Kata e runC runtime usando Docker: uma comparação de 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 de plataforma Docker possui o único propósito e este é de conter todo o ciclo de vida da existência de um contêiner e como um módulo, é possível retirá-lo e conectar outro runtime na sua posição. Com esta possibilidade, manifestou-se outro runtime chamado kata, focando em questões de segurança para o contêiner. Este trabalho apresenta uma comparação aprofundada entre runtime runC e kata, comparando e analisando desempenho de largura de banda e latência, resultando em um desempenho melhor do runC.*

1. Introdução

A tecnologia de virtualização permite que as empresas ou usuários criem vários ambientes virtualizados a partir de um único hardware físico. Isso é alcançado pelo hipervisor, que coordena e compartilha os recursos físicos do computador, servindo como uma interface entre a máquina virtual (MV) e o hardware físico adjacente, certificando que cada MV tem acesso aos recursos de que precisa para executar adequadamente [IBM 2019]. A conteinerização surge como uma opção da virtualização clássica de máquinas. Os contêineres são uma unidade executável de software, na qual o código do aplicativo, bibliotecas, dependências e arquivos de configuração são empacotados juntos para executá-lo em qualquer lugar [Docker 2020b].

Dentro de ferramentas de gerenciamento de contêineres, existe o Docker, que é uma Plataforma como serviço (PaaS)/Infraestrutura como serviço (IaaS) para desenvolver, enviar e executar aplicativos em contêineres [Docker 2020a] e dentro deste software modular, existe uma funcionalidade chamada *runtime* que por padrão no Docker é o runC. Uma nova abordagem existente é o kata *runtime* que enfatiza as questões de segurança relacionadas aos contêineres.

O objetivo deste artigo é analisar uma comparação inicial do desempenho da comunicação em 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 conjunto de empresas criou a Open Container Initiative (OCI) com o propósito de criar padrões para a infraestrutura de contêineres. Neste documento da OCI, são listadas as etapas para fornecer tráfego e comunicação através da rede para os contêineres utilizando três componentes [Poulton 2020]:

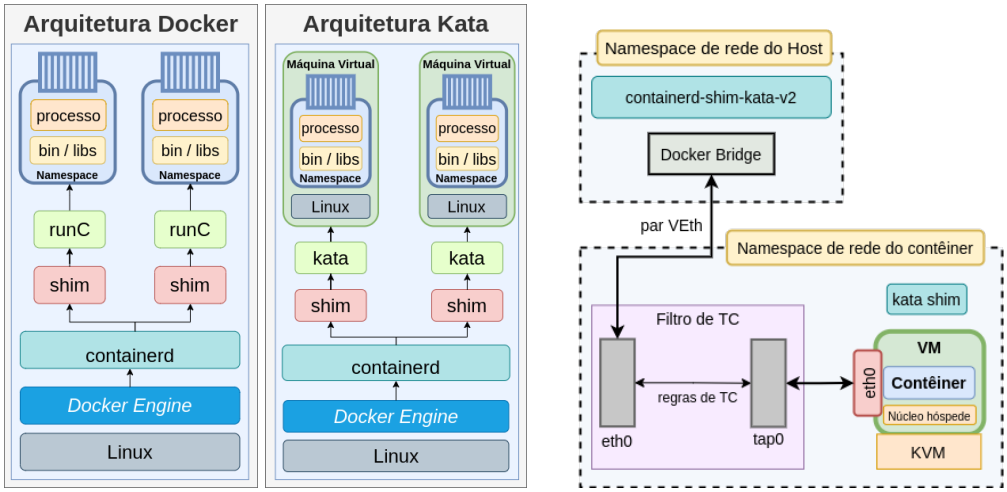
- *sandbox*: a *sandbox* é uma pilha de rede isolada e contém suas configurações, incluindo o gerenciamento de tabelas de roteamento, configurações de DNS, portas e interfaces de contêineres e uma *sandbox*.
- *endpoint*: funciona como uma interface de rede virtual e estes têm a responsabilidade de fazer a conexão de uma *sandbox* para uma rede.
- *driver network*: uma implementação em software da IEEE bridge802.1d e, como tal, estes agrupam e isolam *endpoints*.

