

Prof Hirata e Prof Juliana

Ao invés de simular o sistema distribuído utilizando diferentes processos em terminais (que compartilham recursos de memória e de rede), vamos isolar cada um dos processos em um *container*.

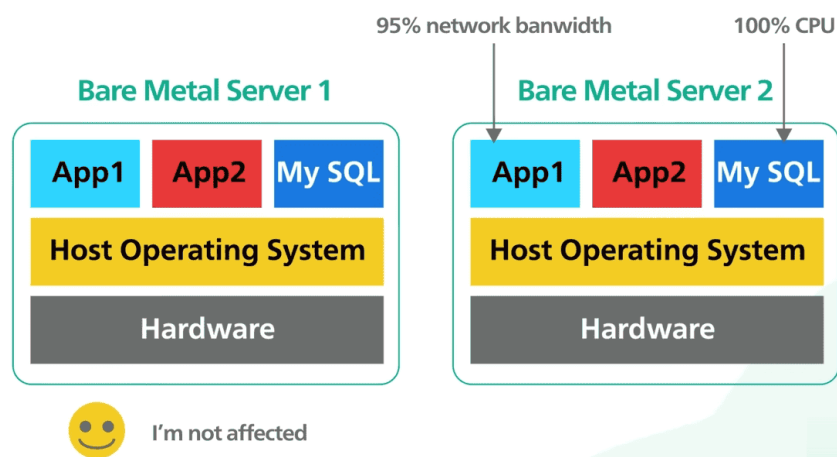
*****Background*****



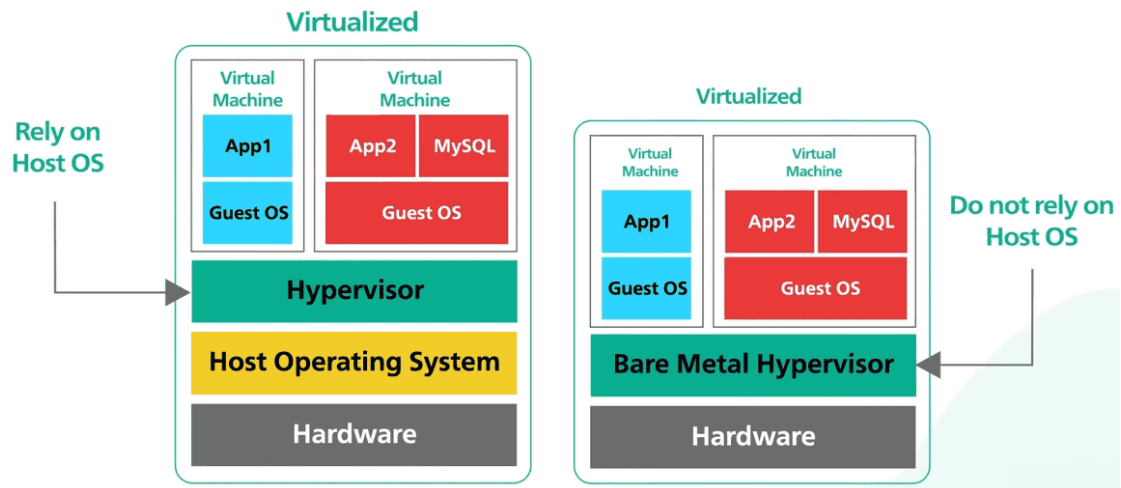
Docker is an open source platform that enables developers to build, deploy, run, update and manage **containers**.

Deploy types:

