Docker client-daemon: Types of Connections

19 September 2024 08:39

For practice Docker in Laptop, we can install "Virtual machines (VM)" on "Docker Desktop" on our Laptop.

The setup look like Below:

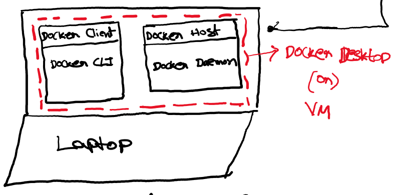


Fig: Client & server inside same machine.

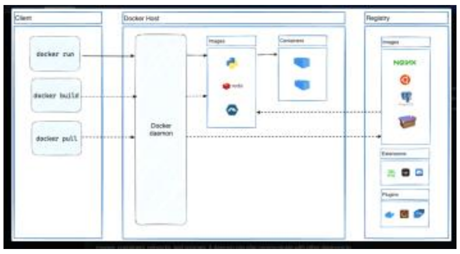
Type 01:-
Client & server on the same machine.

Type 02:-
Client on laptop and Server on VM (or) AWS.

Docker Desktop
Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (`dockerd`), the Docker client (`docker`), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Client-Server Architecture
The Client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters called clients.

Docker Architecture
Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connecta Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



Docker daemon
The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker client
The Docker client (`docker`) is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon

Type 02

Configure remote access for Docker daemon
By default, the Docker daemon listens for connections on a Unix socket to accept requests from local clients. You can configure Docker to accept requests from remote clients by configuring it to listen on an IP address and port as well as the Unix socket.

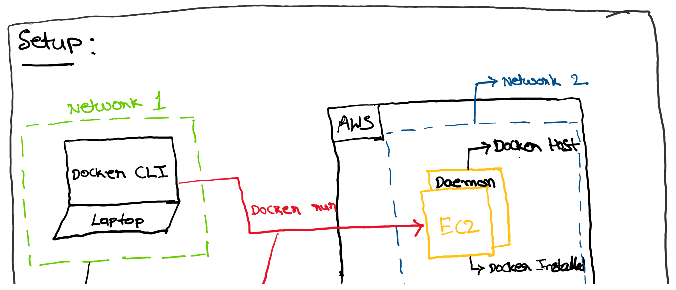
Enable remote access

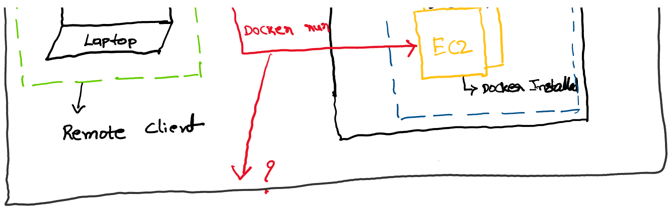
- You can enable remote access to the daemon either using a "docker.service" systemd unit file for Linux distributions using systemd. Or you can use the "daemon.json" file, if your distribution doesn't use systemd.
- Configuring Docker to listen for connections using both the systemd unit file and the "daemon.json" file causes a conflict that prevents Docker from starting.

1. Configuring remote access with systemd unit file
2. Configuring remote access with daemon.json

Note: https://docs.docker.com/engine/daemon/remote-access/

- By default, Docker runs through a non-networked UNIX socket. It can also optionally communicate using SSH or a TLS (HTTPS) socket.





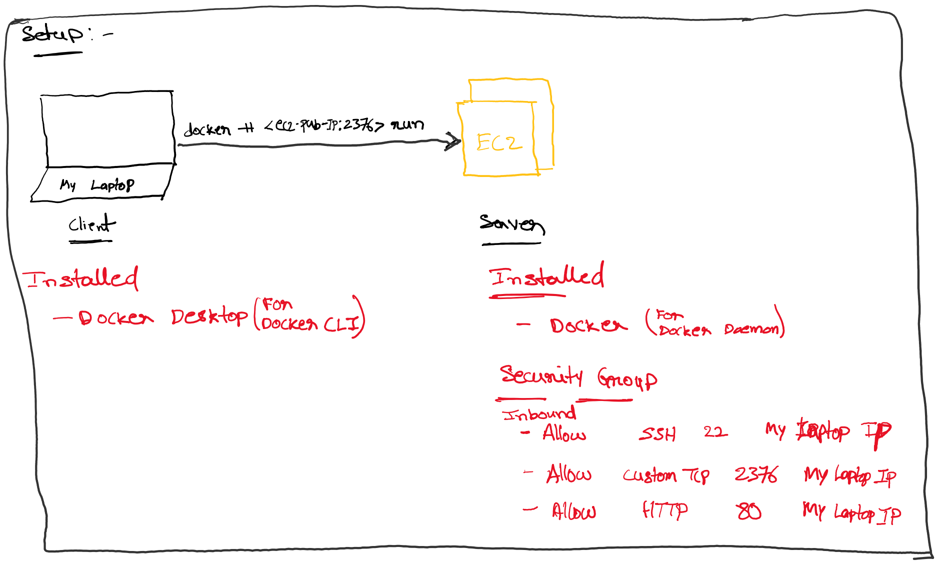
How to do this?

