ChatGPT

Manual de Configuração do Docker Swarm em 3 VPS (Melhores Práticas)

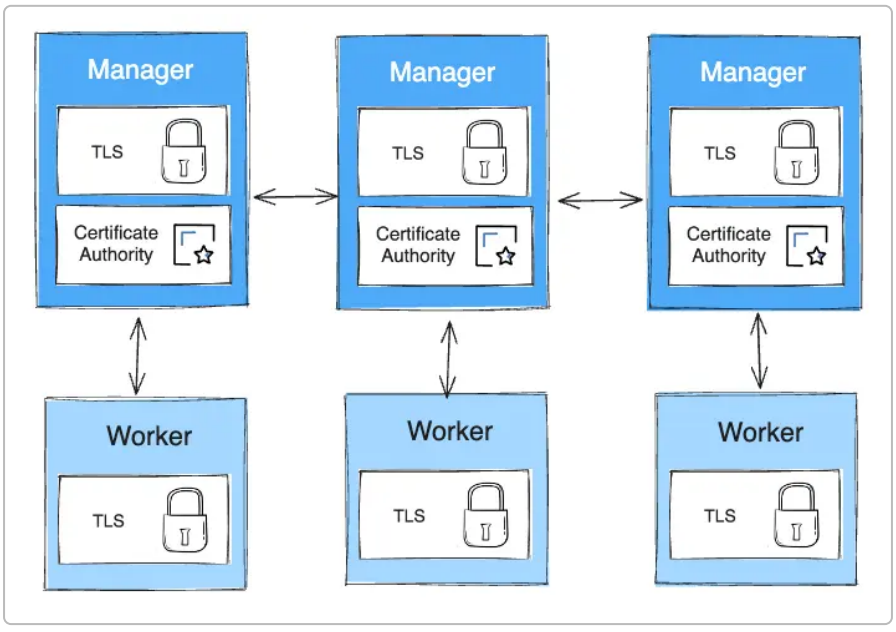
Visão Geral

Este manual descreve o passo a passo para configurar um cluster Docker Swarm com três servidores VPS, seguindo as melhores práticas de segurança e disponibilidade. Teremos três nós (KVM2, KVM4 e KVM8, identificados pelos hostnames informados) atuando **simultaneamente como managers e workers**, devido à limitação de máquinas disponíveis. Esse arranjo de 3 managers proporciona alta disponibilidade: o cluster pode tolerar a queda de um nó gerenciador sem perder funcionamento (quórum), conforme a recomendação de usar um número ímpar de managers [1]. Cada nó worker (inclusive os managers) poderá executar containers agendados pelo Swarm.

Componentes principais da arquitetura:

- **Nó KVM8 (myswarm.cloud)**: será o nó de borda (edge), concentrando componentes extras de infraestrutura. Aqui teremos o proxy reverso (Traefik) responsável por receber o tráfego externo HTTP/HTTPS, além do servidor NFS para armazenamento compartilhado e o serviço de banco de dados (fixado neste nó). Este nó estará exposto nas portas 80/443 para acesso público.
- **Nós KVM2 (inprod.cloud) e KVM4 (otaviomiranda.cloud)**: participarão do cluster Swarm para distribuir cargas de trabalho da aplicação. Eles não receberão tráfego web diretamente do público; todo acesso externo será roteado via o proxy no KVM8.
- **Rede interna**: Os três nós irão se comunicar entre si por meio de portas específicas do Swarm (TCP 2377, TCP/UDP 7946, UDP 4789) e do serviço NFS (porta TCP 2049), as quais serão **estritamente filtradas** para acesso apenas entre os próprios nós (utilizando firewall de borda e UFW em cada servidor). Essa configuração visa minimizar a exposição de portas na Internet.

[2]



O Docker Swarm já inclui recursos de segurança integrados na comunicação interna do cluster. Por exemplo, todos os nós (gerenciadores e workers) utilizam **mutual TLS** para se autenticar e criptografar o tráfego entre si [2]. O diagrama acima, da documentação oficial, ilustra como os managers e workers empregam certificados TLS (emitidos por uma CA interna do Swarm) para garantir conexões seguras dentro do cluster. Além disso, o Swarm renova automaticamente esses certificados dos nós em intervalos regulares (por padrão a cada 90 dias) como medida adicional de segurança [3]. Desta forma, as operações de controle do Swarm e os dados em trânsito nas redes overlay ficam protegidos contra acessos não autorizados.

1. Preparativos Iniciais nos Servidores

Antes de formar o cluster, realize os preparativos básicos em cada VPS:

- **Sistema Operacional e Acesso**: Certifique-se de ter acesso root ou sudo a cada servidor (por SSH, de preferência apenas a partir do seu IP fixo, conforme veremos no firewall). Atualize os sistemas operacionais (ex.: `apt update && apt upgrade` no Ubuntu/Debian) para garantir que todas as correções de segurança estejam aplicadas.

- **Instalação do Docker**: Instale o Docker Engine em cada nó, de preferência usando o repositório oficial do Docker para obter a versão mais recente. Em sistemas Debian/Ubuntu, por exemplo, isso envolve instalar pacotes de pré-requisito (`apt install apt-transport-https ca-certificates curl gnupg`) e adicionar a chave GPG e repositório do Docker, depois instalar o pacote `docker-ce`. Verifique se o Docker está funcionando executando `docker version` ou rodando um container de teste (`docker run hello-world`).

- **Nome de host**: Opcionalmente, defina hostnames lógicos para cada servidor para facilitar identificação (ex.: `hostnamectl set-hostname kvm2-swarm` etc.). O Docker Swarm usa um identificador interno para cada nó, mas ter um hostname reconhecível ajuda na administração. O ideal é também garantir que os nomes (ou IPs) dos nós possam ser resolvidos entre si (via DNS interno ou arquivo `/etc/hosts`), embora não seja estritamente necessário se usarmos os endereços IP diretamente.

- **Sincronização de horário**: Mantenha os relógios das máquinas sincronizados (ex.: instale/configure NTP ou chrony). Isso ajuda na validade de certificados TLS internos e no correto funcionamento de tokens com prazo, evitando problemas sutis de segurança.

2. Configuração de Firewall (Edge e Interno)

Para proteger o ambiente, configuraremos duas camadas de firewall: o firewall de borda da Hostinger **entre os nós** e o firewall local (UFW) em cada VPS. A ideia é **liberar apenas portas essenciais** para o funcionamento do Swarm e da aplicação, bloqueando todo o resto.

2.1. Portas do Docker Swarm a Permitir

O Docker Swarm em modo nativo (Docker 1.12+ swarm mode) requer as seguintes portas abertas entre os nós do cluster [4] :

- **TCP 2377** – Porta de gerenciamento do cluster (usada para comunicação entre managers e para ingressos de novos nós). Deve estar acessível **apenas nos nós managers**. No nosso caso, todos os 3 são managers, então 2377/tcp será permitido mutuamente entre eles [5] .
- **TCP e UDP 7946** – Usada para comunicação de **descoberta de nodes e gossip** do Swarm (control plane) entre todos os nós [6] .
- **UDP 4789** – Usada para tráfego de **rede overlay (VXLAN)** entre os containers nos nós (data plane) [6] .
- *(Opcional)* **Proto 50 (ESP)** – Necessário **apenas se** você ativar criptografia nas redes overlay do Swarm (IPSec). Por padrão, não usaremos overlay criptografada, mas é bom saber que habilitar `--opt encrypted` em uma rede exige permitir o protocolo IP 50 entre os hosts [7] .

Além dessas portas do Swarm, outras portas relevantes para nosso cenário:

- **TCP 22 (SSH)** – Necessária para acesso remoto de administração. Vamos restringir o SSH para apenas seu IP fixo de origem.
- **TCP 80 e 443 (HTTP/HTTPS)** – Necessárias para o tráfego web da aplicação. Apenas o nó de borda (KVM8, `myswarm.cloud`) irá expor essas portas ao público externo. Os demais nós não precisam aceitar conexões HTTP/HTTPS diretamente de fora.
- **TCP 2049 (NFS)** – Porta do serviço NFS (v4) no nó KVM8 que será o **servidor de storage**. Deve ser permitida somente entre o servidor NFS (KVM8) e os clientes (KVM2, KVM4). *Nota:* Usaremos NFSv4.2, que por design opera apenas na porta 2049 e não requer o serviço portmapper separado [8] , facilitando a configuração firewall. (Em NFSv3 ou versões anteriores, seria preciso liberar também portas RPC como 111, mountd, etc., mas o NFSv4 integra essas funções na porta 2049).

