CEEL - ISSN 2596-2221
Universidade Federal de Uberlândia
07 a 11 de dezembro de 2020

ESTABELECENDO UM PARALELO TEÓRICO ENTRE CONTAINERS E WEBASSEMBLY

Victoria Maria Veloso Rodrigues*[1], Márcio José da Cunha[1], Cecília Carneiro e Silva[1] e Ronaldo Pires da Silva[1]

[1]FEELT - Universidade Federal de Uberlândia

***Resumo* - O objetivo deste trabalho é apresentar o método virtualização e desacoplamento de softwares não baseados em web utilizando WebAssembly por meio de uma comparação com a tecnologia mais utilizada atualmente no mercado, Containers. Estratégias e ferramentas utilizadas por ambas para assegurar necessidades básicas de máquinas virtuais, como portabilidade e segurança, são discutidas. E finalmente, conclui-se como, apesar de ainda ser uma tecnologia imatura, WebAssembly pode se estabelecer como opção viável para virtualização em um futuro próximo.**

***Palavras-Chave*- Containers, Docker, WASI, Wasm, WebAssembly.**

ESTABLISHING A THEORETICAL PARALLEL BETWEEN CONTAINERS AND WEBASSEMBLY

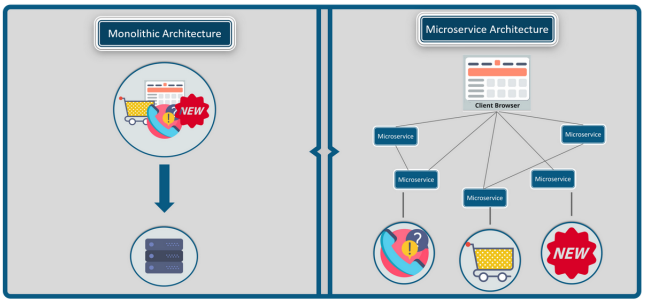
***Abstract* - The objective of this work is to present the virtualization and decoupling method of non-web based software using WebAssembly through a comparison with the most used technology currently on the market, Containers. Strategies and tools used by both to ensure basic needs of virtual machines, such as portability and security, are discussed. And finally, we conclude how, although it is still an immature technology, WebAssembly can establish itself as a viable option for virtualization in the near future.**

***Keywords* - Containers, Docker, WASI, Wasm, WebAssembly.**

I. INTRODUÇÃO

Quando se pensa em desenvolvimento de *software*, o primeiro pensamento que se tem é qual forma na qual esse *software* será desenvolvido. Há pouco tempo, pensava-se em desenvolver um sistema único, monolítico, que tivesse intrinsecamente, as camadas de aplicação, interface e banco de dados. Tudo isso em apenas um único *software*. Com o crescimento das aplicações voltadas para o comércio eletrônico e as redes sociais, tal desenvolvimento se tornou inviável, ou seja, não se podia mais ter apenas um *software* que comportasse todos os módulos e camadas, tendo em vista que, caso fosse necessário fazer algum tipo de manutenção, o *software* deveria ser parado, para que uma nova versão fosse atualizada. Nos dias atuais, é notável a existência de uma quebra de paradigma de desenvolvimento de *software*. Aquela estrutura monolítica perdeu espaço e, atualmente, a arquitetura de desenvolvimento é desenvolvida obedecendo principalmente o conceito de desacoplamento de *software*, ou seja, cada módulo é dividido em pequenas partes que se comunicam entre si. Quando existe a necessidade de atualização, apenas uma dessas pequenas partes é atualizada, sem comprometer o funcionamento das demais. Essas pequenas partes de *software*, hoje, são nomeadas como microsserviços. Na Figura 1, tem-se um comparativo entre as arquiteturas monolíticas e a arquitetura em microsserviço.

Figura 1: Comparação entre sistemas monolíticos e sistemas com arquitetura em microsserviços [1]



Como pode ser observado, quanto mais desacoplado for o serviço, tem-se uma melhora no desempenho, divisão de tarefas, controle de processamento, em alguns casos baixa latência, otimização do consumo de recursos computacionais e organização.

