THE UNIVERSITY OF CHICAGO

MPCS: 52040 Distributed Systems Class 2

Kyle Chard

chard@uchicago.edu

Docker 101

- Download Docker now:
    - https://www.docker.com/products/docker-desktop

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization	Project description 4 REST	**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft	Homework 4 due (Project released)
February 21	8 Consistency and replication	5 FaaS	
February 28	9 Distributed data	6 Distributed data	
March 7	10 Data-intensive computing		**Project due (March 9)** **Final exam (March 14)**

Textbook Chapter 2

Introduction

We established last class that distributed systems are complicated:

- Nodes are distributed and independent
- Nodes may fail (and its difficult to know that they failed)
- There is no global clock and therefore coordination is difficult

How should we go about organizing/managing a distributed system?

- Software architecture: define how software components are organized and how they should interact

Distributed Architecture Styles

1. Layered architectures
2. Object- or service-based architectures
3. Resource-centered architectures
4. Event-based architectures

* In practice, distributed systems combine many architectures

1) Layered architectures

Components are organized into layers and higher layers make calls to lower layers

- Responses are returned from lower to higher levels

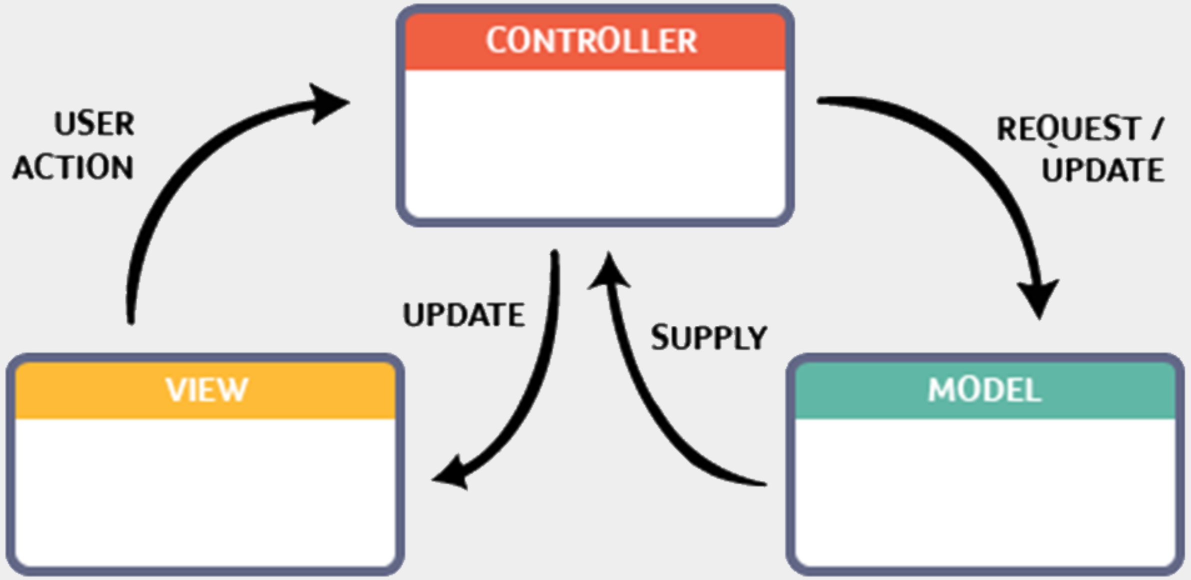
Examples:

- Most applications you use are implemented in layers
  - Python program -> Python libraries -> OS calls
  - Android app -> Android services -> OS calls
- Network communication architectures

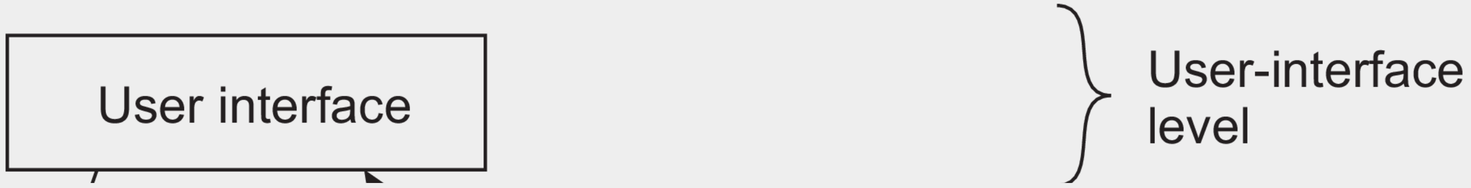
Common layering approach

Many approaches group into common layers:

- Application interface layer: the user interface
- Processing layer: logic and function of the application
- Data layer: state manipulated by application components



Example: search engine



2) Object-based/service-based architectures

What are the advantages of object-oriented programming?

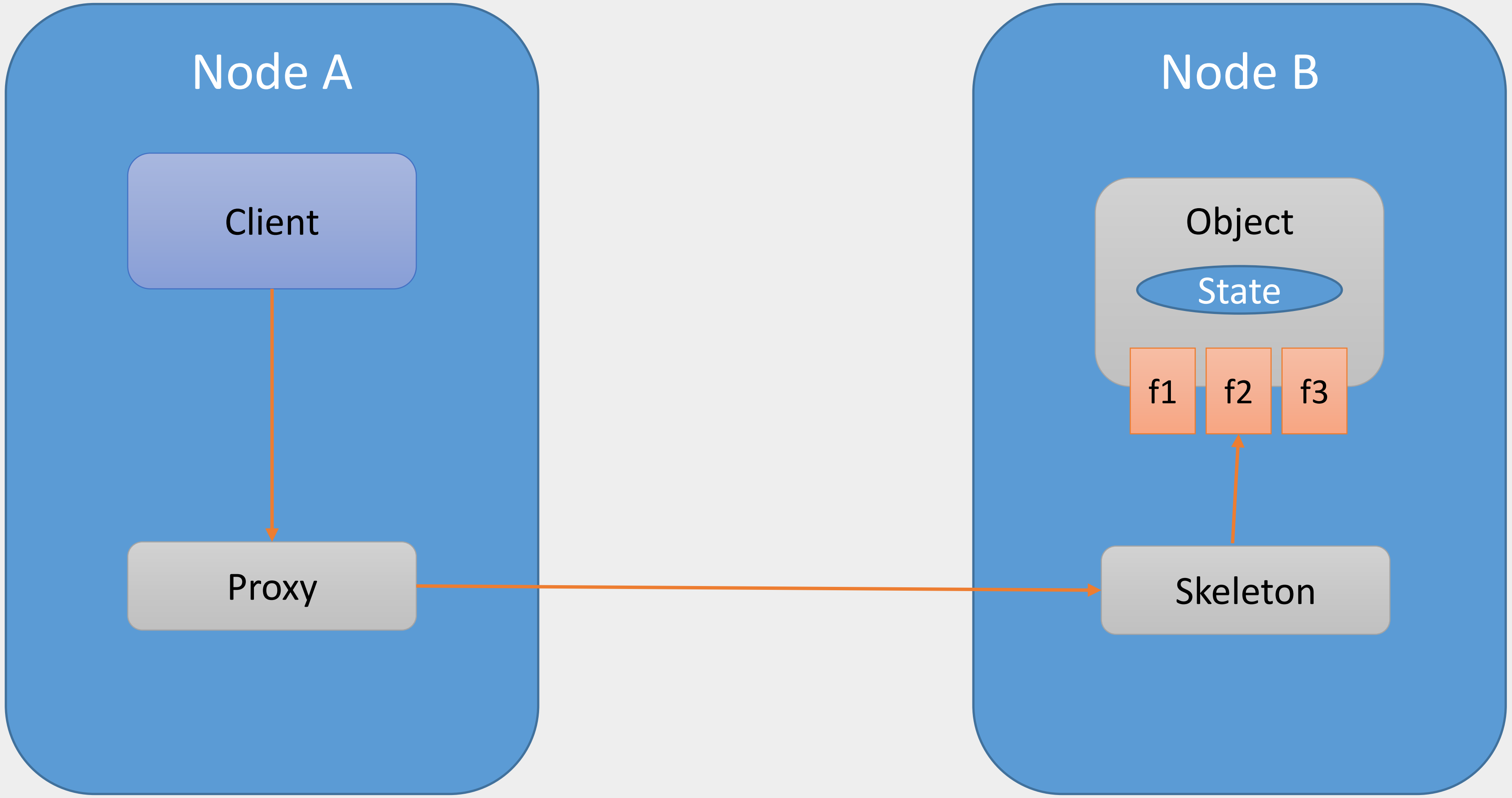
- Modularity, reuse, extensibility, abstraction, encapsulation, polymorphism
- Faster development, easier maintenance, etc.

Object-based architectures:

- Components are objects, connected to each other through procedure calls
- Objects may be placed on different nodes; calls can execute across a network

Objects encapsulate data (object state) and offer well defined methods on that data (without revealing internal implementation)

Example: distributed objects



Example: service oriented architecture (SOA)

The object-based model is similar to a service-based model

- The service is realized as a self-contained entity, that can make use of other services

The distributed application is composed of many different services

- Not necessarily within the same administrative domain (e.g., restaurant review service might rely on a map service)

3) Resource-based architectures

Initial web APIs (e.g., using SOAP) focused on developing rich APIs with a wide range of methods

- Complicated, difficult to understand the interface, hard to version etc.

A simple way of designing a system is to think about *resources* and provide a fixed set of interfaces for manipulating those entities

Representational State Transfer (REST)

Representational State Transfer

- A distributed system as a collection of resources, individually managed by components.
- Resources may be added, removed, retrieved, and modified by (remote) applications

Characteristics:

1. Resources are identified through a single naming scheme
2. All services offer the same interface
3. Messages sent to or from a service are fully self-described
4. After executing an operation at a service, that component forgets everything about the caller

REST replies on HTTP operations

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource
DELETE	Delete a resource
POST	Modify a resource by transferring new state

Example: AWS S3

Simple Storage Service (S3) provides object storage

- Two resources
  - Objects – bundle of bytes representing something
  - Buckets – namespace for organizing objects

Objects are referred to using a URI

- http://BucketName.s3.amazonaws.com/ObjectName

Simple operations to create (PUT), update (PUT/POST), delete (DELETE), and retrieve (GET) an object

4) Event-based systems

As systems scale (more nodes join and leave) its important to decouple dependencies between processes

- Separation between *processing* and *coordination*

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Direct and mailbox

Direct

- Referential coupling appears in form of explicit reference in communication
  - A process can communicate if it knows the name of the other process
- Temporal couple occurs as both processes are online at the same time

Mailbox

- Temporal decoupling – processes are not executing at the same time
- Communication takes place by putting messages in a known mailbox, other processes can then discover messages in the mailbox

Event-based and data spaces

Event-based

- Processes do not know each other explicitly, instead they publish a notification describing the occurrence of an event
- Other processes may choose to subscribe to specific type of event, in which case they will receive notifications about that event

