adquire os arquivos básicos na linguagem adotada por ele (por exemplo, Java) e os integra ao seu aplicativo.

Há um fluxo alternativo (não ilustrado) no qual, ao invés de Bob manifestar interesse pelo serviço e Ana implementá-lo, Ana já dispõe de um serviço (por exemplo, ServAna) que

contém todos os métodos úteis para o aplicativo de Bob. Nessa caso, Bob não precisa registrar uma nova interface, mas apenas consumir ServAna. O procedimento é muito parecido com o descrito anteriormente. A diferença está em quem o inicia, pois, como Ana já possui o serviço, ela deve informar a interface do serviço ao Repositório BIP, enquanto Bob precisa apenas obter os arquivos básicos e integrá-los ao aplicativo cliente em desenvolvimento.

De posse dos arquivos básicos, Bob pode então criar classes que permitam a interação do seu aplicativo de imagens com processos servidores através do Apache Thrift. Além disso, Bob também precisa inicializar o Serviço de Gerenciamento de Duplicação que será o responsável por conversar com os Serviços de Descoberta e de Duplicação do lado servidor para duplicar, de forma automatizada, o processo servidor remoto quando necessário. Para inicializá-lo, Bob deve, primeiramente, anexar ao seu aplicativo a biblioteca do Serviço e criar um objeto *Builder* contendo as classes concretas que implementam as estratégias discutidas anteriormente e o intervalo de tempo (em minutos) desejado para executar as atividades de Duplicar Servidor Remoto e Administrar Servidor Local. Tal objeto deve ser informado ao Serviço de Gerenciamento de Duplicação durante a sua inicialização via *Intent* o como valor do identificador *duplicatorConf*. Uma vez inicializado, o Serviço de Gerenciamento de Duplicação age de forma independente do aplicativo de Bob, sempre seguindo as estratégias previamente configuradas. O Código-fonte 2 apresenta um trecho de código que exemplifica o procedimento descrito nesse parágrafo, onde as Linhas 1 à 16 compreendem a criação do objeto *Builder* e a Linha 18 indica a passagem do objeto criado através do identificador supracitado

Código-fonte 2 – Trecho de código que exemplifica como inicializar o Serviço de Gerenciamento de Duplicação na plataforma Android

```
1  String cachePath = "/data/data/com.app.image/cache";
2
3  DuplicationManager.Builder configuration = new DuplicationManager.Builder()
4          .setManagementStrategy(
5              new LocalServerManagementLazy(cachePath,
6                                  "ImageFilterService",
7                                  "Thrift"))
8          .setDuplicationStrategy(
9              new RemoteServerDuplicationLazy(cachePath,
10                                 "ImageFilterService",
11                                 "Thrift"))
12         .setCleaningStrategy(
13             new CacheCleaningLazy(cachePath,
14                         "ImageFilterService",
```

```
15                                  "Thrift"))
16              .setDuplicatorInterval(2);
17
18  dmService.putExtra("duplicatorConf", configuration);
```

Ana desenvolveu um servidor que interessa a Bob em uma linguagem de programação Y (Go, por exemplo). Portanto, Ana deve registrá-lo na arquitetura para que os binários possam ser produzidos e disponibilizados ao aplicativo cliente. A Figura 27 ilustra o processo de cadastro de um projeto de servidor na arquitetura proposta através de um Diagrama de Sequência. De início, Ana envia uma requisição HTTP POST com o projeto de servidor para o Servidor de Projetos, juntamente com algumas informações adicionais, como a linguagem e o *framework* utilizados durante o desenvolvimento. Ao receber a requisição, o Servidor de Projetos faz algumas verificações gerais, como se falta algum parâmetro obrigatório na requisição. Se estiver tudo em conformidade, ele encaminha todos os dados ao Gerenciador de Projetos.

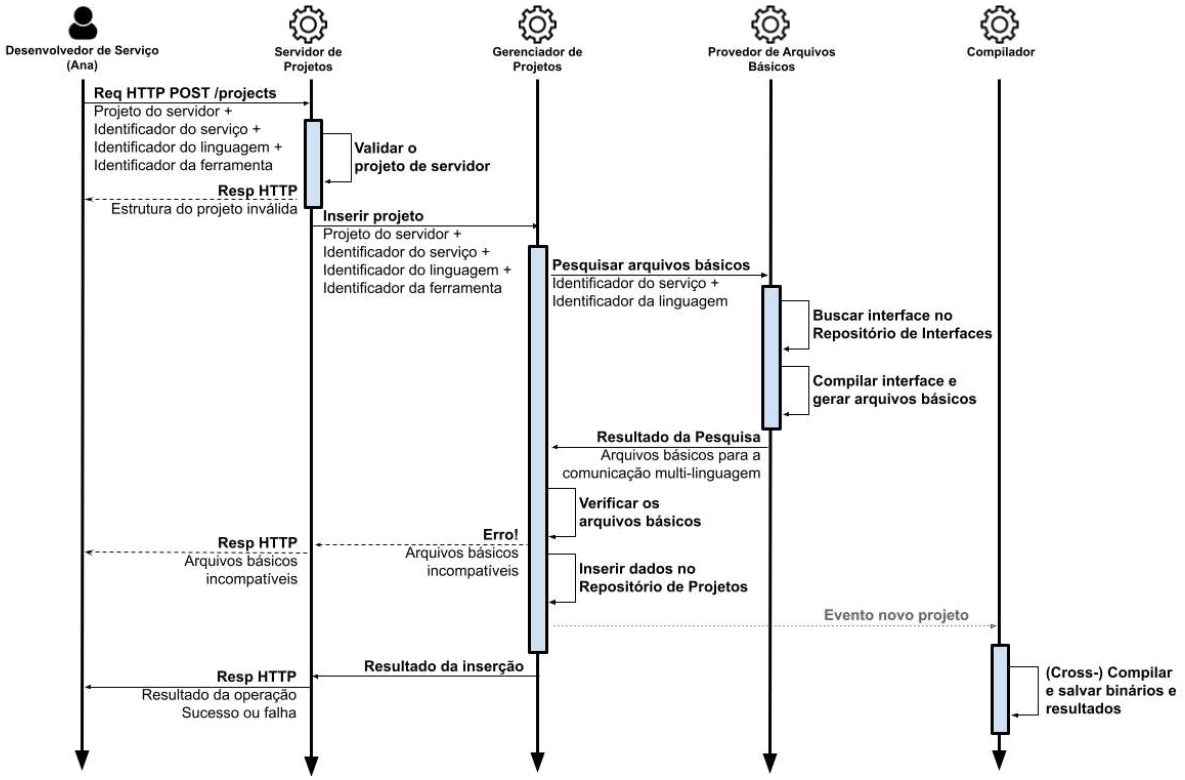


Figura 27 – Ações relacionadas ao registro de projeto de processo servidor

O Gerenciador de Projetos, por sua vez, verifica se aquele projeto está associado a uma interface de serviço conhecida. Para isso, consulta o Provedor de Arquivos Básicos, informado o identificador do serviço e obtém os arquivos básicos de comunicação multi-linguagem. Caso a consulta retorne sucesso, assume-se que o projeto está relacionado a um serviço cadas-

trado. Em seguida, o projeto deve ser armazenado no Repositório BIP. Ana é notificada sobre o sucesso da operação ou sobre o motivo do erro, caso ocorra alguma falha.

Ao inserir um projeto no Repositório BIP, uma notificação é publicada pelo Gerenciador de Projetos e consumida pelo Compilador por intermédio do Publicador de Projetos. Essa notificação contém todas as informações necessárias para que o Compilador (*cross-*)compile o novo projeto. Esse alerta leva o Compilador a recuperar o projeto e os demais dados associados a ele, como os identificadores do serviço, a linguagem e a ferramenta multi-linguagem usados no desenvolvimento. Com todos esses dados, o Compilador (*cross-*)compila o projeto utilizando um compilador previamente configurado para a linguagem em questão. Os binários gerados são salvos no Repositório BIP também.

5.3.2 Fase de Execução

Cid fez o *download* do aplicativo de imagens criado por Bob em seu *smartphone*. Durante a inicialização, o aplicativo configura automaticamente o Serviço de Gerenciamento de Duplicação, sem a necessidade de qualquer ação por parte de Cid. Essa configuração já foi discutida na Seção 5.3.1. Por meio de um Diagrama de Sequência, a Figura 28 ilustra o procedimento de duplicação do processo servidor, que será melhor explicado a seguir.

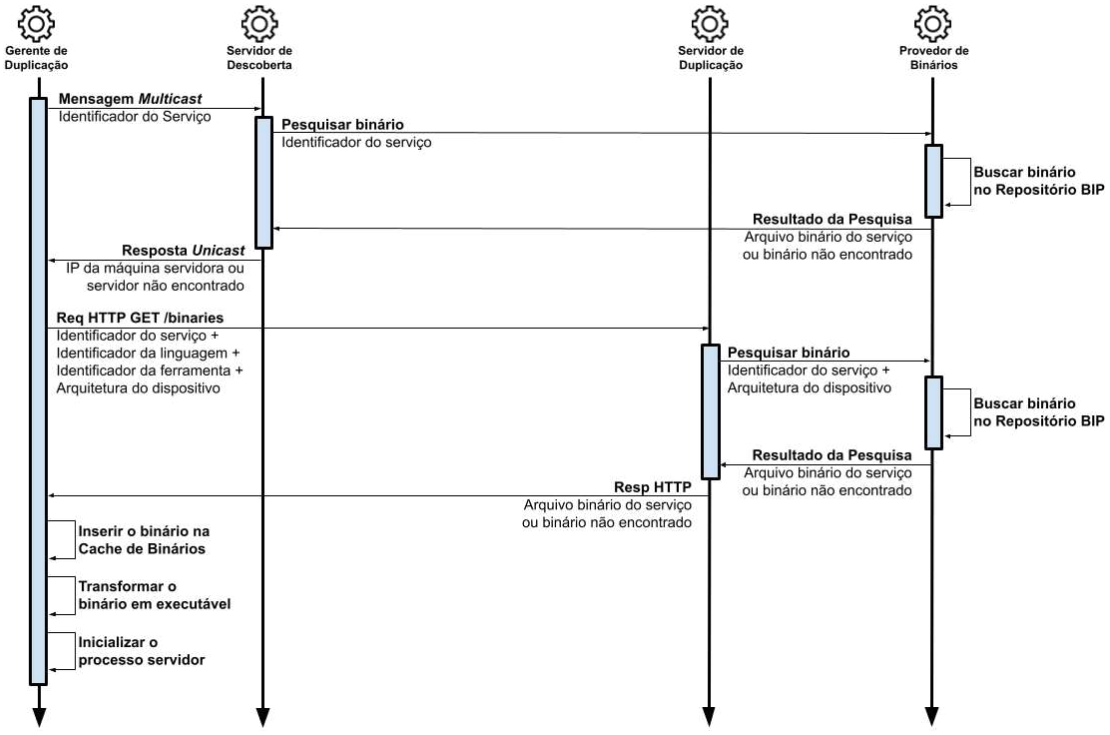


Figura 28 – Ações relacionadas a duplicação de processos servidores remotos

O Gerente de Duplicação, seguindo a Estratégia de Duplicação de Servidor Remoto, determina o melhor momento para duplicar um processo servidor remoto. Para isso, ele analisa o contexto do *smartphone* de Cid periodicamente e, quando a condição definida pela estratégia é atendida, ele inicia o procedimento de duplicação. Diversas métricas podem ser consideradas, em grupo ou separadamente, tais como o nível de intensidade do sinal, de consumo dos recursos computacionais e/ou energéticos do *smartphone* e de largura de banda disponível. A Figura 28 resume os passos necessários para duplicar um processo servidor remoto.

A duplicação começa com a descoberta do endereço IP da máquina servidora que armazena o binário do processo servidor do aplicativo de imagens. Ainda executando a Estratégia de Duplicação de Servidor Remoto, o Gerenciador de Projetos envia mensagens *multicast* com o identificador do serviço desejado. Ao receber uma delas, o Servidor de Descoberta consulta o Provedor de Binários e confirma a disponibilidade do serviço identificado. Caso disponível, o Servidor de Descoberta responde ao Gerente de Duplicação com o IP da máquina servidora que o hospeda. Em caso de erro, o Servidor de Descoberta deve responder com o motivo da falha.

Em seguida, o Gerente de Duplicação, ainda executando a Estratégia de Duplicação de Servidor Remoto, faz o *download* do binário interagindo com o Servidor de Duplicação. Tal interação consiste em enviar uma requisição HTTP GET para o endereço IP obtido na etapa anterior, contendo os identificadores do serviço, da arquitetura de *hardware* do *smartphone*, do *framework* multi-linguagem e da linguagem servidora alvo. Em caso de sucesso, o Servidor de Duplicação responde ao Gerente de Duplicação com o binário do processo servidor que, por sua vez, o salva na *Cache* de Binários e o transforma em executável. Em caso de erro, o motivo da falha deve ser informada pelo Servidor de Duplicação em sua resposta.

