Hermes: um protótipo de ferramenta de entrega serverless com suporte para GPUs

Tiago Martins Nápoli

Monografia de Trabalho de Conclusão de Curso
apresentado à disciplina
MAC0499
(Trabalho de Formatura Supervisionado)

Orientador: Prof. Dr. Alfredo Goldman
Co-Orientador: Renato Cordeiro Ferreira

São Paulo, Abril de 2019

Resumo

Nos últimos anos, as GPUs têm ganhado cada vez mais importância na comunidade científica e na indústria. Apesar disso, o fluxo de desenvolvimento de aplicações aceleradas por GPUs se mantém com diversos inconvenientes. Um desses é a configuração do ambiente de desenvolvimento: tanto o desenvolvimento local quanto o remoto trazem consigo atritos para compilar e executar código. A alternativa local exige disponibilidade de uma GPU por programador, além de configuração de drivers e kits de desenvolvimento; a alternativa remota exige acesso via SSH e alterações no código com editores de linha de comando. Paralelamente a isso, tem crescido na indústria o uso do paradigma *serverless*, que consiste em um modelo de execução de aplicações em que grande parte da configuração do ambiente de execução é abstraído do desenvolvedor.

Dado este contexto, o objetivo deste projeto é criar um protótipo do que chamamos de Hermes, uma ferramenta de entrega *serverless* de execuções, com suporte para linguagens CUDA e C++ e possibilidade do uso de GPUs. Espera-se, com a utilização do Hermes, amenizar o atrito no fluxo de desenvolvimento para GPUs.

Palavras-chave: *serverless*, GPU, Hermes

Conteúdo

Capítulo 1

Introdução

1.1 Motivação e Objetivos

Nos últimos anos, as GPUs têm ganhado cada vez mais importância na comunidade científica e na indústria. Sua arquitetura altamente paralela mostrou-se efetiva [1] em acelerar diversos tipos de aplicações de grande interesse, como aplicações de Aprendizado de Máquina, Inteligência Artificial, Cálculo Numérico, dentre muitas outras [2]. Essa crescente importância das GPUs tem incentivado inovações na maneira como utilizá-las e provisioná-las: Os principais provedores de nuvem hoje oferecem máquinas virtuais com acesso à GPU; *startups* foram fundadas com produtos que tentam facilitar o fluxo de trabalho com GPUs para certos usos [3]; *frameworks* de certas linguagens passaram a oferecer suporte à aceleração com GPUs [4], dentre outros exemplos.

Apesar dessas inovações citadas, o fluxo de desenvolvimento de aplicações para GPUs ainda tem certos obstáculos. Um desses é a configuração do ambiente de desenvolvimento: tanto o desenvolvimento local quanto o remoto trazem consigo inconvenientes para compilar e executar código. A alternativa local exige disponibilidade de uma GPU por programador, além de configuração de drivers e kits de desenvolvimento; a alternativa remota exige acesso via SSH e alterações no código com editores de linha de comando.

Dado esse inconveniente, a proposta desse projeto é criar o Hermes, uma ferramenta de código aberto com o objetivo de eliminar grande parte dessas dificuldades, mantendo certas vantagens tanto do desenvolvimento local quanto do remoto, como o acesso a GPUs mais poderosas, uma vantagem comum do desenvolvimento remoto, e o uso do ambiente local, com o qual o desenvolvedor está confortável.

O objetivo deste trabalho não é criar um sistema para uso em larga escala, muito menos um produto. O foco é criar um protótipo funcional, que resolve o problema sem preocupação demasiada com performance. Também há grande interesse nos aspectos didáticos do projeto ao proporcionar desafios que permitem ao aluno adquirir experiência com ferramentas, tecnologias e paradigmas com crescente importância no mercado de trabalho.

1.2 Organização do Texto

No próximo capítulo (Capítulo 2) serão introduzidos alguns conceitos necessários para compreensão da utilidade do Hermes e sua implementação. Em seguida, no capítulo 3, uma visão geral do sistema, sua usabilidade e funcionalidades serão apresentados. O capítulo 4 introduzirá a arquite-

tura do Hermes e detalhará os serviços existentes em uma instância. A descrição da implementação destes serviços é feita logo após, no capítulo 5. Finalmente o capítulo 6 conclui este trabalho com um panorama do estado atual da implementação do sistema e possíveis trabalhos futuros.

Capítulo 2

Base Teórica

Para compreender o trabalho a ser realizado neste projeto, há três conceitos essenciais que serão utilizados: virtualização, nuvem e GPUs. Este capítulo apresenta os principais conhecimentos necessários sobre esses assuntos que serão posteriormente utilizados no detalhamento da proposta apresentada no capítulo 3.

2.1 Virtualização

Virtualização é a criação de uma versão virtual – baseada em software – de algum recurso, tais como servidores, dispositivos de armazenamento, redes ou sistemas operacionais. A partir do uso desta técnica é possível aumentar o aproveitamento de elementos de hardware, facilitar seu gerenciamento e diminuir custos. A partir de um recurso de hardware é possível, por meio da virtualização, ter várias versões virtuais dele, eliminando em diversos casos a necessidade de compra de mais hardware. A seguir serão apresentadas duas tecnologias muito importantes nos dias de hoje, que utilizam técnicas de virtualização: as máquinas virtuais e os contêineres.

2.1.1 Máquinas Virtuais

Uma Máquina Virtual (*Virtual Machine* - VM) pode ser definida como um software que fornece uma duplicata eficiente e isolada de uma máquina real [5]. A partir do uso das VMs é possível criar, em uma única máquina, diversos ambientes de execução isolados ao invés de um único, que seria o padrão sem o uso de virtualização.

As Máquinas Virtuais são fundamentadas no chamado ***Virtual Machine Monitor***, ou ***hypervisor***. Ele é o software responsável por hospedar as diversas VMs que podem estar sendo executadas na mesma máquina, atuando como uma camada intermediária entre elas e o hardware real. Nesse contexto, as VMs hospedadas pelo *hypervisor* são chamadas **hóspedes** (***guests***), e a máquina real sobre a qual o *hypervisor* executa é chamada **hospedeiro** (***host***).

Para hospedar as VMs, o *hypervisor* virtualiza todos os recursos de hardware do *host* (*e.g.*, processadores, memória, disco, entre outros), fornecendo uma interface para a utilização dos mesmos. O *hypervisor* também controla a alocação de recursos do *host* para os *guests*, e é responsável por escalonar cada VM de maneira similar a como o SO (Sistema Operacional) escalona processos [6]. Além disso, e essa característica é um dos grandes atrativos das VMs, o *hypervisor* é responsável por garantir o isolamento entre cada VM hospedada [7].

Este isolamento garantido pelo *hypervisor* é um dos grandes responsáveis pelas vantagens que as VMs trazem consigo. Algumas delas estão listadas a seguir [6]:

a) **Encapsulamento**: Como as VMs em uma mesma máquina têm isolamento garantido entre elas, os ambientes de execução de cada uma podem ter configuração totalmente diferentes da outra, inclusive sistemas operacionais distintos. Isso permite a empresas e desenvolvedores executarem suas aplicações em VMs diferentes, cada uma com a configuração adequada para o trabalho a ser realizado [8, p. 17].

b) **Segurança, Confiabilidade e Disponibilidade**: O isolamento a nível de hardware trazido pelo *hypervisor* garante que vulnerabilidades ou falhas em aplicações de alguma VM não afete outras VMs na mesma máquina.

c) **Custo**: As VMs eliminam a necessidade de possuir um servidor exclusivo para cada aplicação, algo que era uma prática comum há algumas décadas, para evitar que a execução de uma aplicação interferisse em outras. Como um único servidor pode executar várias VMs e elas têm isolamento entre si, a carga de trabalho de vários servidores é condensada em um só, garantindo economia nos custos de manutenção, refrigeração e energia, além da diminuição do tempo ocioso dos servidores [8, p. 3-14].

d) **Adaptabilidade à variação na carga de trabalho computacional**: Essa variação pode ser tratada deslocando a alocação de recursos computacionais de uma VM à outra. Existem soluções automáticas que fazem essa alocação de recursos dinamicamente [9].

2.1.2 Contêineres

Um Contêiner pode ser definido como um grupo de processos sobre o qual é aplicado uma camada de isolamento entre eles e o resto do sistema. Este isolamento é feito de modo que o contêiner tenha um sistema de arquivos privado e os processos em seu interior sejam limitados a detectar e utilizar somente recursos do sistema que tenham sido atribuídos ao contêiner [10, p. 916]. Os processos conteinerizados reutilizam o *kernel* e outros serviços (e.g., drivers) do SO em que o contêiner foi criado, retirando a necessidade dos contêineres armazenarem um SO completo em seus sistemas de arquivos.