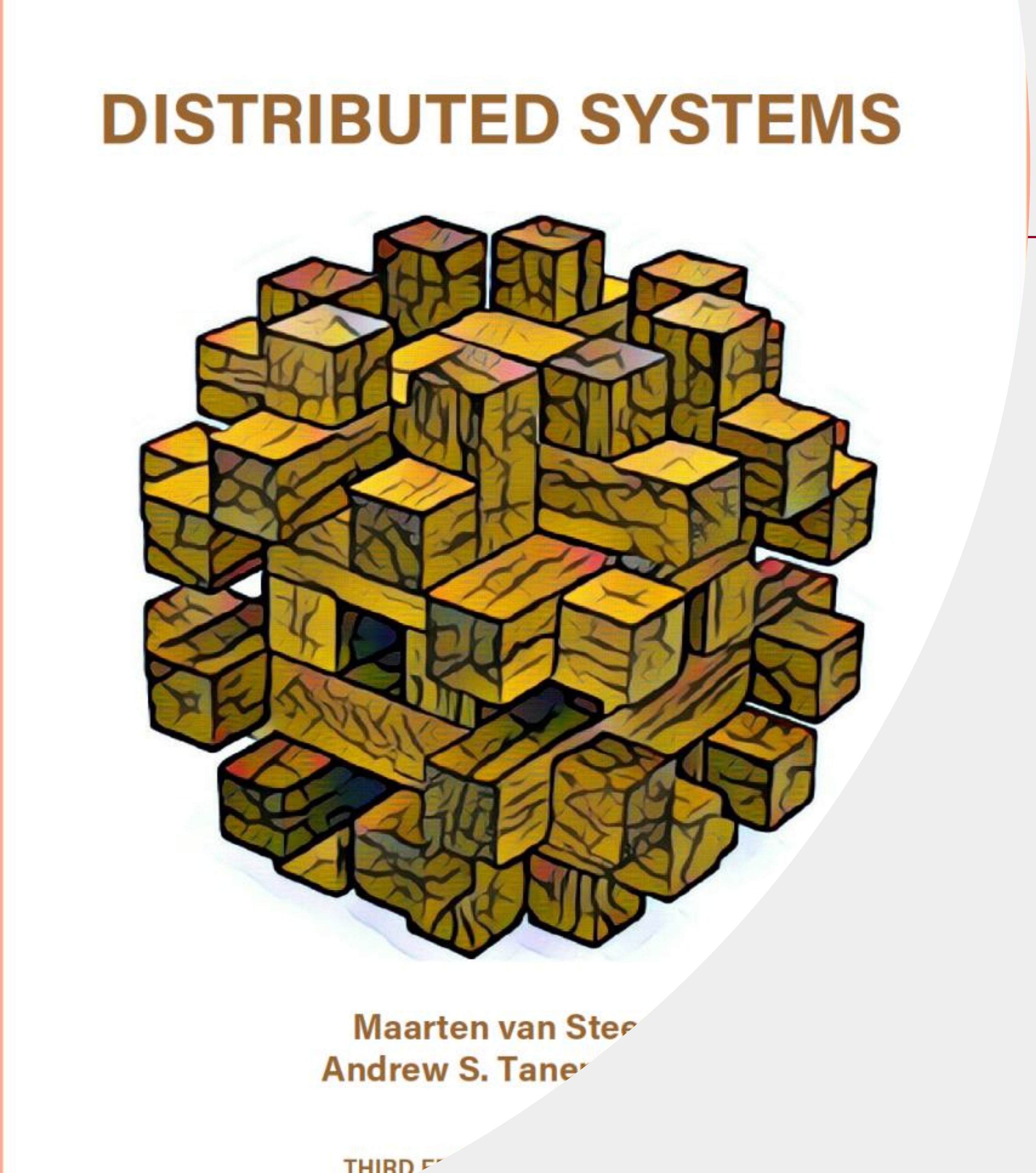
Data space

- Shared data space where processes communicate via tuples
  - Tuples: data records with a number of fields
- Process puts tuple in tuple space, another process can search using patterns
- Example Linda

Summary

1. Layered architectures
2. Object- or service-based architectures
3. Resource-centered architectures
4. Event-based architectures

* In practice, distributed systems combine many architectures



Textbook Chapter 3

Outline

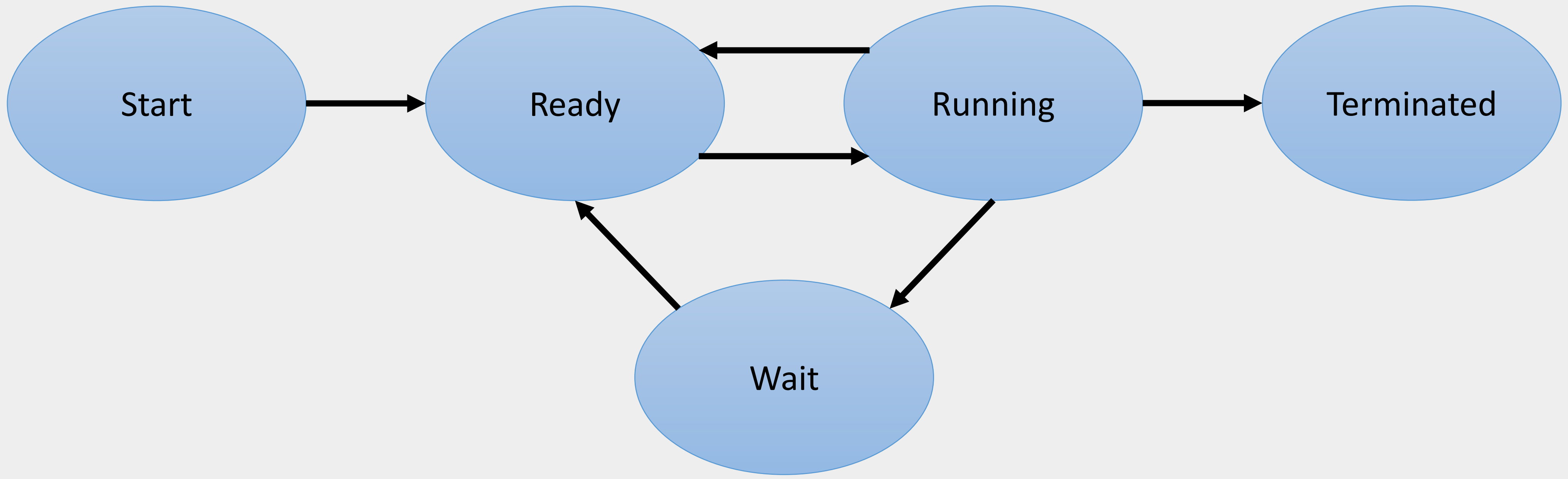
- Part 1: Processes and threads
- Part 2: Virtualization
- Part 3: Client-server organization

Part 1: Processes and threads

What is a process?

Process: a program in execution

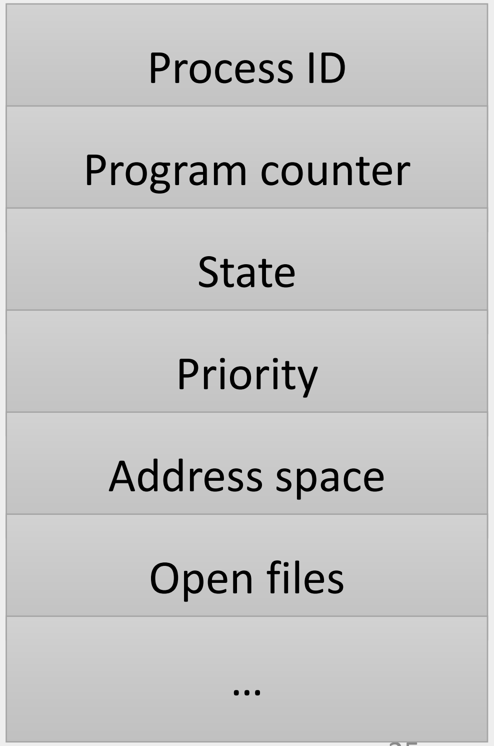
(an instance of a computer program that is being executed)



What is in a process

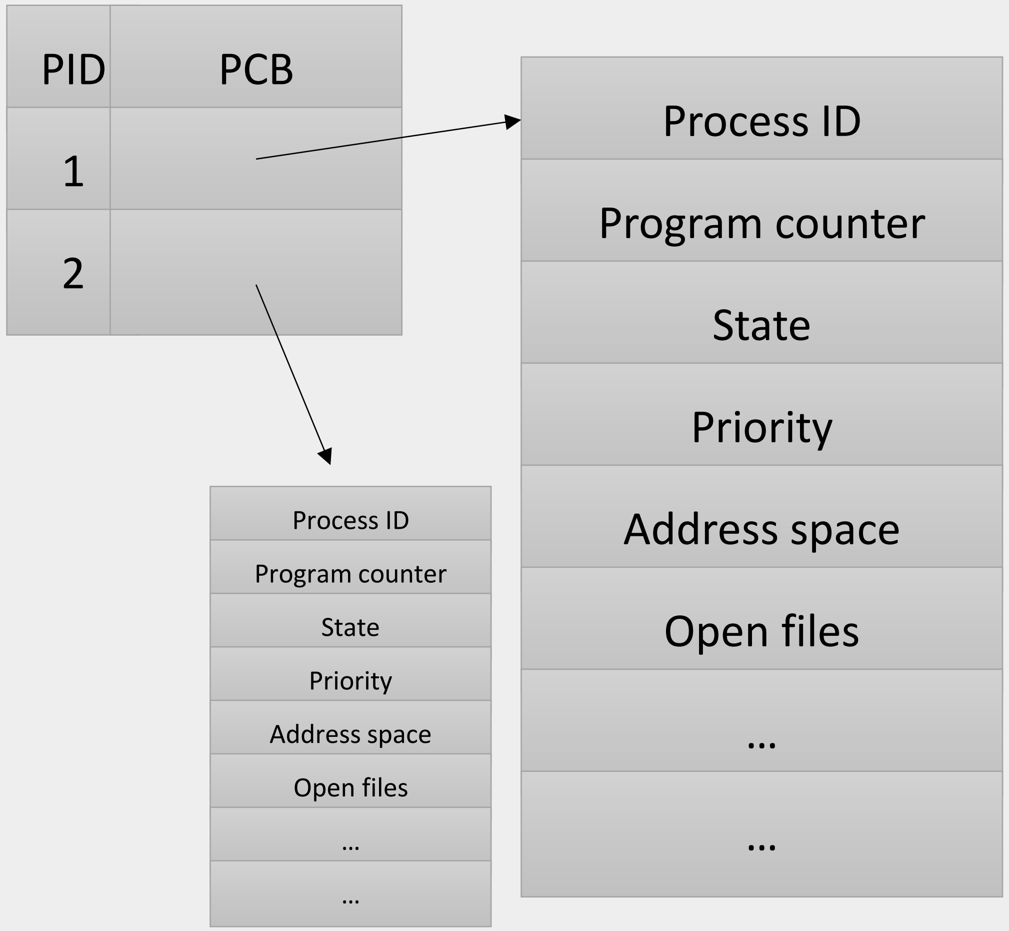
- Machine code that is to be run (i.e., code and ultimately instructions)
- Memory allocation (data, call stack, heap)
- Resources (e.g., files, devices)
- Processor context (e.g., registers, memory addresses)

- This is stored in a **process control block (PCB)**



Let’s dig a little deeper: how does an OS work?

- Virtual Processor:
  - To execute a process the OS creates a virtual processor (each for running a different program)
  - For each program, OS maintains a **process table** to store CPU register values, memory maps, open files, etc.
    - These entries are called a *process context*
    - Changing processes is called *context switching*



Why do we need multiple processes?