Esse desacoplamento exige que o ambiente computacional o qual ele é desenvolvido, dê suporte para sua execução, instalação de ferramentas e configuração, seja do sistema, ou das variáveis de ambiente do sistema operacional. Dessa forma, problemas relacionados à falta de compatibilidade são prati-

*victoria.rodrigues@ufu.br

camente nulos, o que torna o desenvolvimento e implantação menos trabalhosos e onerosos [1].

Uma das principais tecnologias utilizadas para melhorar o desacoplamento são as máquinas virtuais e os containers, por proporcionar tais recursos. Porém, as máquinas virtuais possuem desvantagens com relação aos Containers em aspectos relacionados a utilização de recursos computacionais, tais como memória, processador e disco. Os containers, por outro lado, utilizam de uma outra metodologia, em que os recursos computacionais são compartilhados, minimizando a utilização dos recursos computacionais.

Este artigo tem como proposta fazer um levantamento teórico sobre as tecnologias de virtualização utilizadas atualmente, os Containers e Webassembly, descrevendo seus elementos e fazendo uma comparação entre tais tecnologias. Tais comparações analisam itens referentes ao funcionamento de tais tecnologias, e como elas se diferem e são aplicadas.

O restante do artigo está dividido da seguinte forma: primeiro, revisão da literatura com o objetivo de se criar um banco de dados para referência. Logo após, ambas as tecnologias alvo são introduzidas e descritas a fim de construir uma base teórica para a discussão subsequente. Em seguida, aspectos comuns necessários para a virtualização de Containers e WebAssembly são apresentados de forma paralela com o propósito de apresentar WebAssembly como uma tecnologia viável para desacoplamento de *software* em um futuro próximo.

II. REVISÃO DA LITERATURA

O objetivo desta seção é identificar o atual estado da arte referente ao objeto de estudo para identificar *gaps* de pesquisa existentes no material disponível. A coleta de artigos foi feita através das plataformas Google Schoolar, SpringerLink, IEEExplore e o site da Universidade de Cambridge. WebAssembly, containers, Docker, *serveless*, WASI e Krustlet foram palavras-chave escolhidas para filtrar o conteúdo de interesse. Por ser o container de aplicação mais amplamente utilizado no mercado, Docker foi considerado o estado da arte em seu domínio. Virtualização com Docker é um assunto extensivamente estudado, portanto há diversos textos acadêmicos detalhando características e aplicações dessa tecnologia, além de comparações com máquinas virtuais.

No entanto, a aplicação de WebAssembly para virtualização de software fora dos navegadores é recente, a interface que permite tal uso foi apresentada em 2019 [2]. Portanto, o acervo de textos acadêmicos publicados abordando esse assunto ainda é pequeno. Dessa forma, as documentações publicadas por desenvolvedores das ferramentas citadas neste artigo foram amplamente utilizadas na bibliografia. A escassez de trabalhos acadêmicos se estende para a comparação entre as duas formas de virtualização alvo deste artigo, essas se limitam a propostas de implementação de ambientes de tempo de execução [3] e de interface [4], e uma comparação majoritariamente quantitativa [5]. Dessa forma, este artigo tem como um de seus objetivos suprir a carência de uma comparação teórica entre containers, uma ferramenta solidificada no mercado de software, e a emergente virtualização por WebAssembly.

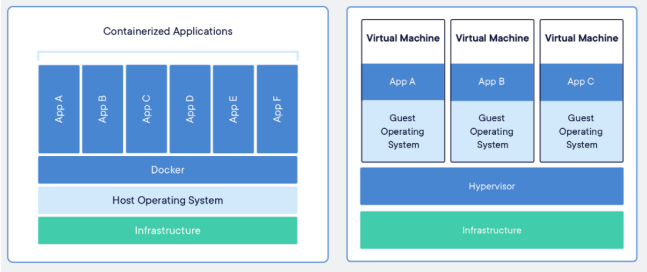
III. FUNDAMENTAÇÃO

Nesta seção é feita uma descrição de forma fundamentada sobre as tecnologias utilizadas, elencando seus principais elementos e características.