Deste modo, os contêineres são capazes de criar ambientes de execução isolados, assim como as Máquinas Virtuais, trazendo consigo os inúmeros benefícios que estas oferecem: segurança, encapsulamento, confiabilidade, disponibilidade, dentre outras. Há, entretanto, diferenças cruciais nos modelos de cada um, o que cria a necessidade de um comparativo entre as vantagens e desvantagens dos dois:

a) **Performance**: Pelo fato das VMs dependerem de um *hypervisor*, uma camada a mais entre as instruções dos processos e a execução pelo hardware, há um atraso inerente a todas as instruções de hardware realizadas dentro das VMs. Com os contêineres isso não ocorre, os processos se comunicam diretamente com o *kernel* e atrasos adicionais são pequenos, comparados às VMs [11].

b) **Inicialização**: As VMs, por terem um SO completo, necessitam de um processo de inicialização (*boot*), algo que os contêineres não necessitam, já que utilizam os recursos e *kernel* do

sistema que os hospeda [10, p. 904-906]. Por isso, o tempo de inicialização dos contêineres é muito inferior [12].

c) **Espaço em Disco**: Mais uma vez, por terem um SO completo, as VMs utilizam muito mais espaço em disco se comparadas aos contêineres. Estes necessitam armazenar em seu sistema de arquivos somente as dependências da aplicação que encapsulam [10, p. 904-906].

d) **Segurança**: No caso dos contêineres, todas as instâncias que estiverem ativas em um SO compartilham do mesmo *kernel* e recursos computacionais. Se houver uma falha nesse pontos, todos os contêineres ativos são afetados. As VMs estão livres dessa vulnerabilidade [12].

e) **Isolamento**: Por virtualizarem em nível de hardware, as VMs têm um isolamento superior. O isolamento realizado pelos contêiner é em nível de processo, havendo compartilhamento do *kernel*, por exemplo, como já mencionado [10, p. 904-906].

f) **Flexibilidade quanto à SO**: Em uma mesma máquina pode-se desejar virtualizar vários SOs diferentes, como Windows, Linux, ou MacOS. Isso não é realizável somente com contêineres. Como eles não carregam consigo um SO, e dependem do *kernel* do SO que os hospeda, não seria possível executar um contêiner com uma aplicação de certo SO se este for executado em outro SO diferente. Este não é um problema que as VMs enfrentam [10, p. 904-906].

2.2 Nuvem

A Nuvem (*Cloud*) pode ser definida como um grande conjunto de recursos virtualizados (*e.g.*, hardware, plataformas de desenvolvimento, serviços) com fácil acesso e usabilidade [13]. Estes recursos podem ser dinamicamente reconfigurados e ajustados segundo as variação da necessidade de uso, garantindo o que é denominado **elasticidade**: a alteração na demanda por certo recurso desencadeia ajustes em seu provisionamento, garantindo uso otimizado [13]. Tipicamente, a utilização da nuvem é cobrada segundo um modelo denominado **pague pelo uso** (***pay-per-use***), em que os usuários pagam somente por quanto e o que usarem, segundo alguma métrica criada pelo provedor de nuvem [13]. Neste modelo de cobrança, os usuários são protegidos pelo **contrato de nível de serviço** (***service level agreement***, **SLA**), que descreve os compromissos do provedor quanto à disponibilidade e tempo de atividade dos recursos oferecidos.

Neste contexto, surgiu também o termo **Computação em Nuvem** (***Cloud Computing***), que se refere à utilização dos recursos oferecidos por um provedor de nuvem [14]. Esta utilização e o gerenciamento dos recursos é feita pelo cliente por meio de uma rede, geralmente a Internet.

Atualmente a nuvem assume um papel extremamente importante para a indústria e tem atraído inclusive a comunidade científica, dependendo do balanço de custos [15]. Este projeto utilizará vários conceitos criados e popularizados em função do *Cloud Computing*, e esta seção oferecerá contexto e definições quanto a esses conceitos.

2.2.1 Serviços da Nuvem

Os diversos serviços oferecidos pelos provedores de nuvem podem ser agrupados, tradicionalmente, nas seguintes categorias:

a) **Infraestrutura como Serviço** (***infrastructure as a service***, **IaaS**): Nesse modelo tipicamente são oferecidos recursos de hardware (*e.g.*, processadores, memória, disco, entre outros) e servidores virtualizados. Os clientes, ao optarem por esse serviço, são cobrados somente pelos recursos consumidos e não precisam se preocupar com compra, instalação e manutenção de hardware e servidores em um *data center*, mas sim com a implantação de seus softwares utilizando os recursos virtualizados disponibilizados (*e.g.*, VMs) e o gerenciamento de tais recursos. Ao provedor de nuvem é depositada a responsabilidade de garantir a disponibilidade de recursos de hardware, servidores e poder computacional, para que os clientes possam escalar suas aplicações facilmente [16].

b) **Plataforma como Serviço** (***platform as a service***, **PaaS**): Nesse modelo, o provedor fornece uma infraestrutura de software sobre a qual os clientes podem desenvolver e lançar determinadas classes de aplicações e serviços. A plataforma oferecida é baseada na infraestrutura do provedor, e sua utilização é cobrada seguindo o modelo *pay-per-use*, segundo a utilização de recursos computacionais pelo cliente. Apesar disso, a infraestrutura é abstraída da plataforma, ou seja, ao cliente não é necessário gerenciar recursos virtualizados: seu foco é o produto que esteja desenvolvendo [16].

c) **Software como Serviço** (***software as a service***, **SaaS**): Nesta modalidade, aplicações completas com funcionamento baseado na infraestrutura de um provedor de nuvem são oferecidas ao usuário final. Os clientes do serviço não precisam se preocupar com gerenciamento de infraestrutura ou da plataforma para garantir seu funcionamento. Um exemplo comum aplicação que segue esse modelo são os *WebMails* [16].

2.2.2 Arquitetura *Serverless*

A Arquitetura *Serverless* refere-se a um padrão de entrega de software em que aplicações, de modo geral funções, sem estado (*stateless*) são utilizadas para compor uma aplicação ou parte dela. Essas funções são executadas na nuvem, sem a necessidade de gerenciamento de infraestrutura por parte do desenvolvedor [17]: o provedor de nuvem é responsável pela execução, monitoramento, manutenção e escalabilidade da aplicação.

Nesse modelo, a cobrança pelo serviço é baseada no tempo de execução das aplicações. Isso é uma grande vantagem pelo fato de que, nessa arquitetura, uma aplicação pode ser escalado a zero, ou seja, se não estiver em uso, não será executada [17]. Essas características permitem ao cliente pagar somente pelo tempo que as aplicações foram de fato usadas.

Por fim, outra característica chave das Arquiteturas *Serverless* é a orientação a eventos [17]: a execução de aplicações é feita em resposta a eventos predefinidos.

2.3 GPUs

As GPUs (Unidades de Processamento Gráfico) são processadores que foram criados inicialmente para lidar com operações de renderização de modo mais eficiente. Apesar de possuírem em princípio essa finalidade, a arquitetura altamente paralela que possuem fomentou o interesse no uso de GPUs em aplicações mais gerais, originando o que foi chamado de GPGPU (Computação de Propósito Geral em GPU). Em virtude do imenso desenvolvimento da arquitetura das GPUs e das

práticas de GPGPU desde a sua criação, hoje a importância delas para a comunidade científica e a indústria é enorme: aplicações aceleradas por GPUs são usadas por áreas como Inteligência Artificial, Aprendizado de Máquina, Física Computacional, Química Computacional, Exploração de Petróleo e Gás Natural, Astrofísica, dentre muitas outras [2].

Capítulo 3

Visão geral e usabilidade do sistema

Hermes é uma aplicação cliente-servidor que permite a um usuário, o cliente, se conectar a um servidor específico para utilizar a instância Hermes instalada nele. O projeto desenvolvido inspirou-se em outros produtos e tecnologias *serverless* existentes, principalmente o projeto *OpenFaas*[18], que oferece uma solução para *deploy* de funções de maneira *serverless*, ou seja, abstraindo do desenvolvedor a preocupação com infraestrutura e configuração do ambiente de execução. Assim como esses projetos, Hermes tem como objetivo eliminar a necessidade do usuário de se preocupar com esses elementos, com o diferencial de permitir a execução de códigos que utilizam GPU. Usando o Hermes o usuário pode:

a) Criar aplicações, chamadas *hermes-functions*, em C, C++ ou CUDA, que utilizam ou não GPU.

b) Registrar as *hermes-functions* criadas em uma instância Hermes.

c) Criar execuções das *hermes-functions*, que utilizarão os recursos do servidor em que a instância Hermes está instalada.

Toda a configuração do ambiente de execução é abstraída do usuário e não há necessidade dele possuir uma GPU para executar binários de códigos CUDA, pois serão utilizados recursos do servidor.

No decorrer deste capítulo serão apresentados mais detalhes sobre o que é o Hermes e como utilizá-lo. As seções a seguir apresentam as *hermes-functions* e a usabilidade, configuração e instalação de um cliente e uma instância Hermes.

3.1 *hermes-functions*

As *hermes-functions* são o elemento básico da usabilidade do Hermes. Elas são as aplicações criadas pelos usuários e registradas em alguma instância Hermes para posterior execução. Uma *hermes-function* é composta pelos seguintes elementos:

a) O código fonte da aplicação.

b) Um arquivo Makefile, que descreve como e quais binários devem ser criados.

c) Um arquivo de configuração chamado *hermes.config.json*.

Este último descreve à instância Hermes na qual a *hermes-function* será registrada e posteriormente executada sobre como esta função será nomeada, se a execução dela exige GPU e, quando executada, qual o binário deverá ser utilizado como ponto de entrada (*entrypoint*). Um exemplo de *hermes.config.json* seria o seguinte:

```
{
  "functionName": "pi-montecarlo",
  "language": "cuda",
  "gpuCapable": true,
  "functionVersion": "1.0.0",
  "handler": "./a.out"
}
```

As configurações mostradas acima são interpretadas da seguinte maneira:

a) *functionName* e *functionVersion*: estes campos, juntamente com o nome do usuário que criou a função, são utilizados para identificar unicamente uma *hermes-function* no servidor. Ao usuário essa identificação será no formato *nomeDeUsuário/functionName:functionVersion*. A função descrita no exemplo acima pelo *hermes.config.json*, criada pelo usuário *tiago*, por exemplo, seria identificada como *tiago/pi-montecarlo:1.0.0*.

b) *language*: especifica a linguagem de programação daquela *hermes-function*. Os valores possíveis são *cpp* ou *cuda*.

c) *gpuCapable*: indica a necessidade ou não de uso de GPU para execução da *hermes-function*. Os valores possíveis para esse campo são *true* ou *false*.

d) *handler*: contém o caminho para o binário que será gerado conforme as instruções do Makefile e que deverá ser executado pelo servidor quando solicitado. Este caminho é relativo à raiz do projeto. Como exemplo, se o binário a ser executado está na raiz da pasta do projeto e é nomeado como *a.out*, o caminho deve ser './a.out'.