O Gerente de Duplicação, agora seguindo a Estratégia de Gerenciamento de Servidor Local, pode inicializar o processo servidor local e encerrar o procedimento de duplicação. Porém, antes de iniciá-lo, o Gerente de Duplicação avalia o contexto do *smartphone* de Cid. Essa análise consiste em verificar se o sistema tem recursos computacionais e/ou energéticos suficientes para hospedar um novo processo. Caso negativo, o procedimento de duplicação é suspenso. Contudo, o binário não é excluído da *Cache*. Isso significa que o Gerente de Duplicação pode, no futuro, repetir a análise e tentar inicializar o processo servidor local novamente. Se o dispositivo móvel tiver recursos suficientes, o binário é executado e o processo servidor local fica ativo. Uma vez ativo, o procedimento de duplicação é encerrado.

Enquanto ativo, o processo servidor local pode receber requisições do aplicativo de

imagens através da interface de *loopback* do sistema e da porta padrão do serviço. O Gerente de Duplicação também tem o poder de encerrar o processo servidor local, mais uma vez, analizando o contexto do *smartphone* de Cid através da Estratégia de Gerenciamento de Servidor Local. Considera-se um momento adequado quando o dispositivo móvel está sobrecarregado e dispõe de poucos recursos computacionais e/ou energéticos.

O Gerente de Duplicação, seguindo a Estratégia de Limpeza de Cache, também define quando liberar espaço na *Cache* de Binários. Para isso, ele verifica se o critério de exclusão definido na estratégia foi atendido periodicamente. Caso positivo, antes de excluir o binário, ele verifica se o processo servidor local ainda está ativo. Se ativo, a remoção do binário é suspensa. Caso contrário, ela é consumada.

5.4 Considerações Finais

Neste capítulo, foram analisados os padrões e tecnologias que sustentam a arquitetura implementada, bem como as decisões de projeto que guiaram seu desenvolvimento, alinhando-a aos requisitos do sistema e às melhores práticas. A versão implementada difere da conceitual, pois foi ajustada para ser uma versão mínima, mas funcional, adequada para testes de prova de conceito. Essas modificações foram feitas para garantir a eficácia na fase inicial de experimentação, cujo detalhamento será abordado no próximo capítulo, que explorará a aplicação prática e a validação da arquitetura.

As decisões de projeto e a seleção das tecnologias refletem um compromisso integral com a criação de uma solução robusta, eficiente e facilmente adaptável. Desde o início, o objetivo central desta tese de Doutorado foi desenvolver uma arquitetura flexível, destinada a ser adotada e personalizada pela comunidade científica para otimizar o processamento de tarefas em dispositivos móveis. Este enfoque oferece não apenas uma alternativa ao *offloading* computacional tradicional, mas também complementa suas capacidades, ampliando as possibilidades de aplicação e adaptação conforme as necessidades emergentes.

Antecipa-se simplificações introduzidas na versão implementada, sejam alvo de investigações e refinamentos contínuos à medida que a arquitetura DADOS evolua através de futuras pesquisas. Por exemplo, as atividades de “Verificar a estrutura do projeto” e “Verificar os arquivos básicos”, atribuídas respectivamente ao Servidor de Projetos e ao Gerenciador de Projetos e apresentadas Figura 27, foram abstraídas. Isso significa que os métodos responsáveis por essas ações até estão presentes no código-fonte, mas não foram implementados de forma

completa, sempre retornando verdadeiro quando invocados. Ao promover um ambiente colaborativo e inovador, este trabalho visa não apenas aperfeiçoar a arquitetura proposta, mas também estimular novas descobertas e avanços significativos na eficiência computacional em dispositivos móveis.

6 EXPERIMENTOS E RESULTADOS

Este capitulo apresenta os experimentos conduzidos para avaliar a viabilidade da implementação da arquitetura DADOS e os potenciais impactos da sua adoção em um cenário tipico de Computação Móvel. A Seção 6.1 descreve o ambiente criado para a realização dos experimentos, apresentando, inicialmente, as configurações dos equipamentos utilizados, as métricas adotadas e a aplicação escolhida para os experimentos. Particularmente, as Subseções 6.1.1 e 6.1.2 apresentam e discutem os principais resultados obtidos nos dois cenários utilizados como referência na condução dos experimentos. Por fim, a Seção 6.2 realiza uma análise crítica dos resultados obtidos e resume as contribuições que a arquitetura proporciona à literatura.

6.1 Descrição dos Experimentos

O ambiente montado para os experimentos foi composto por quatro dispositivos conectados por meio de uma rede sem fio dedicada e montada em uma topologia estrela. Dos quatro dispositivos, três são *smartphones* clientes, enquanto o último é um *notebook* servidor atuando como um *Cloudlet* de um provedor da Nuvem. O *notebook* hospeda todos os serviços que compõem o lado servidor da arquitetura proposta, além do processo apto a computar o *offloading* dos *smartphones* clientes. Já os *smartphones* hospedam a aplicação usada nos experimentos e um Serviço Android capaz de duplicar localmente o processo do servidor remoto. Tal serviço representa o lado cliente da arquitetura proposta. Assume-se que o lado servidor já possui todos os binários dos processos que implementam o serviço requerido pela aplicação cliente. A Tabela 14 resume as configurações dos experimentos realizados.

Em termos de *software*, foi escolhida uma aplicação de processamento de imagem digital conhecida como BenchImage. Essa aplicação possibilita aos usuários aplicar filtros em imagens pré-definidas de diferentes resoluções. A versão original do aplicativo foi modificada para operar com o *framework* multi-linguagem Apache Thrift durante o *offloading* e para iniciar o Serviço Android capaz de duplicar processos servidores. O aplicativo cliente e os processos servidores foram desenvolvidos utilizando os mesmos algoritmos, sem nenhum tipo de paralelismo e com as linguagens de programação Java e Go, respectivamente. A ideia é que os mecanismos avaliados processem igualmente as mesmas tarefas. As tarefas consistem em aplicar filtros na imagem *SkyLine* de baixa (1 MP) e alta resolução (8 MP). Foram adotados dois tipos de filtros: *GrayScale* e *Pencil*. O primeiro foi escolhido por ser um filtro básico, mas bastante

Objetivo(s)	Realizar experimentos de Prova de Conceito com a arquitetura e avaliar os impactos que ela pode proporcionar no processamento de tarefas em uma aplicação móvel.
Sistema (dispositivos utilizados)	**MotorolaGPlay/Cliente** com Qualcomm Snapdragon (1.2GHz, Quad Core), 2GB RAM e Android 7, **SamsungJ5/Cliente** com Qualcomm Snapdragon (1.2GHz, Quad Core), 1.5GB RAM e Android 6, **MotorolaE6/Cliente** com MediaTek (2GHz, Octa Core), 2GB RAM e Android 9, **Notebook/Cloudlet** com Intel Core i7, 12GB RAM e Ubuntu 22.04, **Roteador Netgear WGR612** para construir uma rede sem fio exclusiva de 2,4 GHz entre os dispositivos.
Fatores/ Níveis	Cenários (Dispositivos Juntos e e Dispositivos Separados) Tipos de Processamento (*Local*, *Dew* e *Cloudlet*), Linguagens de Níveis de Programação (Go e Java), Resolução de imagem (1 MP e 8 MP) e Filtro de imagem (*GrayScale* e *Pencil*).
Iterações	Cada experimento foi realizado 50 vezes para cada combinação de Fatores/Níveis; Dentre os resultados obtidos, foram removidos os *outliers* e selecionadas as 40 amostras mais rápidas.
Métricas avaliadas	Tempo de Reposta (tempo necessário para aplicar o filtro na imagem), Consumo de Rede (quantidade de *bytes* transmitidos na rede durante o processamento) e Consumo de Energia (energia dispendida para aplicar o filtro na imagem).

Tabela 14 – Detalhes sobre a configuração dos experimentos realizados

utilizado em operações de processamento de imagem, além de muito usado em pesquisas que avaliam o desempenho do *offloading* computacional. Já o segundo foi selecionado por ser uma composição de dois filtros (*Gaussian* e *Sobel*) aplicados sequencialmente, o que exige mais recursos para computar a tarefa.

Também foi introduzido um novo mecanismo de computação de tarefas, a abordagem *Dew*, onde cada tarefa é processada por um processo servidor recém-duplicado pelo próprio dispositivo cliente. Portanto, o dispositivo cliente só realiza operações na rede externa enquanto duplica o processo servidor hospedado inicialmente no *notebook*. Uma vez em execução, o processo servidor duplicado atende às requisições do aplicativo cliente através da interface de *loopback* do sistema e não realiza mais ações na rede externa. Os demais mecanismos são as abordagens *Local* (realizada pelo próprio aplicativo cliente) e *Cloudlet* (realizada pelo processo servidor hospedado no *notebook*, via *offloading* multi-linguagem).

Nesse estudo, foram avaliadas as métricas Tempo de Resposta, Consumo de Energia e Consumo de Rede. O Tempo de Resposta compreende o tempo necessário somente para aplicar o filtro na imagem. Isso significa que, na abordagem *Dew*, essa métrica não considera o tempo dispendido para duplicar o processo servidor remoto. Em geral, em experimentos iniciais, foi observado que os dispositivos consomem cerca de 50 segundos para duplicar o processo servidor remoto. O Consumo de Energia compreende a energia dispendida para aplicar o filtro, também sem considerar a etapa de duplicação. O Consumo de Rede representa a quantidade de bytes transmitidos na rede durante a execução das abordagens *Dew* e *Cloudlet*. A abordagem *Local* foi desconsiderada dessa métrica, pois ela não realiza nenhuma operação na rede durante a computação da tarefa. Os resultados da métrica Tempo de Resposta foram obtidos na própria

aplicação, ao passo que os resultados da métrica de Consumo de Energia foram obtidos através da ferramenta *dumpsys*. Por fim, os resultados da métrica Consumo de Rede foram coletados com a ferramenta Wireshark[1], executando no servidor e monitorando a porta padrão conforme o tipo de processamento.

Todos os experimentos foram executados 50 vezes com o auxílio da ferramenta Ebserver (OLIVEIRA *et al.*, 2023). Tal ferramenta foi especialmente útil para iniciar a execução simultânea dos *smartphones* no Cenário 2 e para coletar os resultados do Consumo de Energia em ambos os cenários. Dos resultados coletados, foram removidos todos identificados como *outliers* com base na métrica em análise. Neste trabalho, um resultado *outlier* é definido como aquele que está fora da faixa interquartil (IQR), ou seja, acima de $Q3 + 1,5IQR$ ou abaixo de $Q1 - 1,5IQR$. Dos resultados restantes, foram escolhidas as 40 menores entradas com base na mesma métrica. Os resultados são apresentados seguindo dois cenários distintos detalhados nas seções seguintes. Além disso, foram aplicados testes estatísticos e análises *post-hoc* para conferir robustez e precisão estatística aos resultados apresentados em cada métrica. Nas métricas de Tempo de Resposta e Consumo de Energia foram aplicados o teste de Kruskal-Wallis seguido pelo teste *post-hoc* Nemenyi. Já na métrica Consumo de Rede foi aplicado somente o teste Mann-Whitney.

6.1.1 Cenário 1: Dispositivos Separados

O primeiro cenário comparou o desempenho das abordagens considerando cada *smartphone* separadamente (Figura 29). A ideia foi observar o comportamento da abordagem *Dew* em um ambiente com baixa concorrência de acesso à rede sem fio, o que aumenta a largura de banda disponível para a realização de operações remotas. Contudo, é relevante destacar que a rede não foi isolada de interferências externas. Portanto, uma parte da largura de banda foi naturalmente dedicada a eventuais retransmissões em resposta às colisões de sinais causadas por tais interferências. Nesse cenário, a máquina servidora também sofre menos sobrecarga durante a computação do *offloading* na abordagem *Cloudlet*.