Seguindo o modelo criado pelo Docker, e outras empresas do ramo pela Container Network Model (CNM), a biblioteca *libnetwork* é a verdadeira implementação na plataforma Docker [Poulton 2020]. Despachando os componentes citados e chamando *drivers* de rede integrados do *Docker Engine*, que é facilitado pela interface plugável, o Docker pode organizar a comunicação de rede entre vários contêineres automaticamente. Também especificado pela OCI, o *runtime* possui o único propósito de conter o ciclo de vida de um contêiner, incluindo arquivos de configuração e o ambiente de execução, especificado para garantir que os aplicativos em contêineres não falhem entre ações comuns definidos em seu ciclo de vida [OCI 2020].

Uma funcionalidade também indispensável é o *shim* que fica entre o *dockerd* e *runtime* de baixo nível para facilitar a comunicação bidirecional do contêiner e também permitir que os contêineres não sejam vinculados unicamente a um processo *daemon*. Mesmo o kata *runtime* funcionando de modo similar devido a ser uma funcionalidade plugada na configuração da plataforma Docker, a comunicação entre contêineres ocorre de forma mais complexa pelo fato de que o Kata Containers promove o aumento da segurança de um contêiner colocando-os dentro de uma MV. O núcleo padrão fornecido é altamente otimizado para o tempo de inicialização do núcleo e mínimo espaço de memória, fornecendo apenas os serviços exigidos por uma carga de trabalho do 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 de ambos *runtimes* faz com que a arquitetura em alto nível seja diferente.

Com a falta de compartilhamento do núcleo pelos contêineres ao utilizar do *runtime* kata, é preciso utilizar de novos meios para fazer a comunicação fluir para dentro e fora do contêiner. O Kata Containers implementa isto utilizando conceitos de espelhamento providenciado por interfaces TAP. Uma vez que o contêiner é mantido dentro de uma MV, as interfaces TAP são necessárias para a conectividade de rede da MV em que o único par Virtual Ethernet (vEth) não pode ser manipulado adequadamente pelo hipervisor [Kata 2020]. Para superar essa incompatibilidade entre contêiner e MV, Kata Containers conectam interfaces vEth com interfaces TAP usando controle de tráfego (Figura 1(b)), redirecionando os pacotes com base nas regras de controle de tráfego.



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

Figura 1. Diferença e detalhamento de cada arquitetura.

3. Drivers de rede

O Docker utiliza de interfaces de rede do Linux requisitando do núcleo com a assistência da *libnetwork* e neste artigo são descrito quatro *drivers* [Docker 2020b]:

- *bridge*: uma ponte virtual que pode ser vista como um *switch*, utilizando funções como Linux IPtables e Network Address Translation (NAT).
- *host*: existe um compartilhamento de pilha de rede do *host* para os contêineres vinculados a esta interface, removendo segurança de encapsulamento.
- *macvlan*: permite a configuração de múltiplos endereços MAC e IP para cada interface virtual do contêiner, sendo visto como um dispositivo físico na rede.
- *overlay*: permite a distribuição de rede entre múltiplos Docker *daemons*, criando o que é chamado de *Swarm*, permitindo a comunicação de diferentes máquinas dentro da camada de enlace. Esta interface funciona criando uma *bridge* disponível para os contêineres dentro do *Swarm*, uma rede de interface *ingress* com um tunelamento Virtual Extensible LAN (VXLAN). Quando pacotes transitam na rede *ingress*, está rede altera o cabeçalho adicionando um identificador chamado identificador de rede VXLAN (VNID) e encapsula os dados para ser enviado sobre a infraestrutura subjacente.

4. Experimento e Resultados

Para executar os testes, ferramentas de software são utilizadas na assistência na coleta de dados ou simular um sistema com sobrecarga de tarefas para que seja possível identificar o efeito na comunicação da rede. As ferramentas são iPerf [Iperf 2019], sockperf [Mellanox 2021] e stress-ng [Ubuntu 2019]. O iPerf é a ferramenta usada para medir a largura de banda máxima e a taxa de transferência entre os contêineres no qual o sockperf afere a latência e a sensibilidade da rede à latência. Por último, tem-se o stress-ng, capaz de sobrecarregar a CPU para medir o tráfego de rede sob interferência de recursos físicos.

Para o ambiente de testes (Figura 2), 2 MVs pré-alocadas são hospedadas dentro de um servidor integrante da nuvem computacional Tche, hospedada no LabP2D/ UDESC. Cada MV contém 4GB de RAM, 2 processadores virtuais (*host* usa modelo Intel Xeon E3-12xx com 2000MHz) e 20GB de armazenamento em disco rígido, sendo que

cada MV utiliza o GNU/Linux Ubuntu 18.04.5 LTS como SO e está instalado o Docker versão 20.10.2.