3.2 Cliente Hermes

Um cliente Hermes é qualquer utilizador da API de uma instância Hermes. Para usuários comuns (que não *bots* ou programas), foi criada uma interface de linha de comando (*Command Line Interface* - CLI) para interação com a API do servidor. Esta seção tratará desta CLI: instalação, configuração e usabilidade.

3.2.1 Instalação

A CLI para utilização de uma instância Hermes foi disponibilizada como um pacote NPM, um repositório de pacotes NodeJS, e está disponível em [19]. Esta página documenta o passo-a-passo para instalação e os requisitos para uso: possuir o programa Docker[20] instalado na máquina e possuir um usuário em Dockerhub[21]. A necessidade destes requisitos serão discutidas no capítulo 5, sobre detalhes de implementação, e também estão disponíveis em [19]. Além destes requisitos também é necessário possuir o programa yarn[22], um gerenciador de pacotes NodeJS. Ele fará todo o papel de baixar e instalar a CLI mediante o comando:

```
yarn global add @hermes-serverless/cli
```

Após isso a CLI já estará instalada e poderá ser utilizada por meio do comando *hermes* no terminal.

3.2.2 Configuração

A configuração da CLI é simples e é feita utilizando os subcomandos de `hermes config`. Inicialmente é preciso somente configurar o usuário do Dockerhub e a URL da instância Hermes que se deseja utilizar:

a) Nome de usuário do Dockerhub:

```
hermes config docker.username <nomeDeUsuarioDockerhub>
```

b) URL da instância Hermes:

```
hermes config hermes.url <urlDaInstancia>
```

Para a comunidade BCC-IME foi criada uma instância Hermes disponível em:

http://ratel.ime.usp.br:9090

Para utilizá-la bastaria executar:

```
hermes config hermes.url http://ratel.ime.usp.br:9090
```

3.2.3 Usabilidade

O fluxo básico de utilização idealizado envolve a criação de novas *hermes-functions*, compilação destas, para checagem de erros, *deploy* da *hermes-function* na instância Hermes e criação de execuções. Este fluxo, juntamente com outras funcionalidades necessárias, foram base para criação dos comandos da CLI que são resumidos pelo comando de ajuda implementado:

```
Usage: @hermes-serverless/cli <command> [options]

Commands:

  config [property] [newValue]     Change or print a config

  execution list                List your executions
  execution inspect <runID>     Get info on an execution

  function build                                          Build a hermes-function locally
  function delete <functionName> <functionVersion>        Delete a function from remote
  function deploy                                         Deploy a function to remote
  function init                                           Init a hermes-function
  function list                                           List your hermes-functions registered on remote
  function run <functionID>                               Start a hermes-function execution

  remote login <username>       Login into a remote Hermes server
  remote logout                 Logout from an remote Hermes server
  remote register               Register into a remote Hermes server
  remote unregister             Unregister from a remote Hermes server
  remote whoami                 Check the user you're logged on

Options:

  -h, --help  show help information
  -v, --version  show version number
```

Esta seção detalhará as funcionalidades disponíveis, cada uma relacionada a um dos comandos acima.

Criar um novo projeto de *hermes-function*

A criação de novos projetos de *hermes-functions* é facilitada pelo comando `hermes function init`. Quando executado serão solicitadas ao usuário informações sobre a função a ser criada: nome, linguagem e se deveria utilizar GPU. Após isso um diretório com o nome da função será criado e nele três arquivos: *hermes.config.json*, *Makefile* e *main.cpp* ou *main.cu*. O *hermes.config.json* refletirá as configurações desejadas pelo usuário, e o *Makefile* e o arquivo fonte serão um exemplo básico funcional de *hermes-function*.

Compilar uma *hermes-function*

A fim de checar se o projeto compila sem ter a necessidade de comunicação com o servidor foi criado o comando `hermes function build`. O Makefile fornecido pelo usuário será utilizado para compilar os arquivos fonte e algumas checagens de integridade serão realizadas (se o binário especificado no *hermes.config.json* foi gerado, por exemplo). Todo esse processo é feito localmente.

Fazer *deploy* da *hermes-function*

Quando pronta a *hermes-function*, ou para testá-la, o usuário poderá fazer *deploy* dela na instância Hermes por meio do comando `hermes function deploy`. Se o usuário já fez *deploy* anteriormente da mesma função ele poderá utilizar a opção `--update` para substituir a função registrada no servidor. Feito o *deploy* o usuário já poderá realizar execuções.

Listar *hermes-functions*

Todas as *hermes-functions* registradas pelo usuário em determinada instância Hermes podem ser listadas pelo comando `hermes function list`, que exibirá uma tabela como a seguinte:

Function	Language	GPU Capable	Watcher Image
char-printer:1.0.0	cpp	true	tiagonapoli/watcher-char-printer:1.0.0
gpu-pi-montecarlo:1.0.0	cuda	true	tiagonapoli/watcher-gpu-pi-montecarlo:1.0.0
pi-montecarlo:1.0.0	cpp	false	tiagonapoli/watcher-pi-montecarlo:1.0.0
pi-montecarlo:1.0.1	cpp	false	tiagonapoli/watcher-pi-montecarlo:1.0.1
reduce-sum:1.0.0	cpp	false	tiagonapoli/watcher-reduce-sum:1.0.0
waiter:1.0.0	cpp	false	tiagonapoli/watcher-waiter:1.0.0

As informações exibidas são o nome da *hermes-function*, se ela utiliza GPU, a linguagem de programação e a imagem Docker relacionada àquela função, que será utilizada pela instância Hermes (mais informações sobre isso no capítulo 5).

Excluir *hermes-function*

Uma *hermes-function* registrada no servidor pode ser facilmente excluída utilizando o comando `hermes function delete`. A exclusão de uma *hermes-function* eliminará todo o histórico de

execuções associadas a ela.

Criar uma execução

Um usuário pode criar execuções das *hermes-functions* registradas por ele. As execuções, que ocorrem no servidor, podem ser de dois tipos: síncronas ou assíncronas. As síncronas mantém um canal de comunicação aberto com o servidor, permitindo que o usuário receba a saída da execução em tempo real. As assíncronas fecham o canal de comunicação com o servidor, e fornecem ao usuário um ID com o qual pode-se checar o *status* daquela execução. O comando `hermes function run` permite a criação de novas execuções, que serão síncronas ou assíncronas dependendo das opções `--async` ou `--sync` passadas a ele.

Quando executado, este comando solicitará ao usuário a entrada que será repassada ao programa e, após isso, se a execução for síncrona, a saída do programa será exibida ao usuário, senão, se a execução for assíncrona, um ID será exibido. Ele poderá ser usado com o comando `hermes execution inspect` para checar o *status* da execução.

O comando `hermes function run` também aceita a opção `--file caminhoDoArquivo`, que tomará o conteúdo do arquivo como entrada. Isso facilita a utilização desta funcionalidade com *pipes* do Linux, por exemplo:

```
hermes function run tiago/hello-world:1.0.0 --file input --sync | cat
```

Listar execuções

Todas as execuções realizadas pelo usuário podem ser listadas pelo comando `hermes execution list`. Ele exibirá uma tabela como a seguinte:

RunID	Status	Start	End	Elapsed	Function
29	running	----- --:--:--.---	----- --:--:--.---	----- --:--:--.---	tiago/waiter:1.0.0
12	success	10/08 09:48:55.622	10/08 09:48:55.650	00:00:00.28	tiago/reduce-sum:1.0.0
13	success	10/08 09:49:11.500	10/08 09:49:11.540	00:00:00.40	tiago/reduce-sum:1.0.0
14	success	10/08 09:56:29.300	10/08 09:56:29.993	00:00:00.693	tiago/gpu-pi-montecarlo:1.0.0
15	success	10/08 09:57:36.771	10/08 09:57:56.804	00:00:20.33	tiago/waiter:1.0.0
28	success	10/27 16:09:37.578	10/27 16:09:37.611	00:00:00.33	tiago/char-printer:1.0.0

As informações exibidas são o ID da execução, que pode ser usado para inspecioná-la, a data de início e fim, o tempo de execução e a função executada.

Checar status e resultados de execuções

Todas as execuções realizadas podem ser inspecionadas utilizando o comando `hermes execution inspect`, que recebe como parâmetro o ID da execução. Ele exibirá a saída do programa até aquele momento, o tempo de execução, *status* (se terminada ou não), data de início e fim. Esse comando pode ser facilmente utilizado em conjunto com *pipes* do Linux se a opção `--result` for passada, que indica que somente a saída do programa deverá ser exibida.

Gerenciamento de usuário

Todas as operações relacionadas às *hermes-functions* necessitam que o usuário esteja logado em uma instância Hermes. Para isso foram criados comandos simples que permitem que o usuário se registre, faça *login* ou *logout*, confira o usuário logado ou exclua seu usuário:

```
remote login <username>       Login into a remote Hermes server
remote logout                 Logout from an remote Hermes server
remote register               Register into a remote Hermes server
remote unregister             Unregister from a remote Hermes server
remote whoami                 Check the user you're logged on
```

3.3 Servidor Hermes

Um servidor ou instância Hermes pode ser instalado em qualquer computador e disponibilizado para acesso externo. Uma instância expõe uma API que pode ser utilizada via protocolo HTTP por clientes, idealmente a CLI implementada. Uma instância Hermes exige alguns prerrequisitos e configurações que serão abordadas a seguir

3.3.1 Instalação e configuração

A instalação e configuração de uma nova instância Hermes foi facilitada utilizando *scripts* disponíveis no repositório `https://github.com/hermes-serverless/hermes-install` [23].