Uma rede sem fio com uma boa largura de banda disponível é benéfica para o *offloading* computacional, uma vez que essa técnica se baseia na transmissão de dados através da rede. Assim, quanto maior a largura de banda, mais dados são transmitidos e o *offloading* tende a ser mais rápido. É fato que a abordagem *Dew* também se beneficia da maior largura de banda,

[1] https://www.wireshark.org/

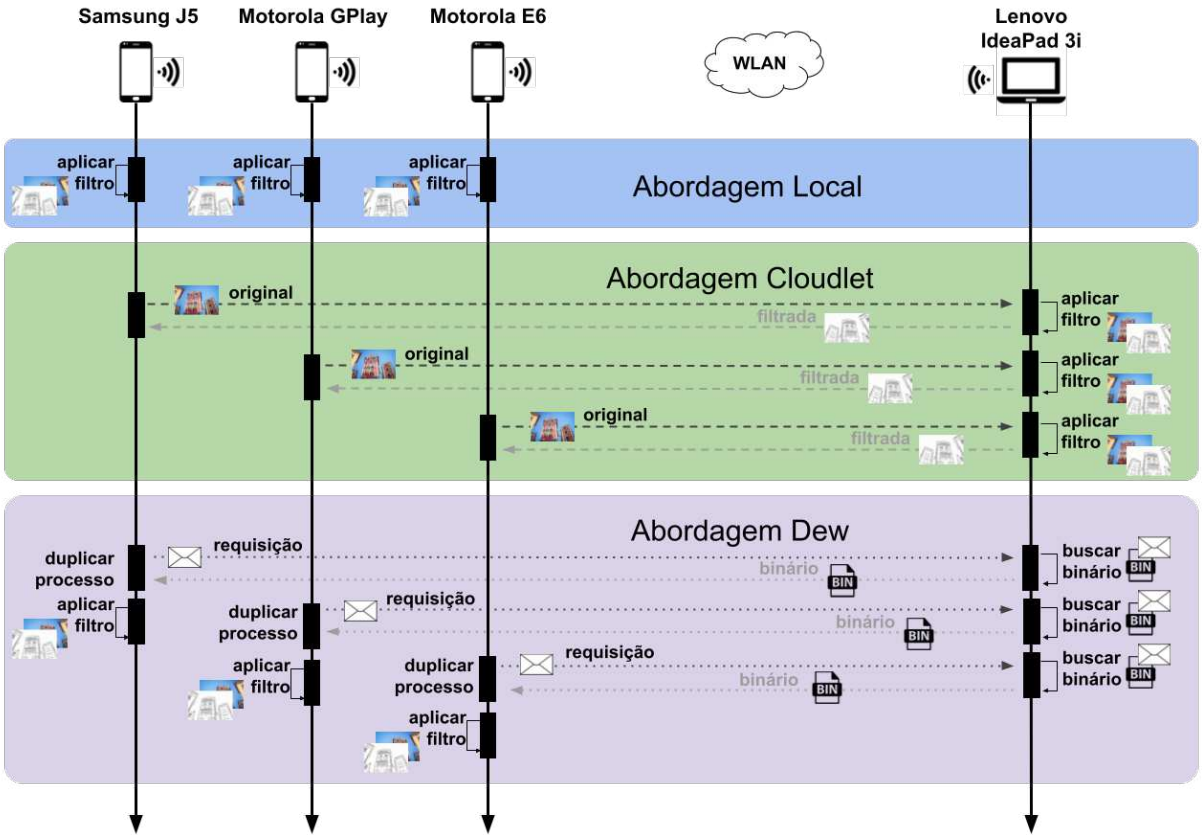


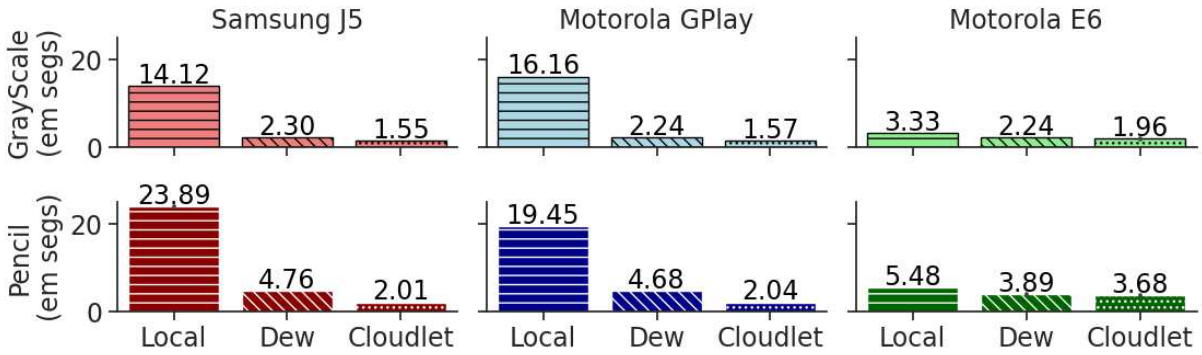
Figura 29 – Visão geral sobre as abordagens *Local*, *Cloudlet* e *Dew* no Cenário 1, onde os *smartphones* clientes interagem com o *notebook* servidor não concorrentemente.

mas apenas durante a duplicação do processo servidor remoto. Uma vez concluída a duplicação, a interação ocorre somente no dispositivo e não envolve ações na rede externa.

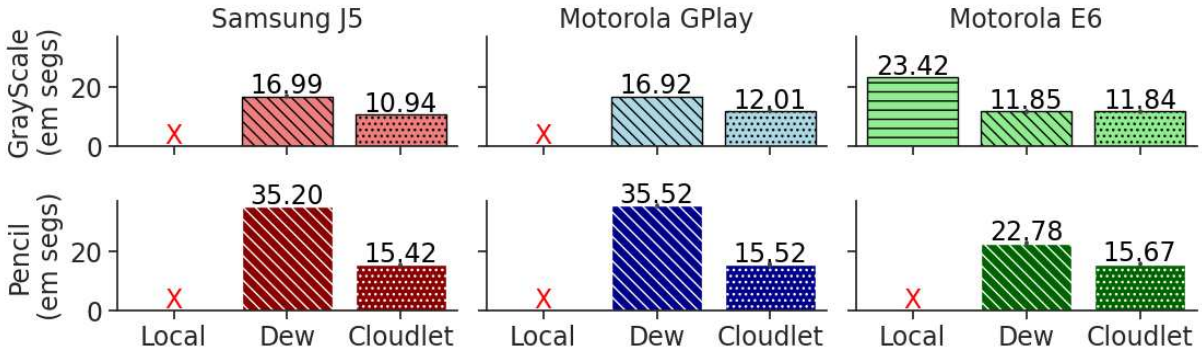
6.1.1.1 Tempo de Resposta

A Figura 30 exibe os resultados relacionados à métrica de Tempo de Resposta. Nela, observa-se que a abordagem *Dew* não superou o desempenho da abordagem *Cloudlet* em nenhuma das configurações avaliadas, independentemente do dispositivo, do filtro usado e da resolução de imagem escolhida. Isso já era esperado, porque, como destacado anteriormente, em situações com baixa disputa de acesso ao meio, o *offloading* multi-linguagem adotado na abordagem *Cloudlet* tende a apresentar um desempenho superior. Além disso, na abordagem *Dew*, a computação da tarefa é realizada pelo próprio *smartphone*, que possui um *hardware* limitado quando comparado ao *notebook* da abordagem *Cloudlet*. Tal diferença de poder computacional pode tornar o processamento da tarefa mais célere ou não.

As maiores diferenças entre as abordagens foram observadas ao aplicar o filtro *Pencil* em imagens de 8 MP, uma tarefa que exige mais recursos computacionais para ser realizada.



(a) Imagem *Skyline* de 1 MP da aplicação *BenchImage*



(b) Imagem *Skyline* de 8 MP da aplicação *BenchImage*

Figura 30 – Tempo médio de resposta no Cenário 1, onde os *smartphones* clientes interagem com o *notebook* servidor não concorrentemente.

Nessa configuração, os resultados mostraram que a abordagem *Cloudlet* foi de 1,4 a 2,2 vezes mais rápida que a abordagem *Dew*. Nos demais casos, também foram constatadas diferenças entre as abordagens, porém menos significativas. As menores diferenças foram observadas com o Motorola E6, o *smartphone* de melhor *hardware* nos experimentos. Por exemplo, ao lidar com imagens de 1 MP, a diferença entre as abordagens *Dew* e *Cloudlet* no Motorola E6 foi de aproximadamente 200 milissegundos para ambos os filtros. O melhor *hardware* faz com que o processo servidor duplicado nesse dispositivo compute tarefas mais simples e/ou que manipulem menos dados mais rapidamente que os demais dispositivos.

Se, por um lado, a abordagem *Dew* não obteve bons resultados quando comparada à abordagem *Cloudlet*, por outro, ela superou a abordagem *Local* em todas as configurações. Em alguns casos, inclusive, a abordagem *Dew* processou tarefas que a abordagem *Local* não conseguiu, devido a problemas de memória. Por exemplo, ao adotar a abordagem *Local*, nenhum *smartphone* foi capaz de aplicar filtros em imagens de 8 MP, com exceção do Motorola E6 ao aplicar o filtro *GrayScale*. Quando a abordagem *Local* teve sucesso, notou-se que ela foi de 1,4 a 7,2 vezes mais lenta que a abordagem *Dew*. O resultado obtido está alinhado com as evidências encontradas na literatura, que indicam que a linguagem Java (utilizada no aplicativo cliente)

computa tarefas de maneira mais lenta em comparação com Go (empregada no processo do servidor duplicado).

Os testes estatístico e *post-hoc* confirmaram existir uma diferença significativa entre todas as abordagens, exceto entre as abordagens *Dew* e *Cloudlet* nos cenários *Pencil*/1 MP (3,89 e 3,68 milissegundos) e *GrayScale*/8 MP (11,85 e 11,84 milissegundos) no Motorola E6. Esses resultados, além de corroborarem as discussões realizadas nesta seção, indicam que as abordagens *Dew* e *Cloudlet* apresentaram um desempenho praticamente equivalente nesta métrica especificamente nesses cenários. Isso reforça a ideia que a abordagem *Cloudlet* pode ser superada pela abordagem *Dew*, especialmente em *smartphones* mais modernos e com melhores configurações de hardware.

6.1.1.2 Consumo de Energia

A Figura 31 compila os resultados relacionados a métrica do Consumo de Energia. Os resultados da métrica de Tempo de Resposta mostraram que as maiores diferenças entre as abordagens avaliadas, independente da resolução da imagem e do filtro usados, foram observadas no dispositivo Samsung J5. Devido a isso, optou-se por avaliar a métrica do Consumo de Energia somente nesse dispositivo.

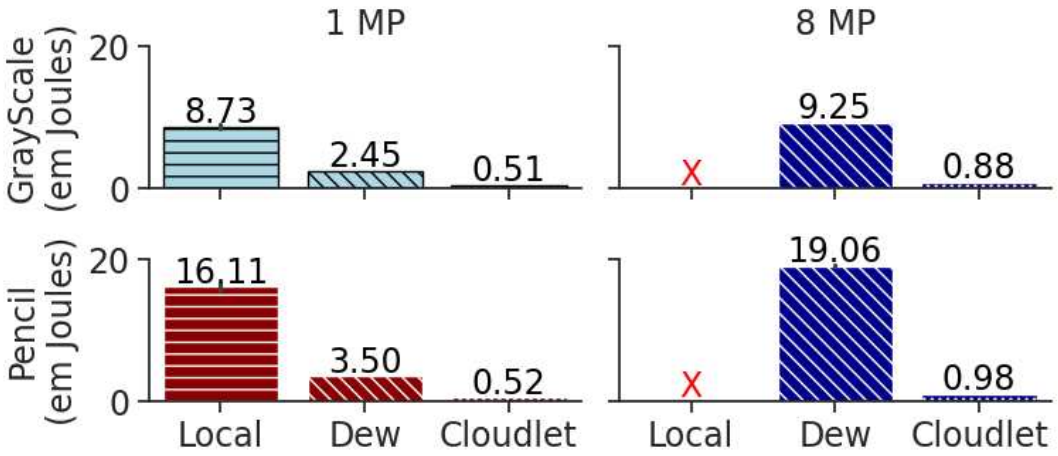


Figura 31 – Consumo médio de energia do Samsung J5 no Cenário 1, onde os *smartphones* clientes interagem com o *notebook* servidor não concorrentemente.

Primeiramente, nota-se que, analogamente à métrica anterior, a abordagem *Cloudlet* economizou mais energia do dispositivo móvel. Por exemplo, ao comparar os resultados das abordagens *Cloudlet* e *Dew*, observa-se que a primeira consumiu 4,8 e 6,7 vezes menos energia que a última ao aplicar filtros *GrayScale* e *Pencil* em imagens de 1 MP, respectivamente. Para

imagens de 8 MP, a economia é ainda maior, chegando a aproximadamente 19 vezes quando foi utilizado o filtro *Pencil*. Tais resultados eram previsíveis, considerando que a abordagem *Dew* exige uma participação ativa do dispositivo móvel, levando ao aumento no consumo de energia. Em contraste, na abordagem *Cloudlet*, o dispositivo móvel delega o processamento da tarefa a outra máquina na rede, limitando-se a aguardar o resultado. Além disso, o menor Tempo de Resposta observado na abordagem *Cloudlet* também contribui para a eficácia desta metodologia.

Também observou-se que a abordagem *Dew* se mostrou mais econômica que a abordagem *Local*. Os resultados mostraram que a abordagem *Dew* consumiu 3,5 e 4,6 vezes menos energia que a abordagem *Local* ao aplicar filtros *GrayScale* e *Pencil* em imagens de 1 MP, respectivamente. Não foi possível realizar essa mesma comparação com imagens de 8 MP, pois não foi possível aplicar os filtros nessa resolução na abordagem *Local*. Assim como no parágrafo anterior, o alto Tempo de Resposta dispendido pela abordagem *Local* para processar a tarefa também influenciou no alto consumo de energia observado.

Os testes estatístico e *post-hoc* aplicados confirmaram haver uma diferença significativa entre os valores apresentados na Figura 31, o que reforça os argumentos apresentados nesta seção. Em resumo, focando apenas no Consumo de Energia, a abordagem *Dew* supera a abordagem *Local* por computar a tarefa adotando uma linguagem de programação mais eficiente. Contudo, ela não supera a abordagem *Cloudlet*, uma vez que o *smartphone* ainda precisa processar a tarefa, o que consome mais energia do que simplesmente esperar pelo resultado do processamento remoto.

6.1.1.3 Consumo de Rede

A Figura 32 exibe os resultados relacionados à métrica do Consumo de Rede. Em geral, os resultados dessa métrica independem do dispositivo utilizado, uma vez que o tráfego de rede gerado é o mesmo para todos eles e varia apenas de acordo com a abordagem utilizada (*Dew* ou *Cloudlet*) e, no caso da abordagem *Cloudlet*, do tipo de filtro aplicado (*GrayScale* ou *Pencil*). Por conta disso, focou-se somente nos resultados obtidos com o *smartphone* Samsung J5.