A. *Containers*

Containers são considerados como uma abstração na camada de aplicação, agrupando o código e suas dependências. Os containers são executados em uma mesma máquina, compartilhando o kernel do sistema operacional com outros containers. Dentre suas vantagens destaca-se a necessidade de se ter pouco espaço, pois as imagens dos containers são consideradas pequenas, e também podem lidar com aplicativos que de certa forma, exigem menos recursos do sistema operacional. Atualmente, os containers são utilizados em todo o processo de desenvolvimento do *software*, desde a produção, testes e *deploy*, onde apenas uma imagem de container é utilizada. Na Figura 2 é mostrado um comparativo entre a arquitetura de um container e das máquinas virtuais [6]

Figura 2: Comparação entre Containers e Máquinas Virtuais [6]



Quando se tem um número considerável de containers é necessário ter uma ferramenta responsável por orquestrar os containers. No mundo Docker, isso é feito por meio de uma tecnologia chamada Kubernets. Eles são responsáveis por automatizar as operações dos containers, eliminando grande parte dos processos manuais necessários para implantar e escalar as aplicações em containers, como subir um container, aumentar o número de containers e fechar os containers, por exemplo.

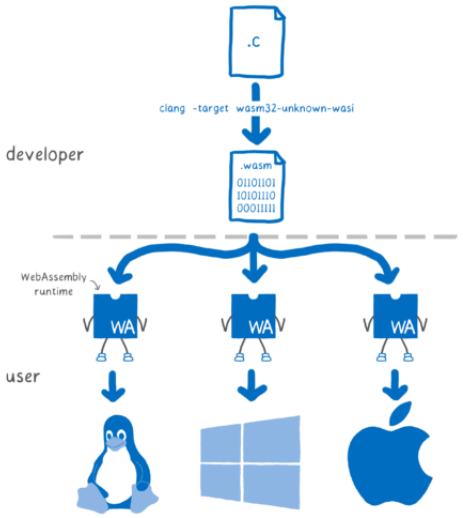
Uma curiosidade sobre empresas que utilizam containers e Kubernets: a Google revelou publicamente que grande parte de suas aplicações utilizam containers, além de possuí-los também nos seus serviços de *Cloud*.

B. *WebAssembly*

WebAssembly (Wasm) é uma tecnologia emergente e promissora desenvolvida atualmente pela *World Wide Web Consortium* (W3C) em cooperação com os principais fornecedores de navegadores como Mozilla, Google, Microsoft e Apple. O projeto foi lançado como Produto Mínimo Viável (MVP) em 2017 e, desde então, vem crescendo em popularidade com a promessa de revolucionar o desenvolvimento web ao permitir que códigos de diversas linguagens sejam executados no navegador em velocidade quase nativa. Wasm é definido como um formato binário para máquinas virtuais baseadas em pilha portátil e eficiente em relação a *load-time* e tamanho, que atua

como um novo alvo de compilação para linguagens de programação. Ou seja, é um código de baixo-nível (assim como Assembly) capaz de habilitar execução na web para aplicações de cliente e servidor escritas em linguagens de alto-nível como C/C++, Rust, Go e C# [7].

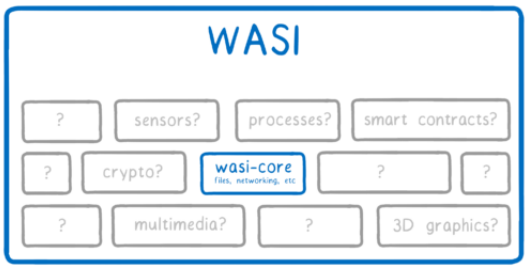
Figura 3: Ilustração da portabilidade de códigos WebAssembly [2]