Para iniciar uma instância, alguns requisitos são necessários, sob certas condições de versão:

a) Docker Engine: v19.03.2 ou superior.

b) NVIDIA Container Toolkit: v2.1.1 ou superior.

c) NVIDIA driver para o sistema operacional em questão.

Os requisitos relacionados à NVIDIA são necessários para o uso da GPU, e o Docker Engine é necessário para todas as operações com contêineres feitas pelo Hermes. Todos esses requisitos e mais informações estão documentadas em [23].

Após a instalação dos requisitos a instância pode ser inicializada executando o *script start.sh* disponível no repositório de instalação. Basta escolher a porta em que o servidor escutará e ele será inicializado escutando requisições em *localhost:PortaEscolhida*. Com isso o administrador do sistema pode personalizar o acesso àquela porta, permitindo algum controle de usuários da instância, algo que, no protótipo criado, não foi implementado.

Capítulo 4

Arquitetura

Este capítulo apresentará a arquitetura de uma instância Hermes, detalhando as funcionalidades de cada serviço e as interações que realiza com outros elementos do sistema. A figura abaixo apresenta uma visão geral da arquitetura, apresentando os serviços existentes e com quem cada um deles interage (uma seta de um componente A para B significa que A interage com B):

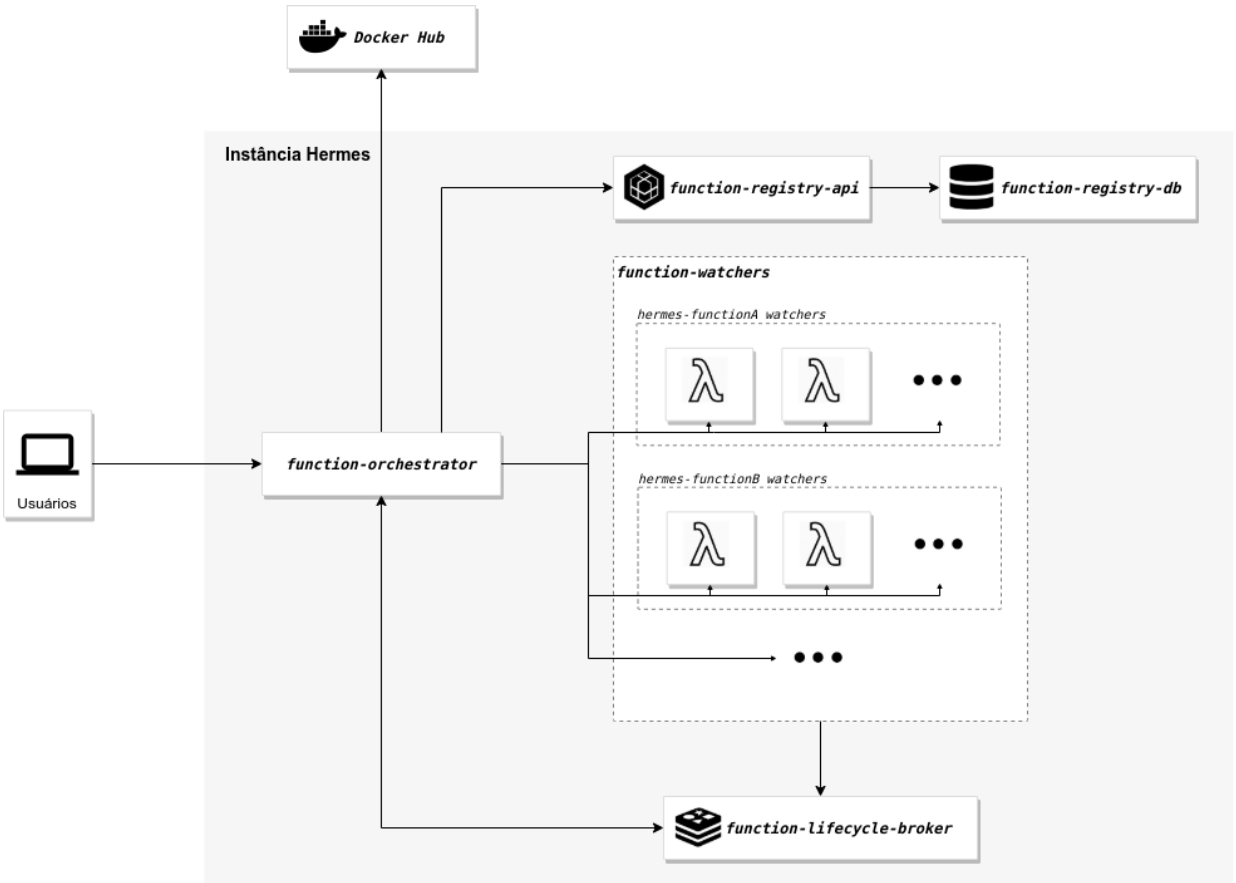

Figura 4.1: Arquitetura de uma instância Hermes

Uma instância Hermes possui os seguintes serviços: *function-orchestrator*, *function-registry-api*, *function-registry-db*, *function-lifecycle-broker*. Também possui diversas instâncias de *function-watchers* (cada uma representada pela letra λ), que são agrupadas logicamente entre *function-watchers* da mesma *hermes-function* (mais detalhes a seguir).

4.1 *function-watcher*

O *function-watcher* é o serviço responsável por todas tarefas relacionadas à execução de uma *hermes-function* específica. Cada instância deste serviço consiste em um embrulho (*wrapper*) ao redor dos binários gerados a partir do código de uma *hermes-function* criada por um usuário. Esse *wrapper* é um servidor HTTP que expõe e implementa uma API para controle do fluxo de execução desses binários. Como cada *hermes-function* gera um tipo novo de *function-watcher*, os *function-watchers* ativos em uma instância podem ser agrupados pela *hermes-function* pela qual aquele tipo é responsável, o que explica a agregação lógica feita na figura 4.1. Nota-se que um servidor Hermes pode possuir diversas instâncias de *function-watchers* ativas.

4.1.1 Funcionalidades

Como já mencionado, as funcionalidades implementadas para o *function-watcher* são relacionadas ao fluxo de execução dos binários da *hermes-function* pela qual a instância é responsável: criação de uma execução, *streaming* de entrada e saída do programa, checagem de status e encerramento de uma execução. Todas essas funcionalidades são expostas pela API servida.

Criação de execução

Esta é a funcionalidade básica do *function-watcher* e se refere à execução do binário da *hermes-function* com uma determinada entrada fornecida pelo usuário[1]. O *streaming* e persistência da saída do programa também são realizados e entram no escopo desta funcionalidade.

Os tipos de execução que podem ser criadas são síncronas ou assíncronas. Esses termos são utilizados para descrever como se dá a comunicação da instância do *function-watcher* com o usuário da API. No caso de execuções síncronas o *streaming* de entrada e saída é feito em tempo real, o que exige que o canal de comunicação entre o usuário e a instância seja mantido. No caso assíncrono isso não é necessário, o usuário envia o *input*, a execução é iniciada e logo em seguida o canal de comunicação é encerrado: o usuário, caso deseje a saída da execução e outras informações, deverá realizar outra requisição ao *function-watcher*.

Verificação do status de uma execução

As informações coletadas e persistidas pelo *function-watcher* são acessíveis ao usuário por meio de uma chamada à API implementada, que devolve um JSON cujos campos contêm dados de:

a) Estado da execução (*running*, *error*, *success*)

b) Momento de início

c) Tempo decorrido

d) Saída do programa

[1] No contexto de uma instância do Hermes o usuário do *function-watcher* é o *function-orchestrator*

Download dos resultados de uma execução terminada

A saída de uma execução terminada e outros metadados, como momento de início e fim da execução e se houve algum erro, são persistidos no disco da instância do *function-watcher* que realizou a execução. Essas informações ficam disponíveis para download por meio de duas rotas da API implementada. Uma delas devolve a saída da execução como um arquivo de texto, e a outra devolve um JSON com os metadados da execução.

Limpeza de arquivos temporários

Os arquivos de saída e metadados de execuções podem ser deletados utilizando uma chamada à API. Esta funcionalidade foi implementada baseada na ideia de que o disco de uma instância de *function-watcher* deveria ser somente um armazenamento temporário dos metadados de execuções. Um armazenamento a longo prazo deveria ser feito em outro serviço.

4.1.2 Ciclo de Vida

O ciclo de vida de uma instância do *function-watcher* começa com a inicialização do servidor HTTP que serve a API de execuções. Assim que o servidor é criado, um evento anunciando esse fato é enviado ao *function-lifecycle-broker* a fim de informar, a quem interesse, que a instância do *function-watcher* está pronta para receber requisições de execução. Em caso de erro na inicialização do servidor um evento também é enviado ao *function-lifecycle-broker*. Deste modo não é necessário ao *function-watcher* saber qual instância de qual serviço o está iniciando.

Quando uma instância do *function-watcher* recebe uma requisição de execução, um *fork* do binário da *hermes-function* é criado e o *streaming* de entrada e saída é realizado. Ao fim da execução um evento é enviado ao *function-lifecycle-broker*, deste modo não é necessário manter um canal de comunicação aberto com o chamador, e o *function-watcher* também não precisa saber quem o chamou.