- Whenever a blocking system call is executed the entire process blocks
  - E.g., think about all the things apps are doing concurrently

- Multiprocessor/multicore machines allow independent functions to be executed concurrently
  - If functions can be run in parallel then overall execution is faster

- Large application design: collection of cooperating programs that communicate with one another

Processes Summary

- A program in execution (i.e., on an OS virtual processor)

+ve: OS ensures that processes are independent (H/W support)

-ve: Creating a process == OS must create a new address space

.. when **context switching** between processes we must save and reload context

- Processes are heavyweight, but provide strong isolation
  - We need something lighter…

Threads

- What is a Thread?
  - Piece of code executing independently (of other threads)
    - 1) Executes its own piece of code (like a process)
    - 2) Independent of other threads
  - However, there is little focus on concurrency transparency
    - That is, it maintains only minimal context that allows the CPU to be shared
    - Thread context => nothing more than processor context (+ minimal info for thread management)
    - Isolation is left entirely to developers (e.g., threads can access shared data in the process)

    => Result is better performance (fast to create, fast to context switch)

Context switching

- Context switching: storing state of process/thread so it can be later restored and execution resumed from that state

- Processes => switching involves the OS (e.g., trapping to the kernel)
- Threads => share address space so context switching can be independent of the operating system

⇒Creating/destroying threads is cheaper than processes

  ⇒Tradeoff **performance** vs **isolation**

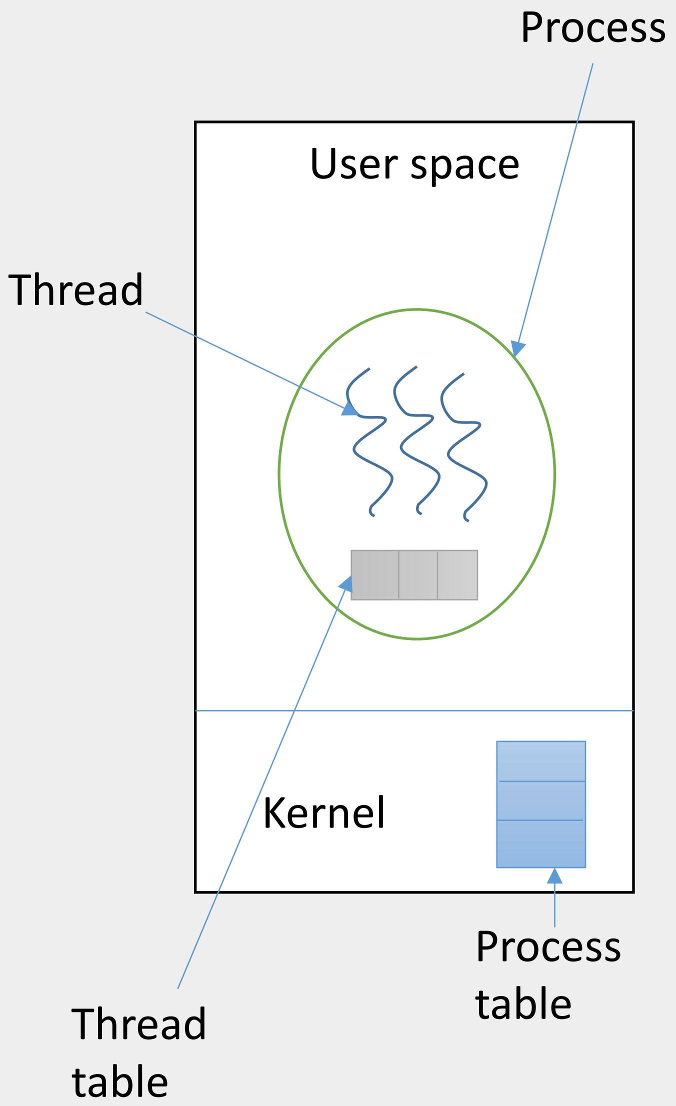
How are threads implemented?

- Thread package
  - Operations to create and destroy threads
  - Synchronization variables (e.g., mutexes, locks, barriers)

- The big question: should the OS implement threads? Or should they be implemented at the user level?

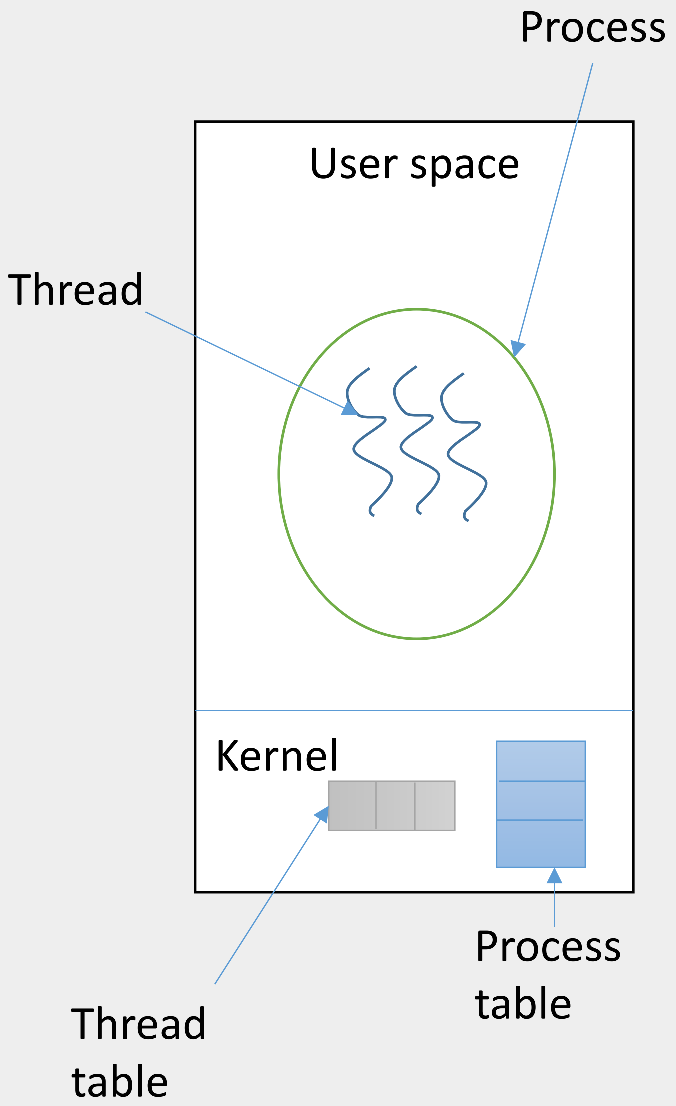
Threads in user space

- Many-to-one model in which many threads are mapped to one kernel process
- Advantages:
  - Cheap: all administration is kept in user address space (cost is basically the cost of allocating memory for the thread's stack)
  - Efficient: all operations handled within a single process, all services provided by the kernel are done on behalf of that process
  - Context switching: can be done in a few instructions (only the values of the CPU registers need to be stored/reloaded)
- Disadvantages:
  - Threads are mapped to a single schedulable entity (at the OS level).. Invocation of a blocking system call will immediately block the entire process to which the thread belongs (and therefore all other threads)



Threads in kernel space

- The kernel implements the thread package so that all operations return as system calls
- One-to-one model: every thread is a schedulable entity
- Operations that block a thread are no longer a problem: the kernel schedules another available thread within the same process
- Handling external events is simple: the kernel (which catches all events) schedules the thread associated with the event.
- Disadvantage: each thread operation (create, delete, sync) has to be carried out by the kernel (requires a system call). Switching contexts might now be as expensive as switching process context.



Threads in distributed systems

- We now understand how threads are used in the OS, how does this apply to distributed systems?
- Multi-threaded clients
- Multi-threaded servers

Multi-threaded clients

- Hiding network latencies:
  - E.g., web browser scans HTML page to find elements that need to be fetched
  - Each file is fetched by a separate thread, each doing a (blocking) HTTP request
  - As files come in, the browser displays them
- Multiple request-response calls to other machines (RPC)
  - A client does several calls at the same time, each one by a different thread
  - It then waits until all results have been returned
  - Note: if calls are to different servers, we may have a linear speed-up

Multi-threaded servers

- Improve performance
  - Having a single-threaded server prohibits simple scale-up to a multiprocessor system
  - As with clients: hide network latency by reacting to next request while previous one is being replied
- Better structure
  - Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure.
  - Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.

Outline

- Part 1: Processes and threads
- **Part 2: Virtualization**
- Part 3: Client-server organization

Virtualization

- Virtualization enables you to run a *virtual* computer system using an abstraction layer above the hardware
  - Extend/replace an existing interface to mimic the behavior of another system

- Virtualization is important for many reasons:
  - Hardware changes faster than software
  - Ease of portability and code migration
  - Isolation of failing or attacked components

Virtualization in distributed systems

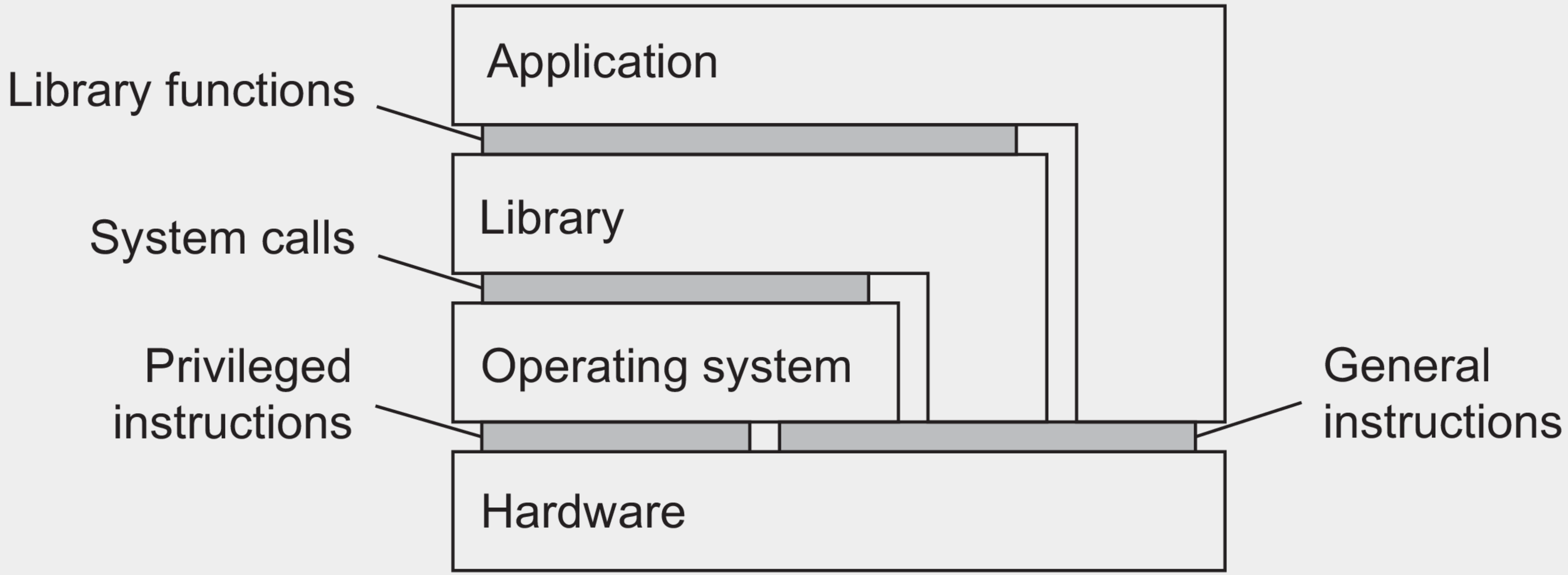
Initially (1970s) method to allow legacy software to run on mainframe hardware

Increasingly common in the 1990s as a method of providing stable interface for slow changing software (provides legacy interfaces to new platforms)

In the 2000s used as the basis for providing elastic computing capacity in the cloud and managing large numbers of heterogeneous servers