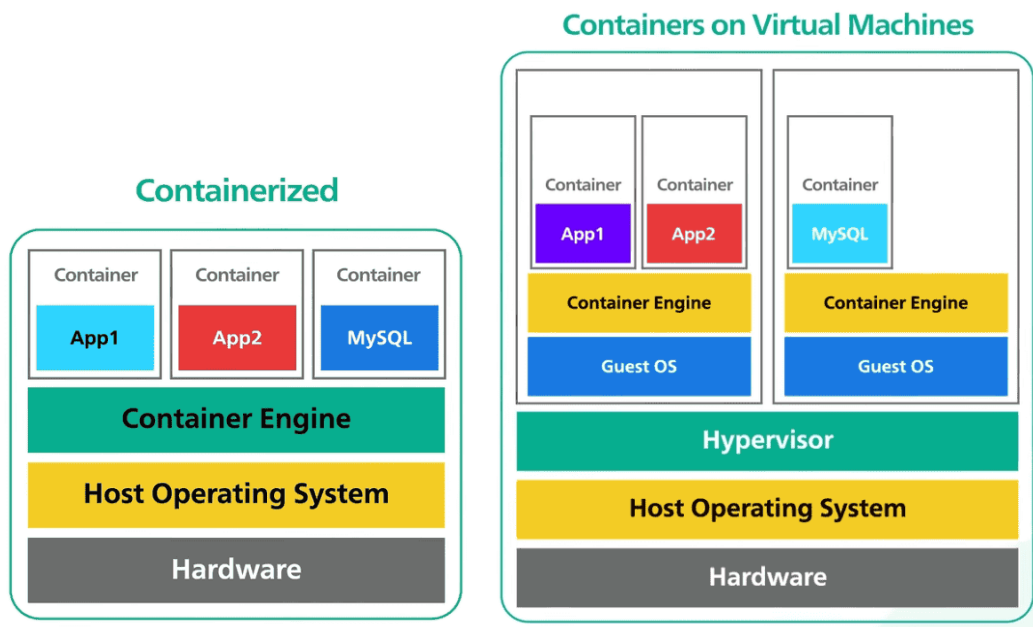
- **Bare metal server (traditional):** a physical computer dedicated to a single tenant.
 - It means that you don't share resources with anyone else!
 - Advantages: complete control, physical isolation, regulatory compliance.
 - Disadvantages: cost, management complexity, scalability.



- **Virtual machine (VM):** a software program that emulates a physical computer. This process is known as **virtualization**.
 - Multiple virtual machines can run on a single piece of hardware, or bare metal.
 - The host operating system runs on top of the bare metal hardware.
 - A hypervisor (or virtual machine monitor) runs on top of the host operating system. It creates and manages virtual machines.
 - Advantages: cost savings, scalability, flexibility.
 - Disadvantages: noisy neighbor problem, security vulnerabilities.