Apesar do foco de desenvolvimento original ter sido aplicações web, algumas características do Wasm o tornaram atrativos para implementação em ambientes não baseados em web. Primeiramente, WebAssembly é agnóstico em relação a hardware e linguagem, ou seja, a ferramenta independe de plataforma, o que garante sua portabilidade com uma única compilação, ilustrada na Figura 3. Dessa forma, é possível executar códigos em qualquer linguagem, desde que compilados para Wasm, e em diversos ambientes sejam eles *mobile apps*, *datacenters* ou dispositivos de Internet das Coisas (IoT). Essa tecnologia também provê uma nova camada de segurança para as aplicações [8], pois seus códigos são executados em um *sandbox*, que isola os impactos de suas vulnerabilidades da interface de execução, podendo ser implementado inclusive em Javascript *Virtual Machines*. Além disso, a memória é disponibilizada linearmente prevenindo bugs e vulnerabilidades como *buffer overflows* e comportamentos imprevisíveis de ponteiros. Por fim, o design da linguagem lhe garante compatibilidade reversa, que, somada às características citadas e o suporte atual de grandes empresas para seu desenvolvimento, estabelece WebAssembly como uma opção segura de formato binário para máquinas virtuais.

Dessa forma, plataformas baseadas em WebAssembly surgem como uma alternativa aos containers e virtual machines, especialmente em sistemas de *Function-as-a-Service* (FaaS). Virtualização por Wasm, não emula *hardware* ou sistemas operacionais (OS), de fato ela ignora o ambiente em geral, incluindo o OS, focando somente no código binário a ser lido pelo sistema de tempo de execução e convertido em um conjunto de instruções nativas [5]. Em outras palavras, o módulo é executado isoladamente de funcionalidades do OS hospedeiro, portanto WebAssembly necessita de uma interface para que possa acessá-las. Wasm é a abstração linguagem assembly para uma máquina conceitual e essa característica, que permite a sua execução em diversos ambientes, induz à necessidade de uma interface para sistema operacional igualmente conceitual [2].

Figura 4: Abordagem modular da interface WASI [2]

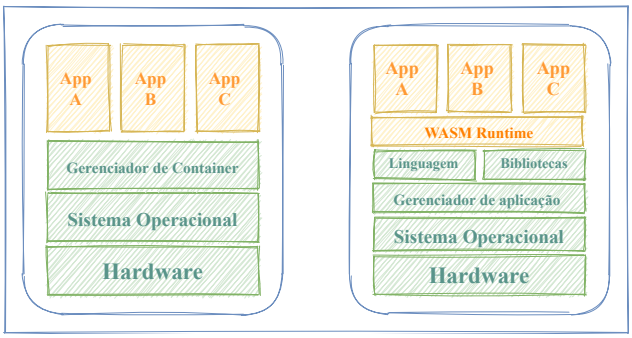


WASI é uma interface de sistema para WebAssembly escrita em Rust e desenvolvida por uma comunidade, cujo objetivo é desenvolver uma ferramenta que dê suporte fora do navegador e mantenha as características de portabilidade e segurança inerentes à linguagem. À vista disso, WASI permite que WebAssembly seja verdadeiramente independente de plataforma e executado de forma nativa em diferentes arquiteturas. A interface o faz iniciando com um módulo fundamental padrão, *wasi-core*, que abrange arquivos, rede e outras funcionalidades, e criando um conjunto de interfaces modulares padrões como ilustra a Figura 4. Em outras palavras, através da abordagem modular, WASI oferece uma boa cobertura padrão de funcionalidades do sistema enquanto limita o uso dessas funções em coerência com as necessidades da plataforma [2]. Assim, portabilidade e segurança são garantidas, pois diferentes implementações e funcionalidades disponíveis podem ser especificadas e implementadas em cada hospedeiro.