Em primeiro lugar, observa-se que o desempenho da abordagem *Dew* foi praticamente o mesmo para todas as resoluções de imagem e filtros nessa métrica. Isso ocorre porque, como destacado anteriormente, a interação com a rede externa só acontece durante a fase de duplicação na abordagem *Dew*. Como o binário é o mesmo em todas as configurações, é natural que o *download* dele consuma aproximadamente os mesmos recursos de rede. Pequenas discre-

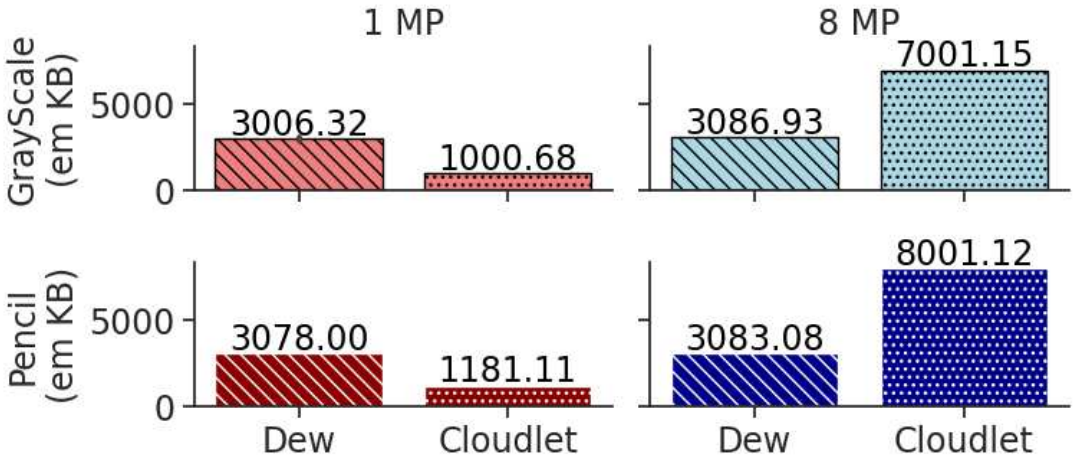


Figura 32 – Consumo médio de rede no Cenário 1, onde os *smartphones* clientes interagem com o *notebook* servidor não concorrentemente.

pâncias podem ser observadas, uma vez que a transmissão sem fio não estava completamente livre de interferências externas, o que poderia levar ao reenvio de pacotes durante a comunicação.

Em segundo lugar, nota-se que, para imagens de 8 MP, a abordagem *Dew* transmitiu cerca de metade dos dados em comparação com a abordagem *Cloudlet*. Esse comportamento não foi observado para imagens de 1 MP, onde, na verdade, a abordagem *Cloudlet* transmitiu cerca de um terço dos dados da abordagem *Dew*. Esse comportamento está diretamente relacionado aos tamanhos das imagens e do binário. Quanto maior a resolução da imagem, maior a quantidade de dados necessária para representá-la. Esses dados devem ser transmitidos na rede externa a cada *offloading* realizado na abordagem *Cloudlet*. A abordagem *Dew* não sofre com isso, pois, uma vez duplicado o processo servidor, a computação ocorre localmente no *smartphone* e totalmente *offline*, ou seja, sem envolver quaisquer operação na rede externa.

Esses resultados indicam que a abordagem *Dew* pode economizar recursos e reduzir o nível de congestionamento da rede. Com uma única interação, a abordagem *Cloudlet* consumirá mais largura de banda que a abordagem *Dew*, caso o tamanho do binário do processo servidor a ser duplicado seja menor que a quantidade de dados a serem transmitidos durante o *offloading*. Além disso, caso as interações sejam recorrentes (aplicar um filtro em um vídeo ou em uma pasta com várias imagens, por exemplo), a abordagem *Dew* pode proporcionar uma economia ainda maior, uma vez que a diferença de dados trafegados entre as abordagens seria mais significativa.

Os testes estatístico aplicados confirmaram haver uma diferença significativa entre os valores apresentados na Figura 32, o que reforça os argumentos apresentados nesta seção.

6.1.2 Cenário 2: Dispositivos Juntos

O segundo cenário comparou as abordagens *Dew* e *Cloudlet* em um ambiente onde a rede sem fio e o *Cloudlet* não estão dedicados a um único *smartphone* cliente (Figura 33). Para isso, os três *smartphones* iniciaram o processamento de suas tarefas simultaneamente a cada repetição. Tal ação aumenta a chance de colisão de sinais durante o *offloading* computacional, o que pode levar à retransmissão de pacotes e à diminuição da largura de banda disponível. Além disso, como o *Cloudlet* terá que processar as tarefas de três dispositivos clientes ao mesmo tempo, também aumenta-se a carga de trabalho sobre ele, o que pode tornar o *offloading* mais lento. Portanto, esse tipo de cenário se mostra mais hostil para a abordagem *Cloudlet* do que o anterior.

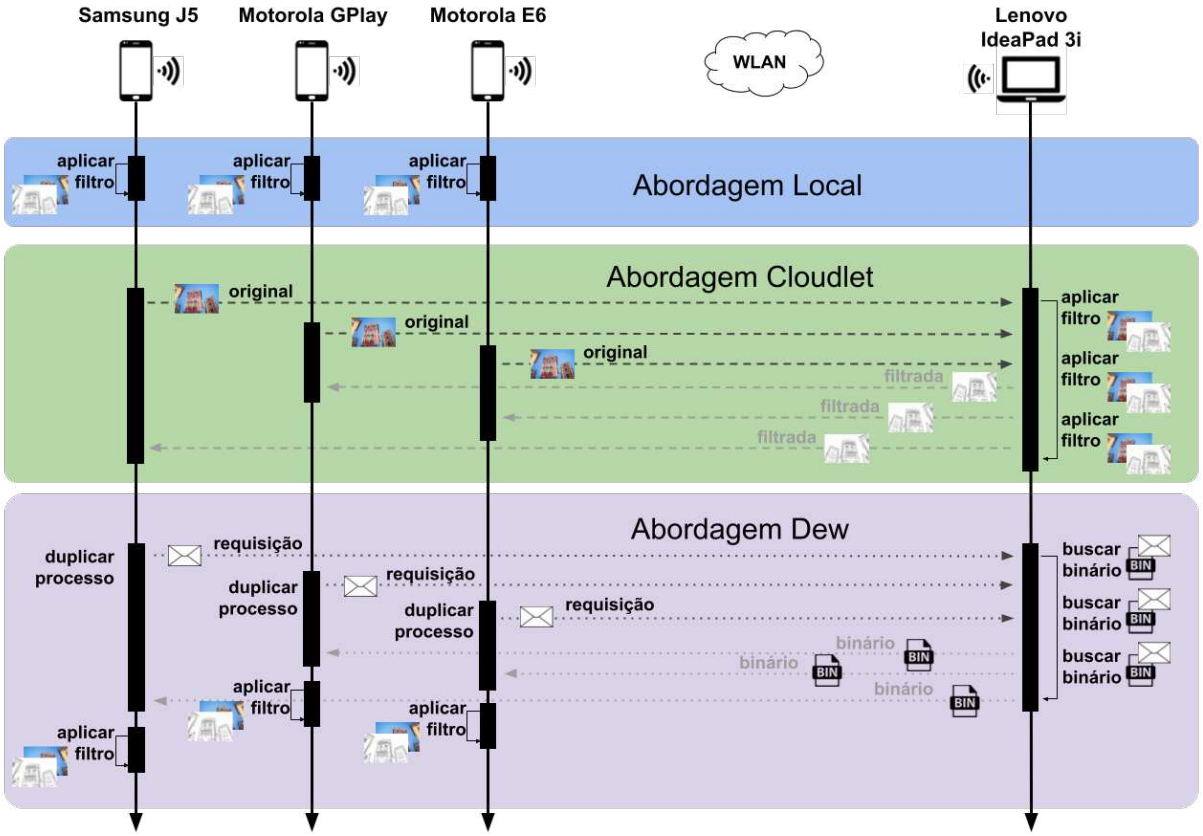


Figura 33 – Visão geral sobre as abordagens *Local*, *Cloudlet* e *Dew* no Cenário 2, onde os *smartphones* clientes interagem com o *notebook* servidor concorrentemente.

A abordagem *Dew* também é afetada negativamente por todo esse compartilhamento, porém menos do que a abordagem *Cloudlet*, uma vez que a interação com a rede e com o servidor externo só acontece durante a fase de duplicação, onde os três *smartphones* entrarão em disputa para adquirir o processo servidor remoto. Encerrada a duplicação, a aplicação cliente solicitará o processamento de suas tarefas ao processo servidor hospedado no mesmo *smartphone* onde ela

está instalada, através da interface de *loopback*, sem envolver ações na rede externa. Assim, uma eventual sobrecarga na rede ou no *Cloudlet* só afetará a abordagem *Dew* durante a duplicação do processo servidor remoto.

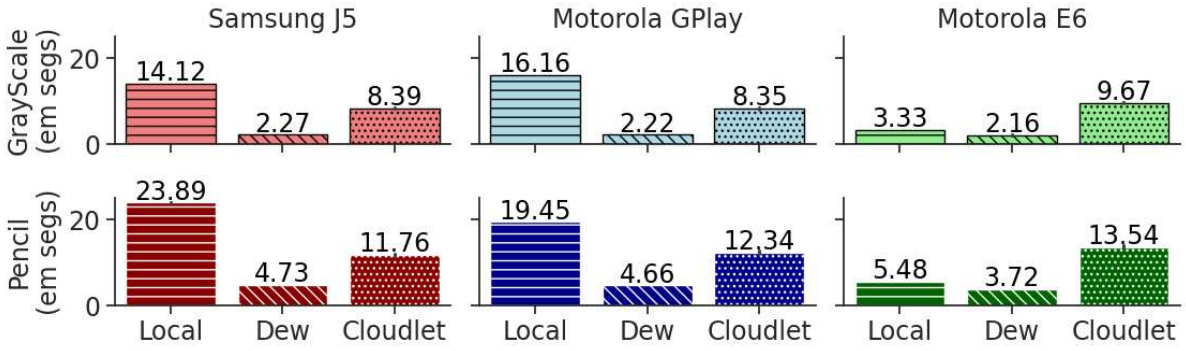
A abordagem *Local* não executa nenhuma operação na rede externa, uma vez que o processamento ocorre completamente no aplicativo cliente. Assim, não importa se a rede externa está sobrecarregada ou não, o desempenho nessa abordagem tende a ser o mesmo dentro dos cenários avaliados. Devido a isso, os resultados da abordagem *Local* obtidos no Cenário 1 foram replicados no Cenário 2. Ao exibir os resultados da abordagem *Local* espera-se simplificar para o leitor comparar os resultados dessa abordagem com as demais consideradas (*Dew* e *Cloudlet*)

6.1.2.1 Tempo de Resposta

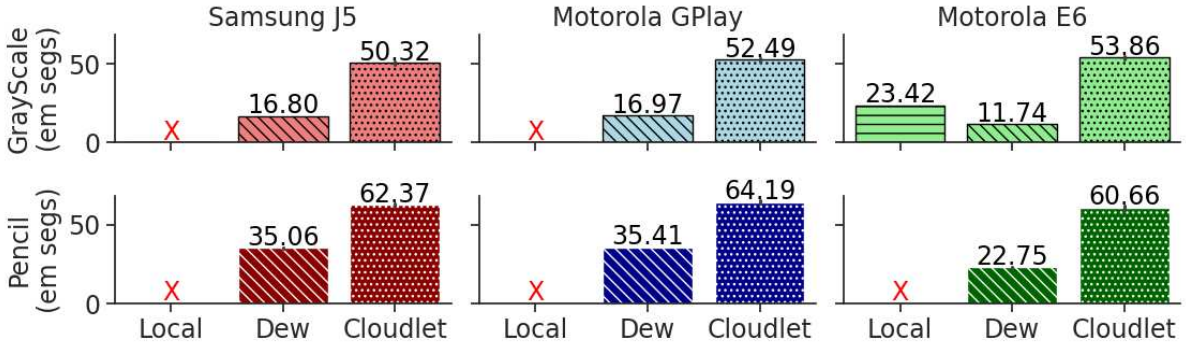
A Figura 34 mostra os resultados obtidos com a métrica Tempo de Resposta. Inicialmente, ao contrário do cenário anterior, a abordagem *Dew* apresentou o melhor desempenho em todos os cenários avaliados. Em geral, a abordagem *Dew* apresentou uma vantagem significativa, com um tempo de resposta de 2,4 a 3,6 vezes menor do que a abordagem *Cloudlet* para imagens de 1 MP e de 1,7 a 4,5 vezes menor para imagens de 8 MP. Embora a diferença entre as abordagens *Dew* e *Local* tenha reduzido, ela ainda pode ser bastante significativa. Por exemplo, ao lidar com imagens de 1 MP, a abordagem *Dew* foi 6,2 vezes mais rápida no filtro *GrayScale* e 5,0 vezes mais rápida no filtro *Pencil*, em comparação com a abordagem *Local* no Samsung J5. Mesmo no Motorola E6, onde se obteve as menores discrepâncias entre as abordagens, a *Dew* foi no mínimo 1,5 vezes mais rápida que a *Local*.