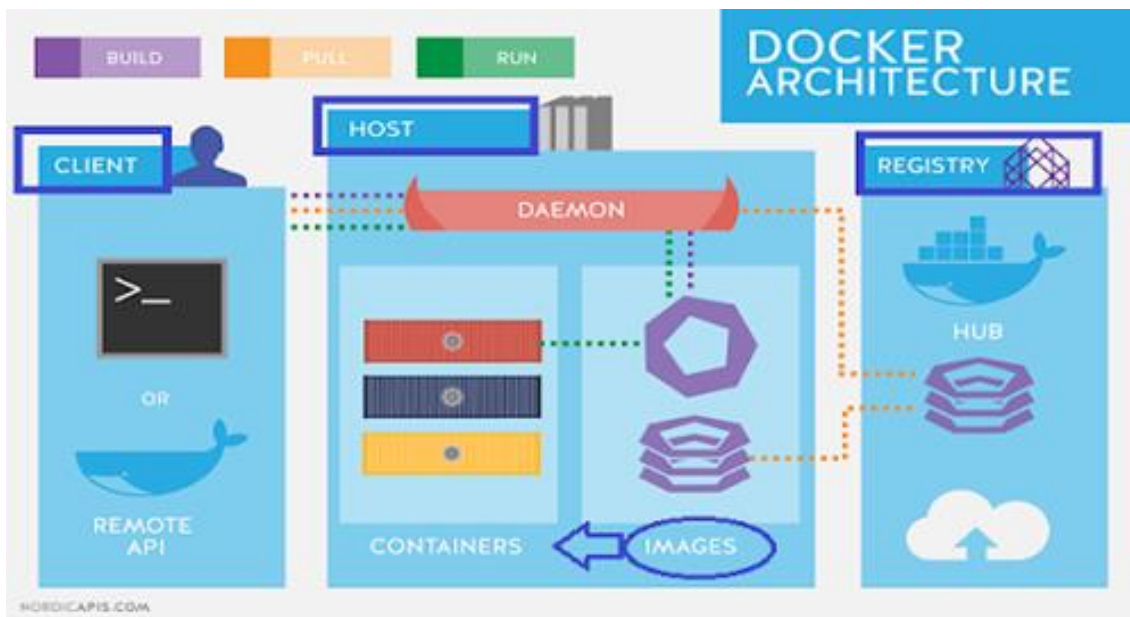


- **Container:** a self-contained package that contains an application along with all its dependencies, such as libraries, frameworks, and runtime.
 - It means that a container has everything it needs to run!
 - Advantages:
 - Lighter weight: Unlike VMs, containers don't carry the payload of an entire OS instance and hypervisor.
 - Improved developer productivity: Containerized applications can be written once and run anywhere. It means portability, since all dependencies are packed in the container.
 - Disadvantages: security risks, complexity (many moving parts in deploy), limited functionality (a challenge for very complex applications).



Docker terms:

- Dockerfile
 - A text file containing instructions for how to build the Docker container image.
- Docker images
 - They contain executable application: source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container.
- Docker containers
 - The running instances of Docker images.
- Docker Desktop
 - An application for Mac or Windows that includes Docker Engine, Docker CLI client, Docker Compose, Kubernetes, and others. It also includes access to Docker Hub.
- Docker daemon
 - A service that creates and manages Docker images, using the commands from the client.
- Docker host
 - The server on which Docker daemon runs.
- Docker registry
 - A scalable open-source storage and distribution system for Docker images.
- Docker Hub
 - The public repository of Docker images. Users can download predefined base images from the Docker filesystem to use as a starting point for any containerization project. Other image repositories exist, as well, notably GitHub.



Docker Compose is a tool for defining and running multi-container Docker applications.

- You use a YAML file to configure your application's services.
- Then, with a single command, you create and start all the services from your configuration.

References:

<https://firstfinger.in/bare-metal-vs-virtual-machines-vs-containers/>
<https://www.ibm.com/topics/docker>

Yet Another Markup Language: linguagem frequentemente usada para arquivos de configuração.

*****Prática*****

Instale o Docker localmente: <https://docs.docker.com/get-docker/>

Note que a instalação para Windows requer que o WSL2 (*Windows Subsystem for Linux 2*) esteja habilitado.

Vamos usar arquivo `Process.go` da Atividade Dirigida 1 (dica 2), onde temos um processo oriundo da junção cliente e servidor. Ou seja, um processo capaz de enviar mensagens (pelo cliente) e receber mensagens (pelo servidor). Nesse exemplo, o processo fica mandando números para os amigos. Lembra?

```
package main

import (
    "fmt"
    "net"
    "os"
    "strconv"
    "time"
)

// Variáveis globais interessantes para o processo
var err string
var myPort string          //porta do meu servidor
var nServers int           //qtde de outros processo
var CliConn []*net.UDPConn //vetor com conexões para os servidores
// dos outros processos
var ServConn *net.UDPConn //conexão do meu servidor (onde recebo
//mensagens dos outros processos)

func CheckError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
        os.Exit(0)
    }
}

func PrintError(err error) {
    if err != nil {
        fmt.Println("Error: ", err)
    }
}

func doServerJob() {
    buf := make([]byte, 1024)
    //Loop infinito mesmo
    for {
        // Ler (uma vez somente) da conexão UDP a mensagem
        n, addr, err := ServConn.ReadFromUDP(buf)
```

```

        PrintError(err)
        // Escrever na tela a msg recebida (indicando o
        // endereço de quem enviou)
        fmt.Println("Received ", string(buf[0:n]), " from ", addr)
    }
}

func doClientJob(otherProcess int, i int) {
    // Enviar uma mensagem (com valor i) para o servidor do processo
    //otherServer.
    msg := strconv.Itoa(i)
    i++
    buf := []byte(msg)
    _, err := CliConn[otherProcess].Write(buf)
    PrintError(err)
}

func initConnections() {
    myPort = os.Args[1]
    nServers = len(os.Args) - 2
    /*Esse 2 tira o nome (no caso Process) e tira a primeira porta (que é a
    minha). As demais portas são dos outros processos*/
    CliConn = make([]*net.UDPConn, nServers)

    /*Outros códigos para deixar ok a conexão do meu servidor (onde recebo
    msgs). O processo já deve ficar habilitado a receber msgs.*/

    ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+myPort)
    CheckError(err)
    ServConn, err = net.ListenUDP("udp", ServerAddr)
    CheckError(err)

    /*Outros códigos para deixar ok a minha conexão com cada servidor dos ou-
    tros processos. Colocar tais conexões no vetor CliConn.*/
    for s := 0; s < nServers; s++ {
        ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+os.Args[2+s])
        CheckError(err)

        /*Aqui não foi definido o endereço do cliente.
        Usando nil, o próprio sistema escolhe. */
        Conn, err := net.DialUDP("udp", nil, ServerAddr)
        CliConn[s] = Conn
        CheckError(err)
    }
}

func main() {

    initConnections()