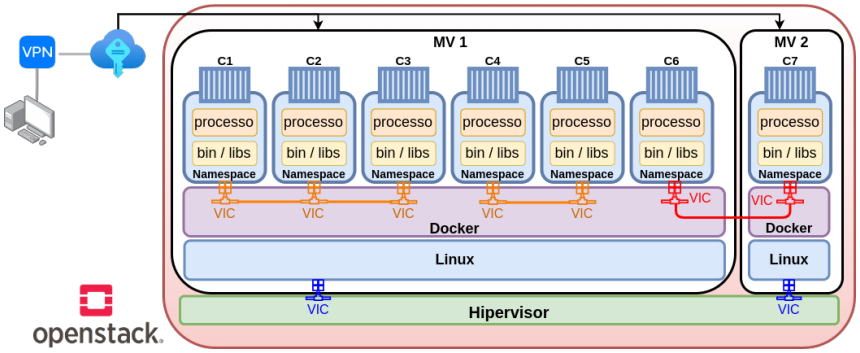
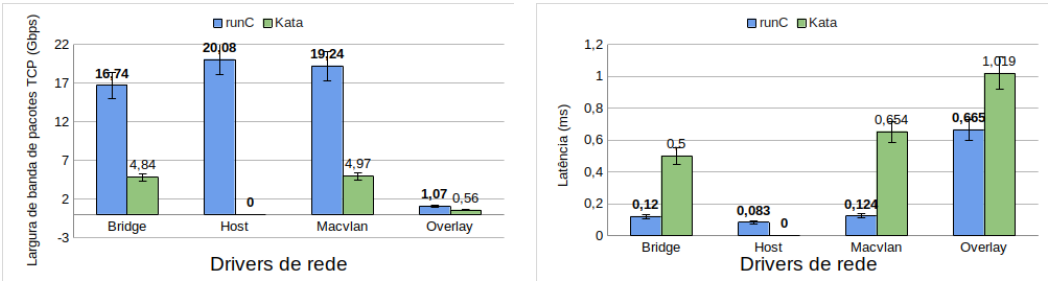


Figura 2. Ambiente de testes utilizado.

Os testes seguem a idealização de pares cliente-servidor no qual um contêiner opera como cliente e outro funciona como servidor. Os pares cliente-servidor são: (*bridge*, *bridge*), (*host*, *bridge*), (*macvlan*, *macvlan*), (*overlay*, *overlay*). Utilizando outros contêineres de pares cliente-servidor, é utilizado da ferramenta stress-ng no contêiner servidor para simular um servidor com as CPUs sobrecarregadas. No total, foram executados 15 testes para cada par cliente-servidor.

Analisando, nota-se que os contêineres de kata têm uma limitação em seu próprio kata *runtime* que é a incapacidade de usar a rede hospedeira. Quando é empregado o *driver* de rede *host*, o contêiner é vinculado à pilha da rede do *host*, permitindo que o contêiner obtenha o melhor desempenho de taxa de transferência, uma vez que não precisa de nenhum tipo de encaminhamento. Contudo, não é possível fazer a conexão com a rede *host* de dentro da MV sem modificar a configuração de rede do contêiner e possivelmente desconfigurando toda a conexão [Kata 2021] e por causa disso, o *driver* de rede *host* é definido como 0 para latência e largura de banda. Um trabalho semelhante, de [Kumar and Thangaraju 2020], também não fornece uma visão completa e um melhor entendimento de como a rede ocorre entre os *runtimes* e como estes diferem.