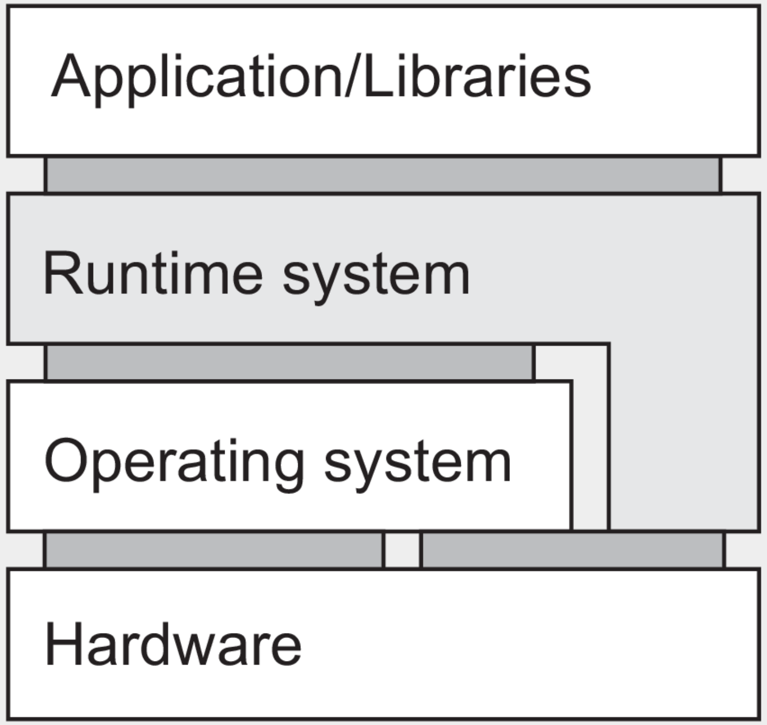
Now useful as a means of porting applications

Virtualization can occur at many levels

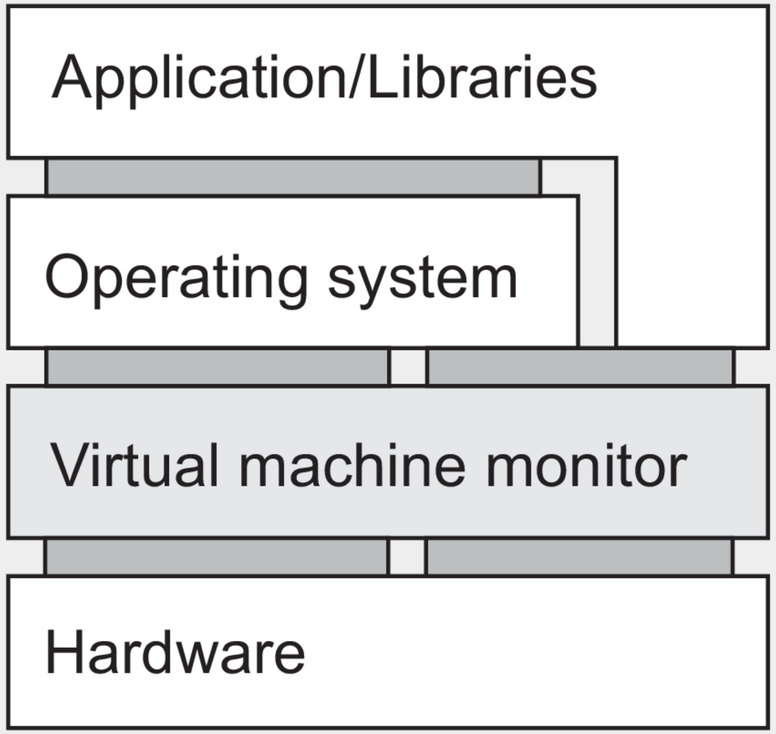


Virtualization can occur at many levels

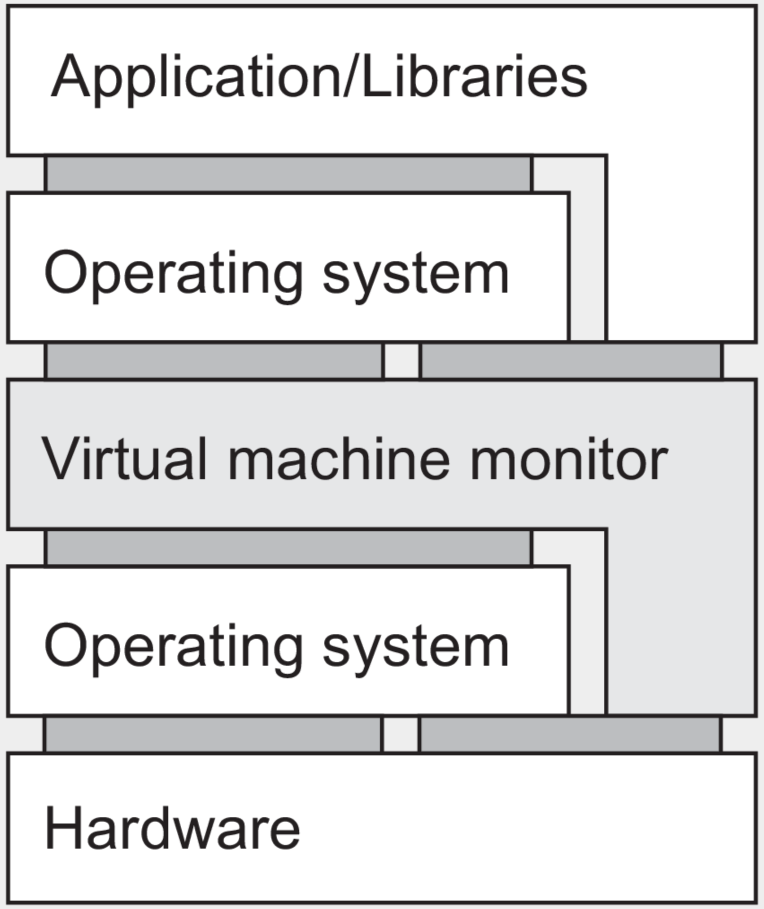
Process virtual machine



Native virtual machine



Hosted virtual machine



Virtual machines

Emulator of a computer system

- "efficient, isolated duplicate of a real computing system"

Allow users to run multiple operating systems (originally to support time-sharing for single task OS)

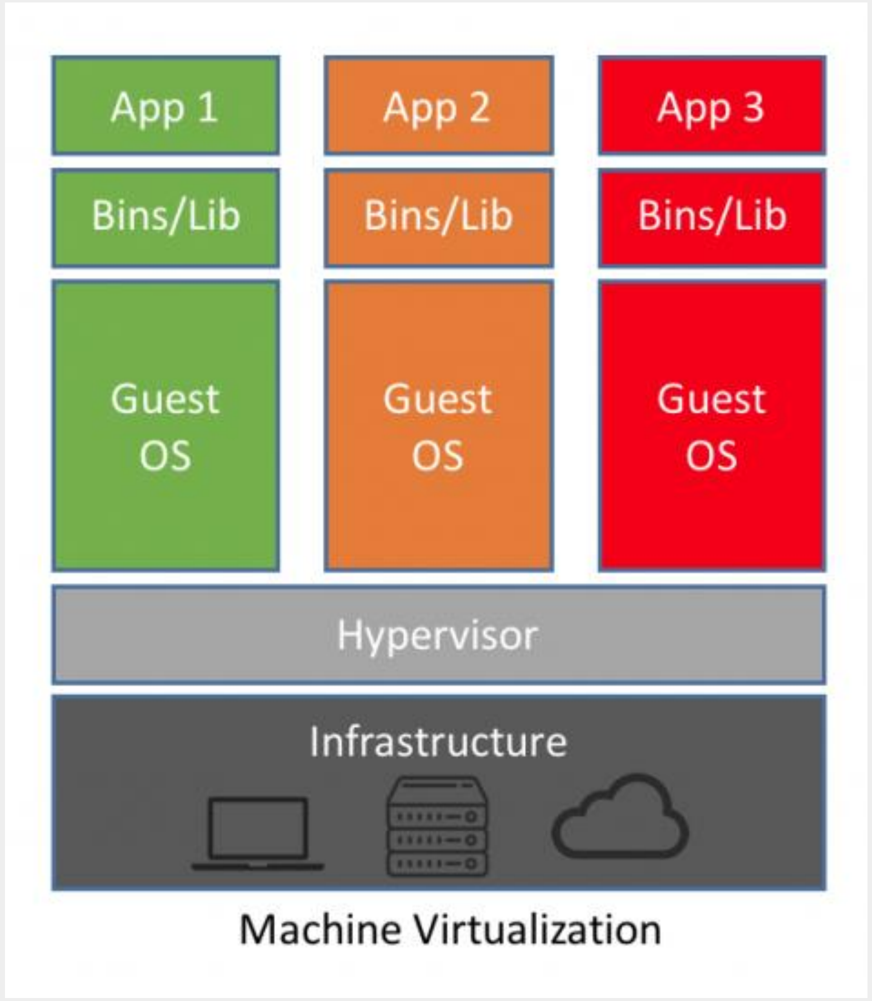
Hypervisor: a program used to run and manage one or more virtual machines on a computer

- Can reallocate resources between VMs

Hardware-assisted virtual machines provide architectural support for VM monitor and allow guest OSes to be run in isolation

- Mid 2000s Intel/AMD implemented hardware support

Examples: VMWare, VirtualBox, KVM, Oracle VM server



Containers

Originally developed to segregate namespaces in Linux OS for security

Linux environments included partitions (jails) which questionable applications could be executed safely (without risk to the kernel)

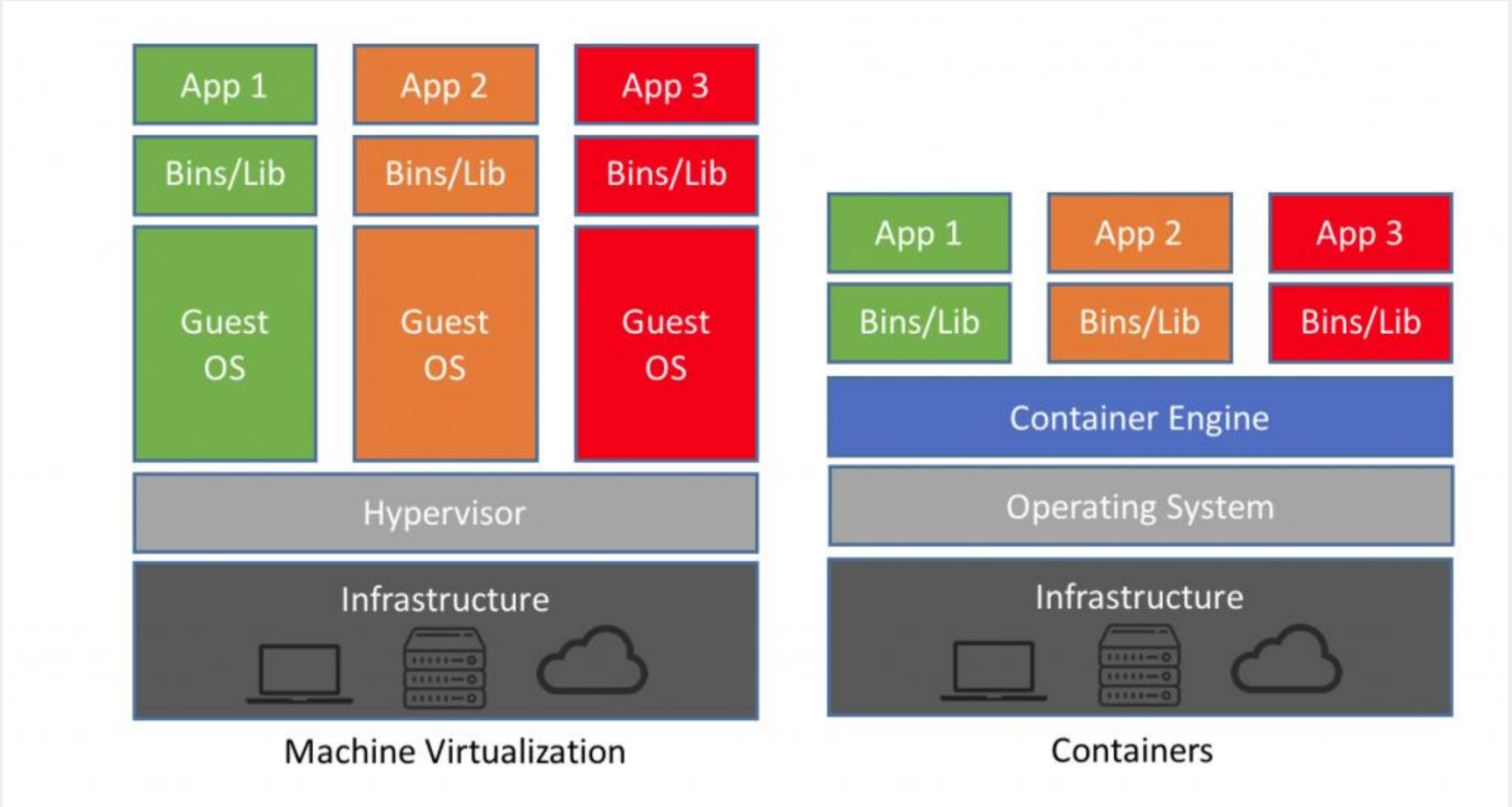
- Kernel responsible for execution via an abstraction layer between kernel and workload