Firewall de Borda (Hostinger): Na interface de gerenciamento da Hostinger, crie regras para permitir **apenas** as portas acima **entre os IPs dos 3 servidores**. Em outras palavras, configure regras liberando 2377/tcp, 7946/tcp, 7946/udp, 4789/udp e 2049/tcp tendo como fonte/destino apenas os endereços IP **internos do cluster** (76.13.71.178, 191.101.70.130, 89.116.73.152). Dessa forma, mesmo que esses serviços estejam ouvindo nas máquinas, tráfego de qualquer outro IP externo será bloqueado pela borda. Essa é uma importante medida de segurança para clusters em redes públicas.

Firewall Local (UFW): Adicionalmente, configure o UFW em cada VPS para reforçar a política de mínimos privilégios:

1. **Política padrão**: Certifique-se de que o UFW esteja com política padrão de **negar conexões de entrada** (`ufw default deny incoming`) e **permitir conexões de saída** (`ufw default allow outgoing`). Isso garante que, por padrão, nada entra no servidor sem uma regra explícita, mas os containers e serviços podem acessar a internet ou outros hosts conforme necessário. (O Docker costuma adicionar regras de iptables próprias; manter o padrão de saída liberada evita problemas de conectividade dos containers para fora).

2. **Permitir SSH restrito**: No **todos os nós**, libere SSH apenas do seu IP fixo de administração. Por exemplo: `ufw allow from <seu_IP_fixo> to any port 22 proto tcp`. Isso impede tentativas de SSH de origens desconhecidas, reduzindo superfície de ataque, ao mesmo tempo permitindo que você gerencie os servidores remotamente.

3. **Permitir portas do Swarm entre nós**: Em cada nó, crie regras UFW para permitir as portas 2377, 7946 (tcp e udp) e 4789 (udp) **somente a partir dos outros nós**. Por exemplo, no KVM8 execute:

```
ufw allow from 76.13.71.178 to any port 2377 proto tcp
ufw allow from 191.101.70.130 to any port 2377 proto tcp
ufw allow from 76.13.71.178 to any port 7946 proto tcp
ufw allow from 191.101.70.130 to any port 7946 proto tcp
ufw allow from 76.13.71.178 to any port 7946 proto udp
ufw allow from 191.101.70.130 to any port 7946 proto udp
ufw allow from 76.13.71.178 to any port 4789 proto udp
ufw allow from 191.101.70.130 to any port 4789 proto udp
```

   Repita logicamente nos demais: em KVM2 permita acesso de KVM4 e KVM8 nessas portas; em KVM4, permita KVM2 e KVM8. (Alternativamente, se os IPs tivessem um bloco comum, poderíamos otimizar com range, mas aqui são distintos.)

4. **Permitir NFS restrito**: No servidor NFS (KVM8), permita a porta TCP 2049 somente para os IPs dos clientes (KVM2 e KVM4):

```
ufw allow from 76.13.71.178 to any port 2049 proto tcp
ufw allow from 191.101.70.130 to any port 2049 proto tcp
```

   Nos nós clientes (KVM2, KVM4) não é necessário abrir porta 2049 para entrada, já que eles apenas iniciam conexão ao servidor. O UFW deles, com política de saída liberada, permitirá acessar o KVM8 na porta 2049.

5. **Permitir HTTP/HTTPS externamente no nó de borda**: No KVM8, permita as portas 80/tcp e 443/tcp para acesso de qualquer origem (já que este será o ponto de entrada público da aplicação). Nos outros nós (KVM2, KVM4), *não* abra essas portas para o mundo. Eles não devem receber tráfego HTTP/HTTPS diretamente de clientes externos.

6. *Observação:* Se desejar, você pode também adicionar regras para permitir HTTP/HTTPS **a partir do seu IP fixo** nos nós internos, para fins de teste ou monitoramento. Por exemplo, se você quiser conseguir acessar diretamente um container web rodando no KVM2 via IP, poderia permitir 80/443 from <IP_fixo>. Porém, em um ambiente de produção isso não é usual – normalmente todo tráfego web entra só pelo proxy central. Portanto, avalie se realmente precisa dessa permissão extra.

7. **Habilitar UFW**: Após adicionar as regras, ative o firewall com `ufw enable` (se já não estiver ativo). Lembre-se de verificar se sua sessão SSH atual continua funcional antes de confirmar (para não se trancar para fora – a regra de SSH do seu IP deve estar aplicada).

8. **Verificar regras**: Use `ufw status numbered` para revisar as regras e garantir que estão conforme o esperado (ordem e especificidade das regras podem importar – o UFW processa na sequência). Todas as regras de permitir específicas devem aparecer, e a política padrão deve ser deny (incoming).

Com o firewall configurado, teremos protegido as portas de gerenciamento do Swarm e demais serviços de forma que: - Somente os nós do cluster conseguem se comunicar nas portas de cluster e storage necessários (2377, 7946, 4789, 2049). - Apenas seu IP admin consegue acessar SSH. - Somente o nó de borda aceita tráfego web público.

Essa segmentação minimiza vetores de ataque e segue o princípio de **expor apenas o essencial**.

(Dica: Em ambiente de produção real, também é recomendado usar redes privadas/VPN entre os nós para essa comunicação, se disponível, mesmo com firewall. No nosso caso usaremos IPs públicos filtrados, o que é aceitável, mas considere que ainda é tráfego passando pela internet. Se houver dados sensíveis trafegando (ex: pelo NFS), avalie usar criptografia adicional ou túnel VPN entre os hosts.)

2.2. Considerações sobre UFW e Docker

Vale notar que o Docker manipula diretamente regras de iptables e, por padrão, ele **ignora** o UFW a nível de cadeia FORWARD. Isso significa que containers podem conseguir trafegar sem passar pelos filtros do UFW, a menos que se configure regras adicionais. Uma boa prática é ajustar o arquivo `/etc/default/ufw` definindo `DEFAULT_FORWARD_POLICY="ACCEPT"` (ou adicionar regras UFW do tipo `ufw route` para permitir tráfego de containers). Em versões recentes do Ubuntu, o UFW já trata melhor o forward em conjunto com Docker, mas é importante testar.

Após subir o Swarm e os serviços, se você notar que containers em hosts diferentes não se comunicam ou que tráfego está sendo bloqueado indevidamente, pode ser necessário ajustar as regras de firewall para incluir tráfego de forward/estabelecido apropriado. Por exemplo, garantir que o UFW permita **pacotes estabelecidos/relacionados** (a maioria das configurações UFW já faz isso automaticamente) e possivelmente usar `ufw allow` em interfaces específicas (como a interface docker_gwbridge ou as veth do overlay) se tiver problemas de conectividade entre containers.

Em resumo: a configuração acima deve funcionar para a maioria dos casos, mas fique ciente dessa interação Docker x UFW. Monitore os logs (`/var/log/ufw.log` ou `dmesg`) se algo não comunicar, para ver se algum pacote está sendo bloqueado pelo firewall local e ajuste conforme necessário.

3. Inicializando o Swarm e Adicionando os Nós

Com os preparativos feitos e a rede/firewall prontos, podemos iniciar o cluster Docker Swarm:

1. **Escolher o nó inicial**: Qualquer dos três servidores pode ser usado para iniciar o Swarm, mas normalmente escolhe-se um que ficará como principal ou que já esteja destinado a ser manager. Aqui podemos usar o **KVM8 (myswarm.cloud)** como nó inicial (faz sentido por ser o de borda e ter serviços centrais). Conecte-se via SSH a este servidor.