Uma análise mais detalhada nos gráficos gerados nos dois cenários propostos revela que essa discrepância surge devido a uma perda significativa de desempenho na abordagem *Cloudlet* em todas as configurações. Isso já era esperado, pois, como foi ressaltado anteriormente, o atual cenário é mais adverso para a abordagem *Cloudlet* do que o anterior, uma vez que os *smartphones* clientes compartilham a rede e o *Cloudlet* durante todo o *offloading* computacional. Tal compartilhamento leva à disputa e divisão dos recursos, o que prejudica o desempenho da abordagem *Cloudlet*. Por exemplo, no cenário anterior, ao aplicar um filtro *Pencil* em imagens de 8 MP no Motorola E6, a abordagem *Cloudlet* consumiu 15,67 segundos, enquanto, no atual cenário, foram necessários 60,66 segundos. Um valor 3,8 vezes maior.

Por outro lado, essa mesma análise indica que a abordagem *Dew* teve um desempenho praticamente constante em todos os cenários. Por exemplo, considerando o mesmo caso anterior,



(a) Imagem *Skyline* de 1 MP da aplicação *BenchImage*



(b) Imagem *Skyline* de 8 MP da aplicação *BenchImage*

Figura 34 – Tempo médio de resposta no Cenário 2, onde os *smartphones* clientes interagem com o *notebook* servidor concorrentemente.

nota-se que, quando o Motorola E6 tinha acesso exclusivo à rede e ao *Cloudlet*, a abordagem *Dew* gastou, em média, 22,78 segundos para aplicar o filtro *Pencil* em imagens de 8 MP, enquanto, no atual cenário, foram necessários 22,75 segundos. Uma diferença de cerca de 300 milissegundos (1%). Como a métrica assume que o processo servidor já está duplicado no *smartphone*, toda a interação entre ele e o aplicativo cliente ocorre independentemente da rede externa. Assim, não importa se a rede ou o *Cloudlet* estão sobrecarregados, o desempenho da abordagem *Dew* tende a ser o mesmo em ambos os cenários.

Por fim, destaca-se que a abordagem *Dew* foi a mais rápida dentre todas as avaliadas para todas as configurações de cenário. Isso indica que, em ambientes com alta sobrecarga, seja na rede ou na máquina servidora, a abordagem *Dew* parece ser a mais vantajosa em termos de tempo de resposta. Esse resultado sugere que a abordagem *Dew* seja a mais eficiente em lidar com situações de alta demanda ou congestionamento, garantindo um desempenho mais estável e rápido em comparação com as outras abordagens. Os testes estatístico e *post-hoc* aplicados confirmaram haver uma diferença significativa entre todos os valores apresentados na Figura 34, o que reforça os argumentos apresentados nesta seção.

6.1.2.2 Consumo de Energia

A Figura 35 mostra os resultados obtidos com a métrica Consumo de Energia. Ao contrário do observado no Cenário 1, a abordagem com o menor Tempo de Resposta (*Dew*) não foi a que apresentou o menor Consumo de Energia. A abordagem *Cloudlet*, mesmo sendo mais lenta, consumiu de 5 a 14 vezes menos energia que a abordagem *Dew*. Como discutido no cenário anterior, na abordagem *Dew*, o *smartphone* hospeda o aplicativo cliente e o processo servidor que deve computar todas as tarefas solicitadas pelo aplicativo cliente. Dessa maneira, ele deve trabalhar ativamente para produzir a resposta de cada requisição, o que tende a aumentar o consumo de energia do dispositivo móvel.

Comparando os resultados dessa métrica nos dois cenários (Figuras 31 e 35), ainda percebe-se que a abordagem *Dew* proporciona uma economia de energia mais significativa que a abordagem *Local*. Em geral, mesmo em um ambiente compartilhado, a adoção da abordagem *Dew* economizou de 3,1 a 4,2 vezes mais energia para o *smartphone* Samsung J5 ao aplicar os filtros *GrayScale* e *Pencil* em imagens de 1 MP. Nesse caso, a adoção de uma linguagem de programação mais eficiente para aplicar os filtros foi responsável por melhorar o desempenho energético do dispositivo móvel.

Por fim, destaca-se que a abordagem *Cloudlet* foi a mais econômica, em termos de consumo energético, dentre todas as avaliadas para todas as configurações de cenário. Isso indica que, mesmo em ambientes com alta sobrecarga, seja na rede ou na máquina servidora, a abordagem *Cloudlet* parece ser a mais vantajosa nessa métrica. Esse resultado sugere que a espera passiva proporcionada pela abordagem *Cloudlet*, ou seja, terceirizar a computação da tarefa e apenas aguardar pelo resultado, economiza mais energia no dispositivo móvel. Os testes estatístico e *post-hoc* aplicados confirmaram haver uma diferença significativa entre todos os valores apresentados na Figura 35, o que reforça os argumentos apresentados nesta seção.

6.1.2.3 Consumo de Rede

A Figura 36 apresenta os resultados obtidos com a métrica de Consumo de Rede. Comparando as Figuras 32 e 36, identificou-se um aumento na quantidade de dados transmitidos em todas as configurações. Isso é justificado pela maior disputa pelo uso do meio compartilhado, uma vez que, nesse cenário, há mais *smartphones* realizando o *offloading* simultaneamente. Quanto maior o tamanho dos arquivos a serem enviados, maior será o número de pacotes a serem

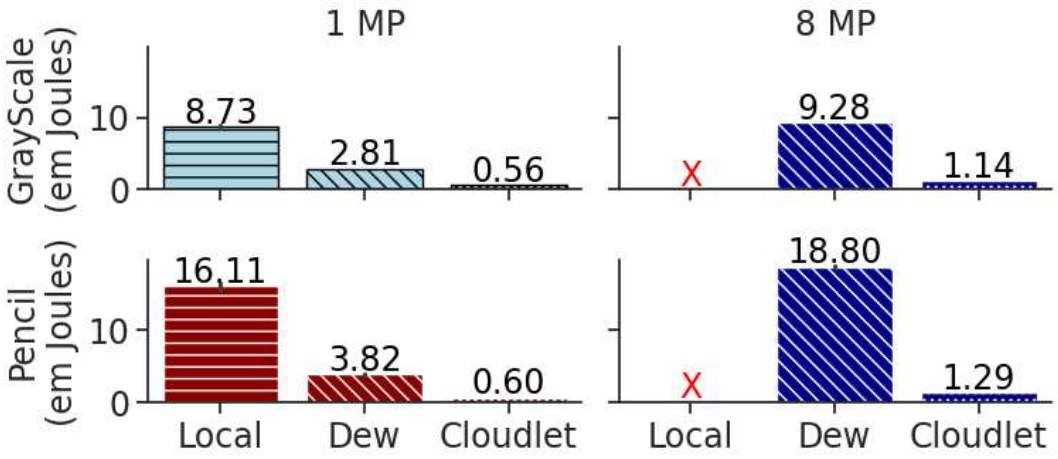


Figura 35 – Consumo médio de energia do Samsung J5 no Cenário 2, onde os *smartphones* clientes interagem com o *notebook* servidor concorrentemente.

transmitidos através da rede. Esse maior número de pacotes aumenta as chances de colisões e, como resultado, pode levar a retransmissões de pacotes e tende a aumentar o Consumo de Rede. Isso é válido tanto para ambas as abordagens. Por exemplo, ao comparar as Figuras 32 e 36, nota-se que o *offloading* da abordagem *Cloudlet* proporcionou um acréscimo máximo de 2,56 KBytes (0,2%) com imagens de 1 MP e mínimo de 1 MByte (12%) com imagens de 8 MP. Já a abordagem *Dew* apresentou um acréscimo entre 736 e 818 KBytes independente do filtro e da resolução da imagem.

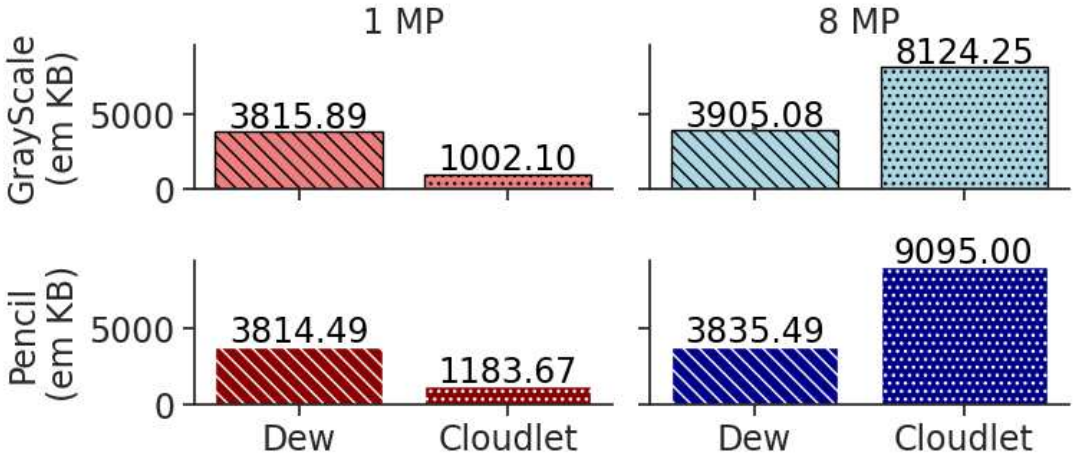


Figura 36 – Consumo médio de rede no Cenário 2, onde os *smartphones* clientes interagem com o *notebook* servidor concorrentemente.

Por fim, não foram identificadas diferenças significativas com as observações levantadas no cenário anterior. Em geral, a abordagem *Dew* continuou economizando mais largura de banda com imagens de 8 MP, ao passo que a abordagem *Cloudlet* continuou sendo a mais econômica com imagens de 1 MP. Esses resultados indicam que, mesmo em um ambiente

sobrecarregado, a abordagem *Dew* continua a ser a melhor solução quando o tamanho do binário do processo a ser duplicado é maior que a quantidade de dados a ser transmitida no *offloading* computacional. Os testes estatístico aplicados confirmaram haver uma diferença significativa entre os valores apresentados na Figura 36, o que reforça os argumentos apresentados nesta seção.

6.2 Considerações Finais

Este capítulo descreveu os experimentos realizados com o objetivo de validar o funcionamento da arquitetura DADOS e o impacto que ela, ao utilizar o paradigma *Dew Computing*, pode proporcionar a computação de tarefas em aplicativos móveis. Para isso, a abordagem *Dew*, proporcionada pela DADOS, foi comparada com abordagens tradicionais da literatura: *Local* (onde a tarefa é executada pelo próprio aplicativo móvel) e *Cloudlet* (onde a tarefa é processada por um servidor remoto via *offloading* computacional). Os resultados indicaram quatro aspectos importantes da abordagem *Dew*, que serão apresentados nesta seção. Nas próximas seções, será feita uma breve reflexão sobre cada um desses aspectos, aprofundando a discussão e identificando as vantagens e desvantagens da abordagem *Dew* em relação às demais avaliadas.

6.2.1 A abordagem Dew foi a abordagem mais rápida que a abordagem Local em todos os cenários avaliados nessa tese

A abordagem *Dew* não somente acelerou o processamento de tarefas do *smartphone*, como também permitiu a ele computar tarefas que o aplicativo cliente não conseguiu nativamente. Isso evidencia que a abordagem *Dew* contribui positivamente com a computação móvel. Por exemplo, considere um ambiente com conexões de rede intermitentes ou lentas. Sem a abordagem *Dew*, o dispositivo móvel potencialmente processaria suas tarefas localmente, mas de forma ineficiente e com possibilidade de não conseguir executá-las. Portanto, a abordagem *Dew* emerge como uma boa alternativa para ambientes desafiadores como esse.

Contudo, é importante observar que os resultados referentes à métrica de Tempo de Resposta não consideraram o tempo necessário para duplicar o processo do servidor remoto. Isso pode ser um fator complicador para a adoção da abordagem *Dew*, uma vez que os *smartphones* levaram, em média, 50 segundos para duplicar completamente o processo remoto. O uso da técnica de *caching* para salvar temporariamente os binários de processos remotos nos *smartphones*

ameniza esse problema. No entanto, como os recursos dos dispositivos móveis são limitados, é necessário desenvolver mecanismos que habilitem o *caching* dos binários, reduzindo ao máximo tanto o espaço de armazenamento quanto o tempo de duplicação do processo servidor.

Outra estratégia que pode acelerar a duplicação do processo do servidor é alterar a forma como o arquivo binário é obtido do servidor remoto. Por exemplo, substituir o protocolo de comunicação ou comprimir os binários antes de transmiti-los pela rede podem ser medidas úteis para acelerar a obtenção do arquivo binário. Essa última opção, inclusive, pode ser adotada em conjunto com a técnica de *caching* para também ajudar a economizar espaço de armazenamento do dispositivo móvel. No entanto, é importante avaliar o impacto da descompressão do arquivo binário durante a duplicação em ambos os casos.