Por fim, quando uma instância do *function-watcher* já não é mais necessária, é possível desligá-la e removê-la utilizando uma rota de *shutdown* implementada.

4.2 *function-lifecycle-broker*

O *function-lifecycle-broker* implementa o padrão de design *event-broker*. Sua única função é receber eventos publicados por seviços e repassá-los a quem se inscreveu para receber aquele tipo de evento. No *Hermes*, o *function-lifecycle-broker* atua recebendo e repassando eventos relacionados ao ciclo de vida de instâncias do *function-watcher* e ao ciclo de execuções das *hermes-functions*, realizadas também pelos *function-watchers*.

4.3 *function-registry*

A persistência de dados relativos aos usuários de uma instância do *Hermes*, *hermes-functions* registradas e execuções criadas são função do *function-registry*. Este se divide em dois serviços, o *function-registry-db*, que é simplesmente um banco de dados SQL, e o *function-regsitry-api*, um

servidor HTTP que implementa uma API para CRUD de usuários, *hermes-functions* e informações de execuções.

4.4 *function-orchestrator*

Toda a integração entre os serviços de uma instância Hermes para realizar de fato uma funcionalidade do sistema é feita pelo *function-orchestrator*. Ele tem três funções primárias (que, em princípio, poderiam gerar três serviços, mas por simplicidade foram condensadas em um só):

a) *Gateway*: O *function-orchestrator* é o nó de comunicação de uma instância Hermes com o mundo externo. É ele quem expõe a API que implementa as funcionalidades disponibilizadas pelo Hermes a usuários externos – as CLIs utilizadas por usuários, principalmente.

b) Orquestrador de *function-watchers*: o *function-orchestrator* também é responsável por criar e remover instâncias de *function-watchers*, dependendo da necessidade (requisições de execução) ou falta de uso (instâncias há algum tempo sem utilização).

c) Armazenador de resultados de execuções: o *function-orchestrator* persiste em seu disco as saídas e metadados de execuções terminadas, permitindo posterior acesso a esses dados.

Nas seções a seguir serão detalhadas as funcionalidades de cada subserviço do *function-orchestrator* e suas interações com outros elementos da instância Hermes.

4.4.1 Serviço de Orquestração

No quesito de orquestração, o *function-orchestrator* é responsável pelo gerenciamento de instâncias de *function-watcher*s. Ele cria e remove instâncias, dependendo das requisições de execução de certo *hermes-function*, ou inatividade de instâncias de *function-watchers*.

Criação de novas instâncias de *function-watchers*

O serviço de orquestração pode criar novas instâncias de *function-watchers* de determinada *hermes-function*. Quando recebe uma solicitação para isso ele solicita ao *function-registry-api* os metadados da função, que possuem informações da localização da imagem do contêiner daquele *function-watcher*, além da informação se aquela *hermes-function* deve ter em seu ambiente de execução a disponibilidade de GPUs. Com essas informações o serviço de orquestração pode criar uma nova instância do *function-watcher*. Após criado, entretanto, a nova instância não está imediatamente disponível, pois precisa de alguns instantes para inicialização. Para resolver esse problema e ser notificado quando uma nova instância está preparada, o *function-orchestrator* se inscreve em uma lista de eventos de inicialização específica para aquela instância no *function-registry-api*. Quando estiver pronto, o *function-watcher* enviará um evento ao *function-lifecycle-broker*, que por sua vez notificará os inscritos naquela lista, no caso o *function-orchestrator*, que saberá então a partir de que momento uma instância está de fato pronta.

Gerenciamento das instâncias ativas

O *function-orchestrator* mantém uma lista de instâncias ativas de cada tipo de *function-watcher* (que variam dependendo da *hermes-function* que servem). Ele mantém informações de quantas

execuções estão sendo feitas naquela instância e quanto tempo ela está inativa, podendo, assim, gerenciar para qual instância será enviada uma nova execução.

Desligamento de instâncias

Quando uma instância está há muito tempo inativa, sem receber requisiçõess de execução, o *function-orchestrator* solicita seu desligamento, a fim de economizar recursos computacionais.

4.4.2 Serviço de *Gateway*

CRUD de usuários e *hermes-functions*

Uma instância do Hermes persiste informações de usuário e metadados de *hermes-functions*, detalhados na seção de implementação, assim foi necessária uma API, exposta para o mundo externo pelo *function-orchestrator*, que permite CRUD de usuários e *hermes-functions*.

O fluxo de requisição para esta funcionalidade é simples: quando o *function-orchestrator* recebe uma requisição de CRUD ele, de modo geral, somente realiza algumas validações e chama o *function-registry-api*, solicitando a realização da operação desejada com os dados recebidos. Os dados recebidos como resposta são filtrados se necessário (de modo geral alguns dados, como ID de um objeto no banco de dados não deveriam ser repassados ao usuário), e repassados ao usuário. Todo esse processo mantém o canal de comunicação com o chamador aberto, até que a resposta seja enfim enviada a ele.

Execução de *hermes-functions*

O *function-orchestrator* expõe para os usuários daquela instância do Hermes as funcionalidades de execuções síncronas e assíncronas dos *function-watchers*.

Ao receber uma solicitação de execução de uma *hermes-function*, o serviço de *gateway* solicita ao serviço de orquestração uma instância de *function-watcher* para aquela *hermes-function*. Ao receber uma instância, a requisição é repassada a ela. Se a execução é síncrona o canal de comunicação é mantido aberto, entre usuário com *function-orchestrator* e *function-orchestrator* com aquele *function-watcher*. Se a execução é assíncrona o *function-orchestrator* espera a resposta da instância de *function-watcher* e a repassa ao usuário, fechando, em seguida, o canal de comunicação.

Status ou saídas de execuções

O *function-orchestrator* permite a solicitação do status ou saída de uma certa execução. A interação com outros serviços ou sistemas depende do estado da execução em questão, se terminada ou não.

- Se a execução já foi terminada, é solicitado ao serviço de persistência o arquivo de saída da execução, se essa foi a requisição, ou ao *function-registry-api* os dados referentes àquela execução: momento de início e fim, *hermes-function* executada, status da execução (se successo ou erro).
- Se a execução não terminou a requisição de status ou saída da execução é repassada à instância do *function-watcher* responsável. Com a resposta, o *function-orchestrator* simplesmente a repassa ao chamador.

4.4.3 Serviço de Persistência

O *function-orchestrator* também possui um serviço simples de persistência. Essa escolha arquitetural, que não é ideal, foi feita por questões de simplicidade para a implementação inicial. Esse serviço é unicamente responsável por solicitar as saídas de execuções realizadas aos respectivos *function-watchers* e armazenar esses dados em disco, para posterior consulta.

Saída de execuções

Quando uma execução começa, o *function-orchestrator* se inscreve no *function-lifecycle-broker* para ser notificado acerca do término daquela execução. Assim que terminada, o *function-orchestrator* solicita o arquivo de saída da execução e o persiste no disco da instância do *function-orchestrator*, disponibilizando-o para posterior visualização.

Capítulo 5

Implementação

Este capítulo apresentará alguns detalhes de implementação que auxiliam o entendimento do funcionamento dos serviços de uma instância Hermes. Alguns deles, sendo projetos já prontos e documentados em outras fontes, ou padrões comuns de arquitetura de sistema, não serão ricamente documentados. Outros, que possuem funcionalidades complexas ou várias interações com outros serviços, serão mais detalhados. Estes mais ricamente documentados serão o *function-watcher* e o *function-orchestrator*.

Todo o código dos serviços de uma instância Hermes e pacotes auxiliares criados estão disponíveis e livres para contribuição em [24].

5.1 *function-watcher*

Os tópicos a seguir tratarão de aspectos importantes para o entendimento do *function-watcher*. Serão abordados detalhes sobre a montagem e estrutura das imagens Docker, fluxos de funcionamento e questões de segurança resolvidas ou em aberto.

5.1.1 Imagens Docker

Como já mencionado no capítulo 4, cada *hermes-function* gera um novo tipo de *function-watcher*. Isso ocorre porque para cada *hermes-function* existe uma imagem Docker de *function-watcher* associada a ela. As distinções entre imagens são os binários de *hermes-function* que possuem e, em caso de linguagens de programação diferentes, os programas necessários para que a *hermes-function* seja executada. A seguir serão detalhados o processo de montagem e estrutura de pastas dessas imagens, que evidenciam as diferenças citadas.

Montagem

O processo de montagem de uma imagem de *function-watcher* foi feito de modo a tentar minimizar o tempo de montagem e o tamanho final da imagem. Para isso são utilizadas imagens auxiliares pré-montadas. A seguir estão detalhadas essas imagens – para que servem e onde se localizam seus *Dockerfiles* no repositório do *function-watcher*[25]:

a) **`hermeshub/watcher-base-$language`**: Esta imagem possui os programas necessários para executar binários da linguagem de programação em questão (`$language`), além de

possuir o programa *node*, necessário para executar o servidor do *function-watcher*. Como exemplo, para a linguagem CUDA a imagem é nomeada `hermeshub/watcher-base-cuda` e contém os programas necessários para comunicação com GPUs NVIDIA.

Os *Dockerfiles* dessas imagens estão na pasta `watcherBase`, na raiz do repositório do *function-watcher*, nomeados como `$linguagem.Dockerfile`. A escolha de criar essa imagem base permite uma fácil extensão das linguagens suportadas – basta criar um novo Dockerfile com os programas necessários para rodar os binários ou o código dessa linguagem de programação.

b) **`hermeshub/watcher-$language`**: Esta imagem usa como base a anterior e adiciona os arquivos relacionados ao servidor HTTP do *function-watcher*, que está codificado em *Typescript* e contido na pasta `src` do repositório. Os arquivos dessa pasta são compilados, as dependências são instalados e por fim é gerada uma imagem que contém todo o necessário para executar o servidor, faltam somente os binários da *hermes-function*.