2. **Executar** `docker swarm init`: No KVM8, rode o comando de inicialização:

   ```
   docker swarm init --advertise-addr 89.116.73.152
   ```

   Use o IP público do próprio KVM8 no `--advertise-addr` (ou o IP privado, se houvesse, mas aqui é o público mesmo). Este comando inicializa o Swarm e torna esse nó o primeiro **manager**. O Docker irá gerar automaticamente um token de ingresso e exibir na saída o comando que outros nós devem executar para se juntar ao cluster [9] . Algo como:

```
Swarm initialized: current node (...) is now a manager.
To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-... <KVM8_IP>:2377
To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

Importante: Copie e guarde esses tokens exibidos. O token de manager é diferente do de worker. Como planejamos adicionar os próximos nós também como managers, precisaremos do token de **manager**. Caso você perca o token ou precise reexibir, pode usar `docker swarm join-token manager` a qualquer momento no nó manager existente para mostrar novamente.

3. **Ingressar os outros nós**: Agora, em cada um dos outros servidores (KVM2 e KVM4), conecte via SSH e execute o comando de join fornecido:

```
docker swarm join --token <TOKEN_MANAGER> 76.13.71.178:2377
```

(Obs: Use o IP do nó inicial/manager – no exemplo acima coloquei 76.13.71.178 como se o KVM2 fosse inicial, corrija para o IP do KVM8, que é 89.116.73.152, conforme mostrado no comando `docker swarm init`*.)*
Ao rodar o join em KVM2 e KVM4 com o token de manager, eles irão se conectar ao KVM8, autenticar via TLS (usando o CA e segredo contidos no token) [10], e integrar-se ao cluster como nós managers adicionais. Se tudo der certo, cada comando retornará uma mensagem de sucesso do tipo *"This node joined a swarm as a manager."*.

4. **Verificar o cluster**: De volta ao nó KVM8 (ou em qualquer manager), rode:

```
docker node ls
```

Isso deve listar os três nós do cluster com seus IDs, hostname, e status Reachable/Leader. Você deverá ver um como Leader (provavelmente KVM8 que iniciou) e os outros como Reachable, todos em disponibilidade *Active*. Neste ponto, o Swarm está formado com 3 gerenciadores.

Por termos 3 managers, nosso cluster está tolerante a falhas de 1 nó gerenciador sem perder o controle (quórum de 2 de 3). Esse é um arranjo clássico de alta disponibilidade para Swarm [1]. *Nota:* Embora todos sejam managers, eles também por padrão funcionam como workers (aceitando tarefas). O Docker permite marcar managers como "Drain" caso não queiramos que executem cargas, mas aqui manteremos todos ativos para distribuir aplicações, já que temos poucos nós [11].

Proteção do Token e Rotação: Vale lembrar que o token de join do manager dá poderes completos a quem o usar (pois ingressa como manager). **Guarde-o com segurança** (trate-o como uma senha do cluster) [12]. Se você suspeitar que o token possa ter vazado, é possível rotacioná-lo a qualquer momento com:

```
docker swarm join-token --rotate manager
```

Isso invalida o token antigo e gera um novo [13]. Da mesma forma, existe `--rotate worker` para o token de worker. Após rotacionar, nós já presentes não são afetados, mas ninguém mais consegue ingressar com o token antigo.

Bloqueio do Swarm (Autolock): Como medida extra, você pode ativar o **autolock** do Swarm. Com isso, sempre que os daemons Docker nos managers reiniciarem, será necessário fornecer uma chave de desbloqueio para o cluster iniciar as funções de orquestração. Essa chave é gerada uma única vez ao ativar o recurso. O autolock protege a chave TLS interna e a chave de criptografia do Raft (log do cluster) de acesso fácil caso alguém obtenha acesso ao sistema de arquivos de um manager [14] [15]. Para ativar já no init poderia ter usado `docker swarm init --autolock` [16]. Para habilitar agora em um cluster já formado, execute em um manager:

```
docker swarm update --autolock=true
```

Isso retornará uma mensagem mostrando a nova chave de desbloqueio do swarm [17] (guarde-a em local seguro, pois será necessária após um reboot do Docker). Com autolock ativo, se um atacante desligar o Docker ou reiniciar o host, o swarm não iniciará automaticamente sem a senha, mitigando o risco de alguém roubar as chaves de um manager offline. A desvantagem é a necessidade de intervenção manual em restarts planejados ou não. Use este recurso conforme seu nível de exigência de segurança.

4. Configurando Armazenamento Persistente Compartilhado (NFS)

Para armazenamento de dados persistentes da aplicação (por exemplo, arquivos enviados pelo usuário, certificados SSL gerados, etc.), utilizaremos um servidor **NFS** centralizado no nó KVM8. Os outros nós montarão esse compartilhamento, permitindo que containers em qualquer nó acessem os mesmos dados. A escolha por NFS foi feita pela simplicidade em um cluster pequeno – embora não seja a opção mais robusta para produção de larga escala, é aceitável para um setup simples se bem restrita ao uso interno [18].

Passo 4.1: Instalar e configurar o servidor NFS no KVM8

- **Instalar servidor NFS**: No KVM8, instale o pacote NFS server (ex.: `apt install nfs-kernel-server` em Ubuntu/Debian). Certifique-se que o serviço `nfs-server` seja habilitado para iniciar no boot.

- **Criar diretório de compartilhamento**: Escolha ou crie um diretório para servir os dados. Por exemplo, podemos usar `/srv/docker-nfs` (ou `/mnt/nfs_share`). Crie o diretório e defina as permissões apropriadas. Exemplo:

  ```
  mkdir -p /srv/docker-nfs
  chown root:root /srv/docker-nfs
  chmod 1770 /srv/docker-nfs
  ```

  Aqui ajustamos permissão 1770 apenas como exemplo se quisermos permitir acesso de um grupo específico; você pode decidir permissões dependendo de quais usuários/containers vão escrever. Em

muitos casos deixar root:root 755 ou 775 é suficiente se containers rodarem como root ou um usuário específico.

- **Exportar o diretório via NFS**: Edite o arquivo `/etc/exports` e adicione uma entrada para compartilhar o diretório apenas com os IPs dos nós do cluster. Por exemplo:

```
/srv/docker-nfs 76.13.71.178(rw,sync,no_subtree_check)
191.101.70.130(rw,sync,no_subtree_check)
```

Isso permite que os hosts KVM2 e KVM4 montem `/srv/docker-nfs` com acesso de leitura/escrita. As opções recomendadas:

- `sync`: garante que gravações sejam feitas de forma síncrona (mais seguro para integridade dos dados, embora um pouco mais lento).
- `no_subtree_check`: melhora desempenho e evita certos problemas de permissões ao não re-checar subárvores na exportação.

- *Opcional:* Você pode adicionar `no_root_squash` se quiser que o usuário root dos clientes NFS seja tratado como root no servidor (por padrão, o NFS mapeia root remoto para *nobody* por segurança – root squash). **Atenção:** `no_root_squash` pode representar risco de segurança, pois dá poderes de root remoto via NFS; só use se necessário e em ambiente controlado. Sem no_root_squash, containers rodando como root podem ter problemas de permissão, pois no servidor as ações deles serão vistas como usuário nobody. Uma abordagem segura é criar usuários/grupos correspondentes nos hosts e adequar permissão dos diretórios. Simplificando: se enfrentar problema de permissão com NFS, pode considerar ativar no_root_squash sabendo dos riscos, ou ajustar UIDs de containers.)

- **Aplicar e verificar export**: Após editar, rode `exportfs -ra` para aplicar. Verifique com `exportfs -v` se a exportação está listada corretamente. Também confirme se o serviço NFS está rodando (`systemctl status nfs-server`).

- **Firewall**: Já configuramos via UFW e firewall de borda a permissão da porta 2049/tcp apenas para os dois clientes. O NFSv4 utiliza somente essa porta [8], então estamos cobertos. (Se eventualmente você usar NFSv3 ou notar tentativas na porta 111, seria o portmapper – mas isso não deve ocorrer se os clientes usarem v4 exclusivamente.)

Passo 4.2: Instalar clientes NFS e montar o volume nos nós KVM2 e KVM4

- **Instalar NFS client**: Nos nós KVM2 e KVM4, instale o pacote cliente NFS (ex.: `apt install nfs-common` em Debian/Ubuntu).

- **Criar ponto de montagem**: Escolha um caminho onde o compartilhamento será montado localmente em cada nó. Por consistência, use o mesmo caminho em todos – por exemplo, também `/srv/docker-nfs` ou `/mnt/nfs`. Vamos supor `/mnt/nfs`. Crie o diretório: `mkdir -p /mnt/nfs`.

- **Montar manualmente (teste)**: Tente montar o compartilhamento do KVM8 para verificar conectividade:

```
mount -t nfs -o vers=4.2 89.116.73.152:/srv/docker-nfs /mnt/nfs
```

Use o IP do KVM8 (ou o hostname `myswarm.cloud` se DNS resolver) seguido de `:/caminho/exportado` e o diretório local. A opção `vers=4.2` força uso da versão NFS 4.2. Se tudo estiver correto, o comando deve completar sem erros. Verifique com `df -h` se o FS apareceu, e teste criar um arquivo: `touch /mnt/nfs/test.txt`. Em seguida, no KVM8 confira se `/srv/docker-nfs/test.txt` surgiu.

- **Configurar montagem automática**: Adicione a entrada no `/etc/fstab` para montar no boot. Por exemplo, em cada cliente:

```
89.116.73.152:/srv/docker-nfs   /mnt/nfs   nfs   defaults,vers=4.2   0 0
```

Assim, no reinício do nó, o compartilhamento será montado automaticamente. (Teste com `umount /mnt/nfs && mount /mnt/nfs` ou reiniciando a máquina, para garantir que a fstab está ok.)

Agora, todos os três nós compartilham o diretório `/mnt/nfs` (com o KVM8 servindo e KVM2/KVM4 montados como clientes). Esse diretório será usado para armazenar dados que precisem ser persistidos ou compartilhados entre containers. Por exemplo, arquivos de certificado TLS gerados pelo Traefik, arquivos de upload da aplicação, etc., podem residir ali.

Dicas de Segurança para o NFS:

- Mantemos o NFS restrito apenas aos IPs internos de confiança. Isso evita acesso não autorizado de outras fontes, já que a porta 2049 nem está acessível externamente e o próprio serviço NFS verifica IPs permitidos via `/etc/exports` e wrappers de host (TCP wrappers, se aplicável).
- Dados no NFS trafegam não criptografados pela rede. Como estamos usando rede pública filtrada, há um risco teórico de intercepção se alguém conseguisse inserir-se no caminho entre os servidores. Se os dados no NFS forem altamente sensíveis, considere opções como configurar uma VPN IPSec entre os nós ou usar Kerberos para NFS (montagem autenticada e criptografada), o que complica bastante a configuração. Para uma aplicação web simples isso normalmente não se justifica, mas é bom estar ciente.
- Monitore logs do NFS (`/var/log/syslog` costuma registrar mounts etc.) e use ferramentas como `nfsstat` para diagnosticar performance. NFS funciona melhor em redes locais de baixa latência; em redes geograficamente distantes ou com latência alta pode apresentar lentidão.
- Considere implementar backup periódico do conteúdo desse NFS, pois é ponto único de falha para os dados persistentes. Um simples script de tar/rsync para outro local já ajuda, dependendo da criticidade dos dados.

5. Deploy do Proxy Reverso (Traefik) no Swarm

Com o cluster ativo e o storage pronto, podemos configurar os serviços Docker propriamente ditos. O primeiro serviço que iremos criar é o **proxy reverso Traefik**, que funcionará como **porta de entrada (edge router)** do nosso cluster. O Traefik receberá conexões HTTP/HTTPS na borda (KVM8) e roteará para os containers adequados internamente, permitindo termos múltiplos serviços web e certificados SSL automáticos via Let's Encrypt.

Por que Traefik? Ele é uma ótima opção nativa para ambiente Docker Swarm, integrando-se via labels do Docker para descobrir automaticamente os serviços e suas rotas, além de gerenciar certificados SSL de forma dinâmica. Conforme destacado em materiais de referência, o Traefik consegue **expor serviços com base em domínios, lidar com múltiplos domínios (hosts virtuais), fazer offload de HTTPS, e obter certificados da Let's Encrypt automaticamente** [19] – tudo isso de forma centralizada. Vamos configurá-lo seguindo boas práticas:

5.1. Rede overlay pública para o Traefik

Primeiro, crie uma **rede overlay** que será usada para conectar o Traefik aos outros serviços web que ele irá proxyar. É recomendável usar uma rede separada dedicada ao Traefik (em vez da rede ingress padrão), pois isso dá mais controle e isolamento.

No swarm, execute em um manager (por exemplo KVM8):

```
docker network create -d overlay traefik-public
```

Essa rede overlay permitirá comunicação entre o container Traefik e quaisquer containers de aplicações web que coloquemos nela. Por ser overlay, ela existirá em todos os nós e trafegará via VXLAN (porta 4789/udp) entre hosts.

Dica: Ativar criptografia IPSec na rede overlay (com `--opt encrypted`) seria possível aqui para aumentar segurança do tráfego interno; entretanto, isso adiciona overhead e requer liberar protocolo 50 no firewall. Em ambiente controlado e já filtrado, pode não ser necessário. Mencionamos para completude: se quiser ativar, recrie a rede com `docker network create -d overlay --opt encrypted traefik-public`.

5.2. Definir rótulo no nó de borda e volume para certificados

Queremos que o Traefik **rode apenas no nó KVM8**, porque somente lá as portas 80/443 estão abertas e porque manteremos lá o storage de certificados. Para garantir que o serviço Traefik não seja movido pelo Swarm para outro host, usaremos **constraints** de nó baseadas em label.

Uma abordagem: adicionar um label específico ao nó KVM8 indicando que ele detém os certificados, e então configurar o serviço Traefik para ser agendado somente em nó com esse label. Por exemplo, vamos conectar no KVM8 (manager) e definir:

```
docker node update --label-add traefik.certificates=true $(docker info -f
'{{.Swarm.NodeID}}')
```

Acima usamos um comando que pega o NodeID do nó atual (que é KVM8 se rodado lá) e atribui o label `traefik.certificates=true`. Você também pode referenciar pelo nome, ex: `docker node update --label-add traefik.certificates=true kvm8`. Verifique com `docker node inspect kvm8 -f '{{.Spec.Labels}}'` se apareceu o label.

Esse label servirá para fixar o Traefik. Também iremos criar um **volume nomeado** para armazenar os certificados do Traefik (arquivo acme.json com certificados TLS obtidos do Let's Encrypt). Como o

volume ficará local ao nó, com o label garantimos que Traefik sempre vá para o mesmo node onde o volume reside, preservando os certificados entre recriações do container [20] [21].

No Swarm Mode, volumes nomeados criados por um serviço são locais ao nó onde o serviço roda, a não ser que usemos um driver externo. Neste caso, poderíamos até armazenar os certificados no NFS compartilhado, mas geralmente opta-se por mantê-los localmente e persistentes no host por segurança e desempenho. Apontar para NFS também funcionaria, mas vamos seguir a prática de volume local + constraint.

5.3. Implantar o serviço Traefik via stack (Docker Compose)

Vamos utilizar um arquivo YAML de stack para definir o serviço Traefik com todos os parâmetros necessários (é possível usar `docker service create` com vários `--label` e opções, mas o compose facilita a organização).

Crie um arquivo `traefik-stack.yml` (pode ser no KVM8 ou em qualquer manager, já que ao aplicar via stack ele distribui ao cluster). Conteúdo sugerido:

```
version: "3.8"

services:
  traefik:
    image: traefik:v2.10  # versão atual do Traefik v2
    ports:
      - 80:80     # Porta HTTP
      - 443:443   # Porta HTTPS
    networks:
      - traefik-public   # conecta-se à rede overlay pública
    deploy:
      placement:
        constraints:
          - node.labels.traefik.certificates == true   # garante rodar no
KVM8
      replicas: 1
      labels:
          # Habilita o Traefik a se auto-configurar através das labels
(piloto do próprio Traefik service)
        - traefik.enable=true
          # Define que Traefik só vai considerar serviços com um certo label
(evita conflito se houver outro Traefik interno em outro stack)
        - traefik.constraint-label=traefik-public
          # Dashboard do Traefik exposto em domínio restrito (opcional):
        - traefik.http.routers.traefik-
dashboard.rule=Host(`traefik.myswarm.cloud`)
        - traefik.http.routers.traefik-dashboard.entrypoints=https
        - traefik.http.routers.traefik-dashboard.tls=true
        - traefik.http.routers.traefik-dashboard.service=api@internal
          # Autenticação básica no dashboard (crie user/pass e gere hash
conforme docs Traefik)
        - traefik.http.middlewares.auth.basicauth.users=admin:$$apr1$
```

```
$XYZ...  # (hash gerado do password)
        - traefik.http.routers.traefik-dashboard.middlewares=auth
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
# dá acesso de leitura ao socket do Docker para Traefik ler serviços e suas
labels
      - traefik-certificates:/etc/traefik/acme        # volume para guardar
certificados ACME (acme.json)
    command:
      - --providers.docker=true
      - --providers.docker.swarmMode=true          # habilita modo Swarm
      - --providers.docker.exposedByDefault=false  # só expõe serviços com
traefik.enable=true
      - --providers.docker.constraints=Label(`traefik.constraint-label`,
`traefik-public`)
      - --entryPoints.http.address=:80
      - --entryPoints.https.address=:443
      - --entryPoints.http.http.redirections.entryPoint.to=https   #
redireciona tudo de HTTP->HTTPS
      - --entryPoints.http.http.redirections.entryPoint.scheme=https
      - --api=true      # habilita dashboard/api
      - --log=true      # habilita log do Traefik (pode configurar loglevel)
      - --accesslog=true # habilita access log (requests)
      - --certificatesResolvers.le.acme.email=email@seu-dominio.com
# email para registro Let's Encrypt
      - --certificatesResolvers.le.acme.storage=/etc/traefik/acme/acme.json
      - --certificatesResolvers.le.acme.httpChallenge.entryPoint=http  # usa
desafio HTTP para obter cert
      # Observação: poderíamos usar tlsChallenge em vez de httpChallenge se
preferir, ou DNS challenge, etc.