6.2.2 A abordagem Dew é mais rápida que a abordagem Cloudlet apenas em cenários com alta disputa pelos recursos compartilhados (rede e máquina servidora)

A abordagem *Dew* se destacou ao superar a abordagem *Cloudlet* em ambientes sobrecarregados, além de demonstrar que mantém o desempenho relativamente constante, independentemente da carga de tráfego na rede e/ou do processamento no *Cloudlet*. Isso é um ponto importante, pois indica a abordagem *Dew* como uma solução promissora para processar tarefas efetivamente, mesmo sob condições adversas da rede. Por exemplo, considere uma rede em que o tráfego de dados varia bastante durante um período de tempo, como a rede de um campus universitário. A abordagem *Dew* pode ser uma garantia de um desempenho satisfatório, embora por vezes subótimo, na computação das tarefas.

Contudo, é importante notar que os experimentos adotaram uma quantidade reduzida de dispositivos móveis e um *Cloudlet* com recursos limitados quando comparado a outras máquinas disponíveis no mercado. Portanto, os desempenhos de ambas as abordagens podem variar se aumentar o poder computacional do *Cloudlet* e, principalmente, o número de dispositivos móveis clientes. Porém, mesmo sob essas condições, espera-se que a abordagem *Dew* supere a abordagem *Cloudlet*, pois o uso da abordagem *Dew* evita ações na rede compartilhada, que comumente representa o principal gargalo no desempenho do *offloading*, e explora os recursos computacionais cada vez melhores que os *smartphones* mais modernos oferecem aos seus usuários. Mais testes são necessários para confirmar se esse comportamento ocorre na prática.

6.2.3 A abordagem Dew tende a economizar largura de banda quando o offloading envolve uma quantidade significativa de dados

O fato de a abordagem *Dew* não gerar tráfego extra de dados na rede, devido à duplicação do processo servidor, pode beneficiar outras aplicações nessa rede. Por exemplo, considere uma rede universitária com diversas aplicações, como videoconferência. Quanto mais dispositivos usam a abordagem *Dew*, menor tende a ser a carga de dados transmitida na rede por eles a longo prazo. Isso libera largura de banda para que aplicações, como a de videoconferência, ofereçam uma experiência mais estável e com menos interrupções aos seus usuários. Além disso, também foi observado que, em alguns casos, duplicar o processo servidor remoto para computar a tarefa localmente consome menos largura de banda do que enviar a tarefa para ser computada por esse processo servidor remotamente. Isso reforça a ideia que a abordagem *Dew* pode auxiliar a reduzir eventuais sobrecargas na rede e poupar recursos.

Essa última observação só é válida se o tamanho do binário do processo servidor for menor que a quantidade de dados trocados durante o *offloading* computacional. Nos experimentos deste trabalho, o tamanho do binário do processo a ser duplicado foi maior que os parâmetros de entrada e saída do *offloading* para imagens de 1 MP (independentemente do tipo de filtro) e menor que os parâmetros do *offloading* para imagens de 8 MP (independentemente do tipo de filtro). Por conta disso, observou-se uma maior economia para imagens maiores e maior consumo de largura de banda para imagens menores. Contudo, a ideia da economia a longo prazo ainda se mantém. Por exemplo, considerando a Figura 36, se a abordagem *Cloudlet* fosse usada para aplicar um filtro *GrayScale* em quatro imagens de 1 MP, seria necessário transmitir 4008,40 KB (4x1002,10 KB), o que excederia o consumo exigido para duplicar o processo servidor na abordagem *Dew* (3815,89 KB).

6.2.4 A abordagem Dew apresenta um desempenho intermediário de consumo de energia em relação as abordagens Local e Cloudlet

Por fim, em todos os cenários considerados, a abordagem *Dew* obteve um desempenho energético intermediário em comparação com as abordagens *Local* e *Cloudlet*, em todos *smartphones*. Mesmo no Cenário 2, onde a abordagem *Dew* foi mais rápida que a abordagem *Cloudlet*, a primeira consumiu mais energia que a última. Porém, em ambos os cenários, a abordagem *Dew* foi mais rápida e consumiu menos energia que a abordagem *Local*. Estes resultados

indicam que a abordagem *Dew* equilibra o desempenho e a eficiência energética, tornando-se uma opção vantajosa em ambientes com acesso limitado à rede ou com alta concorrência por recursos, como é o caso do Cenário 2.

Contudo, é essencial ressaltar que, nos experimentos da Etapa 4, foram observados resultados em que a abordagem *Dew* consumiu menos energia do que a abordagem *Cloudlet* (Figura 19). Isso sugere que, sob circunstâncias específicas, a abordagem *Dew* pode ser mais vantajosa que a abordagem *Cloudlet* não somente em desempenho, mas também em termos de energia. Por exemplo, na Etapa 4, o *offloading* realizado pelo aplicativo envolvia a transmissão constante de uma grande quantidade de dados e o processo servidor foi implementado em Rust. Além disso, os códigos adotados na Etapa 4 foram desenvolvidos por terceiros com o intuito de processar tarefas o mais rápido possível e consumindo menos recursos do dispositivo. Portanto, novos experimentos são necessários para aprofundar o estudo comparativo entre as abordagens *Dew* e *Cloudlet* em relação ao consumo de energia.

7 CONCLUSÃO

Um dos desafios de pesquisa mais relevantes associados à computação móvel nos últimos anos é como melhorar o desempenho de dispositivos com restrição de energia e computacional ao processar aplicativos que se tornam cada vez mais complexos e que exigem cada vez mais recursos dos dispositivos que os executam. Uma das formas propostas pela academia para atacar esse problema é a técnica de *offloading*, onde dispositivos mais limitados enviam através da rede tarefas para processamento ou arquivos para armazenamento remoto em equipamentos menos restritivos. Entretanto, o uso da técnica de *offloading* ainda possui alguns desafios relevantes como a garantia de um sistema escalável, que consumo recursos computacionais e energéticos de forma eficiente e que mantenha um bom desempenho mesmo sob condições adversas da rede como, por exemplo, em cenários de sobrecarga.

Por outro lado, alguns pesquisadores tem comparado o desempenho de várias linguagens de programação e tem indicado que, independente da plataforma usada, algumas delas são menos apropriadas para computar determinados tipos de tarefas do que outras. Algumas das mais apropriadas, inclusive, são bastante incomuns no desenvolvimento de aplicativos móveis, como Rust e Go. Além das diferenças de desempenho computacional, esses estudos também mostraram que as linguagens possuem diferentes padrões de consumo de energia e que nem sempre a linguagem mais rápida é aquela que consome menos energia. Esses resultados motivam a criação de alternativas que possibilitem a adoção de linguagens de programação e de algoritmos mais eficientes para computar tarefas específicas em plataformas móveis especialmente, onde os recursos computacionais e energéticos são limitados.

Diante desse cenário, inicialmente, esta pesquisa focou em avaliar a possibilidade do *offloading* computacional entre processos desenvolvidos em diferentes linguagens de programação, assim como os benefícios que essa técnica poderia proporcionar aos dispositivos móveis. Os resultados dos primeiros estudos foram bastante encorajadores, pois demonstraram não apenas a viabilidade da aplicação da técnica, mas também que, ao simplesmente modificar a linguagem do processo servidor, é possível processar tarefas até 38% mais rápido e economizar até 25% de energia do dispositivo móvel. Além disso, verificou-se que a adoção de linguagens distintas nos lados cliente e servidor pode melhorar a escalabilidade do sistema. Por exemplo, ao utilizar uma linguagem de alto desempenho no lado servidor, foi possível atender até 24 clientes simultaneamente e ainda proporcionar um tempo de resposta menor que a computação local. Contudo, todos os estudos destacaram que a rede é um fator crucial para assegurar o desempenho

ideal do *offloading* computacional, dado que as operações de rede podem consumir no mínimo 50% do tempo total envolvido no *offloading*.

Como forma de mitigar os efeitos da rede no desempenho da computação de tarefas em aplicativos móveis, a pesquisa optou por aplicar o mecanismo de duplicação proposto pelo paradigma *Dew Computing*. Em resumo, esse mecanismo consiste em copiar ao menos uma parte do servidor remoto para o dispositivo cliente, permitindo que ele próprio atenda suas requisições e as requisições de dispositivos próximos. Em experimentos preliminares, ao duplicar o processo servidor remoto no dispositivo móvel cliente, observou-se uma redução significativa na quantidade de dados transmitidos pela rede, sem comprometer o desempenho computacional e energético. Por exemplo, aplicar essa estratégia em tarefas complexas e com grandes volumes de dados diminuiu a quantidade de dados transmitidos em até 97%, melhorou o tempo de resposta da tarefa em até 87% e poupou até 25% da energia do dispositivo móvel. Contudo, não existia uma ferramenta que facilitasse a implantação desse mecanismo em ambientes de computação móvel e permitisse sua exploração completa pelos desenvolvedores de aplicativos móveis.

Esta tese então propôs a arquitetura DADOS (*Dew Architecture for Distribution of Offloading Servers*) para atender tal demanda. A DADOS é uma arquitetura que une o paradigma *Dew Computing* e a interação multi-linguagem entre processos com o objetivo de simplificar a duplicação de processos servidores de *offloading* programados com linguagens distintas (e potencialmente mais eficientes) do que aquela usada no desenvolvimento do aplicativo móvel. Ao facilitar esse procedimento, a DADOS pode proporcionar um melhor desempenho e uma maior eficiência energética na execução de tarefas geradas pelos aplicativos móveis. Além disso, ao permitir a duplicação do processo servidor no dispositivo móvel, a DADOS também contribui para 1) reduzir a carga de trabalho em servidores na *Fog*, *Edge* ou *Cloud*, ao transformar o dispositivo móvel em servidor e habilitá-lo a atender as demandas de dispositivos próximos a ele e 2) reduzir a dependência da rede para a computação eficiente da tarefa, permitindo que o dispositivo móvel atenda suas próprias demandas.

Nesse documento, a arquitetura DADOS foi descritas em duas versões, uma conceitual e outra concreta. A versão concreta inicial foi gerada com base em algumas simplificações da versão conceitual com o intuito de produzir uma versão mínima, porém funcional, para avaliar a viabilidade da arquitetura. Em linhas gerais, a versão concreta foi estruturada em duas partes principais: o Lado Servidor e o Lado Cliente. O Lado Servidor foi criado com uma forte ênfase na abordagem de microsserviços, o que permitiu uma divisão funcional em sete

componentes distintos, cada um responsável por uma parte específica da arquitetura. A escolha pela abordagem de microsserviços foi motivada, principalmente, para facilitar a modificação dos serviços em pesquisas futuras. O Lado Cliente, por sua vez, foi implementado como um Serviço Android configurável através de Políticas. Mais uma vez, a opção por Políticas visa permitir a personalização simplificada do comportamento do Lado Cliente de modo a facilitar pesquisas futuras.

Foram realizados experimentos com o objetivo de validar a versão inicial da DADOS e avaliar seu impacto na computação de tarefas móveis. Para isso, montou-se um ambiente com quatro dispositivos reais, um *notebook* servidor Linux (Lenovo IdeaPad 3) e três *smartphones* clientes Android (Samsung J5, Motorola GPlay e Motorola E6), conectados através de uma rede WiFi exclusiva. Os testes conduzidos consideraram três métricas (Tempo de Resposta, Consumo de Rede e Consumo de Energia do *smartphone*) em dois cenários, um onde cada *smartphone* teve acesso exclusivo à rede e ao *notebook* servidor, e outro onde eles compartilharam esse acesso. Durante os experimentos, os *smartphones* poderiam processar as tarefas de três modos distintos: 1) *Local*: quando o próprio aplicativo Java Java executa a tarefa; 2) *Cloudlet*: quando o aplicativo Java submete, através do *offloading* computacional, a tarefa para ser processada pelo processo Go hospedado no *notebook*; 3) *Dew*: quando o aplicativo Java submete a tarefa para ser processada pelo processo Go recém-duplicado e hospedado no próprio *smartphone*. As tarefas envolviam aplicar filtros (*GrayScale* e *Pencil*) em imagens de diferentes resoluções (1 MP e 8 MP). Todos os experimentos foram executados 50 vezes, dos quais foram escolhidos os 40 menores valores após a retirada de eventuais *outliers*. Também foram aplicados testes estatísticos e *post-hoc* com intuito de validar e comparar diferenças significativas entre grupos, garantindo robustez e confiabilidade na interpretação dos resultados.

Os resultados obtidos com a versão seminal da arquitetura foram bastante promissores. Em primeiro lugar, a abordagem *Dew* superou a *Local* em ambos os cenários, processando tarefas até 7,2 vezes mais rápido e consumindo até 4,6 vezes menos energia do *smartphone*. Além disso, a abordagem *Dew* permitiu que os *smartphones* processassem tarefas que eles não conseguiram computar com a abordagem *Local*, o que evidencia sua capacidade de lidar com cargas de trabalho mais complexas e intensivas que ultrapassam as limitações da linguagem de programação nativa do aplicativo móvel. Esses resultados destacam os benefícios potenciais da adoção da abordagem *Dew* para melhorar o desempenho e a eficiência energética em ambientes de computação móvel, especialmente em cenários com conectividade de rede limitada ou ausente.