First example LXC (Linux Containers)

- Docker, Singularity, Shifter

Much lower overhead than traditional VMs

Virtual machines vs containers



Docker

- Enterprise containers (2013)

- Developed for Linux
  - Cgroups to limit resource usage (CPU/Mem)
  - Kernel namespaces limit app view of the operating environment (process trees, network, file systems)
  - Union-capable file system (isolated file system)

- Now available in Windows and supported by cloud providers

Virtualization in cloud computing

Virtualization was (arguably) the single-most important technology for realizing cloud computing

Infrastructure as a Service

- Cloud providers rent out VM
- May share physical machine with others
- Allows for isolation, resource sizing, etc.

Container services (e.g., Elastic Container Service)

- Provide interfaces for running containers

Function as a service?

Summary

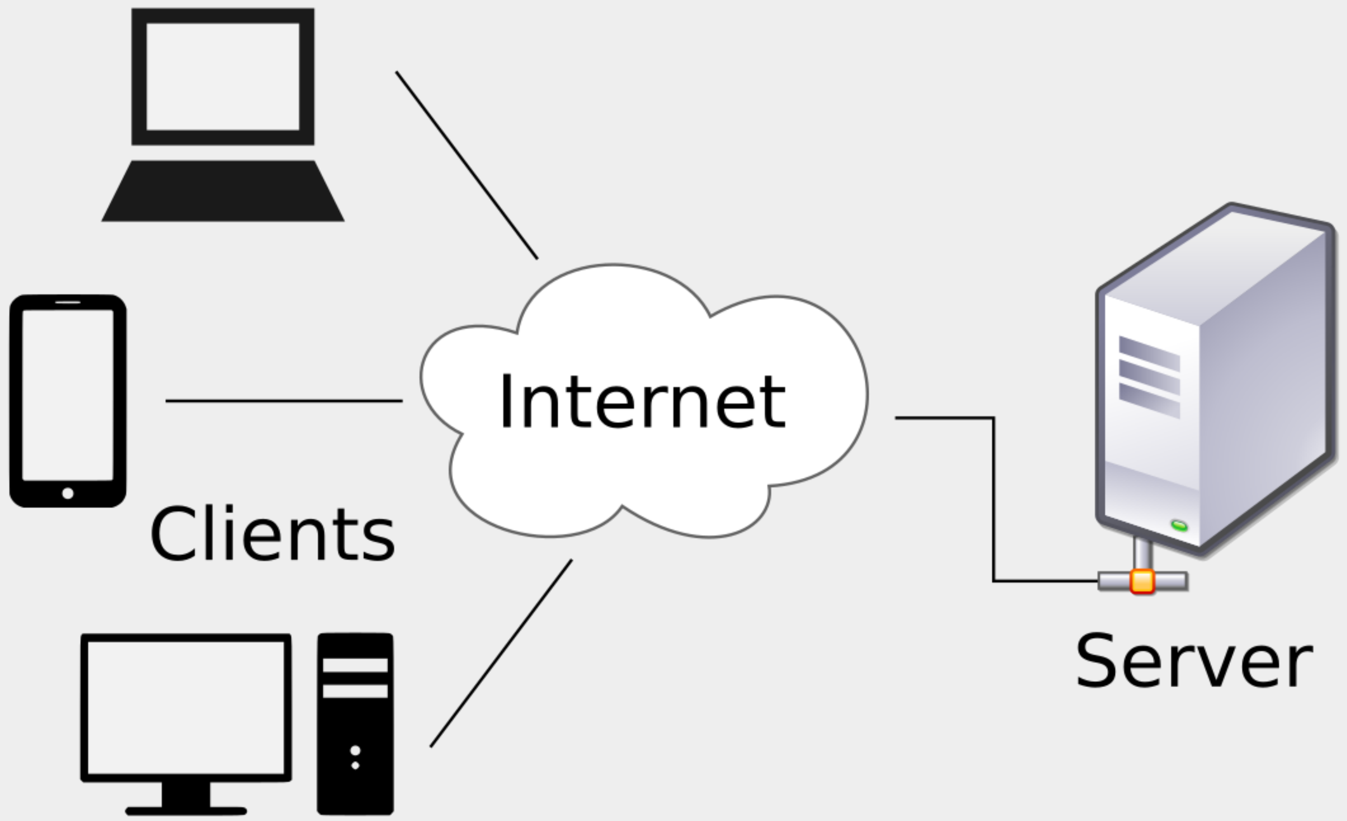
- Virtualization enables you to run a *virtual* computer system using an abstraction layer above the hardware
- Virtual machines (supported by hardware) allow for multiple operating systems to be run on a single machine
- Containers provide lightweight virtualization on top of an operating system
  - Enables application portability

Part 3: Client-server organization

Outline

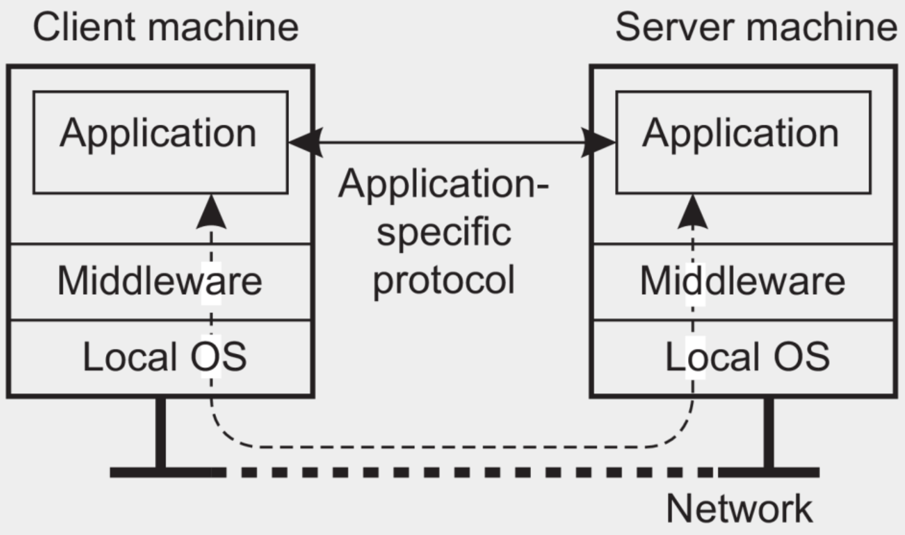
- Part 1: Processes and threads
- Part 2: Virtualization
- **Part 3: Client-server organization**

Client-server model

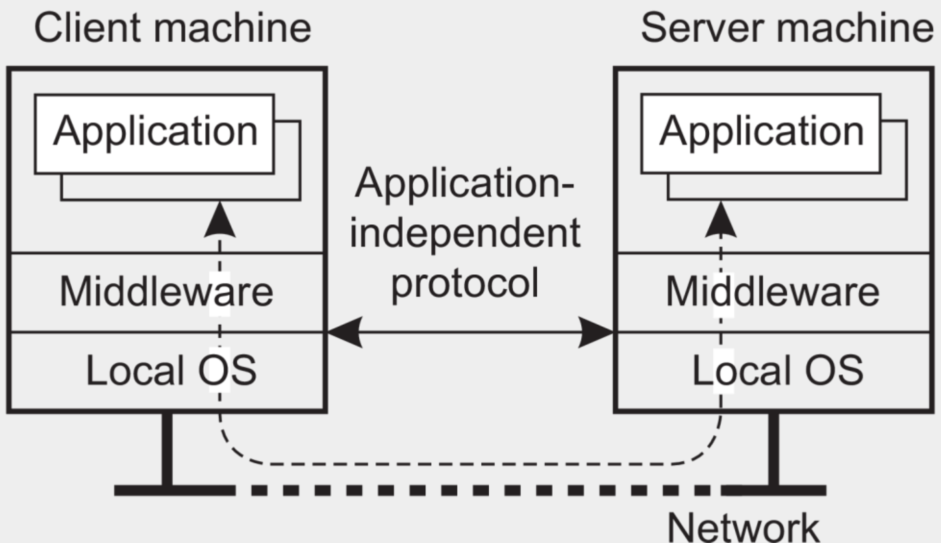


Client-server organization: clients

- Thick client
  - Remote server has a separate client that contacts the server over the network



- Thin client
  - Direct access to remote services with a single user interface (client is effectively a terminal with no local storage)



Client-side distribution transparency

Often clients maintain state and perform processing (primarily as a means of either distributing load or abstracting distribution)

- E.g., embedded client software (e.g., ATMs, TV boxes, etc.) much processing is on the client side

How does a client provide transparency?

1. Access transparency: client stubs for RPC
2. Location/migration transparency: client tracks actual location
3. Replication transparency: E.g., multiple servers, client will handle replicated invocations to each server
4. Failure transparency: client must support if masking server/communication failures

Client-server organization: servers

What is a server:

- “A process implementing a specific service on behalf of a collection of clients. It waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.”

- General design issues:
  1. Concurrent vs iterative
  2. How to contact a server
  3. Interrupting a server
  4. Stateless vs stateful

1. Concurrency

Iterative server: Server handles the request before attending a next request.

Concurrent server: Uses a dispatcher, which picks up an incoming request that is then passed on to a separate thread/process.

2. Contacting a server

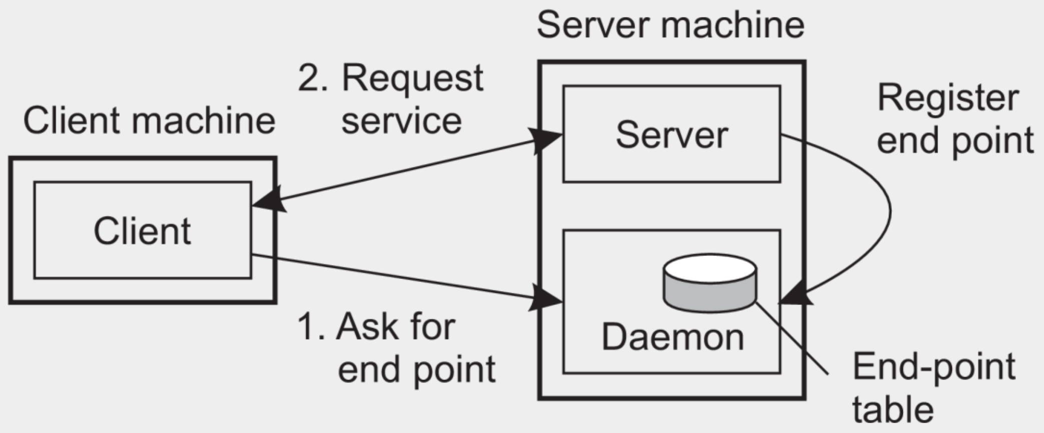
Server listens on a specific port

- How do clients know this port?