```

```

    //O fechamento de conexões deve ficar aqui, assim só fecha //conexão
    quando a main morrer
    defer ServConn.Close()
    for i := 0; i < nServers; i++ {
        defer CliConn[i].Close()
    }

    /*Todo Process fará a mesma coisa: ficar ouvindo mensagens e mandar infi-
    nitos i's para os outros processos*/
    go doServerJob()
    i := 0
    for {
        for j := 0; j < nServers; j++ {
            go doClientJob(j, i)
        }
        // Espera um pouco
        time.Sleep(time.Second * 1)
        i++
    }
}

```

Testamos assim antes:

- Terminal 1: Process :10002 :10003 :10004
- Terminal 2: Process :10003 :10002 :10004
- Terminal 3: Process :10004 :10002 :10003

Da forma implementada, a primeira porta é a do processo corrente e as demais portas (em qualquer ordem) são dos outros processos.

Essas portas são as do *server* dos processos. O IP havíamos colocado fixo 127.0.0.1 (*localhost*) no código, pois rodamos tudo na mesma máquina. Agora vamos simular máquinas distintas com o Docker! No momento de rodar, informaremos **IP:porta**. Precisamos então dos seguintes ajustes no início do código e na função `initConnections`:

```

//var myPort string          //porta do meu servidor //ANTIGO
var myAddress string        //endereço do meu servidor //NOVO

```

```

//myPort = os.Args[1] //ANTIGO
myAddress = os.Args[1] //NOVO

```

```

//ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+myPort) //ANTIGO
ServerAddr, err := net.ResolveUDPAddr("udp", myAddress) //NOVO

```

```

//ServerAddr, err := net.ResolveUDPAddr("udp", "127.0.0.1"+os.Args[2+s])
//ANTIGO
ServerAddr, err := net.ResolveUDPAddr("udp", os.Args[2+s]) //NOVO

```

Teste parcial

A ideia é saber que o seu código `Process.go` funciona normalmente.

Compile `Process.go` e rode assim:

- Terminal 1: Process 127.0.0.1:10002 127.0.0.1:10003 127.0.0.1:10004
- Terminal 2: Process 127.0.0.1:10003 127.0.0.1:10002 127.0.0.1:10004
- Terminal 3: Process 127.0.0.1:10004 127.0.0.1:10002 127.0.0.1:10003

Para usar o Docker, vamos **incluir os arquivos Dockerfile e docker-compose.yml** **correspondentes no diretório desta atividade** (onde está o `Process.go` devidamente modificado).

Sobre o arquivo Dockerfile

O arquivo `Dockerfile` é um meio para criar nossas próprias imagens, como uma receita do que queremos para compor o *container*.

Esse arquivo não possui extensão.

- Para criá-lo no Windows, basta usar um editor (ex: bloco de notas) e salvar como "Dockerfile" (usando aspas duplas mesmo para ficar sem extensão). Obs: Pode usar VSCode, se desejar.

```
FROM golang:1.23.0-alpine3.20
```

Importa um sistema operacional simplificado Alpine Linux com suporte a golang 1.23. Alpine é uma opção que usa pouca memória em disco. Outras imagens de golang em: https://hub.docker.com/_/golang

```
WORKDIR '/app'
```

Define o diretório base de operação inicial do container.

```
COPY ./Process.go .
```

Copia o código fonte da máquina local para o container.

```
RUN go build Process.go
```

Executa o build do código .go na máquina inicializada.

```
ENTRYPOINT [ "./Process" ]
```



Representa a criação do processo de entrada do container (para ele rodar como um executável).