IV. PARALELO ENTRE AS ARQUITETURAS

Para atender as demandas de aplicações atuais, soluções para virtualização de recursos devem garantir portabilidade e segurança através de isolamento e controle. A seguir, a implementação desses atributos em ambas plataformas são comparados individualmente, assim como ferramentas de orquestração:

Figura 5: Comparação entre Containers e Wasm



A. Portabilidade

A portabilidade de uma aplicação em ambientes virtuais está fortemente correlacionada com a forma de encapsulamento que a ferramenta fornece. Containers oferecem virtualização a nível de sistema operacional, ou seja, eles são executados no *user system* do *kernel* hospedeiro, e, portanto, são dependentes desse sistema. Em razão dessa dependência, muitos container só podem ser executados em OS iguais ou similares ao do hospedeiro em que foi desenvolvido [9]. A fim de garantir a portabilidade entre diferentes arquiteturas, aplicações em Containers como Docker são encapsuladas em uma imagem com o ambiente de execução da linguagem e dependências embutidas e individuais para cada uma. É possível estabelecer comunicação entre aplicações, mas essa condição de encapsulamento impacta negativamente seu consumo de memória [3].

Assim como a abstração de Containers ocorre em um nível superior a de máquinas virtuais, a virtualização por Wasm ocorre em nível de abstração mais alto que Containers. O foco da portabilidade de WebAssembly não é encapsulamento do ambiente de produção para garantir a execução correta de uma aplicação, um arquivo Wasm deve ser capaz de ser executado em diferentes sistemas e ambientes de execução em uma compilação única. Dessa forma, um único binário pode ser executado dentro do navegador e também em diferentes arquiteturas e sistemas independentemente da linguagem original em que foi programado [10]. Essa nova camada de abstração introduzida em WebAssembly concede maior liberdade à virtualização, permitindo que dependências de aplicação não precisem ser embutidas individualmente no encapsulamento e que possam ser compartilhadas conforme mostra a Figura 5.

Nesse cenário, a portabilidade de aplicações em sistemas não baseados em *web* fica a cargo da integração entre a interface WASI e o ambiente de execução (*runtime*) escolhido pelo desenvolvedor. A natureza modular de WASI permite uma padronização do encapsulamento enquanto garante a liberdade para que as plataformas utilizem apenas as partes que sejam relevantes para a aplicação. O módulo fundamental *wasi-core* foi inspirado em POSIX (Interface Portátil entre Sistemas Operativos) e estabelece suas funcionalidades a partir de chamadas do sistema, no entanto, não suporta operações de processo como fork [2].

B. Segurança

No aspecto da segurança, o objetivo principal é proteger a aplicação de *bugs* ou módulos maliciosos, provendo ao desenvolvedor primitivas úteis com mitigação de riscos. Em WebAssembly, módulos são executados em *sandbox*, ou seja, em um ambiente isolado do ambiente de tempo de execução hospedeiro, de forma que as aplicações sejam executadas deterministicamente e seja incapazes de escapar da *sandbox* sem a supervisão de Interfaces de Programação de Aplicação (API) apropriadas [12]. Para que a virtualização provida seja segura, é indispensável que a ferramenta possua mecanismos para garantia de isolamento e controle de recursos.

Isolamento

Containers como Docker utilizam a funcionalidade de kernel do Linux, namespaces, para isolar processos, redes e sistema de arquivos. Namespaces envolvem a sistema global de recursos em uma abstração que permite que um processo, dentro desse *namespace*, se comporte como se tivesse acesso isolado a todos esses recursos [11]. Dessa forma, Docker provê isolamento de *filesystems*, processos e redes disponibilizando um sistema de arquivos raíz único para cada Container, executando cada um em seu próprio ambiente e separando entre eles interfaces virtuais e endereçamento de protocolo da internet (IP), respectivamente [9].

Através de atributos do próprio WebAssembly é possível alcançar resultados semelhantes ao isolamento por *namespaces*. A representação de memória utilizada em Wasm oferece acesso a *bytes* crus sem permitir acesso direto a memória, por exemplo ponteiros não são permitidos [4]. Essa representação é de uma memória linear isolada, como um *array* de *bytes*, de tamanho fixo e endereçável pelo usuário em tempo de compilação. Assim referências a essa memória são computadas com precisão e WebAssembly garante que a aplicação acesse somente a memória destinada a ela mesma enquanto simplifica checagem de limites durante o processamento [12].