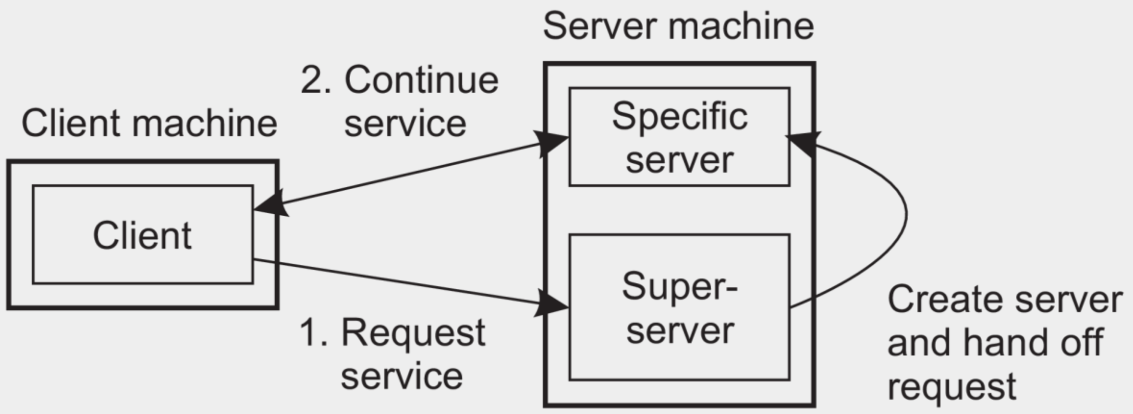
1. Some common services have well-known ports
   - FTP == TCP 21
   - HTTP == TCP 80
   - HTTPS == TCP 443

2. Otherwise, clients have to discover the port somehow

Discovering a server



- Daemon process used to register servers and clients must first ask for the port



- Superserver manages creation of server on port within predefined range

3. Interrupting a server

Is it possible to interrupt a server once it has accepted (or is in the process of accepting) a service request?

1. Use a separate port for urgent data
   - Server has a separate thread/process for urgent messages
   - Urgent message comes in -> associated request is put on hold
   - Note: requires OS supports priority-based scheduling
2. Use facilities of the transport layer
   - Example: TCP allows for urgent messages in same connection
   - Urgent messages can be caught using OS signaling techniques

4. State

Stateless server

- Does not keep information on state of client
- Can change state without informing clients
- In practice, most servers do store some state on clients, but if that state is lost it won’t disrupt the service

Advantages

- Independence: clients/servers are independent
- Inconsistency: if client/server crashes there is no need to sync

Disadvantage

- Performance: server cannot anticipate client behavior

Stateful servers

Server maintains persistent information on its clients

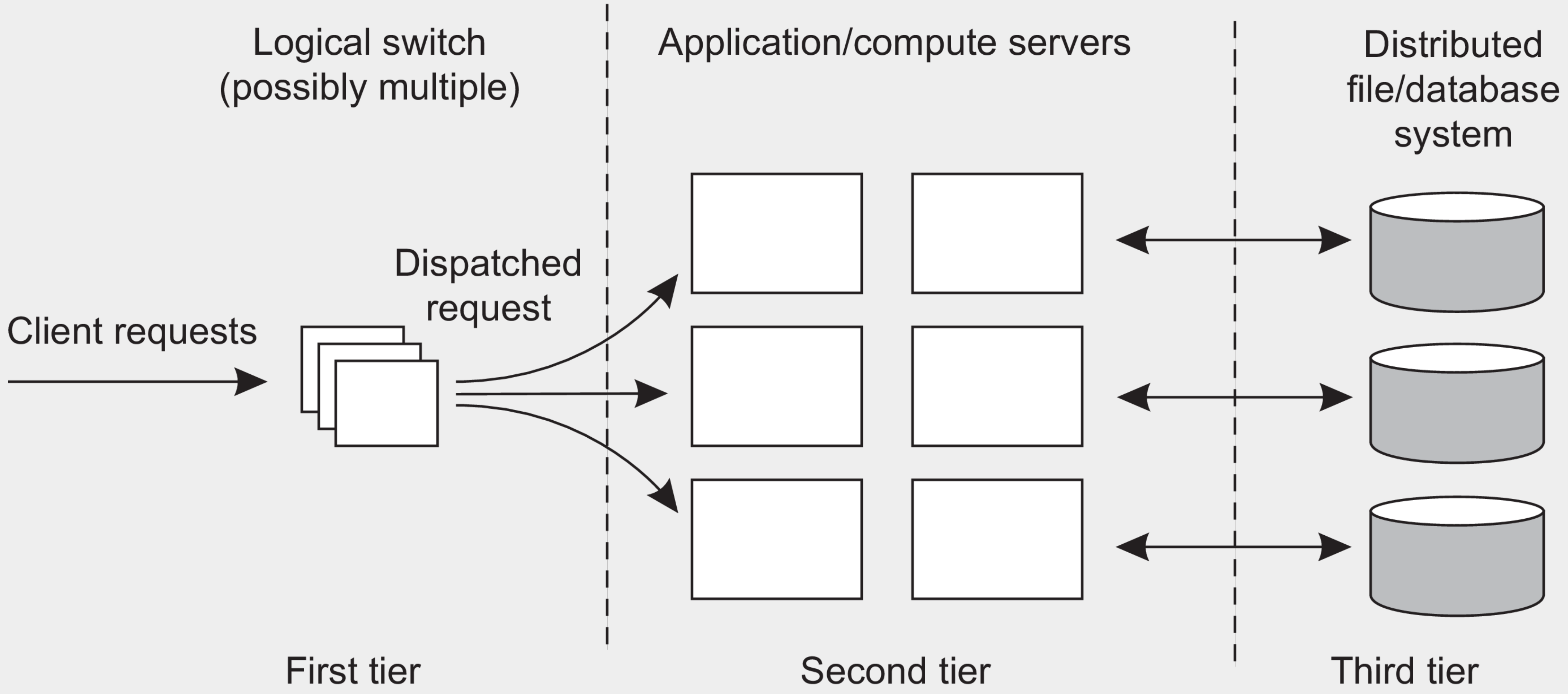
- Information must be explicitly deleted by the server

Example: file server

- Record that a file has been opened, so that prefetching can be done
- Knows which data a client has cached, and allows clients to keep local copies of shared data

Advantage: performance (at the expense of reliability)

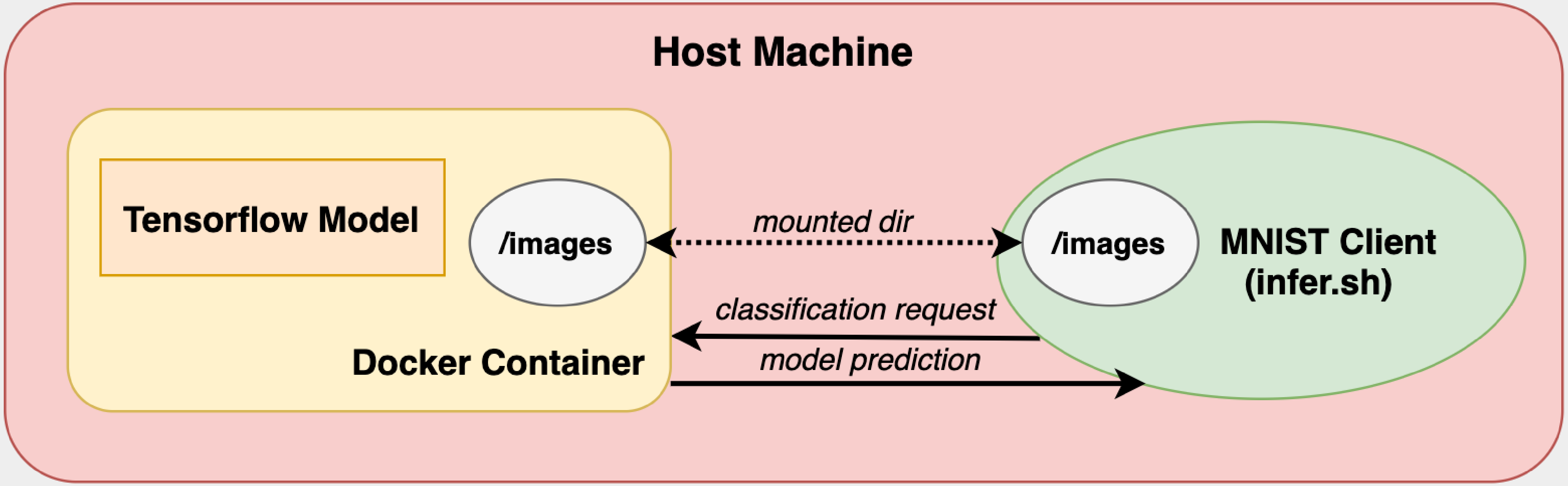
Client-server organization: server clusters



Questions?

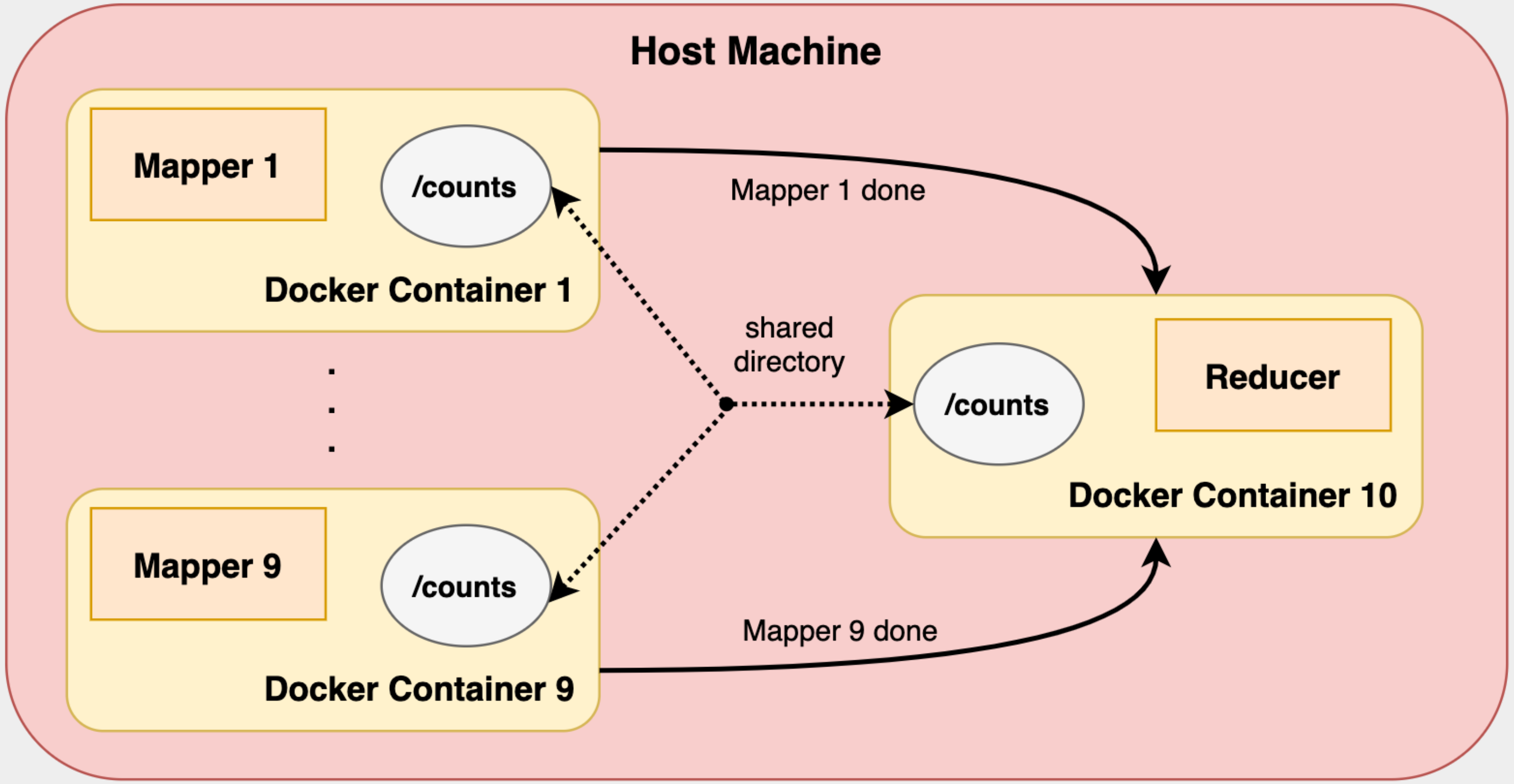
Homework 2

- Part 1
  - Deploy a classifier model in a docker container and use it to classify images