Ans:- Currently I know 3 ways to configure remote access to Docker daemon.

1. Tcp protocol — without using TLS certificates → Just for practice (not recommended)
2. using TLS certificates → prefered for production env
3. SSH Connection

} Both are secure ways to connect Docker daemon

1. **Configure remote access for Docker daemon (with systemd unit file) - without TLS certificates**



Installed
— Docker Desktop (For Docker CLI)

Installed
— Docker (For Docker Daemon)

Security Group
Inbound

- Allow	SSH	22	My ~~laptop~~ IP
- Allow	custom Tcp	2376	My Laptop Ip
- Allow	HTTP	80	My Laptop IP

For Ubuntu - Machine:

Setup details:
Client: my laptop Docker
Host (contain docker daemon): AWS EC2

step 0: AWS
1. launch ec2 instance (Ubuntu)
   security groups
   - allow SSH 22 - my Ip
   - allow custom TCP 2376 - my IP
   - for nginx container: allow http 80 my Ip
2. install docker (Ubuntu)
   - sudo apt-get update
   - sudo apt-get install docker.io -y
   - sudo systemctl start docker
   - sudo systemctl status docker
   - sudo systemctl enable docker # To start the Docker service automatically when the instance starts.
   - sudo usermod -a -G docker $(whoami) # if we are not added user to docker group, then we need to use "sudo" example: "sudo docker run"
   - newgrp docker # apply the above user group changes

step 1: docker configuration file (ec2)
docker default configuration file location: /usr/lib/systemd/system/docker.service
Override file location: /etc/systemd/system/docker.service.d/override.conf

Type 1:
sudo systemctl edit docker.service --> this automatically creates "/etc/systemd/system/docker.service.d/override.conf" file
enter the below lines
```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://<aws-ec2-private-ip>:2376 --
containerd=/run/containerd/containerd.sock
```
Type 2:
sudo mkdir -p /etc/systemd/system/docker.service.d
sudo nano /etc/systemd/system/docker.service.d/override.conf
enter the below lines
```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://<aws-ec2-private-ip>:2376 --
containerd=/run/containerd/containerd.sock
```

Note:
- aws-ec2-private-ip not change even the instance is restarted.
- I also tried using the public ip in the "override.conf" file and I faced an issue while running "sudo systemctl restart docke r" command

step 2: Reload the systemd configuration and restart Docker:
```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

step 3: Verify the Network Configuration

```
ip a | grep <aws-ec2-private-ip> # make sure is IP is valid IP address configured on the server and that port 2376 is not already in
use
sudo netstat -tuln | grep 2376
(or)
sudo netstat -lntp | grep dockerd
```

step 4: Connect Docker CLI on Windows to the AWS EC2 Docker Host & Verify the Connection: (my laptop)

In laptop - open powershell:

```
set DOCKER_HOST=tcp://<your-ec2-public-ip>:2376
docker version
docker run -d nginx
(or)
docker -H <your-aws-public-ip:2376> run -d nginx
docker -H 3.71.35.109:2376 container ls
```

For Amazon Linux - Machine:

Setup details:
Client: my laptop Docker
Host (contain docker daemon): AWS EC2

only change in step 1 and step 0 - install docker commands:

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://<aws-ec2-private-ip>:2376 --containerd=/run/containerd/containerd.sock
$OPTIONS $DOCKER_STORAGE_OPTIONS $DOCKER_ADD_RUNTIMES
```

Note:
- you can use any port like 2375, but add in ec2 security group inbound list.

2. **Configure remote access for Docker daemon (with systemd unit file) - with TLS (HTTPS) certificates**
   https://docs.docker.com/engine/security/protect-access/

Overview

1. Generate TLS certificates on the Docker host (AWS EC2).
2. Configure the Docker daemon to use these certificates and allow remote access.
3. Open port 2376 in the EC2 security group.
4. Transfer client certificates to your Windows laptop.
5. Configure the Docker client on Windows to connect to the remote Docker daemon using the certificates.