Teste parcial

Diretório 'local' na sua máquina.

A ideia é ver que o seu código `Process.go` roda no Docker.
No diretório da atividade faça o seguinte:

- Crie a sua imagem (chamada `p1`): `docker build -t p1 .`
 - Entre no Docker Desktop (aba *Images*) para ver a imagem criada.

| <input checked="" type="checkbox"/> | Name | Tag | Status | Created | Size | Actions |
|-------------------------------------|------------------------------------|--------|------------------------|---------------|-----------|---|
| <input checked="" type="checkbox"/> | p1 37c78b24992b | latest | In use | 26 minutes ag | 358.05 MB |   |

Opção para dar um nome à imagem, no caso `p1`.

Com esse ponto aqui!
Obs: Esse processo pode demorar...

- Crie o *container* (oriundo dessa imagem): `docker run p1 127.0.0.1:10002 127.0.0.1:10003`
 - Como saída temos erro de conexão justamente porque o processo está tentando mandar mensagem para outro inexistente.

```
C:\golangExample\dica2_docker>docker run p1 127.0.0.1:10002 127.0.0.1:10003
Error: write udp 127.0.0.1:35835->127.0.0.1:10003: write: connection refused
Error: write udp 127.0.0.1:35835->127.0.0.1:10003: write: connection refused
Error: write udp 127.0.0.1:35835->127.0.0.1:10003: write: connection refused
```

- Entre no Docker Desktop (aba *Containers*) para ver o *container* criado.

| <input type="checkbox"/> | Name | Image | Status | CPU (%) | Port(s) |
|--------------------------|---|--------------------|------------|---------|---------|
| <input type="checkbox"/> | reverent_shaw 7c31c0f45ea2 | p1 | Exited (2) | | |

'Running' se estiver rodando.
'Exited' se for encerrado.

O nome do container foi gerado automaticamente.
Se desejar especificar um nome, usar:
`docker run --name container_p1 p1 127.0.0.1:10002 127.0.0.1:10003`

Precisamos do **Docker Compose** para trabalhar com mais de um processo!
Mas vamos continuar precisando o **Dockfile**.

Sobre o arquivo `docker-compose.yml`

Esse arquivo é forma a orquestrar a inicialização e execução simultânea de processos (no caso, três), com respectivos *logs* aparentes via interface com terminal.

```
services:
  first_process:
    container_name: p1
    build:
      context: .
    command: ["p1:10002", "p2:10003", "p3:10004"]
    stdin_open: true
    tty: true
  second_process:
    container_name: p2
    build:
      context: .
    command: ["p2:10003", "p1:10002", "p3:10004"]
    stdin_open: true
    tty: true
  third_process:
    container_name: p3
    build:
      context: .
    command: ["p3:10004", "p1:10002", "p2:10003"]
    stdin_open: true
    tty: true
```

Vai compor o nome da imagem

Nome do processo

Argumentos passados na chamada do processo no formato IP:porta do processo atual e dos amigos.

Essa linha e a próxima servem para habilitar interação via teclado. Isso é útil apenas com os demais códigos Process.go (quando interagimos via teclado). Pode até comentar (usando #) essas linhas.

Teste final

Para compor o nome da imagem, usou o nome da pasta, no caso “dica2_docker”.

No diretório da atividade faça o seguinte:

















- Crie as imagens dos processos: `docker compose build`
 - Entre no Docker Desktop (aba *Images*) para ver as imagens criadas.
 - Atenção: O *build* é necessário sempre que quisermos alterar algo na imagem, por exemplo, se mudarmos o `.go`.

| <input type="checkbox"/> | Name | Tag | Status | Created | Size | Actions |
|--------------------------|---|--------|------------------------|------------|-----------|---|
| <input type="checkbox"/> | dica2_docker-second_process 30944c0cae1e | latest | In use | 6 days ago | 358.05 MB | ▶ ⋮ 🗑 |
| <input type="checkbox"/> | dica2_docker-third_process d48d0ea529bd | latest | In use | 6 days ago | 358.05 MB | ▶ ⋮ 🗑 |
| <input type="checkbox"/> | dica2_docker-first_process f3fd3cc6dc10 | latest | In use | 6 days ago | 358.05 MB | ▶ ⋮ 🗑 |