Homework 2

- Part 2
  - Develop a map-reduce computation to count unique words in a set of documents
  - Run several containers in parallel to count words



Class Exercise 1: Docker

- Please submit to GitHub (link on the final slide)

What is a container

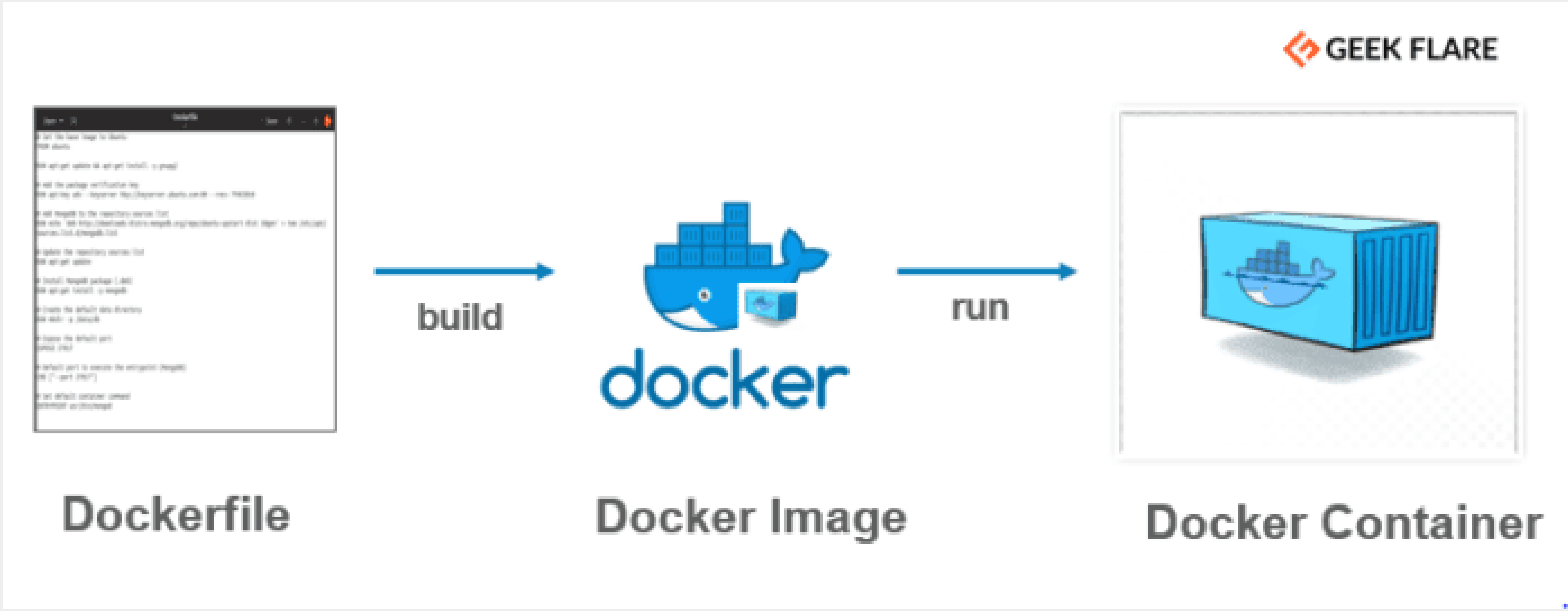
- A package of code and dependencies that can be deployed and executed in different computing environments
- Provides lightweight virtualization and portability between operating systems
- No requirement to move around the entire OS-application stack, instead just the layers on top of the operating system
- Containers are lightweight, contain only applications and libraries, so they are typically small and many can be deployed on the same machine

Docker

Docker is a platform for developers and sysadmins to ***build, run, and share*** *applications with containers. The use of containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is.*

-- Docker website

Dockerfiles, images, and containers



```
[illegible]
```

Docker 101

- Download Docker:
  - https://www.docker.com/products/docker-desktop

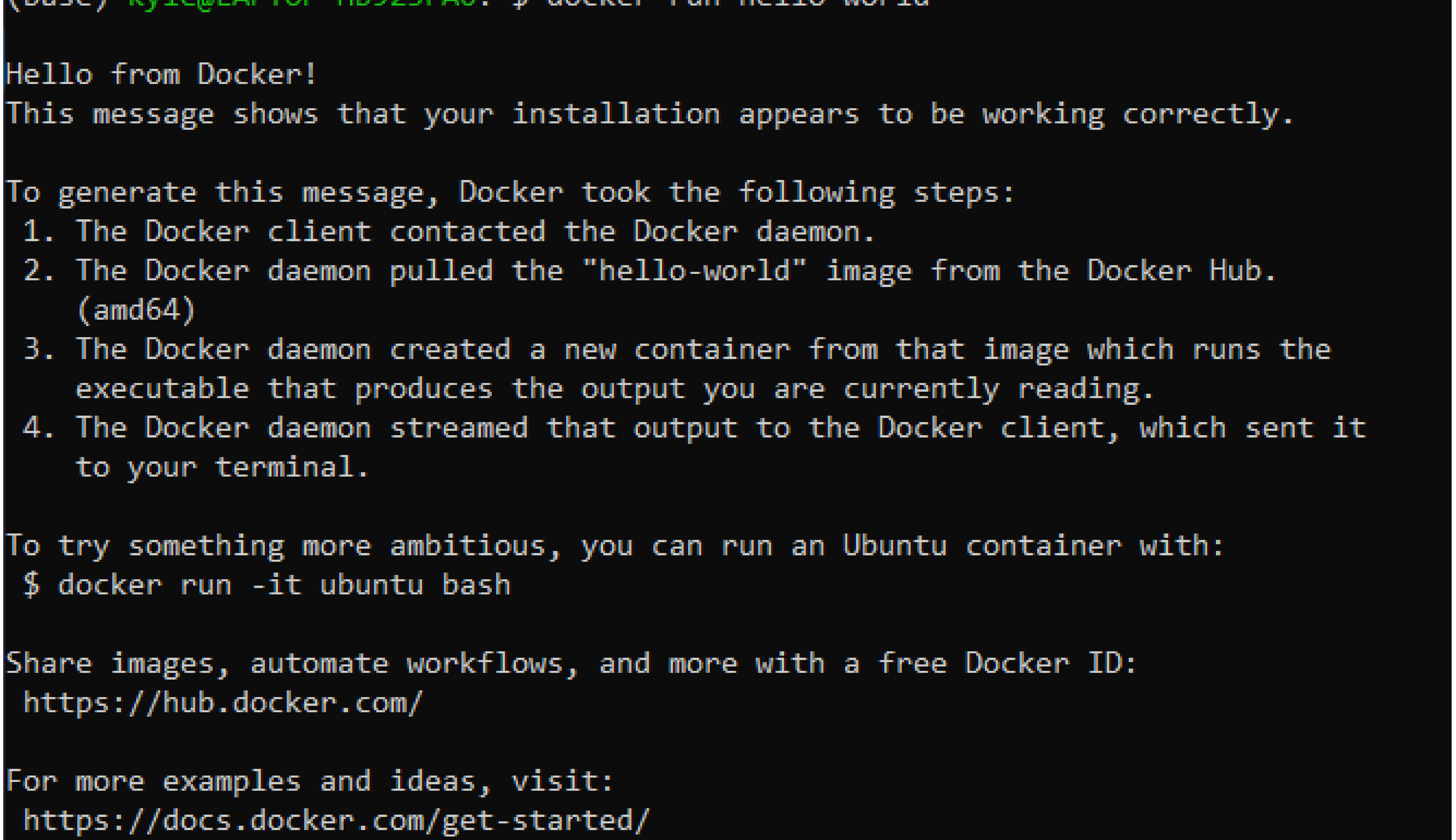
- Steps:
  1. Run a published container
  2. Create and run our own Docker image
  3. Interactive Docker container
  4. Look at mounting a local folder

Hello World

- One of the good things about Docker is that there is a huge repository of existing Docker images that can be run..

- Run our first Docker image
  - $ docker run hello-world

- List images to see what has been downloaded
  - $ docker image ls



```
(base) kyle@LAPTOP-HB925FA6: $ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Write our own “Hello MPCS”

- First write a Python script (hello.py)
    - Print(“Hello MPCS 52040”)

Intro to Dockerfile (Must be called “Dockerfile”)

- FROM: Sets the base image to start from
  - FROM ubuntu:latest; FROM python:3
- ADD: copies files or directories to the container
  - ADD hello.py
- RUN: runs a command to update the state of the image
  - RUN apt-get update –y
  - RUN ./hello.py
  - RUN chmod 777 hello.py
- EXPOSE: expose a particular port
- WORKDIR: sets the current working dir for RUN, CMD, etc.
  - WORKDIR mpcs
  - RUN ./hello.py
- CMD: default command when executing the container (only once in a Dockerfile)
  - CMD [“python” ,”asdad”]

Example Dockerfile