O *Dockerfile* dessa imagem é o arquivo *watcher.Dockerfile*, localizado na raiz do projeto.

As imagens acima são criadas quando uma nova versão do *function-watcher* é lançada. No repositório há um *script*, *deployWatcherImages.sh*, que monta essas imagens e as envia ao Docker Hub do Hermes, *hermeshub*, onde ficarão públicas para download.

O último processo de montagem é feito no computador do usuário, quando ele executa o comando `hermes function deploy` da CLI. Esse processo possui duas etapas:

a) **Geração dos binários da *hermes-function*:** Nesse passo será utilizado um *Dockerfile* disponível no repositório [26] com instruções para geração dos binários da linguagem de programação. O *Dockerfile* utilizado será o de nome *$linguagem.Dockerfile*. Esse arquivo é baixado e utilizado para criar a imagem `$usuarioDockerhub/build-$nomeDaFuncao:$versaoDaFuncao`, que ficará temporariamente armazenada no computador do usuário, para uso no próximo passo. Para criá-la, a montagem do *Dockerfile* é realizada usando o diretório da *hermes-function* como contexto de montagem.

b) **Criação da imagem final do *function-watcher***: Nesse passo será gerada a imagem `$usuarioDockerhub/watcher-$nomeDaFuncao:$versaoDaFuncao`, que é o *function-watcher* definitivo para aquela *hermes-function*. Essa imagem é gerada usando como base a `hermeshub/watcher-$language` – o único passo de montagem é a cópia dos binários da *hermes-function* em `$usuarioDockerhub/watcher-$nomeDaFuncao:versaoDaFuncao`.

Após a montagem de `$usuarioDockerhub/watcher-$nomeDaFuncao:$versaoDaFuncao`, ela é enviada para o Docker Hub do usuário, tornando-se pública para *download*. O nome da imagem e outros metadados relacionados à função, como nome e versão, são enviados à instância Hermes e guardados no banco de dados. Desse modo, ao receber uma requisição de execução dessa *hermes-function* o sistema saberá qual imagem deverá ser utilizada e poderá baixá-la do Docker Hub.

Estrutura de pastas

A estrutura de pastas da imagem final de um *function-watcher* facilita a compreensão de seu funcionamento. Essa estrutura é criada na montagem da `hermeshub/watcher-$language` e com-

pletada com a criação da imagem `$usuarioDockerhub/watcher-$nomeDaFuncao:$versaoDaFuncao`, que é a imagem final do *function-watcher*:

a) **`/app/server`:** Nesse diretório estão os arquivos necessários para execução do servidor HTTP – a aplicação NodeJS que expõe e implementa a API do *function-watcher*. Esse diretório contém os arquivos *js* compilados a partir do projeto *Typescript* do *function-watcher*, e as dependências necessárias para executá-los. Esses arquivos são gerados no processo de montagem da imagem `hermeshub/watcher-$language`, que ocorre somente quando uma nova versão de *function-watcher*[25] é lançada.

b) **`/app/function`:** Nesse diretório são copiados todos os arquivos da *hermes-function* criada pelo usuário, juntamente com os binários gerados. Esses arquivos são copiados da imagem de montagem dos binários – `$usuarioDockerhub/build-$nomeDaFuncao:$versaoDaFuncao`.

c) **`/app/io/in`:** Nesse diretório serão criados os arquivos de entrada de execuções. O nome de cada arquivo será o ID da execução ao qual ele se refere. Esse diretório é criado vazio na montagem de `hermeshub/watcher-$language`.

d) **`/app/io/all`:** Nesse diretório estarão os arquivos de saída das execuções. Novamente cada arquivo terá como nome o ID da execução ao qual se refere, e o conteúdo de cada arquivo será a intercalação do *stdout* e *stderr* da *hermes-function* executada. Esse diretório também é criado vazio na montagem de `hermeshub/watcher-$language`.

5.1.2 Funcionamento

Uma instância de *function-watcher* é um container da seguinte imagem:

`$usuarioDockerhub/watcher-$nomeDaFuncao:$versaoDaFuncao`

Este container, quando iniciado, executa o comando `node index.js` no diretório `/app/server`, iniciando o servidor HTTP. Este servidor, que não varia entre diferentes imagens de *function-watcher*, é responsável pelas funcionalidades disponíveis, que serão detalhadas nos tópicos a seguir.

Execução da *hermes-function*

A funcionalidade de execução da *hermes-function* foi implementada utilizando o pacote *execa*[27], que consiste em um *wrapper* ao redor da biblioteca *child_process*, nativa do NodeJS. Esse pacote permite:

a) Criar processos filhos que executam binários especificados.

b) Fazer *streaming* da entrada e saída do processo (o processo pai envia entrada para o processo filho e recebe sua saída).

c) Saber quando um processo filho criado é concluído.

d) Matar processos filhos.

Uma solicitação de execução da *hermes-function* ao *function-watcher* é feita por meio de uma requisição HTTP na rota `/run/async` ou `/run/sync` com o campo *x-run-id* no cabeçalho da requisição. Este campo especifica o ID daquela execução. O corpo da requisição deverá conter a entrada que será utilizada na execução da *hermes-function*.

Quando uma requisição ao servidor é feita na rota `/run/sync`, uma execução síncrona será criada. O fluxo de funcionamento será o seguinte:

a) Arquivos com nome equivalente ao ID da execução são criados em `/app/io/all` e `/app/io/in`, um em cada diretório.

b) A *stream* relacionada ao corpo da requisição HTTP é acoplada ao arquivo `/app/io/in/$ID`, assim todo dado que chegar no corpo da requisição é também armazenado nesse arquivo.

c) Um processo filho é criado, executando o binário especificado no campo *handler* do *hermes.config.json*.

   A *stream* de entrada desse processo é acoplada à *stream* do corpo da requisição HTTP, assim todos os dados do corpo da requisição são repassados como entrada do programa.

   A *stream* de saída do processo, que consiste na intercalação do *stdout* e *stderr*, é acoplada tanto ao arquivo `/app/io/all/$ID` quando à *stream* de resposta da requisição HTTP, assim a saída é armazenada no arquivo e também repassada ao chamador da API.

d) Quando a execução termina e o *stream* da saída é concluído, um *trailer* é adicionado na resposta da requisição. Este conterá um campo *x-result*, com valor *error* ou *success*, indicando o resultado da execução – deste modo o chamador da API poderá saber se tudo passou com sucesso. Além disso, um evento anunciando o término daquela execução (ela possui um ID identificando-a) é enviado ao *function-lifecycle-broker*.

Quando uma requisição ao servidor é feita na rota `/run/async`, uma execução assíncrona será criada. O fluxo é mais simples:

a) Arquivos com nome equivalente ao ID da execução são criados em `/app/io/all` e `/app/io/in`, um em cada diretório.

b) O corpo da requisição HTTP incidente é totalmente drenado e direcionado ao arquivo `/app/io/in/$ID`.

c) A requisição HTTP é encerrada assim que seu corpo seja totalmente drenado. Uma resposta é enviada – um JSON com o ID da execução e o instante de encerramento do canal de comunicação.

d) Um processo filho é criado, executando o binário especificado no campo *handler* do *hermes.config.json*.

   A *stream* de entrada do processo é acoplada ao arquivo `/app/io/in/$ID`, que contém todo o corpo da requisição.

   A *stream* de saída do processo filho, que consiste na intercalação do *stdout* e *stderr*, é direcionada ao arquivo `/app/io/all/$ID`.

e) Quando a execução termina, um evento anunciando este fato é enviado ao *function-lifecycle-broker*.

Checagem de status e resultados

O arquivo `/app/io/all/$ID` possui a saída da execução. Se a execução está em andamento ele possui a saída até aquele instante, senão, se a execução ja foi concluída, ele contém toda saída do programa. A qualquer momento durante ou após o término da execução, o status ou resultado dela pode ser requisitado. As seguintes rotas são relacionadas a essa funcionalidade:

a) **`/$runID/status`**: Esta rota pode ser utilizada enquanto a execução ainda está em andamento. Ela devolverá um JSON com as informações de tempo de início da execução, tempo decorrido e os últimos 10KB de saída do programa.

b) **`/$runID/result/info`**: Esta rota pode ser utilizada quando a execução foi terminada. Ela devolverá um JSON com as informações de tempo de início, fim e tempo decorrido.

c) **`/$runID/result/output`**: Esta rota pode ser utilizada quando a execução foi terminada. Ela devolverá o conteúdo do arquivo `/app/io/all/$ID`, ou seja, toda a saída do programa.