Em segundo lugar, a abordagem *Dew* não superou consistentemente a abordagem *Cloudlet*, exceto no cenário de rede compartilhada, onde processou tarefas até 4,5 vezes mais rápido, mas consumiu até 14 vezes mais energia do *smartphone*. Além disso, a *Dew* reduziu a quantidade de dados transmitidos na rede em até 2,6 vezes para imagens de 8 MP comparada a abordagem *Cloudlet*, porém para imagens de 1 MP, transmitiu até 3,8 vezes mais dados. Esses resultados sugerem que a escolha entre as abordagens depende do contexto específico de uso. Se o foco é poupar energia do *smartphone*, a abordagem *Cloudlet* parece ser mais vantajosa, mesmo em redes congestionadas. Já para otimizar o tempo de processamento de tarefas, a escolha da melhor abordagem dependerá do congestionamento da rede. Em redes congestionadas, a abordagem *Dew* supera a *Cloudlet*, enquanto em redes não congestionadas, a *Cloudlet* pode ser mais eficiente.

7.1 Principais Publicações Relacionadas a Este Trabalho

Durante a condução da pesquisa que motivou o desenvolvimento do sistema proposto nesse documento, quatro artigos científicos foram escritos, aceitos e publicados em conferências nacional e internacionais relevantes na área. Além disso, um outro artigo foi submetido e, atualmente, está sob avaliação dos revisores do jornal alvo. Ainda não foi produzido nenhum artigo relacionado à arquitetura DADOS, pois a intenção é realizar todas as correções recomendadas pela banca antes de iniciar a escrita dos artigos. Contudo, planeja-se a redação de pelo menos dois artigos. O primeiro se concentrará nos resultados obtidos com os experimentos conduzidos com a versão atual da arquitetura, conforme apresentados no Capítulo 6 e será escrito tão logo a versão final do documento da tese esteja finalizada. O segundo artigo focará na avaliação do desempenho da arquitetura sob a perspectiva dos Desenvolvedores de Serviço e de Aplicativo. Por exigir experimentos adicionais ainda não inicializados, espera-se que este trabalho demore um pouco mais para ser desenvolvido. A lista abaixo faz referência a todos os artigos já produzidos e relacionados a essa tese:

1. *"An empirical study about the adoption of multi-language technique in computation offloading in a mobile cloud computing scenario"* **PUBLICADO em *11th International Conference on Cloud Computing and Services Science (CLOSER)*** em 2021;
2. *"Secure computational offloading with grpc: A performance evaluation in a mobile cloud computing environment"* **PUBLICADO em *ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications (DIVANET)*** em 2021;

3. *"Multi-language offloading service: An android service aimed at mitigating the network consumption during computation offloading"* **PUBLICADO em *Brazilian Symposium on Multimedia and the Web (WEBMEDIA)*** em 2022;
4. *"Evaluating offloading scalability using a multi-language approach on cellular networks"* **PUBLICADO em *IEEE Consumer Communications Networking Conference (CNCC)*** em 2023;
5. *"A Comparative Study About the Performance of Multi-Language Tools in Computation Offloading Scenarios"* **SUBMETIDO COM *MAJOR REVIEW*** para o *The Journal of Supercomputing* em 2024.

7.2 Trabalhos Futuros

O desenvolvimento da arquitetura DADOS representa um passo significativo em direção a implantação de sistemas distribuídos eficientes e escaláveis para a computação de tarefas em dispositivos móveis. No entanto, existem várias direções que podem ser exploradas para aprimorar a arquitetura e sua aplicação prática. Alguns trabalhos futuros sugeridos são:

1. **Aperfeiçoar da Arquitetura DADOS:** A Arquitetura DADOS apresentada nesta tese foi projetada como uma versão simplificada, com o propósito de ser minimamente funcional para testes e validação de conceitos. O objetivo era demonstrar o potencial da arquitetura e, após essa validação, disponibilizá-la publicamente para permitir aprimoramentos contínuos à medida que novas pesquisas fossem realizadas. Portanto, sugere-se como trabalho futuro o aprimoramento da versão inicial proposta. Em termos de Arquitetura Teórica, recomenda-se melhorar a formalização e a documentação da arquitetura. No aspecto da Arquitetura Implementada, é aconselhável incorporar novas tecnologias e padrões emergentes para otimizar sua eficiência, escalabilidade e confiabilidade.
2. **Avaliar o uso de LLMs na Arquitetura DADOS:** A utilização de Modelos de Linguagem de Grande Escala (LLMs) para traduzir códigos-fonte entre linguagens de programação está se tornando uma ferramenta poderosa no desenvolvimento de software. Tais modelos, treinados em grandes conjuntos de dados de programação, facilitam a conversão entre linguagens e economizam tempo ao evitar a reescrita manual do código. Embora prometam preservar a lógica original, ainda é necessário validar e ajustar manualmente a tradução para lidar com as nuances de cada linguagem. Portanto, sugere-se como trabalho futuro avaliar o impacto que a adoção de LLMs pode proporcionar a Arquitetura DADOS especialmente

em seus atores.

3. **Testar com Novas Linguagens e *Frameworks*:** Os experimentos originais desta tese se restringiram à linguagem servidora Go e ao *framework* multi-linguagem Apache Thrift. Reconhecendo a amplitude de opções disponíveis, propomos a expansão do escopo da pesquisa para incluir linguagens servidoras alternativas, como Rust, e *frameworks* multi-linguagem distintos, como gRPC. Essa iniciativa visa fortalecer a robustez da arquitetura DADOS, validando sua efetividade em um cenário mais amplo e diverso. Além disso, a adoção de novas alternativas podem reforçar ou refutar os resultados obtidos nessa tese, contribuindo positivamente para as pesquisas na área.
4. **Aplicar da Arquitetura DADOS em Novos Contextos:** Nesta tese, os experimentos se concentraram em uma única aplicação e em um ambiente fixo com uma máquina servidora e, no máximo, três *smartphones* clientes simultâneos. Para aprofundar a compreensão da arquitetura DADOS, propõe-se a expansão da pesquisa para explorar a aplicabilidade da arquitetura em diferentes tipos de aplicativos e com diferentes requisitos de processamento e comunicação. Por exemplo, pode-se considerar a adoção de uma aplicação de *streaming* de imagens como um novo contexto. Além disso, também sugere-se aumentar o número de dispositivos, clientes e/ou servidores, simultâneos para analisar o comportamento da arquitetura sob carga mais elevada, simulando cenários mais realistas de uso.
5. **Desenvolver de Políticas do Lado Cliente:** Nos experimentos realizados nesta tese não foi utilizada nenhuma Política efetiva no Lado Cliente da arquitetura DADOS. Visando um melhor controle do comportamento dos *smartphones*, adotou-se Políticas simples que não avaliavam a situação do *smartphone* e/ou da rede externa para tomar suas decisões. Portanto, propõe-se o desenvolvimento de pesquisas voltadas para definir Políticas mais realistas que considerem o contexto do *smartphone* (como a carga de bateria disponível) e da rede externa (como o nível de intensidade do sinal) para decidir o melhor momento para duplicar um processo servidor remoto, parar um processo servidor duplicado e limpar a *Cache* de Binários.
6. **Avaliar o impactos da abordagem *Dew* nos paradigmas *Edge*, *Fog* e/ou *Cloud*:** Como discutido previamente, uma das vantagens em duplicar o processo servidor e hospedá-lo no *smartphone* é aumentar a oferta de dispositivos aptos a atender requisições de *offloading* e, potencialmente, reduzir a carga de trabalhos em máquinas servidoras alocadas na *Edge*, *Fog* e/ou *Cloud*. Assim, indica-se como trabalho futuro avaliar o nível de contribuição

que a *Dew* pode proporcionar aos demais paradigmas, quantificando os benefícios em termos de performance, redução de custos e otimização de recursos. Além disso, também é importante investigar os desafios para integrar a abordagem *Dew* aos demais paradigmas.

Essas sugestões representam apenas algumas das possíveis direções para futuras pesquisas na área da arquitetura DADOS. Explorar essas e outras possibilidades pode contribuir significativamente para o avanço do conhecimento e prática para a computação de tarefas em dispositivos móveis

REFERÊNCIAS

ALVES, P. P.; MATOS, F. de; REGO, P. A.; TRINTA, F. A. A multi-criteria context-sensitive approach to offloading decision making. *In*: **2023 IEEE 12th International Conference on Cloud Networking**. New York, US: Institute of Electrical and Electronics Engineers, 2023. p. 112–119.

ARAÚJO, M.; MAIA, M. E. F.; REGO, P. A. L.; SOUZA, J. N. D. Performance analysis of computational offloading on embedded platforms using the grpc framework. *In*: **8th International Workshop on ADVANCEs in ICT Infrastructures and Services**. Villeurbanne, FR: Hal Open Science, 2020. p. 1–8.

BARBOSA, R. A.; REGO, P. A. L. A mobile cloud computing testbed based on lightweight virtualization. *In*: **2019 IEEE Global Communications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2019. p. 1–6.

BERA, S.; DEY, T.; MUKHERJEE, A.; BUYYA, R. E-cropreco: a dew-edge-based multi-parametric crop recommendation framework for internet of agricultural things. **The Journal of Supercomputing**, Springer, v. 79, p. 11965–11999, 2023.

BHATTACHARYA, A.; DE, P. A survey of adaptation techniques in computation offloading. **Journal of Network and Computer Applications**, Elsevier, v. 78, p. 97–115, 2017.

BILAL, K.; KHALID, O.; ERBAD, A.; KHAN, S. U. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. **Computer Networks**, Elsevier, v. 130, p. 94–120, 2018.

BUGDEN, W.; ALAHMAR, A. The safety and performance of prominent programming languages. **International Journal of Software Engineering and Knowledge Engineering**, WorldScientific, v. 32, p. 713–744, 2022.

CHAMAS, C. L.; CORDEIRO, D.; ELER, M. M. Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis. *In*: **IEEE 9th Latin-American Conference on Communications**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2017. p. 1–6.

CHISNALL, D. The challenge of cross-language interoperability. **Communications of the ACM**, Association for Computing Machinery, v. 56, p. 50–56, 2013.

CHUANG, Y.-T.; HUNG, Y.-T. A real-time and aco-based offloading algorithm in edge computing. **Journal of Parallel and Distributed Computing**, Elsevier, v. 179, p. 1–14, 2023.

CISCO. **Cisco Annual Internet Report (2018–2023) White Paper**. 2020. Disponível em: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Acessado em: 04-02-2020.

COSTA, P. B.; REGO, P. A. L.; ROCHA, L. S.; TRINTA, F. A. M.; SOUZA, J. N. de. Mpos: A multiplatform offloading system. *In*: **Proceedings of the 30th Annual ACM Symposium on Applied Computing**. New York, US: Association for Computing Machinery, 2015. p. 577–584.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems: Concepts and Design**. 5. ed. Hoboken: Addison-Wesley, 2011.

COUTO, M.; PEREIRA, R.; RIBEIRO, F.; RUA, R.; SARAIVA, J. a. Towards a green ranking for programming languages. *In*: **Proceedings of the 21st Brazilian Symposium on Programming Languages**. New York, US: Association for Computing Machinery, 2017. p. 1–8.

CUERVO, E.; BALASUBRAMANIAN, A.; CHO, D.-k.; WOLMAN, A.; SAROIU, S.; CHANDRA, R.; BAHL, P. Maui: Making smartphones last longer with code offload. *In*: **Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services**. New York, US: Association for Computing Machinery, 2010. p. 49–62.

DE, D. **Mobile Cloud Computing: Architectures, Algorithms and Applications**. 1. ed. Boca Raton: CRC Press, 2016.

DREDGE, S. **5G Multi-Access Edge Computing with cloudlets in fog creating mist, and why the hell does every networking journal read like a London weather report these days?** 2018. Disponível em: https://www.metaswitch.com/blog/5g-multi-access-edge-computing-with-cloudlets-in-fog-creating-mist. Acessado em: 28-09-2023.

FERNANDO, N.; LOKE, S.; RAHAYU, W. Mobile cloud computing: A survey. **Future Generation Computer Systems**, Elsevier, v. 29, p. 84–106, 2013.

FISHER, D. E.; YANG, S. Doing more with the dew: A new approach to cloud-dew architecture. **Open Journal of Cloud Computing (OJCC)**, RonPub, v. 3, p. 8–19, 2016.

GARROCHO, C. T. B.; OLIVEIRA, R. A. R. Counting time in drops: views on the role and importance of smartwatches in dew computing. **Wireless Networks**, Springer, v. 26, p. 3139–3157, 2020.

GEORGIOU, S.; KECHAGIA, M.; LOURIDAS, P.; SPINELLIS, D. What are your programming language's energy-delay implications? *In*: **2018 IEEE/ACM 15th International Conference on Mining Software Repositories**. New York, US: Association for Computing Machinery, 2018. p. 303–313.

GEORGIOU, S.; SPINELLIS, D. Energy-delay investigation of remote inter-process communication technologies. **Journal of Systems and Software**, Elsevier, v. 162, p. 1–14, 2019.

GOMES, F. A. A.; REGO, P. A. L.; ROCHA, L.; SOUZA, J. N. d.; TRINTA, F. Caos: A context acquisition and offloading system. *In*: **2017 IEEE 41st Annual Computer Software and Applications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2017. p. 957–966.

GU, F.; NIU, J.; QI, Z.; ATIQUZZAMAN, M. Partitioning and offloading in smart mobile devices for mobile cloud computing: State of the art and future directions. **Journal of Network and Computer Applications**, Elsevier, v. 119, p. 83–96, 2018.

GUBEROVIC, E.; LIPIC, T.; CAVRAK, I. Dew intelligence: Federated learning perspective. *In*: **2021 IEEE 45th Annual Computers, Software, and Applications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2021. p. 1819–1824.