```
FROM ubuntu:latest

RUN apt-get update -y

ADD test.sh /

CMD [“.test.sh”]
```

Write the docker file

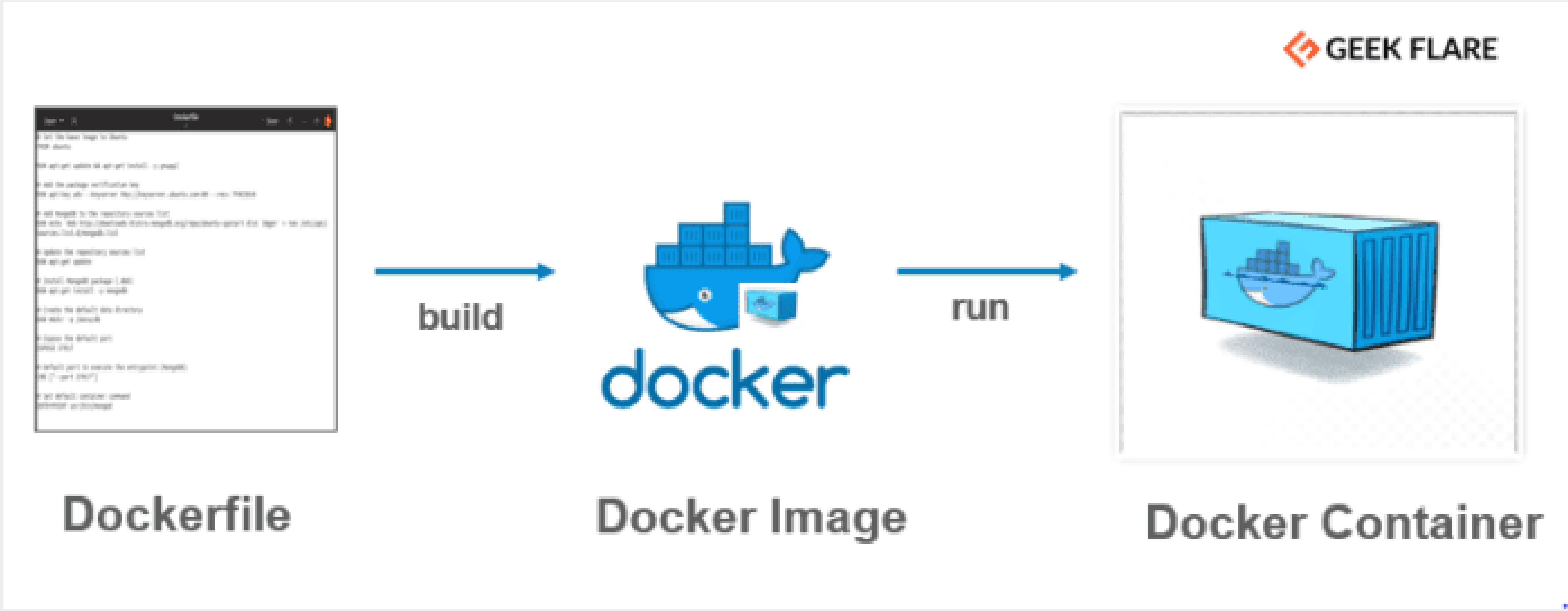
- We should start from an image that has Ubuntu? Python?
- We need to include our program
- We need to run that program

```
FROM python:3

ADD hello.py /

CMD [ "python" "./hello.py" ]
```

Dockerfiles, images, and containers

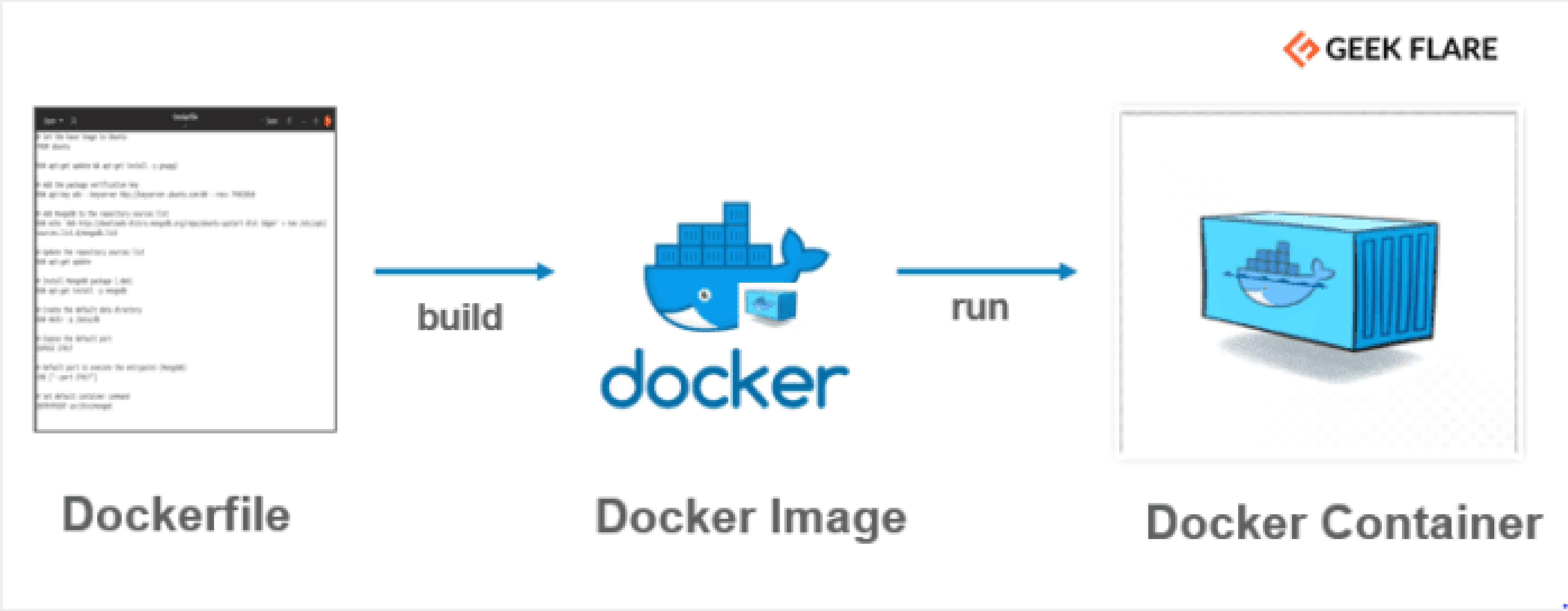


Build the image

- Build the image with “docker build … ”

```
$ docker build -t hello-mpcs .
```

Dockerfiles, images, and containers



Docker run options

docker run <container tag>

- -d (--detach) run container in background
- -I (--interactive) Keep STDIN open
- --rm remove the container when its finished
- -v map a volume
- -p (--publish) publish container ports to the host (for networking)

Run our container

- Build the image

```
$ docker build -t hello-mpcs .
```

- Run the container

```
$ docker run hello-mpcs
```

Interactive access to a running container

- Run a standard ubuntu container:

```
$ docker run -it ubuntu:latest /bin/bash
```

Accessing shared folders

- Docker images are designed to be sandboxed such that they can’t access external resources

*** Enable shared folders in the Docker Desktop application**

- When running use the format:

```
 -v HostFolder:ContainerVolumeName
docker run –v /kyles-stuff:/tmp/files mpcs-hello
```

List directory container

List directory container

```
import os
import sys

print(os.listdir(sys.argv[1]))
```

Dockerfile

- Start with an image? (let's start with ubuntu this time)
- Install Python3
- Set the directory we want to run in
- Copy in our program
- (optionally run the program)

```
FROM ubuntu:latest

RUN apt-get update -y
RUN apt-get install -y
python3

WORKDIR /list

COPY ./list.py .
```

Build and Run

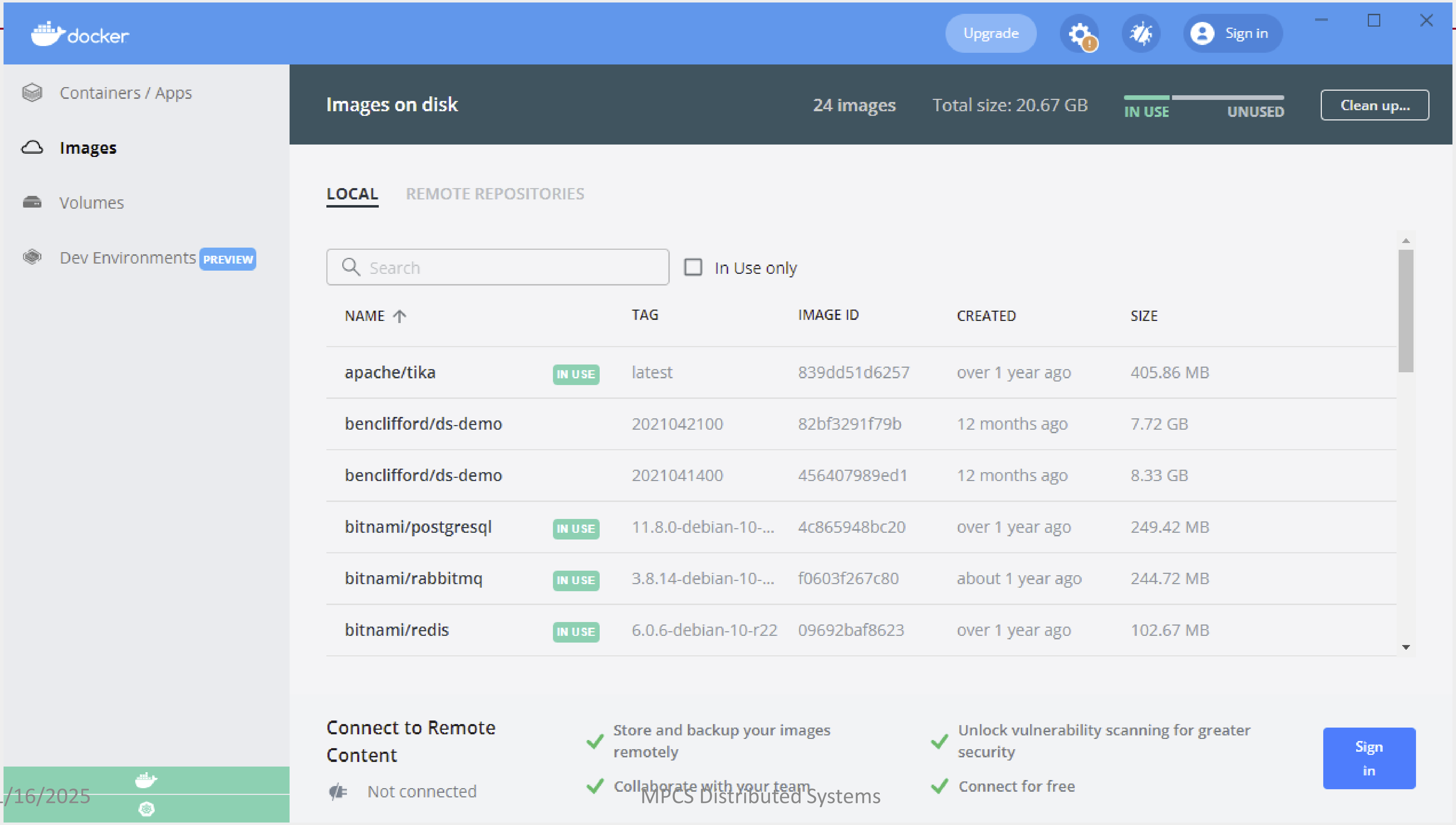
- Build

```
$  docker build -t list-dir .
```

- Run

```
$ docker run -it -v /<local_path>:/<container_path> list-dir
$ docker run -it -v "$(pwd)/shared:/shared" list-dir
```

Clean up containers and images



NAME		TAG	IMAGE ID	CREATED	SIZE
apache/tika	IN USE	latest	839dd51d6257	over 1 year ago	405.86 MB
benclifford/ds-demo		2021042100	82bf3291f79b	12 months ago	7.72 GB
benclifford/ds-demo		2021041400	456407989ed1	12 months ago	8.33 GB
bitnami/postgresql	IN USE	11.8.0-debian-10-...	4c865948bc20	over 1 year ago	249.42 MB
bitnami/rabbitmq	IN USE	3.8.14-debian-10-...	f0603f267c80	about 1 year ago	244.72 MB
bitnami/redis	IN USE	6.0.6-debian-10-r22	09692baf8623	over 1 year ago	102.67 MB

Connect to Remote Content

Not connected

✓ Store and backup your images remotely

✓ Collaborate with your team

✓ Unlock vulnerability scanning for greater security

✓ Connect for free

Sign in

Exercise submission

- Please submit:
  - Hello MPCS
    - Dockerfile
    - Python file
  - List dir
    - Docker file
    - Python file

https://classroom.github.com/a/-E5kcsnn