- Crie os respectivos *containers*: `docker compose up`
 - Já é possível ver a saída de cada processo!

```
C:\golangExample\dica2_docker>docker compose up
[+] Running 4/4
 ✓ Network dica2_docker_default Created
 ✓ Container p1 Created
 ✓ Container p2 Created
 ✓ Container p3 Created
Attaching to p1, p2, p3
p3 | Received 0 from 172.18.0.3:49451
p3 | Received 0 from 172.18.0.4:39026
p1 | Received 0 from 172.18.0.2:37634
p2 | Received 0 from 172.18.0.2:56346
p1 | Received 0 from 172.18.0.4:49424
p2 | Received 0 from 172.18.0.3:44310
p3 | Received 1 from 172.18.0.3:49451
p2 | Received 1 from 172.18.0.3:44310
```

- Entre no Docker Desktop (aba *Containers*) para ver os *containers* criados.

| <input type="checkbox"/> | Name | Image | Status | CPU (%) | Port(s) | Last | Actions |
|--------------------------|--|----------------------------|---------------|---------|---------|-------|---|
| <input type="checkbox"/> |  dica2_docker | | Running (3/3) | N/A | | 0 sec |    |
| <input type="checkbox"/> |  p1 3974b608a767 | dica2_dock | Running | N/A | | 0 sec |    |
| <input type="checkbox"/> |  p3 21a41aa8b983 | dica2_dock | Running | N/A | | 0 sec |    |
| <input type="checkbox"/> |  p2 4caed6b2f4f1 | dica2_dock | Running | N/A | | 0 sec |    |

- Para interagir com um processo separadamente (ex: p1), abra um novo terminal e digite: `docker attach p1`

```
PS C:\golangExample\dica2_docker> docker attach p1
Received 48 from 172.18.0.4:60352
Received 48 from 172.18.0.3:50537
Received 49 from 172.18.0.3:50537
Received 49 from 172.18.0.4:60352
Received 50 from 172.18.0.4:60352
```

***** Extra: Lab1 com Docker *****

Que tal usar Docker para rodar o seu lab1?
Veja como seria...

1. Criar um arquivo Dockerfile para Process.go.
 - Sugestão de nome “Dockerfile.Process”

```
FROM golang:1.23.0-alpine3.20

WORKDIR '/app'

COPY ./Process.go .

RUN go build Process.go

ENTRYPOINT [ "./Process" ]
```

2. Criar um arquivo Dockerfile para SharedResource.go.
 - Sugestão de nome: “Dockerfile.SharedResource”

```
FROM golang:1.23.0-alpine3.20

WORKDIR '/app'

COPY ./SharedResource.go .

RUN go build SharedResource.go

CMD [ "./SharedResource" ]
```

3. Criar o arquivo docker-compose.yml

```
services:
  shared_resource:
    container_name: shared
    build:
      context: .
      dockerfile: "Dockerfile.SharedResource"
    stdin_open: true
    tty: true
  first_process:
    container_name: p1
    build:
      context: .
      dockerfile: "Dockerfile.Process"
    command: ["1", "p1:10004", "p2:10003", "p3:10002"]
    depends_on:
      - shared_resource
    stdin_open: true
    tty: true
  second_process:
    container_name: p2
    build:
      context: .
      dockerfile: "Dockerfile.Process"
    command: ["2", "p1:10004", "p2:10003", "p3:10002"]
    depends_on:
      - shared_resource
    stdin_open: true
    tty: true
  third_process:
    container_name: p3
    build:
      context: .
      dockerfile: "Dockerfile.Process"
    command: ["3", "p1:10004", "p2:10003", "p3:10002"]
    depends_on:
      - shared_resource
    stdin_open: true
    tty: true
```

SharedResource tem o seu Dockerfile!

Process tem o seu Dockerfile!

Lembrar que aqui devemos usar os processos sempre em ordem, pois temos o *id* do processo corrente.

Isso faz com que SharedResource suba antes de p1, pois há dependência.

Bom estudo!