Kata e runC runtime usando Docker: uma comparação de desempenho na perspectiva de rede

Henrique Z. Cochak[1], **Charles C. Miers**[1]

[1]Departamento de Ciência da Computação (DCC)
Universidade do Estado de Santa Catarina (UDESC)

henrique.zc@edu.udesc.br,

charles.miers@udesc.br

Resumo. *O módulo runtime na plataforma Docker possui o propósito de gerenciar o ciclo de vida da existência de um contêiner. Como um módulo, é possível retirá-lo e conectar outro runtime na sua posição para buscar outras características de interesse. Assim, o runtime kata foi elaborado focando em questões de segurança para o contêiner. Este trabalho apresenta uma comparação entre runtimes runC e kata, comparando o desempenho de rede (largura de banda e latência). Os resultados iniciais evidenciam um desempenho melhor do runC.*

1. Introdução

A tecnologia de conteinerização, sendo uma opção da virtualização clássica, funciona criando uma unidade executável de software chamado de contêiner no qual o código do aplicativo, bibliotecas, dependências e arquivos de configuração são empacotados para ser executado em qualquer ambiente [Docker 2020].

Dentro de ferramentas de gerenciamento de contêineres, existe o Docker, que é uma *Platform-as-a-Service* (PaaS)/ *Infrastruture-as-a-Service* (IaaS) que entrega estes pacotes e dentro deste software modular, existe uma funcionalidade chamada *runtime* que por padrão no Docker é o runC. Uma nova abordagem é o kata *runtime* que é desenvolvido para fornecer recursos aprimorados de segurança relacionados à contêineres.

O objetivo deste artigo é expor uma comparação inicial do desempenho da comunicação de rede usando ferramentas de medição para capturar e analisar os dados coletados. O artigo está organizado da seguinte forma: uma descrição de como ocorre a comunicação de rede para ambos *runtimes*, runC e kata, na Seção 2. Uma explicação sobre as interfaces de rede do Linux e os *drivers* utilizados em ambos *runtimes* na Seção 3. Detalhes sobre a configuração dos experimentos e resultados são apresentados na Seção 4.

2. Comunicação no *runtime* runC e *runtime* kata

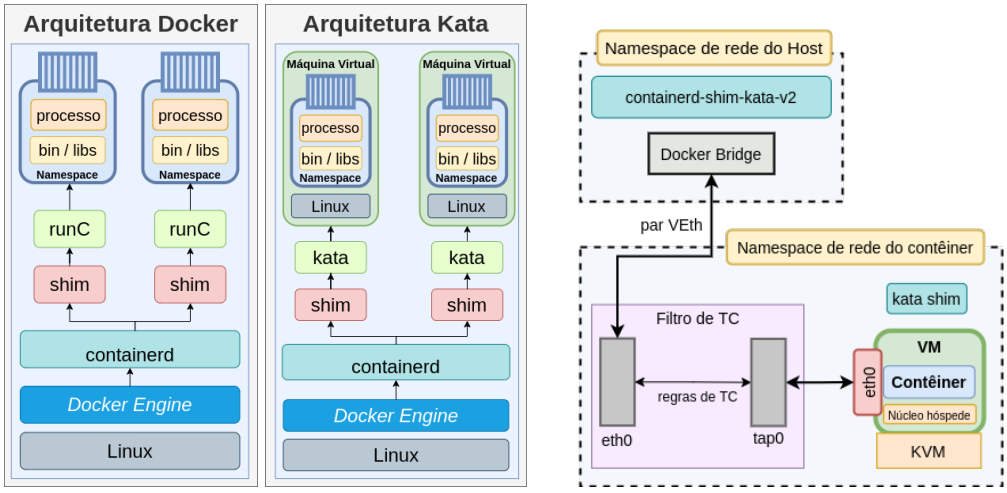
A tecnologia de contêiner Docker foi lançada pela primeira vez em 2013 como uma PaaS de código aberto, aproveitando conceitos do Linux como unionFS (atualmente *overlay*), *cgroups* e *namespaces*. No ano de 2015, um grupo de empresas criou a Open Container Initiative (OCI) com a finalidade de padronizar a infraestrutura de contêineres e nestes documentos, são listadas as etapas para fornecer o tráfego e a comunicação através da rede para estes contêineres, utilizando de três componentes [Poulton 2020], uma *sandbox* que exerce a função de uma pilha de rede isolada, um *endpoint* que funciona como uma

interface de rede virtual que contém a responsabilidade de fazer a conexão de uma *sandbox* com uma rede e por último o *network driver* que é uma implementação em software da IEEE bridge802.1d e, como tal, estes agrupam e isolam *endpoints*.