GUO, X.; DONG, C.; WEN, W. Dynamic computation offloading strategy with dnn partitioning in d2d multi-hop networks. *In*: **2021 9th International Conference on Communications and Broadband Networking**. New York, US: Association for Computing Machinery, 2021. p. 172–178.

GUSEV, M. What makes dew computing more than edge computing for internet of things. *In*: **2021 IEEE 45th Annual Computers, Software, and Applications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2021. p. 1795–1800.

GUSHEV, M. Dew computing architecture for cyber-physical systems and iot. **Internet of Things**, Elsevier, v. 11, p. 1–9, 2020.

HAUNG, Y.-R. A qoe-aware strategy for supporting service continuity in an mcc environment. **Wireless Personal Communications**, Springer, v. 116, p. 629–654, 2021.

HIRSCH, M.; MATEOS, C.; ZUNINO, A.; MAJCHRZAK, T. A.; GRøNLI, T.-M.; KAINDL, H. A task execution scheme for dew computing with state-of-the-art smartphones. **Electronics**, MDPI, v. 10, p. 1–22, 2021.

HOANG, D. T.; LEE, C.; NIYATO, D.; WANG, P. A survey of mobile cloud computing: architecture, applications, and approaches. **Wireless Communications and Mobile Computing**, Wiley, v. 13, p. 1587–1611, 2013.

HORT, M.; KECHAGIA, M.; SARRO, F.; HARMAN, M. A survey of performance optimization for mobile applications. **IEEE Transactions on Software Engineering**, Institute of Electrical and Electronics Engineers, v. 48, p. 2879–2904, 2022.

JUNIOR, F. M. R.; KAMIENSKI, C. A. A survey on trustworthiness for the internet of things. **IEEE Access**, Institute of Electrical and Electronics Engineers, v. 9, p. 42493–42514, 2021.

KANE, S.; MATTHIAS, K. **Docker: Up & Running: Shipping Reliable Containers in Production**. 2. ed. Sebastopol: O'Reilly, 2018.

KOCHHAR, P. S.; WIJEDASA, D.; LO, D. A large scale study of multiple programming languages and code quality. *In*: **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2016. p. 563–573.

KOEDIJK, L.; OPRESCU, A. Finding significant differences in the energy consumption when comparing programming languages and programs. *In*: **2022 International Conference on ICT for Sustainability (ICT4S)**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2022. p. 1–12.

KOSTA, S.; AUCINAS, A.; HUI, P.; MORTIER, R.; ZHANG, X. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. *In*: **2012 Proceedings IEEE International Conference on Computer Communications**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2012. p. 945–953.

KUMAR, K.; LIU, J.; LU, Y.-H.; BHARGAVA, B. A survey of computation offloading for mobile systems. **Mobile Networks and Applications**, Springer-Verlag, v. 18, p. 129–140, 2013.

LARICCHIA, F. **Number of smartphones sold to end users worldwide from 2007 to 2021**. 2022. Disponível em: https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/. Acessado em: 28-09-2023.

LI, Z.; QI, X.; YU, Q.; LIANG, P.; MO, R.; YANG, C. Multi-programming-language commits in oss: An empirical study on apache projects. *In*: **2021 IEEE/ACM 29th International Conference on Program Comprehension**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2021. p. 219–229.

LION, D.; CHIU, A.; STUMM, M.; YUAN, D. Investigating managed language runtime performance: Why JavaScript and python are 8x and 29x slower than c++, yet java and go can be faster? *In*: **2022 USENIX Annual Technical Conference**. Berkeley, US: USENIX Association, 2022. p. 835–852.

LIU, J.; AHMED, E.; SHIRAZ, M.; GANI, A.; BUYYA, R.; QURESHI, A. Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions. **Journal of Network and Computer Applications**, Elsevier, v. 48, p. 99–117, 2015.

MAKRIS, A.; KONTOPOULOS, I.; PSOMAKELIS, E.; XYALIS, S. N.; THEODOROPOULOS, T.; TSERPES, K. Performance analysis of storage systems in edge computing infrastructures. **Applied Sciences**, MDPI, v. 12, p. 1–10, 2022.

MANOTAS, I.; BIRD, C.; ZHANG, R.; SHEPHERD, D.; JASPAN, C.; SADOWSKI, C.; POLLOCK, L.; CLAUSE, J. An empirical study of practitioners' perspectives on green software engineering. *In*: **Proceedings of the 38th International Conference on Software Engineering**. New York, US: Association for Computing Machinery, 2016. p. 237–248.

MATOS, F. D.; OLIVEIRA, W.; CASTOR, F.; REGO, P.; TRINTA, F. Multi-language offloading service: An android service aimed at mitigating the network consumption during computation offloading. *In*: **Proceedings of the Brazilian Symposium on Multimedia and the Web**. New York, US: Association for Computing Machinery, 2022. p. 329–338.

MATOS, F. D.; REGO, P. A. L.; TRINTA, F. Evaluating offloading scalability using a multi-language approach on cellular networks. *In*: **2023 IEEE 20th Consumer Communications Networking Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2023. p. 125–130.

MATOS, F. F. S. B. de; REGO, P. A. L.; TRINTA, F. A. M. An empirical study about the adoption of multi-language technique in computation offloading in a mobile cloud computing scenario. *In*: INSTICC. **Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER,**. Online: SciTePress, 2021. p. 207–214. ISBN 978-989-758-510-4.

MATOS, F. F. S. B. de; REGO, P. A. L.; TRINTA, F. A. M. Secure computational offloading with grpc: A performance evaluation in a mobile cloud computing environment. *In*: **Proceedings of the 11th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications**. New York, US: Association for Computing Machinery, 2021. p. 45–52.

MEDHI, K.; AHMED, N.; HUSSAIN, M. I. Dew-based offline computing architecture for healthcare iot. **ICT Express**, The Korean Institute of Communications Information Sciences, v. 8, p. 371–378, 2022.

NANZ, S.; FURIA, C. A. A comparative study of programming languages in rosetta code. *In*: **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2015. p. 778–788.

NEWMAN, S. **Building Microservices**. 1. ed. Sebastopol: O'Reilly, 2021.

NGUYEN, Q.-H.; DRESSLER, F. A smartphone perspective on computation offloading: A survey. **Elsevier Computer Communications**, Elsevier, v. 159, p. 133–154, 2020.

OLIVEIRA, W.; MORAES, B.; CASTOR, F.; FERNANDES, J. P. Ebserver: Automating resource-usage data collection of android applications. *In*: **2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2023. p. 55–59.

OLIVEIRA, W.; OLIVEIRA, R.; CASTOR, F. A study on the energy consumption of android app development approaches. *In*: **2017 IEEE/ACM 14th International Conference on Mining Software Repositories**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2017. p. 42–52.

OLIVEIRA, W.; OLIVEIRA, R.; FILHO, F. C.; PINTO, G. H. L.; FERNANDES, J. P. Improving energy-efficiency by recommending java collections. **Empirical Software Engineering**, Springer, v. 26, p. 1–45, 2021.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. a. P.; SARAIVA, J. a. Energy efficiency across programming languages: How do energy, time, and memory relate? *In*: **Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering**. New York, US: Association for Computing Machinery, 2017. p. 256–267.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. Ranking programming languages by energy efficiency. **Science of Computer Programming**, Elsevier, v. 205, p. 1–30, 2021.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de Software: Uma Abordagem Profissional**. 8. ed. Porto Alegre: Bookman, 2016.

RAJ, C. R.; TOLETY, S. B. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. *In*: **2012 Annual IEEE India Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2012. p. 625–629.

RAY, P. P. An introduction to dew computing: Definition, concept and implications. **IEEE Access**, Institute of Electrical and Electronics Engineers, v. 6, p. 723–737, 2018.

RAY, P. P. Minimizing dependency on internetwork: Is dew computing a solution? **Transactions on Emerging Telecommunications Technologies**, John Wiley & Sons, v. 30, p. 1–13, 2019.

RAY, P. P.; SKALA, K. A vision of dew-iot ecosystem: Requirements, architecture, and challenges. *In*: **2021 IEEE 45th Annual Computers, Software, and Applications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2021. p. 1783–1788.

REGO, P. **Applying Smart Decisions, Adaptive Monitoring and Mobility Support for Enhancing Offloading Systems**. 115 p. Tese (Doutorado em Ciência da Computação) — Pró-Reitoria de Pesquisa e Pós-Graduação, Universidade Federal do Ceará, Fortaleza, 2016.

REGO, P. A.; COSTA, P. B.; COUTINHO, E. F.; ROCHA, L. S.; TRINTA, F. A.; SOUZA, J. N. d. Performing computation offloading on multiple platforms. **Computer Communications**, Elsevier, v. 105, p. 1–13, 2017.

RINDOS, A.; WANG, Y. Dew computing: The complementary piece of cloud computing. *In*: **2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications**

(SustainCom) (BDCloud-SocialCom-SustainCom). Piscataway, US: Institute of Electrical and Electronics Engineers, 2016. p. 15–20.

SABIN, S. **Smartphone Owners Prefer Simple Features Like Battery Life, Durability, Camera Quality**. 2018. Disponível em: https://morningconsult.com/2018/11/15/smartphone-owners-prefer-simple-features-like-battery-life-durability-camera-quality/. Acessado em: 19-10-2021.

SANTOS, G. B. D.; TRINTA, F. A. M.; REGO, P. A. L.; SILVA, F. A.; SOUZA, J. N. D. Performance and energy consumption evaluation of computation offloading using caos d2d. *In*: **2018 IEEE Global Communications Conference**. Piscataway, US: Institute of Electrical and Electronics Engineers, 2018. p. 1–7.

SATYANARAYANAN, M.; BAHL, P.; CACERES, R.; DAVIES, N. The case for vm-based cloudlets in mobile computing. **IEEE Pervasive Computing**, Institute of Electrical and Electronics Engineers, v. 8, p. 14–23, 2009.

SHAKARAMI, A.; SHAHIDINEJAD, A.; GHOBAEI-ARANI, M. A review on the computation offloading approaches in mobile edge computing: A game-theoretic perspective. **Software: Practice and Experience**, Wiley, v. 50, p. 1719–1759, 2020.

SHERIF, A. **Market share of mobile operating systems worldwide 2009-2023**. 2024. Disponível em https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. Acessado em: 09-02-2024.

SINGH, P.; GABA, G. S.; KAUR, A.; HEDABOU, M.; GURTOV, A. Dew-cloud-based hierarchical federated learning for intrusion detection in iomt. **IEEE Journal of Biomedical and Health Informatics**, Institute of Electrical and Electronics Engineers, v. 27, p. 722–731, 2023.

SINGH, P.; KAUR, A.; AUJLA, G. S.; BATTH, R. S.; KANHERE, S. S. Daas: Dew computing as a service for intelligent intrusion detection in edge-of-things ecosystem. **IEEE Internet of Things Journal**, Institute of Electrical and Electronics Engineers, v. 8, p. 12569–12577, 2021.

SKALA, K.; DAVIDOVIC, D.; AFGAN, E.; SOVIC, I.; SOJAT, Z. Scalable distributed computing hierarchy: Cloud, fog and dew computing. **Open Journal of Cloud Computing**, RonPub, v. 2, p. 16–24, 2015.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Addison-Wesley, 2011.

SUWANSRIKHAM, P.; KUN, S.; HAYAT, S.; JACKSON, J. Dew computing and asymmetric security framework for big data file sharing. **Information**, MDPI, v. 11, p. 1–28, 2020.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos: princípios e paradigmas**. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

VASCONCELOS, D. R.; ANDRADE, R. M. C.; SEVERINO, V.; SOUZA, J. N. D. Cloud, fog, or mist in iot? that is the question. **ACM Transactions on Internet Technology**, Association for Computing Machinery, v. 19, p. 1–20, 2019.

WANG, J.; PAN, J.; ESPOSITO, F.; CALYAM, P.; YANG, Z.; MOHAPATRA, P. Edge cloud offloading algorithms: Issues, methods, and perspectives. **ACM Computing Surveys**, Association for Computing Machinery, v. 52, p. 1–23, 2019.

WANG, Y. Cloud-dew architecture. **International Journal of Cloud Computing**, Inderscience, v. 4, p. 199–210, 2015.

WANG, Y. **The Initial Definition of Dew Computing**. 2015. Disponível em: http://www.dewcomputing.org/index.php/2015/11/10/the-initial-definition-of-dew-computing/. Acessado em: 28-09-2023.

WANG, Y. Definition and categorization of dew computing. **Open Journal of Cloud Computing**, RonPub, v. 3, p. 1–7, 2016.

WANG, Y. A blockchain system with lightweight full node based on dew computing. **Internet Things**, Elsevier, v. 11, p. 1–6, 2020.

YANNIBELLI, V.; HIRSCH, M.; TOLOZA, J.; MAJCHRZAK, T. A.; ZUNINO, A.; MATEOS, C. Speeding up smartphone-based dew computing: In vivo experiments setup via an evolutionary algorithm. **Sensors**, MDPI, v. 23, p. 1–22, 2023.

ZHU, S.-f.; CAI, J.-h.; SUN, E.-l. Mobile edge computing offloading scheme based on improved multi-objective immune cloning algorithm. **Wireless Networks**, Springer, v. 29, p. 1737–1750, 2023.