5.1.3 Questões de Segurança

A execução de código não-confiável criado por usuários é ainda um problema complexo a ser resolvido para o Hermes. Algumas vulnerabilidades encontradas, e algumas possíveis soluções (algumas já adotadas), são:

a) ***Fork bomb***: Um usuário pode criar uma *hermes-function* que crie um número muito grande de *threads*, o que pode exaurir os recursos do servidor. Uma possível solução para isso é limitar o uso de recursos de cada instância de *function-watcher*.

b) **Acesso à rede**: Uma *hermes-function* poderia acessar a rede e, de alguma maneira, prejudicar outros computadores na rede em que a instância Hermes se encontra. Uma solução seria utilizar as funcionalidades do Docker para limitar as requisições de uma instância de *function-watcher* somente ao *function-lifecycle-broker*. Entretanto isso ainda permitiria um ataque ao *function-lifecycle-broker* – talvez seria necessário a implementação de algum tipo de autenticação no sistema, a fim de que o *function-lifecycle-broker* recebesse somente requisições do servidor do *function-watcher* (não o código do usuário).

c) **Acesso aos processos em execução**: É possível à *hermes-function* acessar os processos em execução e até interrompê-los. Isso inutilizaria a instância de *function-watcher* em questão. Executar o processo da *hermes-function* como um usuário com permissões reduzidas resolveria esse problema. É possível criar usuários com menos permissões quando se cria uma imagem Docker, e o pacote *execa* também permite criar processos filhos com um usuário específico.

d) **Acesso aos arquivos**: Uma *hermes-function* pode acessar, modificar e deletar os arquivos contidos em uma instância de *function-watcher*. Semelhantemente ao item anterior, isso pode inutilizar a instância, e pode ser resolvido utilizando um usuário com permissões mais restritas para executar a *hermes-function*

e) **Tempo grande de execução**: Uma execução pode ter tempo arbitrariamente grande durante o qual ficará consumindo recursos do servidor. Em um ambiente em que os usuários não pagam

pelo tempo utilizado isso poderia exaurir os recursos do servidor e prejudicar outros usuários. Esse problema se mantém em aberto.

f) **Muitas requisições de execução**: Uma instância Hermes pode receber muitas requisições de execução, acarretando a criação de muitas instâncias de *function-watchers*. Isso poderia exaurir os recursos do servidor. Esse problema se mantém em aberto.

5.2 *function-lifecycle-broker*

A implementação do *function-lifecycle-broker* é simples: ele consiste em uma instância de Redis não modificada em que são utilizadas as funcionalidades nativas de publicação e subscrição de eventos. O Redis é um serviço de armazenamento em memória de estruturas de dados de chave-valor [28] que é comumente usado como banco de dados, cache e intermediário de eventos e mensagens (*event broker* ou *message broker*).

5.3 *function-registry-api*

O *function-registry-api* é um servidor HTTP implementado em NodeJS que atua como mediador entre o cliente e o banco de dados (*function-registry-db*). A implementação da interação com o banco de dados, que é o elemento mais importante e diferencial deste serviço, foi feita com o pacote *Sequelize*, uma ferramenta de mapeamento objeto-relacional (ORM - *Object Relational Mapper*) para PostgreSQL. Um ORM permite que as tabelas do banco de dados sejam representadas através de classes e os registros de cada tabela sejam instâncias de cada classe. As operações de criação, atualização, deleção e busca são fornecidas como métodos estáticos ou métodos das instâncias, facilitando todo o processo de interação com o banco de dados, de modo que não é necessário escrever código SQL.

As rotas fornecidas pelo *function-registry-api* seguem o padrão de arquitetura REST e implementam as operações de CRUD de usuários, *hermes-functions* e execuções.

5.4 *function-registry-db*

O *function-registry-db* foi implementado utilizando o PostgreSQL, um banco de dados relacional de código aberto. Na primeira execução do sistema em um servidor, o *script* de instalação e execução da instância Hermes[23] configura as tabelas e restrições no banco de dados de modo a implementar o seguinte modelo entidade relacionamento:

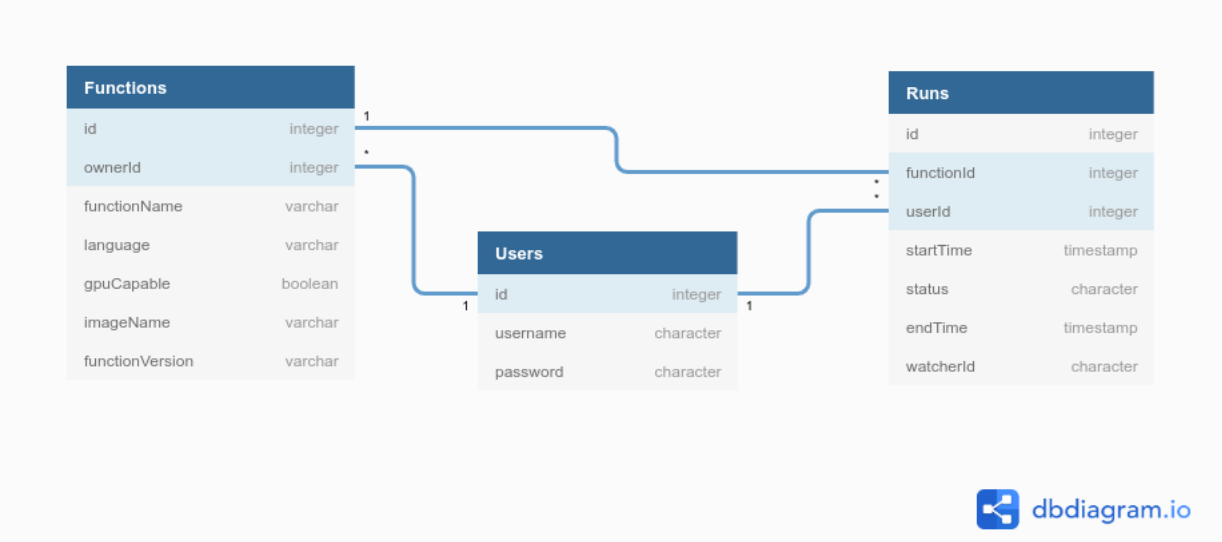

Figura 5.1: Modelo entidade relacionamento do sistema

5.5 *function-orchestrator*

O *function-orchestrator* é um servidor HTTP implementado em NodeJS que integra todos os serviços do Hermes e atua como o nó de comunicação entre clientes externos e a instância. Ele expõe e implementa a API que ficará disponível para os clientes externos. Esta seção trará mais detalhes de implementação sobre as principais funcionalidades expostas para os clientes.

5.5.1 Criação de Execuções

Como já mencionado no capítulo 4, o *function-orchestrator* é responsável por criar ou excluir contêineres de *funciton-watchers*, dependendo da demanda. Esta funcionalidade foi implementada permitindo o acesso da instância de *function-orchestrator* ao *Docker socket* da máquina em que a instância está executando. Os binários do *Docker* foram instalados na imagem do *function-orchestrator* e a execução destes binários pelo servidor NodeJS permite a manipulação dos contêineres de *function-watchers*.

O *function-orchestrator* mantém uma estrutura de dados em memória com todos os *function-watchers* ativos, criados por ele, e mantém controle sobre quais estão executando uma *hermes-function* ou há quanto tempo estão ociosos. Por razões de segurança discutidas em 5.1.3, e por simplicidade, escolheu-se que cada instância de *functon-watcher* executará no máximo uma vez sua *hermes-function*, deste modo uma execução não afetará a outra, e após a execução a instância será desligada. Todo esse controle é feito com auxílio desta estrutura de dados, que recebe informações do *function-lifecycle-broker* para manter a consistência dos dados. A seguir está detalhado o fluxo simplificado de funcionamento de uma execução, do momento que a requisição chega ao *function-orchestrator* ao seu fim, com sucesso ou erro:

a) Uma requisição de execução é feita na rota `/$username/run/$functionOwner/$functionName/$functionVersion`. Da URL o *function-orchestrator* retira as informações de qual usuário está requisitando a execução (*username*), qual usuário é o dono daquela *hermes-function* (*functionOwner*) e quais são

o nome (*functionName*) e versão (*functionVersion*) da função. O cabeçalho da requisição também deve conter o campo *x-hermes-run-type*, com valor *sync* ou *async*, especificando o tipo de execução que será realizado.

b) Com as informações recebidas, o *function-orchestrator* solicita ao *function-registry-api* os metadados daquela *hermes-function*: ID no banco de dados, nome da imagem Docker e se deve utilizar GPU.

c) O ID da função é utilizado para checar se existem *function-watchers* responsáveis por aquela *hermes-function* ativos. Esses dados são mantidos em memória em uma estrutura de dados. Se já existe algum *function-watcher* ativo o fluxo pula para o passo *f*, senão segue para o passo seguinte.

d) O binário do Docker é executado com argumentos indicando o nome da imagem docker e a informação se deveria utilizar GPU. Um exemplo de execução deste binário feito pelo *function-orchestrator* é o seguinte: *docker run --gpus all --name=watcher-1_ee0fb23676633489 --network=hermes -e REDIS_CHANNEL=watcher-1_ee0fb23676633489 tiagonapoli/watcher-char-printer:1.0.0*. Nessa execução são especificados para o Docker:

   i. A imagem a ser utilizada para criar o contêiner. No caso é *tiagonapoli/watcher-char-printer:1.0.0*. O Docker baixará esta imagem do DockerHub se já não estiver disponível localmente.

   ii. O nome que será atribuído ao contêiner, especificado pelo argumento *--name=watcher-1_ee0fb23676633489*. Os nomes foram padronizados no seguinte modelo: `watcher-$functionID_$randomHexNumber`.

   iii. A variável de ambiente *REDIS_CHANNEL*, que estará disponível durante a execução do *function-watcher*, e acessível a ele. Ela especifica o nome do canal do Redis em que serão enviados eventos referentes àquela instância.

   iv. Se o contêiner utilizará GPU. A opção *--gpus all* permite ao contêiner a utilização das GPUs da máquina. Este argumento é utilizado somente se os metadados da *hermes-function* especificam que ela utilizará GPU.