networks:
  traefik-public:
    external: true   # usar a rede overlay já criada fora do stack

volumes:
  traefik-certificates:
    name: traefik-public-certificates
    driver: local
```

Explicações dos principais pontos na definição acima:

- Estamos usando Traefik v2.10 (versão atual em 2025). A sintaxe de configurações via labels e comandos é referente à v2 (v3 do Traefik terá sintaxe similar, com alguns ajustes, mas podemos manter v2.x pois é estável).
- **Placement constraint**: `node.labels.traefik.certificates == true` garante que o container suba somente no nó com aquele label (KVM8), onde guardaremos os certs.
- **Replicas 1**: Mantemos uma instância única do Traefik. Poderíamos rodar Traefik em alta disponibilidade (replicas múltiplas), mas isso complica o compartilhamento do certificado ACME. Uma instância já é suficiente para este cluster; se ela falhar, precisaremos recuperá-la (poderia

até ser reimplantada em outro node apontando para o mesmo NFS de certificados, mas vamos focar no básico).

- **Labels no service**: As labels configuradas no próprio serviço Traefik controlam seu comportamento:
- `traefik.enable=true` expõe o próprio Traefik através de si mesmo (autoexposição). Com isso e as regras definidas, estamos expondo o dashboard do Traefik em `traefik.myswarm.cloud` por HTTPS (como exemplo). *Essa parte é opcional e só para poder visualizar a UI do Traefik; você pode remover se não quiser abrir dashboard.* Note que protegemos com auth básica via middlewares labels.
- `traefik.constraint-label=traefik-public` e a correspondente configuração no command `providers.docker.constraints=Label(traefik.constraint-label, traefik-public)` faz com que **este Traefik ignore qualquer container que não tenha o label** `traefik.constraint-label=traefik-public`. Isso é útil se, no futuro, quisermos ter outro Traefik interno rodando para outros fins, ou simplesmente para não pegar serviços que não desejamos expor. Então, toda aplicação que quisermos expor publicamente terá que incluir `traefik.constraint-label=traefik-public` nas suas labels também.
- Redirecionamento HTTP->HTTPS automático via configuração de entryPoints (dessa forma, qualquer requisição na porta 80 será redirecionada para 443 com o mesmo host).
- Configuração do **resolver Let's Encrypt** chamado "le": definimos email de contato e modo de desafio. Aqui usamos o **HTTP challenge**, que requer que Traefik sirva um token em `http://seu_dominio/.well-known/acme-challenge/` quando Let's Encrypt verificar. Por isso é importante ter a porta 80 aberta e sem redirecionamento para algumas rotas específicas (Traefik cuida disso automaticamente para o challenge). Alternativamente, poderíamos usar TLS challenge (que requer estar acessível por 443 sem cert válido, etc.) ou DNS challenge (requere API DNS, etc.). O HTTP challenge é simples e funciona bem quando podemos expor porta 80.

- O arquivo de certificados `acme.json` será salvo em `/etc/traefik/acme/acme.json` dentro do container, que mapeamos para o volume nomeado `traefik-public-certificates` (pasta host persistente). Garantimos assim que, ao recriar o container Traefik, ele lembrará dos certificados já obtidos e não precisará reemitir sempre (evitando rate limit do Let's Encrypt).

- **Volumes**:

- Montamos o socket Docker no container Traefik como somente-leitura. Isso permite ao Traefik ler as informações dos serviços Docker e suas labels para configurar rotas dinamicamente. É um ponto sensível de segurança, pois dar acesso ao socket equivale a dar privilégios de root no host (via API Docker). Estamos montando read-only e confiaremos no Traefik, já que é um software de terceiros amplamente usado para esse fim. (Em ambientes super críticos, algumas pessoas preferem usar Traefik em modo Provider "docker.sock proxy" ou com menor acesso, mas foge do escopo aqui).

- Volume `traefik-certificates`: definimos explicitamente o nome para garantir que não crie um por stack deployment. Nomeamos `traefik-public-certificates` (poderia ser qualquer nome identificável) e driver local. Assim, ele criará um volume local no nó KVM8 (devido à constraint) para guardar os certs.

    - *Observação:* Se preferisse guardar os certs no NFS (para poder restaurar em outro node se precisar rodar Traefik lá), poderia trocar isso por:

```
volumes:
  - /mnt/nfs/traefik/acme:/etc/traefik/acme
```

assumindo que montou NFS em `/mnt/nfs` e criou uma pasta traefik/acme nele. No entanto, isso significa que outro container ou processo poderia teoricamente acessar esse arquivo de certificados via NFS. Mantendo local, limitamos escopo. Vamos manter local pelo exemplo.

- **Rede**: Conectamos o serviço Traefik na rede overlay `traefik-public` (que marcamos como externa no compose, pois já criamos fora). Traefik precisará também acessar a rede de cada serviço que ele for proxyar. Geralmente, você vai conectar os serviços web **tanto** na rede `traefik-public` quanto em sua própria rede interna de backend (se tiver). Dessa forma, Traefik pode alcançá-los via `traefik-public`.

Agora implante o stack no cluster, executando no manager:

```
docker stack deploy -c traefik-stack.yml infra
```

Aqui chamamos a stack de "infra" (pode ser outro nome). O Docker Swarm irá criar o serviço Traefik com as definições dadas.

Verifique com `docker stack services infra` e `docker stack ps infra` se o Traefik subiu corretamente no nó desejado. Ele deve mapear as portas 80/443 no host KVM8. Você pode testar acessando `http://<IP_KVM8>` e ver se redireciona para HTTPS (vai dar provavelmente um cert inválido se Let's Encrypt ainda não gerou, ou um 404 se nenhum router corresponde porque não definimos default). Quando apontarmos uma aplicação, veremos o funcionamento.

> **Nota:** Até obtermos certificados válidos, o Traefik servirá com um certificado self-signed padrão. Assim que configurarmos a aplicação e Traefik puder solicitar o cert para o domínio, ele fará isso automaticamente. Certifique-se de ter apontado o DNS do(s) domínio(s) que usará para o IP do KVM8 antes de testar HTTPS, pois Let's Encrypt precisará resolver e conectar.

5.4. Testando o Traefik (opcional)

Para confirmar que o Traefik está funcionando: - Tente acessar `http://myswarm.cloud` (ou o domínio configurado) em um navegador. Deve converter para `https://myswarm.cloud` e provavelmente mostrar um erro de certificado inicialmente (até LE emitir um válido). - No servidor, veja os logs do Traefik: `docker service logs -f infra_traefik` para ver se há mensagens de erro. Você deverá ver algo sobre "Certificates obtained successfully" após um tempo se Let's Encrypt conseguiu emitir.

Se você configurou o dashboard em `traefik.myswarm.cloud` conforme exemplo, após ter um certificado válido você pode acessar esse hostname e autenticar com o user/password configurado para ver a interface do Traefik com os routers e services monitorados. Lembre de criar o registro DNS `traefik.myswarm.cloud` apontando para o IP antes, se for usar o dashboard.

6. Deploy da Aplicação e Demais Serviços no Swarm

Com o proxy reverso em funcionamento, podemos implantar a aplicação e seus componentes (por exemplo, um serviço web e um banco de dados). A ideia é que o Traefik irá encaminhar as requisições para o serviço web adequadamente via hostnames.

Vamos seguir as orientações gerais para a aplicação:

- **Container da aplicação web**: Suponhamos que temos uma aplicação (por exemplo, um app Python Flask ou Node.js) que escuta na porta 5000 internamente. Vamos implantá-la como um serviço Docker no Swarm, replicado em múltiplos nós para alta disponibilidade, conectado à rede `traefik-public` para ser acessível pelo Traefik, e usando volumes para quaisquer dados persistentes (por exemplo, armazenar uploads no NFS).
- **Banco de dados**: Será um serviço separado, rodando apenas no nó KVM8 (conforme planejado, usando uma constraint), já que não vamos clusterizar o banco nesse momento. Pode ser um container MySQL, Postgres ou outro. Ele ficará acessível apenas internamente (sem portas expostas externamente) e a aplicação se conectará a ele via rede overlay.

6.1. Configurando variáveis e secrets (credenciais)

Antes do deploy, trate variáveis sensíveis apropriadamente: - Em vez de codificar senhas nos YAML, use o recurso **Docker Secrets** do Swarm para armazenar, por exemplo, a senha do banco de dados. Por exemplo:

```
echo "minhaSenhaSeguro123" | docker secret create db_root_password -
```

Isso armazenará a senha como um segredo chamado `db_root_password` no cluster. Esse valor ficará criptografado e replicado somente entre managers [22], e será disponibilizado em arquivo de memória dentro do container que precisar dele (ex: no path `/run/secrets/db_root_password`) [23]. O Swarm garante que segredos só sejam acessíveis pelos serviços aos quais você explicitamente os vincular, e nunca expõe em plaintext fora dos containers destinados [24] [25]. É uma ótima prática para não colocar senhas em variáveis de ambiente ou código.

Faremos isso para o banco de dados (ex.: senha do root ou usuário admin). Para fins de exemplo, criamos acima o secret.

(Lembre-se de criar qualquer segredo antes de rodar o deploy da stack que os utiliza, pois senão a implantação falha. Alternativamente, você pode declarar no compose e o Swarm vai reclamar se não existe.)

6.2. Arquivo de definição (stack) da aplicação

Vamos criar um compose file, por exemplo `app-stack.yml`, com a aplicação web e o banco de dados. Ajuste conforme sua stack real, mas segue um modelo simples:

```
version: "3.8"
services:
  webapp:
    image: nome-da-imagem-da-app:latest   # substitua pela imagem da sua
```

```
aplicação
    networks:
      - traefik-public   # conecta à rede do Traefik para ser acessível
      - app-backend      # rede interna para comunicacao com banco, se
preferir segregar
    deploy:
      replicas: 2       # executa 2 instâncias para alta disponibilidade
      labels:
        - traefik.enable=true
        - traefik.constraint-label=traefik-public    # faz match com o
Traefik edge
        - traefik.http.routers.myapp.rule=Host(`myswarm.cloud`)   # roteia
host principal para este serviço
        - traefik.http.routers.myapp.entrypoints=https
        - traefik.http.routers.myapp.tls.certresolver=le         # usa Let's
Encrypt resolver configurado
        - traefik.http.services.myapp.loadbalancer.server.port=5000  # porta
interna do container que serve a app
        # (Acima assumimos que a app escuta na 5000. Ajuste se for 80, 3000,
etc.)
      resources:
        limits:
          cpus: "0.5"
          memory: 512M
        reservations:
          cpus: "0.2"
          memory: 256M
      restart_policy:
        condition: on-failure
    volumes:
      - app-data:/app/data   # exemplo de volume persistente para uploads/etc
    environment:
      - DB_HOST=db   # host do serviço de BD (ver abaixo)
      - DB_USER=admin
      - DB_PASSWORD_FILE=/run/secrets/db_root_password   # ler a senha do
secret injectado
    secrets:
      - db_root_password

  db:
    image: mysql:8.0        # exemplo com MySQL
    networks:
      - app-backend
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.labels.traefik.certificates == true
# garantir que roda no KVM8 (poderia usar outro label tipo role=db)
      resources:
        limits:
```

```
          cpus: "0.5"
          memory: 1G
    volumes:
      - db-data:/var/lib/mysql
# volume nomeado local no KVM8 para os dados do banco
    environment:
      - MYSQL_ROOT_PASSWORD_FILE=/run/secrets/db_root_password
      - MYSQL_DATABASE=appdb   # define um db inicial
      - MYSQL_USER=admin
      - MYSQL_PASSWORD_FILE=/run/secrets/db_root_password
    secrets:
      - db_root_password

networks:
  traefik-public:
    external: true
  app-backend:
    driver: overlay

volumes:
  app-data:
    driver: local
    driver_opts:
      type: none
      device: "/mnt/nfs/app-data"
      o: bind
  db-data:
    name: db-data
    driver: local
```

Explicando a definição acima:

- **Serviço webapp**:
- Conectado a duas redes: `traefik-public` (para ingress) e `app-backend` (rede overlay interna para comunicar com o DB). Podemos isolar assim o tráfego de DB numa rede que o Traefik não participa, se quisermos mais segurança. Mas não seria problema colocar DB também na traefik-public e usar regras de firewall no serviço... de qualquer forma, optamos por segregar.
- Labels:
  - `traefik.enable=true` e `traefik.constraint-label=traefik-public` sinalizam para o Traefik edge que este serviço deve ser considerado (por conta do constraint label bater com o que configuramos no Traefik).
  - A rota (router) é definida para host `myswarm.cloud` (ajuste para o domínio real da app). Isso quer dizer: Traefik vai encaminhar requisições cujo Host HTTP seja "myswarm.cloud" para este serviço. Se quiser usar subdomínios ou múltiplos domínios, pode acrescentar regras ou usar regex, etc.
  - Definimos que o entrypoint é HTTPS e que deve usar o resolver "le" (Let's Encrypt) para TLS. Isso instruirá o Traefik a obter certificado para `myswarm.cloud` automaticamente e aplicar.
  - Indicamos qual porta interna o Traefik deve direcionar, através de `loadbalancer.server.port=5000`. (Isso é necessário porque, em Swarm, Traefik não

consegue auto-detectar a porta de exposição da app a menos que seja publicada. Como não publicamos porta, precisamos dizer qual porta do container ouvir).

- Replicas 2: teremos dois containers da app rodando (o Traefik fará load balance entre eles automaticamente).
- Volume `app-data` montado em `/app/data`: suponha que a aplicação grava uploads ou certificados locais; isso vai para o NFS. Note que definimos o volume com driver local e `device: /mnt/nfs/app-data` bind, ou seja, estamos **bind-montando** a pasta `/mnt/nfs/app-data` do host dentro do container. Como em todos os nós `/mnt/nfs` aponta para o mesmo storage, independente de em qual nó a tarefa da app rodar, ela estará montando uma pasta do NFS. Isso nos dá armazenamento compartilhado entre réplicas.
  - Não esquecemos de criar a pasta `/srv/docker-nfs/app-data` no servidor NFS e dar permissão (por exemplo, deixar aberta 777 para testes ou ajustar dono).
  - Usamos `driver_opts` no volume para contornar o Swarm não suportar diretamente volumes bind para múltiplos nós via compose. Essa é uma técnica: como o NFS já está montado no host, esse volume na verdade é local para cada nó (mas apontando para o mesmo recurso NFS).
  - Alternativamente, poderíamos simplesmente usar um bind mount direto na definição do serviço (ex: `volumes: - /mnt/nfs/app-data:/app/data`), contudo, o Docker Stack exige que bind mounts referenciem paths presentes em todos os nós alvo. Como garantimos que cada nó tem /mnt/nfs montado, poderia funcionar; mas usar volumes nomeados com device none nos permite gerir nomes e garantir presença via deploy (por isso optei por volume).
- Variáveis de ambiente da app: apontamos DB_HOST para "db" (que é o nome do serviço do database - o Swarm DNS irá resolver `db` para o IP do container do serviço **db** dentro da rede `app-backend`). A senha e usuário estão sendo fornecidos via secrets (no container, o entrypoint do MySQL lerá a variável `MYSQL_ROOT_PASSWORD_FILE` e carrega o conteúdo do arquivo). Na aplicação, usamos `DB_PASSWORD_FILE` para indicar que ela deve ler a senha daquele path (claro, isso depende de como a app foi programada; se não suporta, pode mapear para env normal mas aí evitaríamos pôr a senha em plain text - se necessário, leia o arquivo manualmente no entrypoint do app).

- Incorporamos o secret `db_root_password` no serviço, assim o Swarm montará o arquivo `/run/secrets/db_root_password` dentro do container webapp. Mesma coisa no db.

- **Serviço db**:

- Usamos MySQL como exemplo. Está na rede `app-backend` (somente).
- Replicas 1 e uma constraint para rodar no nó KVM8. Aqui reuso o label `traefik.certificates==true` simplesmente porque sabemos que é KVM8. Em produção seria melhor criar um label específico tipo `role=db=true` para não confundir semanticamente, mas para simplificar usamos o mesmo já aplicado no KVM8. Assim o Swarm agendará o DB container no KVM8 sempre. Isso facilita pois o volume `db-data` será local nesse nó.
- O volume `db-data` é nomeado e local. Ao criar, o Swarm alocará no KVM8 (pois a tarefa está restrita lá) e guardará os dados do banco. Mesmo se o container reiniciar no mesmo nó, dados permanecem. Se por acaso KVM8 fosse perdido, perderíamos o volume e o banco, mas isso faz parte da nossa opção (não estamos fazendo cluster de banco). **Backup do banco** deve ser feito regularmente exportando dados.
- Variáveis do MySQL configuradas para usar o secret ao invés de expor a senha. O container MySQL oficial suporta essa convenção `_FILE` para ler do arquivo do secret automaticamente.

- Externamente, não publicamos porta do MySQL. Então ele só é acessível pelos containers na rede overlay (no nosso caso, os webapps). Traefik também não está nessa rede, então não exporia mesmo que quisesse.

- **Networks**:

- Declaramos `traefik-public` como externa (pois já existe).

- Criamos `app-backend` como overlay para esse stack. O Swarm vai criar ela automaticamente quando subir a stack (poderíamos criar antes, mas não precisa). Essa rede não é acessível por Traefik, logo ainda que Traefik tenha constraint label igual, ele nem enxerga serviço DB pois não compartilham rede. As rotas definidas só apontam pro webapp mesmo.

- **Volumes**:

- `app-data`: aqui demonstro o uso do NFS montado via bind local. Usamos driver local + driver_opts para apontar para `/mnt/nfs/app-data`. Isso efetivamente diz: em qualquer nó que rodar um container usando `app-data`, montar essa pasta local do host. No KVM8, /mnt/nfs/app-data é NFS; no KVM2 idem; KVM4 idem. Então as instâncias do webapp em nós diferentes verão a mesma pasta compartilhada (ideal para conteúdo estático, etc.).
    - Tenha certeza que a pasta existe no NFS e que os clientes tenham montado. Caso contrário, se por exemplo o container subir antes do NFS mount, ele acabará criando localmente um `/mnt/nfs/app-data` vazio dentro do host e escrevendo lá (desconectado do NFS). Para evitar isso, você pode usar a opção `nofail` e script de checagem de mount ou até no command do container checar presença... Em ambiente simples, apenas certifique manualmente que montou NFS antes de subir stack.
- `db-data`: definimos o nome e deixamos driver local. Assim ele fica em `/var/lib/docker/volumes/db-data/_data` no KVM8. Poderíamos também mapear para NFS se quiséssemos mover o banco facilmente, mas banco de dados rodando sobre NFS pode ter degradação de performance e problemas de locking. **Melhor mantê-lo local** para performance. Em caso de migração, faria dump e restore manual, o que ok.

Com esse compose pronto e os secrets criados, execute o deploy:

```
docker stack deploy -c app-stack.yml app
```

Isso irá criar a stack "app" com serviços `app_webapp` e `app_db`. Espere baixar as imagens (certifique que as imagens referenciadas existem e são acessíveis; se é uma imagem privada, precisaria configurar credenciais de registro ou docker login antes).

Verifique `docker stack ps app` para ver se as tasks subiram. O webapp deve ter 2 réplicas distribuídas entre os nós (pode ser que caia 1 no KVM8 e 1 no KVM2, ou KVM4, etc., dependendo do scheduler – todos têm peso igual já que todos são managers/workers). O db deve aparecer como running no KVM8.

6.3. Testando a aplicação via Traefik

Com a stack da aplicação rodando, o Traefik deve detectar o serviço webapp: - Ele verá o label `traefik.http.routers.myapp.rule=Host(myswarm.cloud)` e deve registrar um router para esse host apontando para os containers do serviço. - Se tudo configurado certo, acessar **https://**

myswarm.cloud no navegador deve agora entregar a aplicação (serviço webapp). O Traefik fará o TLS termination (usando certificado da Let's Encrypt que provavelmente ele já obteve) e encaminhar por trás via HTTP para as tasks do webapp.

Confira novamente os logs do Traefik (`docker service logs infra_traefik`) e do webapp (`docker service logs app_webapp`). No Traefik deve constar algo como router created, service created, etc., e nenhum erro de resolução. No log do webapp, você deve ver requisições chegando.

Se a página carregar via HTTPS com cadeado verde, significa que o Traefik conseguiu emitir o certificado e está roteando ok. Se tiver um certificado inválido ainda, pode ser que Let's Encrypt não conseguiu verificar (ex: DNS não apontava certo ou porta 80 bloqueada). Resolva esses issues (ver logs de erro ACME no Traefik) e tente novamente (Traefik re-tenta automaticamente em intervalos, ou você pode escalar o container ou reiniciar para forçar retry, mas cuidado com rate limit do LE se fez muitos fails).

6.4. Observações sobre escalabilidade e atualização

- Para escalar o serviço webapp, basta ajustar `deploy.replicas` no compose e fazer `docker stack deploy` novamente, ou usar `docker service scale app_webapp=3`. O Traefik notará a nova instância e começará a distribuir requests adicionalmente.
- Em caso de atualização da aplicação (nova imagem), use `docker stack deploy` apontando para a nova imagem (ou tag). O Swarm fará *rolling update* (pode configurar detalhes em `deploy.update_config`).
- Mantenha o Traefik sempre com apenas 1 replica neste design, pois temos um único ponto de entrada. Se quisesse redundância do Traefik, precisaria de IP Virtual flutuante ou DNS round-robin e compartilhar o storage de certificados (por ex, colocando acme.json no NFS). Isso é avançado e não faremos aqui, mas é bom saber.
- O banco de dados é single-point. Para HA real, você poderia no futuro usar um serviço gerenciado ou replicar manualmente (ex: um cluster MariaDB Galera, ou Patroni se fosse Postgres). No entanto, isso traz muita complexidade. Para uma app pequena, um container único de DB com backup diário pode ser suficiente.
- Use **Docker configs** para arquivos não sensíveis de configuração (em vez de bind-mount ou baked in image). Por exemplo, se sua app tem um arquivo config.yml, você poderia armazená-lo via `docker config` e montar no container. Isso ajuda a gerenciar configurações separadamente do container e de forma segura se não contiver segredos.
- Monitoramento: Considere adicionar um serviço de monitoramento leve. Há opções no modo Swarm como o Portainer (para UI de administração) e o cAdvisor/Prometheus + Grafana para métricas. O Portainer, por exemplo, pode ser implantado facilmente para visualizar stacks, logs, etc., em UI web protegida. Apenas não exponha abertamente sem proteção robusta.

7. Considerações Finais de Segurança e Boas Práticas

Para encerrarmos, vamos recapitular as principais medidas de segurança e melhorias implementadas, além de outros pontos gerais recomendados:

- **Firewall restritivo**: Abrimos no mínimo necessário (2377/tcp apenas entre managers; 7946/tcp+udp e 4789/udp entre nós; 2049/tcp entre nós; 80/443 público só no edge). Isso reduz a superfície de ataque e cumpre os requerimentos de rede do Swarm [26].

- **Comunicação interna segura**: O Docker Swarm garante que toda comunicação de controle entre os daemons é cifrada com TLS automaticamente [2]. Cada nó tem um certificado individual emitido pela CA do swarm, e esses certs são renovados periodicamente [3]. Essa infraestrutura de PKI interna mitiga ataques de interceptação no tráfego de orquestração.

- **Proteção de segredos**: Utilizamos **Docker Secrets** para senhas sensíveis (do banco). Assim, essas credenciais não aparecem em nenhum arquivo em disco ou comando de texto plano. Os secrets são armazenados no log Raft do Swarm de forma **criptografada em repouso** e só são distribuídos aos nós managers (e aos workers que de fato rodam uma tarefa que precisa dele) [22]. Dentro do container, o segredo fica em tmpfs (memória) e é removido quando não é mais usado [23]. Isso previne vazamentos acidentais de senhas em logs ou imagens.

- **Isolamento de redes**: Segregamos as redes overlay para diferentes propósitos – uma pública (`traefik-public`) para ingress e outra apenas interna (`app-backend`) para comunicação back-end. Isso evita que, por exemplo, o serviço de DB apareça em descobertas de serviço na rede pública. Também ativamos firewall de nível de serviço por não expor porta DB externamente. Adicionalmente, mencionamos a possibilidade de ativar criptografia nas redes overlay se julgado necessário (apesar do overhead) – isso garantiria que mesmo o tráfego dos containers (VXLAN) seja cifrado ponto a ponto [27].

- **Constrait de localização de serviços**: Ao fixar o Traefik e o DB no nó KVM8 via rótulos, garantimos previsibilidade e evitamos latências de rede adicionais para acessar volumes locais (NFS e volume de DB). Além disso, isso simplifica firewall – por exemplo, se o Traefik fosse movido a outro node sem atualizar regras de borda, o tráfego externo poderia ser bloqueado. Portanto, definimos onde os serviços críticos rodam.

- **Autenticação no painel**: Protegemos o dashboard do Traefik com auth básica e ele está servido via HTTPS. Nunca exponha interfaces de gerenciamento sem proteção. Da mesma forma, se usar Portainer ou outra ferramenta, utilize senhas fortes e acesso restrito (Portainer permite rodar em cluster e proteger com JWT, etc.).

- **Manutenção do Swarm**: Lembre-se de aplicar atualizações de segurança do Docker assim que possível. Mantenha o engine Docker e o SO atualizados. Teste updates em ambiente de staging se tiver, antes de produção.

- Sobre managers: Nunca exceda 7 managers nem use número par sem necessidade (3 ou 5 são ideais) [28]. Mais managers aumentam overhead e o cluster ainda tolera apenas (N-1)/2 falhas.
- Monitore o estado do swarm: `docker node ls` para saber se algum caiu (por padrão, se um manager falhar, as tasks em execução nos workers continuam rodando, mas você quer substituí-lo ASAP para recuperar resiliência).