Step 1: Generate TLS Certificates on the Docker Host (AWS EC2)

To securely connect to the Docker daemon remotely, you need to generate TLS certificates. This setup involves creating:
a. Certificate Authority (CA) certificate.
b. server certificate for the Docker daemon.
c. client certificate for the Docker client (Windows laptop).

1. Install OpenSSL (if not already installed):

On Ubuntu Machine:

```
sudo apt-get update
sudo apt-get install -y openssl
```

2. Create a directory for the certificates:

```
mkdir -p ~/docker-certs
cd ~/docker-certs
```

3. Generate a Certificate Authority (CA): The CA certificate is used to sign the server and client certificates.

```
openssl genrsa -aes256 -out ca-key.pem 4096  # CA private key note: enter atleast 4 digits
openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
```

4. Create a Server Certificate and Key for Docker Daemon:

Generate a private key for the server:

```
openssl genrsa -out server-key.pem 4096
```

Create a Certificate Signing Request (CSR) for the server:

```
openssl req -subj "/CN=<your-ec2-public-ip>" -new -key server-key.pem -out server.csr
```

note: Replace <your-ec2-public-ip> with the public IP address of your EC2 instance.

Create an extfile for the server certificate:

```
echo "subjectAltName = IP:<your-ec2-public-ip>,IP:127.0.0.1" > extfile.cnf
echo "extendedKeyUsage = serverAuth" >> extfile.cnf
```

Generate the server certificate:

```
openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out server-cert.pem -extfile extfile.cnf
```

5. Create a Client Certificate for Your Laptop:

Generate a private key for the client:

```
openssl genrsa -out key.pem 4096
```

Create a CSR for the client:

```
openssl req -subj '/CN=client' -new -key key.pem -out client.csr
```

Create an extfile for the client certificate:

```
echo "extendedKeyUsage = clientAuth" > extfile-client.cnf
```

Generate the client certificate:

```
openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem -extfile extfile-client.cnf
```

6. After generating cert.pem and server-cert.pem you can safely remove the two certificate signing requests and extensions config files:

```
rm -v client.csr server.csr extfile.cnf extfile-client.cnf
```

7. Set the Correct Permissions on the Certificates:

- To protect your keys from accidental damage, remove their write permissions. To make them only readable by you, change file modes as follows:

```
chmod 0400 ca-key.pem key.pem server-key.pem
chmod 0444 ca.pem server-cert.pem cert.pem

or

chmod -v 0400 ca-key.pem key.pem server-key.pem
chmod -v 0444 ca.pem server-cert.pem cert.pem
```

8. Copy the Server Certificates to Docker's Directory:

```
sudo mkdir -p /etc/docker/certs
sudo cp ca.pem server-cert.pem server-key.pem /etc/docker/certs/
```

Step 2: Configure the Docker Daemon for Remote Access

- You need to configure the Docker daemon to use the generated certificates and bind to the public IP.

1. Edit Docker's systemd service file:
    Create or edit /etc/systemd/system/docker.service.d/override.conf:

```
sudo mkdir -p /etc/systemd/system/docker.service.d
sudo nano /etc/systemd/system/docker.service.d/override.conf
```

    Add the following content to the override.conf file:

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376 --tlsverify --tlscacert=/etc/docker/certs/ca.pem --tlscert=/etc/docker/certs/server-cert.pem --tlskey=/etc/docker/certs/server-key.pem      # in a single line
```

    note:
        -H tcp://0.0.0.0:2376 makes Docker listen on all network interfaces. You could specify a specific IP address instead of 0.0.0.0 for more control.

2. Reload the systemd configuration and restart Docker:

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Step 3: Open Port 2376 in Your AWS Security Group

Step 4: Transfer Client Certificates to Your Windows Laptop
- Copy the ca.pem, cert.pem, and key.pem files from your EC2 instance (~/docker-certs/) to your Windows laptop using a secure method (e.g., scp, WinSCP, or similar).

- Place them in a directory on your laptop, e.g., C:\Users\<YourUsername>\docker-certs.

example: `scp -i ec2key.pem ec2-user@<your-ec2-public-ip>:~/docker-certs/{ca,cert,key}.pem C:/Users/pavan/docker_certs`

test: `docker --tlsverify --tlscacert=./docker_certs/ca.pem --tlscert=./docker_certs/cert.pem --tlskey=./docker_certs/key.pem -H=<your-ec2-public-ip>:2376 info`

Step 5: Configure Docker Client on Your Windows Laptop

Type 1: (for single section: testing)

1. Open Command Prompt or PowerShell on your Windows laptop.

2. Set the environment variables to use the remote Docker host:

```
set DOCKER_HOST=tcp://<your-ec2-public-ip>:2376
set DOCKER_TLS_VERIFY=1
set DOCKER_CERT_PATH=C:\Users\<YourUsername>\docker-certs
```

Note:
- Replace <your-ec2-public-ip> with the public IP of your EC2 instance.
- Replace <YourUsername> with your actual Windows username.

3. Test the Remote Docker Connection:

```
docker info
```

Type 2: Persisting Environment Variables

1. Create a .docker directory on your laptop if it doesn't already exist:

```
mkdir %USERPROFILE%\.docker
```

2. Move the client certificates (ca.pem, cert.pem, key.pem) to this directory.

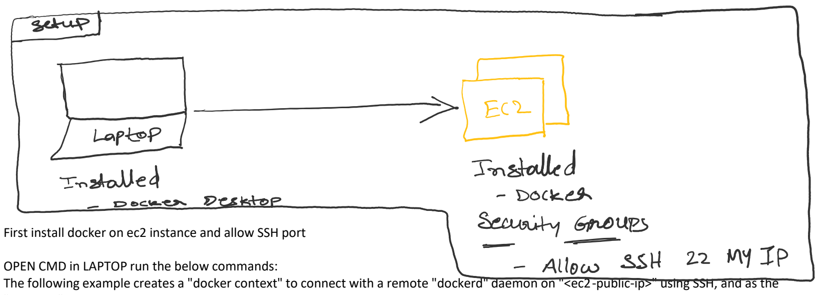
3. Create or edit a Docker configuration file:

    Create a file named config.json in %USERPROFILE%\.docker and add the following:

```
{
  "tlsverify": true,
  "tlscacert": "%USERPROFILE%\\.docker\\ca.pem",
  "tlscert": "%USERPROFILE%\\.docker\\cert.pem",
"tlskey": "%USERPROFILE%\\.docker\\key.pem",
"hosts": ["tcp://<your-ec2-public-ip>:2376"]
}
```

4. docker info

3. SSH: https://docs.docker.com/engine/security/protect-access/



1. First install docker on ec2 instance and allow SSH port

2. OPEN CMD in LAPTOP run the below commands:
   The following example creates a "docker context" to connect with a remote "dockerd" daemon on "<ec2-public-ip>" using SSH, and as the "ec2-user" user on the remote machine.
   command 1:

```
docker context create --docker host=ssh://ec2-user@<ec2-public-ip> --description="Remote engine" my-remote-engine
or
docker context create --docker "host=ssh://ec2-user@ec2-3-68-187-98.eu-central-1.compute.amazonaws.com" --description="Remote engine" my-remote-engine

message: Successfully created context "my-remote-engine"
```

   command 2:

```
docker context use my-remote-engine

message: Current context is now "my-remote-engine"
```

   command 3:

docker info

error message:
while using ec2 public ip or DNS: stderr=ec2-user@<ec2-public-ip>: Permission denied (publickey,gssapi-keyex,gssapi-with-mic)

Note:
docker context create --docker host=ssh://ec2-user@<ec2-private-ip> --description="Remote engine" my-remote-engine
docker context use my-remote-engine
docker info
error message:
while using ec2 private ip: stderr=banner exchange: Connection to UNKNOWN port-1: Connection refused

To resolve above issue:

go to file location IN LAPTOP: C:/Users/pavan/.ssh/config

add below details: (to specify the SSH key for the remote host)

```
# AWS EC2 instance remote docker configuration using "docker context"
Host <aws-ec2-public-ip>
        User ec2-user # if Ubuntu
        IdentityFile <full-path-of ec2-instance-pem-key>.pem
        StrictHostKeyChecking no
```

OPEN CMD in LAPTOP

- run the command 1 to 3 one by one.

note: I also checked using "aws-ec2-private-ip", but I got same error message "Connection to UNKNOWN port -1: Connection refused" mentioned earlier

Testing: CMD in LAPTOP
- First check the current host using command "docker context show"
- docker run nginx # login to ec2 instance and check running containers (docker ps -q)

Hints:
- To know more commands just type "docker context" in CMD and it provides list
- switch back to default host: "docker context use default"