   Um momento antes da execução do binário Docker para criação do contêiner o *function-orchestrator* cria uma subscrição no *function-lifecycle-broker* para ouvir eventos no *REDIS_CHANNEL* especificado.

e) Assim que o contêiner do *function-watcher* inicia seu servidor com sucesso ou erro, um evento é enviado ao *function-lifecycle-broker* que por sua vez informa ao *function-orchestrator* que a instância está pronta ou se houve erro. Em caso de erro a resposta da requisição é feita informando o erro e o fluxo acaba. Em caso de sucesso a nova instância é registrada na estrutura de dados que mantém os *function-watchers* ativos e o fluxo continua.

f) O *function-orchestrator* requisita ao *function-registry-api* a criação de uma novo objeto *Run* (execução) no banco de dados, com valores nulos em seus campos. O objeto retornado pelo *function-registry-api* possui somente o ID do objeto *Run* no banco de dados, que será utilizado em breve.

g) Um *proxy* da requisição de execução é feito para o *function-watcher*, alterando levemente a requisição para o formato aceito pela API e acrescentando à requisição o ID da execução no banco de dados (ID do objeto *Run* do passo f).

h) Quando a execução termina, com sucesso ou erro, o *function-lifecycle-broker* envia um evento ao *function-orchestrator*, que então solicita os dados daquela execução ao *function-watcher* responsável, arquiva-os em disco, envia uma requisição de desligamento para aquele *function-watcher* e envia ao *function-registry-api* uma atualização dos dados referentes àquela execução.

5.5.2 Operações de CRUD

Algumas das rotas implementadas pelo *function-orchestrator* fornecem uma API REST para CRUD de *hermes-functions*, usuários e execuções. Quando uma requisição deste tipo é realizado, o *function-orchestrator* solicita os dados ao *function-registry-api* e os envia como resposta.

5.5.3 Checagem de status e resultados de execuções

Uma requisição de checagem de status e resultados de execuções pode ser feita a uma das seguintes rotas, que devolve as informações especificadas:

a) **`/:username/run/:runId/status`:** Se a execução ainda está em andamento devolve um JSON com metadados de data de início, tempo decorrido e os últimos 10Kb de saída do programa. Se a execução estiver terminada esta rota devolve o mesmo que `/:username/run/:runId/result-info`.

b) **`/:username/run/:runId/result-info`:** Se a execução estiver em andamento devolve um JSON erro, senão devolve os metadados de tempo de início, fim e tempo decorrido de execução.

c) **`/:username/run/:runId/result-output`:** Se a execução estiver em andamento devolve um erro, senão devolve a saída da execução.

Quando uma requisição a elas é feita, o *function-orchestrator* extrai o ID da execução da URL e checa se aquela execução ainda está em andamento em algum *function-watcher* ativo. Se estiver, e a rota solicitada é `/:username/run/:runId/status`, a requisição é repassada ao *function-watcher* responsável e sua resposta é devolvida ao usuário. Se a execução estiver terminada, os metadados da execução podem ser solicitados ao *function-registry-api*, e a saída da execução estará armazenada no disco da instância do *function-orchestrator*, assim, dependendo da rota de resultado solicitada, um desses dados serão devolvidos.

Capítulo 6

Conclusão e Trabalhos Futuros

O protótipo criado conseguiu resolver decentemente o problema de entrega *serverless*, apresentando suporte para execução de código acelerado por GPU. O usuário consegue ter uma experiência *serverless* de execução de suas aplicações codificadas em CUDA, para uso com GPU, ou C++, sem GPU. O sistema, no estado atual, pode ser utilizado em pequena escala, suportando apenas algumas execuções simultâneas – isso foi evidenciado criando *bots* que realizam execuções: infelizmente algumas poucas execuções simultâneas já sobrecarregam o sistema e, nesse sentido, ainda há grande espaço para melhora.

Existem diversos desafios ainda a serem resolvidos, melhorias e possíveis trabalhos futuros para o Hermes. Alguns deles são:

- **Melhoria da orquestração dos *function-watchers*:** Atualmente a orquestração é realizada pelo *function-orchestrator*, entretanto hoje existem softwares de código aberto que realizam uma orquestração muito mais sofisticada, oferecendo inclusive um controle do tempo de utilização de CPU por cada contêiner, possibilitando um melhor compartilhamento de recursos. Um desses softwares que poderiam ser utilizados seria o *Kubernetes*. O melhor compartilhamento dos recursos de GPU, entretanto, se mantém um problema em aberto para a comunidade de computação [29], então ainda seria um problema para o Hermes.

- **Mudanças no modo como as imagens de *function-watchers* são criadas:** Um dos problemas na utilização do Hermes é, algumas vezes, o tempo para *download* da imagem do *function-watcher* a ser iniciado no servidor. Uma mudança arquitetural permitiria que existisse somente uma imagem de *function-watcher* e que ela baixasse de um serviço armazenador o código a ser executado.

- **Coleta e centralização de *logs*:** Existem softwares atualmente cuja função é coletar e armazenar *logs* enviados a ele, permitindo consulta e análise posterior. Um desses serviços é, por exemplo, o *Fluentd*. Os *logs* dos serviços do Hermes poderiam ser enviados a ele, o que facilitaria muito testes e rastreio de *bugs*.

- **Melhorias de usabilidade da CLI:** A experiência de usuário ainda tem muito espaço para evolução – as mensagens de erro podem ser melhores e ainda é necessário criar mais documentação.

- **Melhores mensagens de erro dos serviços:** Algumas dessas mensagens são repassadas

ao usuário, então melhorá-las tornaria a experiência de uso melhor. Também facilitaria, juntamente com a coleta e centralização de *logs*, a identificação de problemas no sistema.

Sendo um projeto de código aberto, as contribuições são muito bem vindas. Esta monografia é uma boa documentação para entendimento da arquitetura atual do sistema, o que facilitará trabalhos futuros e contribuições, que certamente são uma grande oportunidade de aprendizado como todo este projeto foi.

Bibliografia

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," **Journal of Parallel and Distributed Computing**, vol. 68, no. 10, pp. 1370 – 1380, 2008, general-Purpose Processing using Graphics Processing Units. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731508000932 2

[2] NVIDIA. (2019) Gpu-accelerated applications. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf 2, 8

[3] Spell. Deep learning and ai development platform. [Online]. Available: spell.run 2

[4] PyTorch. A python-based scientific computing package. [Online]. Available: https://pytorch.org/ 2

[5] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," **Communications of the ACM**, vol. 17, no. 7, pp. 412–421, 1974. 4

[6] D. A. Menascé, "Virtualization: Concepts, applications, and performance modeling," in **31th International Computer Measurement Group Conference, December 4-9, 2005, Orlando, Florida, USA, Proceedings**, 2005, pp. 407–414. 4, 5

[7] L. P. Lotti and A. B. Prado, "Sistemas virtualizados – uma visão geral," 2010. 4

[8] M. Portnoy, **Virtualization Essentials**, 1st ed. Wiley, 2012. 5

[9] M. N. Bennani and D. A. Menasce, "Resource allocation for autonomic data centers using analytic performance models," in **Second International Conference on Autonomic Computing (ICAC'05)**, June 2005, pp. 229–240. 5

[10] E. Nemeth, G. Snyder, T. R. Hein, B. Whaley, and D. Mackin, **UNIX and Linux System Administration Handbook (5th Edition)**, 5th ed. Addison-Wesley Professional, 2017. 5, 6

[11] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in **2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**, March 2015, pp. 171–172. 5

[12] K. Agarwal, B. Jain, and D. E. Porter, "Containing the hype," in **Proceedings of the 6th Asia-Pacific Workshop on Systems**, ser. APSys '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:9. [Online]. Available: http://doi.acm.org/10.1145/2797022.2797029 6

[13] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: Towards a cloud definition," **SIGCOMM Comput. Commun. Rev.**, vol. 39, no. 1, pp. 50–55, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1496091.1496100 6

[14] V. Rajaraman, "Cloud computing," **Resonance**, vol. 19, no. 3, pp. 242–258, Mar 2014. [Online]. Available: https://doi.org/10.1007/s12045-014-0030-1 6

[15] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in **SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing**, Nov 2008, pp. 1–12. 6

[16] C. N. Höfer and G. Karagiannis, "Cloud computing services: taxonomy and comparison," **Journal of Internet Services and Applications**, vol. 2, no. 2, pp. 81–94, Sep 2011. [Online]. Available: https://doi.org/10.1007/s13174-011-0027-x 7

[17] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," **CoRR**, vol. abs/1706.03178, 2017. [Online]. Available: http://arxiv.org/abs/1706.03178 7

[18] A. Ellis. (2019) Serverless functions, made simple. [Online]. Available: https://www.openfaas.com/ 9

[19] T. Nápoli. (2019) Hermes cli - npm package. [Online]. Available: https://www.npmjs.com/package/@hermes-serverless/cli 10

[20] Docker. (2019) Docker installation. [Online]. Available: https://docs.docker.com/v17.09/engine/installation/ 10

[21] ——. (2019) Library and community for container images. [Online]. Available: https://hub.docker.com/ 10

[22] Yarn. (2019) Yarn - fast, reliable, and secure nodejs dependency management. [Online]. Available: https://yarnpkg.com/en/ 10

[23] T. Nápoli. (2019) Hermes instance installation scripts. [Online]. Available: https://github.com/hermes-serverless/hermes-install 14, 26

[24] ——. (2019) Hermes serverless - github project. [Online]. Available: https://github.com/hermes-serverless 21

[25] ——. (2019) function-watcher repository. [Online]. Available: https://github.com/hermes-serverless/function-watcher 21, 23

[26] ——. (2019) project building images repository. [Online]. Available: https://github.com/hermes-serverless/project-building-base-images 22

[27] sindresorhus. (2019) Execa – process execution for humans. [Online]. Available: https://github.com/sindresorhus/execa 23

[28] AWS. (2019) O que é o redis? [Online]. Available: https://aws.amazon.com/pt/elasticache/what-is-redis/ 26

[29] P. McQuighan. (2019) Lightning talk: Sharing a gpu among multiple containers. [Online]. Available: https://www.youtube.com/watch?v=VNZsooht3K8 30