Além disso, *sandbox* fornecida por WASI limita o acesso a recursos e as funcionalidades do sistema. Isso ocorre porque as funções e, consequentemente, as chamadas de sistema permitidas para cada aplicação são determinadas pelo ambiente de tempo de execução. O *runtime* instância os módulos WASI que serão utilizados decidindo quais das suas funções poderão ser acessadas pela aplicação por meio de importações. Dessa forma, ações de possíveis módulos maliciosos ou falhos são limitadas e suprimidas protegendo a aplicação e o sistema de execução [2].

Ao contrário do Docker, WebAssembly não possui um conjunto de diretrizes específicos para acesso a arquivos além da *sandbox*, o ambiente de tempo de execução deve implementar essa função. É responsabilidade do *runtime* estender funcionalidade e impor segurança para qualquer processo Wasm [4]. *Runtimes* são encarregados de especificar modo de compilação e execução, quais plataformas são suportadas, interoperabilidade com outras linguagens, funções adicionais, assim como definir se o código compilado pode ou não ser executado em ambientes não baseados em web. Diversos sistemas de tempo de execução para diferentes fins estão sendo desenvolvidos com o propósito de ampliar a aplicabilidade e portabilidade do WebAssembly [13] [14]. Atualmente, os principais ambientes de tempo de execução que permitem esse tipo de virtualização são wasmtime, wasmer e Lucet.

Controle de recursos

Em ambientes virtuais compartilhados por múltiplas aplicações, é necessário limitar tempo de execução e utilização de memória dos processos. Plataformas baseadas em Containers alocam individualmente para cada Container recursos como CPU e memória através de *cgroups*, ferramenta para controle de grupos em kernel do Linux reponsável por impor limites de recurso no sistema[9].

Paralelamente, em WebAssembly uma memória única e linear é disponibilizada para cada aplicação. Essa é criada em um tamanho inicial, seja ele default ou definido pelo desenvolvedor, em tempo de carregamento, mas com capacidade para crescer posteriormente, de forma dinâmica, até um valor máximo estabelecido [4]. O tempo máximo de execução é definido pela interface do sistema.

C. Orquestração

Com o advento da computação em nuvem, *big data* e IoT, Containers se popularizaram e hoje são amplamente utilizados em sistemas distribuídos. Porém, a implementação de Containers nesse novo ambiente gerou novas demandas de gerenciamento típicas desses sistemas. A solução encontrada para esse problema foi a utilização de plataformas de gerenciamento de *workloads* e serviços. Essas ferramentas devem prover serviços de balanceamento de carga, orquestração de armazenamento, gerenciamento de falhas e monitoramento. Uma das ferramentas de orquestração mais utilizadas hoje é o Kubernetes, uma plataforma *open-source* desenvolvida pela Google [15].

Em abril de 2020, foi anunciado pelo DeisLabs o Krustlet [16], Kubernetes-rust-kubelet, é uma ferramenta para executar *workloads* WebAssembly nativamente em Kubernetes [17]. Krustlet age como um nó em um cluster Kubernetes, de forma que quando um usuário escala uma confirmação de entrega (Pod), o API da Kubernetes o direciona para um nó Krustlet que o executa em um sistema de tempo de execução baseado em WASI. Krustlet ainda é uma tecnologia altamente experimental, com fim de integrar *workloads* WebAssembly e Containers tradicionais orquestrados por Kubernetes [16].

V. CONCLUSÕES

Neste trabalho foi discutido o estado da arte de técnicas de virtualização, desacoplamento, isolamento e portabilidade de aplicações baseadas em containers e WebAssembly. Os containers foram representados principalmente por meio do Docker, ferramenta bem estabelecida no mercado. E WebAssembly foi introduzido como uma opção virtualização em desenvolvimento para aplicações não baseadas em *web*. Ao comparar as duas ferramentas, ficou nítida a capacidade que WASM possui para prover em uma máquina virtual leve, segura, com execução em velocidade quase nativa e alta portabilidade entre linguagens e plataformas.