- Se for fazer manutenção em um manager, pode usá-lo como pretexto para testar o failover (pare o Docker de um manager e veja o líder reeleito, etc). Documente qual nó é líder – embora o Swarm lide sozinho, é bom monitorar logs para qualquer eleição instável.

- **Logs e backups**: Configure log rotation para containers (Docker driver `json-file` normalmente limita tamanho, mas verifique). Para logs da aplicação, use uma solução de agregação se necessário (ELK stack, etc., dependendo da complexidade).

- Faça backup regular dos dados persistentes: isso inclui o volume do banco de dados (dump SQL), arquivos em NFS e possivelmente o arquivo `acme.json` do Traefik (pode salvar uma

cópia periódica em outro local, assim se perder o nó inteiro, você tem os certs e evita limites do Let's Encrypt reemitindo).

- **Proteja o daemon Docker**: No nosso caso não expusemos a porta de API Docker (2375/2376). Ela está apenas local. Isso é bom, pois a API do Docker sem TLS é uma porta aberta para root access. Se um dia precisar expor o Docker para um sistema externo (como CI/CD), use **TLS client certificates** ou SSH túnel, nunca exponha 2375 abertamente sem proteção [29].

- **Gerenciamento de acesso**: Considere quem pode acessar esses hosts e cluster. Utilize contas de usuário separadas ou ao menos proteja bem as credenciais. Cada manager dá controle total do cluster, então acesso root nesses servidores deve ser bem guardado. Evite usar root diretamente – crie usuário sudoer e use chave SSH.

- **Planejamento de capacidade**: Três nós pequenos podem não suportar picos muito altos dependendo da aplicação. Monitore uso de CPU, RAM, I/O. O Swarm tem configurações de resources (como fizemos `limits` e `reservations`) para ajudar a evitar sobrecarregar um nó. Se notar um nó saturado e outros ociosos, verifique se o scheduler não ficou limitado por constraints desnecessárias. Você pode adicionar mais nós workers facilmente (ex: se amanhã puder ter mais VPS, basta ingressar com token de worker; managers adicionais só se necessário aumentar resiliência de controle).

- **Testes de falha**: Finalmente, teste cenários de falha de forma controlada. Por exemplo, desligue o servidor KVM2 e veja se o app continua acessível (deve continuar, pois outra réplica no KVM4 ou KVM8 assume todo tráfego). Desligue o KVM8 (nó edge) – isso vai derrubar tudo que está nele (Traefik, DB, etc.). Esse é um ponto fraco: o KVM8 concentra componentes, então se cair, o site sai do ar (mesmo que KVM2 e KVM4 estejam rodando instâncias do app, sem Traefik e DB elas não têm utilidade). Para melhorar isso, no futuro poderia-se separar o load balancer em uma instância dedicada redundante, ou usar dois Traefik com DNS round-robin e DB replicado, etc. Mas inicialmente, saiba das consequências e tenha contingência (ex.: backup do DB para restaurar em outro nó e redeploy Traefik com flag ajustada se KVM8 morrer permanentemente).

Em resumo, seguimos práticas sólidas de segurança e configuração para um cluster Docker Swarm básico de 3 nós. Resumidamente, aplicamos: isolamento de portas e redes, TLS interno, secrets, persistência em volumes compartilhados, e mantivemos serviços expostos minimamente (apenas Traefik na borda). Com isso, você deve ter um ambiente orquestrado pronto para rodar sua aplicação atual e futuras, com a tranquilidade de facilitar escalabilidade e manutenção centralizada via Swarm, e com atenção aos principais pontos de segurança para evitar armadilhas comuns. Boa implantação!

Referências Utilizadas:

- Documentação do Docker Swarm sobre requisitos de porta e firewall [30], confirmando as portas 2377, 7946 e 4789 necessárias e o uso do SSH e outras portas de serviço conforme demanda.
- Guia da Red Hat destacando que o NFSv4 opera apenas na porta TCP 2049 e não requer portmapper (rpcbind) [8], simplificando a configuração de firewall para o serviço de arquivos.
- Recomendações da Docker/Aqua Security de usar número ímpar de managers (três) para tolerância a falhas e quórum adequado no Swarm [1].
- Documentação oficial do Docker sobre o sistema de PKI interno do Swarm, uso de mutual TLS e renovação automática de certificados a cada 90 dias [2] [3].
- Artigo da Docker explicando o recurso de **autolock** para proteger as chaves de criptografia do Swarm em repouso, exigindo desbloqueio manual após reinício [14] [16].

- Referência da Docker Secrets que reforça a criptografia e alta disponibilidade dos segredos no log Raft do Swarm [22], garantindo que dados sensíveis (senhas, chaves) não fiquem expostos em texto plano.
- Exemplo do DockerSwarm.rocks demonstrando deploy do Traefik v2 em Swarm com constraint por label e storage persistente de certificados [20] [21], padrão que adotamos para o proxy reverso.
- Trechos da documentação do Traefik e do guia DockerSwarm.rocks confirmando as capacidades de routing por domínio e obtenção automática de certificados Let's Encrypt pelo Traefik [19].
- Recomendações gerais de segurança para tokens de join do Swarm (proteger e rotacionar quando necessário) [12] [13], uma vez que posse do token de manager equivale a controle do cluster.

Espero que este manual o ajude a configurar seu Docker Swarm com sucesso, unindo as peças (containers, rede, storage) de forma segura e eficiente. Com essa base, você poderá expandir e atualizar sua infraestrutura com confiança nas próximas etapas do projeto. Boa sorte! [31] [32]

---

[1] [11] [12] [13] [28] Docker Swarm
https://www.aquasec.com/cloud-native-academy/docker-container/docker-swarm/

[2] [3] [10] Manage swarm security with public key infrastructure (PKI) | Docker Docs
https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/

[4] [5] [6] [26] [29] [30] How to Configure the Linux Firewall for Docker Swarm on Ubuntu 16.04 | DigitalOcean
https://www.digitalocean.com/community/tutorials/how-to-configure-the-linux-firewall-for-docker-swarm-on-ubuntu-16-04

[7] [27] [32] Docker Swarm Port Requirements, both Swarm Mode 1.12+ and Swarm Classic, plus AWS Security Group Style Tables · GitHub
https://gist.github.com/BretFisher/7233b7ecf14bc49eb47715bbeb2a2769

[8] Chapter 9. Network File System (NFS) | Reference Guide | Red Hat Enterprise Linux | 4 | Red Hat Documentation
https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/4/html/reference_guide/ch-nfs

[9] Scaling Made Easy: Deploying a Robust 3-Node Docker Swarm Cluster | by Jeremy Reid | Medium
https://medium.com/@jeremyreid757/scaling-made-easy-deploying-a-robust-3-node-docker-swarm-cluster-b9edece121e1

[14] [15] [16] [17] Lock your swarm to protect its encryption key | Docker Docs
https://docs.docker.com/engine/swarm/swarm_manager_locking/

[18] Best practice for persistent data using swarm - General
https://forums.docker.com/t/best-practice-for-persistent-data-using-swarm/38897

[19] [20] [21] Traefik Proxy with HTTPS - Docker Swarm Rocks
https://dockerswarm.rocks/traefik/

[22] [23] [24] [25] Manage sensitive data with Docker secrets | Docker Docs
https://docs.docker.com/engine/swarm/secrets/

[31] What is Docker Swarm? - Sysdig
https://www.sysdig.com/learn-cloud-native/what-is-docker-swarm