A verdadeira implementação destes documentos pelo Docker é a biblioteca chamada de *libnetwork* [Poulton 2020] e funciona fazendo a comunicação direta com o núcleo Linux para requisições de funções do sistema e assim o Docker pode organizar a comunicação de rede entre vários contêineres de forma automatizada.

Uma funcionalidade também indispensável é o *shim* que é uma Application Programming Interface (API) entre o *dockerd* e *runtime* de baixo nível para facilitar a comunicação bidirecional do contêiner. Por fim, existe o *runtime* com o único propósito de conter o ciclo de vida de um contêiner.

Usando a tecnologia tradicional de contêiner para criar e isolar contêineres e adicionando um núcleo altamente otimizado como um núcleo convidado [Kata 2020], com uma interface de virtualização de hardware completa, é possível ter uma camada extra de encapsulamento, aumentando o nível de segurança do sistema *host*. Esta escolha (Figura 1(a)) do nível dos *runtimes* faz com que a arquitetura em alto nível seja diferente.



(a) *runtime* runC e kata (b) Comunicação do Kata Containers

Figura 1. Detalhamento de cada arquitetura de *runtime*.

Com a falta de compartilhamento do núcleo pelos contêineres ao utilizar do *runtime* kata, é preciso dispor de novos meios para fazer a comunicação fluir entre um contêiner e a rede. O Kata Containers faz esta implementação utilizando conceitos de espelhamento providenciado por interfaces TAP.

Uma vez que o contêiner é mantido dentro de uma máquina virtual (MV), as interfaces TAP são necessárias para a conectividade de rede da MV em que um único par Virtual Ethernet (vEth) não pode ser manipulado adequadamente pelo hipervisor. [Kata 2020]. Para superar esta incompatibilidade entre contêiner e a MV, Kata Containers conecta pares vEth com interfaces TAP usando controle de tráfego (Figura 1(b)), assim redirecionando os pacotes com base nas regras de controle.

3. Drivers de rede

O Docker utiliza de rede do Linux requisitando acesso de funcionalidades do núcleo com a assistência da *libnetwork*. Neste artigo são descritos quatro *network drivers* [Docker 2020], *bridge*, *host*, *macvlan* e *overlay*, cada um com sua especificidade.

4. Experimento e Resultados

Para o ambiente de testes (Figura 2), 2 MVs pré-alocadas são hospedadas dentro de um servidor da nuvem computacional Tche, hospedada no LabP2D/ UDESC. Cada MV contém 4GBs de RAM, 2 processadores virtuais (*host* com modelo Intel Xeon E3-12xx com 2000MHz) e 20GBs de armazenamento em disco, sendo que cada MV utiliza o GNU/Linux Ubuntu 18.04.5 LTS e a ferramenta Docker na versão 20.10.2.

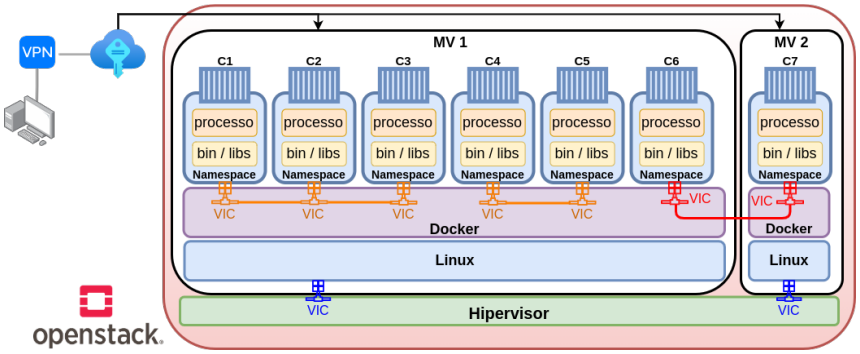


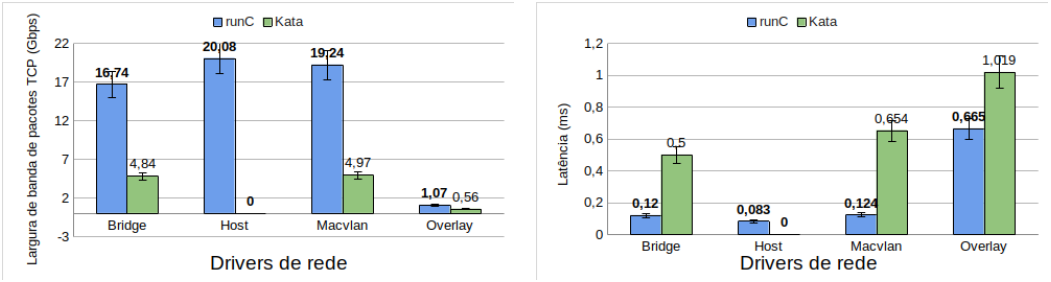
Figura 2. Ambiente de testes utilizado.