Essas características tornam WebAssembly atrativo para aplicações de diversas naturezas, mas principalmente para IoT e outros empregos de *Function-as-a-Service*. Contudo, ainda que algumas ferramentas para implementação desse recurso já possuam versões estáveis, esta ainda é uma tecnologia imatura, que possui múltiplos módulos ainda em fase experimental. Dessa forma, o uso comercial e industrial de WebAssembly para execução de aplicações em ambientes virtuais e desacoplamento ainda não é viável, mas esse cenário deve mudar em um futuro próximo.

VI. TRABALHOS FUTUROS

Como objetivo futuro, serão desenvolvidos protótipos de WebAssembly, para validar a proposta do artigo. Outro ponto importante a ser estudado é a integração com os tipos de sistema. Tal integração se dará com um sistema distribuído, composto de dispositivos IoT, onde as informações geradas serão consumidas e enviadas para uma base de dados referente ao processo que o sistema IoT está conectado. O resultado dessa integração é a construção de um *Big Data* para análises de processos industriais.

REFERÊNCIAS

[1] S. Newman, *Building Microservices*, Sam Newman, 2018.

[2] L. Clark *Standardizing WASI: A system interface to run WebAssembly outside the web*. Acedido em 01 de Outubro de 2020, em: `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/`

[3] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken and G. Parmer, "Challenges and Opportunities for Efficient Serverless Computing at the Edge"*Symposium on Reliable Distributed Systems (SRDS)*, pp. 261-2615, Lyon, France, 2019.

[4] A. Hall and U. Ramachandran, "An Execution Model for Serverless Functions at the Edge", *Proceedings of the International Conference on Internet of Things Design and Implementation*, 225–236, 2019.

[5] J. Napieralla, "Considering WebAssembly Containers for Edge Computing on Hardware-Constrained IoT Devices", Dissertation, 2020.

[6] Containers *Containers - Docker descrição*. Acedido em 01 de Outubro de 2020, em: `https://www.docker.com/resources/what-container`.

[7] *WebAssembly Project 2018*. Acedido em 01 de Dezembro de 2020, em: `https://webassembly.org/`.

[8] Disselkoen, Craig Renner, John Watt, Conrad Garfinkel, Tal Levy, Amit Stefan, Deian, "Position Paper: Progressive Memory Safety for WebAssembly"*HASP '19: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1-8, 2019.

[9] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*, James Turnbull, 2014.

[10] D. Gohman *WASI: WebAssembly System Interface*. Acedido em 30 de Novembro de 2020, em: `https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md`

[11] Linux Programmer's Manual. 2018 *namespaces - overview of Linux namespaces*. Acedido em 01 de Outubro de 2020, em: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[12] WebAssembly Project 2018 *Security*. Acedido em 01 de Outubro de 2020, em: https://webassembly.org/docs/security/.

[13] appcypher *Awesome WebAssembly Runtimes*. Github. Acedido em 01 de Outubro de 2020, em: https://github.com/appcypher/awesome-wasm-runtimes.

[14] appcypher *Awesome WebAssembly Languages*. Github. Acedido em 01 de Outubro de 2020, em: https://github.com/appcypher/awesome-wasm-langs.

[15] Kubernetes Documentation *What is Kubernetes?*. Acedido em 01 de Outubro de 2020, em: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[16] DeisLabs *Introducing Krustlet, the WebAssembly Kubelet*. Acedido em 01 de Outubro de 2020, em: https://deislabs.io/posts/introducing-krustlet/.

[17] M. Fisher (bacongobbler) *What is Krustlet?*. Github. Acedido em 01 de Outubro de 2020, em: https://github.com/deislabs/krustlet/blob/master/docs/intro/intro.md.