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



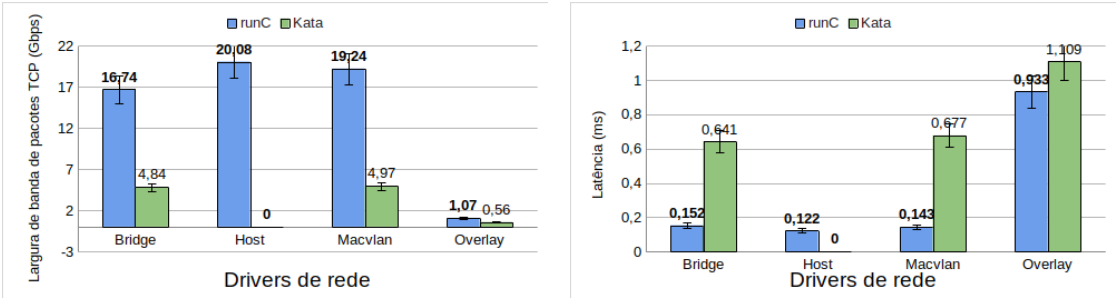
(a) Largura de banda (b) Latência

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

O runC usa o núcleo do *host* para manipular e separar os processos dentro de seus próprios *namespaces* e deste modo quase não há etapas extras para proceder a informação entre processos e *sockets* para os processos e redes isolados dos contêineres, todos dentro do mesmo sistema e nível de abstração.

O kata *runtime* precisa implementar etapas extras para criar a comunicação dentro do contêiner, para manter toda a abstração extra da MV conforme (Figura 1(b)), mostrando que as etapas extras necessárias para a comunicação impactam no desempenho para manter a segurança, resultando em uma largura de banda menor para o kata (Figura 3(a)). O modo *Swarm* que utiliza o *driver overlay* tem a menor largura de banda dos pacotes TCP e está relacionado à latência e como o *driver overlay* trabalha para fazer os ajustes necessários para a comunicação dos contêineres.

Se a CPU é estressada pela ferramenta stress-ng (Figura 4), o runC sofre menos 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). Como o contêiner servidor está lidando com vários pedidos intensivos da CPU feitos pela ferramenta stress-ng, a fila está em alta demanda. Como cada *overhead* do pacote TCP precisa ser desencapsulado e as informações devidamente tratadas, os pacotes entram na fila e passam a maior parte do seu tempo de ciclo ocioso já que a CPU está sobrecarregada.



(a) Largura de banda

(b) Latência

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

A medida que os dados trafegam pelas camadas ocorre o encapsulamento /desencapsulamento as informações e, além disso, se a informação transmitida for muito grande para o tamanho da rede, esta deve ser fragmentada em pacotes menores (somente no caso do IPv4, visto que o IPv6 trata isso de modo diferente). Ambos conceitos precisam ser computados dentro de uma CPU, mas se a fila está cheia devido a processos solicitando todo os seus recursos, isto atrasará a execução desses processos. Com o tempo, o período parado afeta a quantidade de dados transmitidos pela rede, influenciando a largura de banda final transmitida pelo teste da ferramenta iPerf.

Com o aumenta da latência (Figura 4(b)), a largura de banda sofre perdas, induzindo uma largura de banda reduzida quando comparada a um sistema não estressado (Figura 3(a)) já que a largura de banda é mensurada em quantidade de dados que flui em bits por segundo.

5. Considerações

O Kata Containers buscam desempenho enquanto impulsionam a segurança do contêiner, empregando uma nova MV virtualizada com QEMU/KVM nos seus contêineres. Em todos os experimentos, o *runtime* do Docker, runC, registrou uma latência menor para cada *driver* de rede do que o kata *runtime*. Além disso, o Docker alcançou uma largura de banda maior para todos os casos e cenários também.

Como identificado, 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, mas a segurança é um requisito necessário para diminuir as chances de acesso não autorizado e resolver algumas falhas que a tecnologia do contêiner possui, mesmo a um custo de desempenho na comunicação da rede.

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

Referências

Docker (2020a). Docker overview. Disponível em: https://docs.docker.com/get-started/overview/. Acesso em: 17 Jun. 2021.

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

IBM (2019). What is virtualization? Disponível em: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide. Acesso em: 20 Mai. 2021.

Iperf (2019). Iperf. Disponível em: https://iperf.fr/. Acesso em: 10 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.

Kata (2021). Kata limitations. Disponível em: https://github.com/kata-containers/kata-containers/blob/main/docs/Limitations.md. Acesso em: 04 Jun. 2021.

Kumar, R. and Thangaraju, B. (2020). Performance analysis between runc and kata container runtime. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–4. IEEE. Acesso em: 17 Jun. 2021.

Mellanox (2021). Sockperf. Disponível em: https://github.com/Mellanox/sockperf. Acesso em: 10 Jun. 2021.

OCI (2020). Runtime and lifecycle. Disponível em: https://github.com/opencontainers/runtime-spec/blob/master/runtime.md/. Acesso em: 23 Mai. 2021.

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

Ubuntu (2019). Stress-ng manual. Disponível em: https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html. Acesso em: 06 Jun. 2021.