Os testes seguem uma organização baseada em pares de cliente-servidor no qual um contêiner opera como cliente e outro opera como um servidor. Os pares cliente-servidor são: (*bridge*, *bridge*), (*host*, *bridge*), (*macvlan*, *macvlan*), (*overlay*, *overlay*). Utilizando outros contêineres de pares cliente-servidor, utiliza-se da ferramenta stress-ng no contêiner servidor para simular um servidor com CPUs sobrecarregadas. São executados 15 testes para cada par cliente-servidor.

Nota-se que o Kata Containers possui uma limitação em seu próprio *runtime* com a incapacidade de usar o *driver* de rede *host*. Logo é definido o valor 0 tanto para latência quanto para a largura de banda para este *driver*.

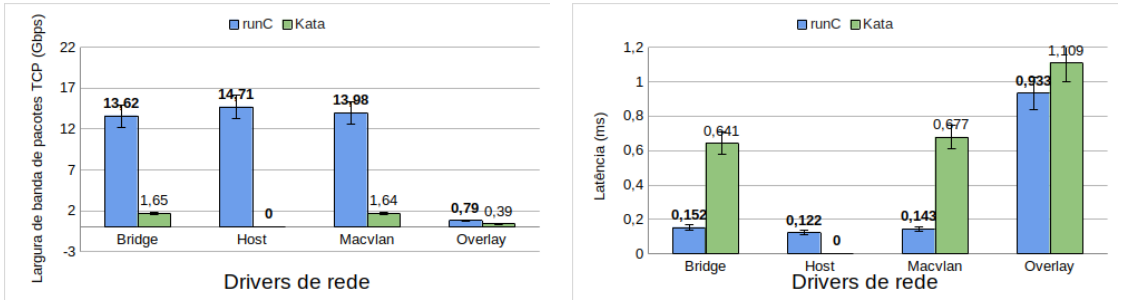
Observando a Figura 3(b), há uma lacuna entre os *runtimes* em que o *runtime* runC possui a latência mais baixa para todos os *drivers* testados. Este é o resultado de como o Kata Containers implementa a rede enquanto fornece um nível extra de encapsulamento.

O runC usa o núcleo do *host* para manipular e separar os processos dentro de seus próprios *namespaces*. Deste modo quase não há etapas extras para proceder a informação entre processos e soquetes para os processos e redes isolados dos contêineres, todos dentro do mesmo sistema e nível de abstração. O *runtime* kata precisa implementar etapas extras para criar a comunicação dentro do contêiner para manter toda a abstração extra da MV conforme (Figura 1(b)), resultando em uma largura de banda menor (Figura 3(a)). Se as CPU são estressadas pela ferramenta stress-ng (Figura 4), o runC possui melhor desempenho que o kata *runtime* com aumento de latência em todos os *drivers* de rede. A razão na queda da largura de banda está relacionado com a fila de processo da Central Process Unit (CPU), na qual grande parte do tempo os pacotes TCP ficam retidos pela CPU estar sobrecarregada.



(a) Largura de banda (b) Latência

Figura 3. Testes feitos em contêineres sem sobrecarregar os processadores.



(a) Largura de banda (b) Latência

Figura 4. Testes feitos em contêineres utilizando da ferramenta stress-ng.

A medida que dados trafegam pelas camadas, ocorre o encapsulamento/desencapsulamento das informações e caso a informação transmitida for muito grande para o tamanho da rede, esta deve ser fragmentada em pacotes menores (somente para IPv4). Ambos conceitos precisam ser computados dentro de uma CPU, mas se a fila está cheia, isto atrasará a execução desses processos, afetando a quantidade total de dados transmitidos. Com o aumenta da latência (Figura 4(b)), a largura de banda sofre perdas, induzindo uma largura de banda menor ao comparada a um sistema não estressado (Figura 3(a)).

5. Considerações

O Kata Containers busca desempenho enquanto impulsiona a segurança do contêiner, empregando uma nova MV virtualizada com QEMU/KVM nos seus contêineres. Entretanto, em todos os experimentos, o *runtime* do Docker, runC, registrou uma latência menor para cada *driver* de rede do que o kata *runtime*. Deste modo o Kata Containers possui um longo caminho para realmente atingir a velocidade dos contêineres do Docker enquanto mantém a segurança obtida através da camada de encapsulamento extra.

Agradecimentos: Os autores agradecem o apoio do LabP2D/UDESC e a FAPESC.

Referências

Docker (2020). What is container. Disponível em: https://www.docker.com/resources/what-container. Acesso em: 05 Jun. 2021.

Kata (2020). Kata architecture. Disponível em: https://github.com/kata-containers/kata-containers/blob/main/docs/design/architecture.md. Acesso em: 16 Apr. 2021.

Poulton, N. (2020). *Docker Deep Dive: Zero to Docker in a single book*. Packt Publishing.