[544] gRPC and Docker Compose

Tyler Caraza-Harter

Learning Objectives

- describe the functionality that HTTP provides (beyond what TCP alone provides)
- call functions remotely via gRPC
- configure SSH tunneling and Docker port forwarding to communicate with an app in a container on a different machine
- deploy multi-container apps with Docker compose

Outline

HTTP

gRPC

Docker Port Forwarding

Docker Compose

HTTP (Hypertext Transfer Protocol)

HTTP

TCP or UDP

Internet Protocol

Ethernet or Wi-Fi

URL (domain/IP + port + resource)

IP addresses + ports

IP addresses

MAC addresses

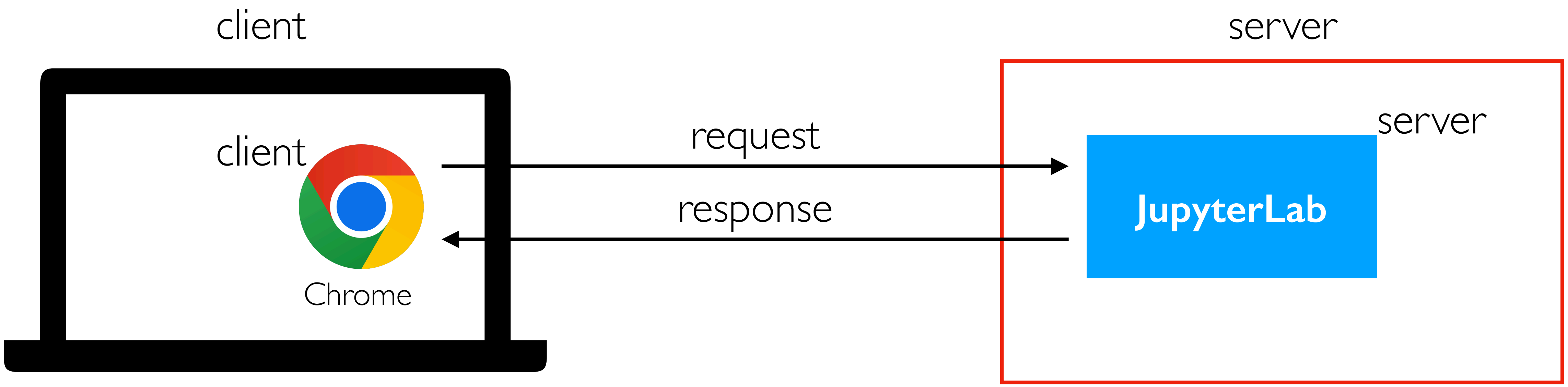
`https://tyler.caraza-harter.com:443/cs544/s23/schedule.html`

domain name
(mapped to an IP)

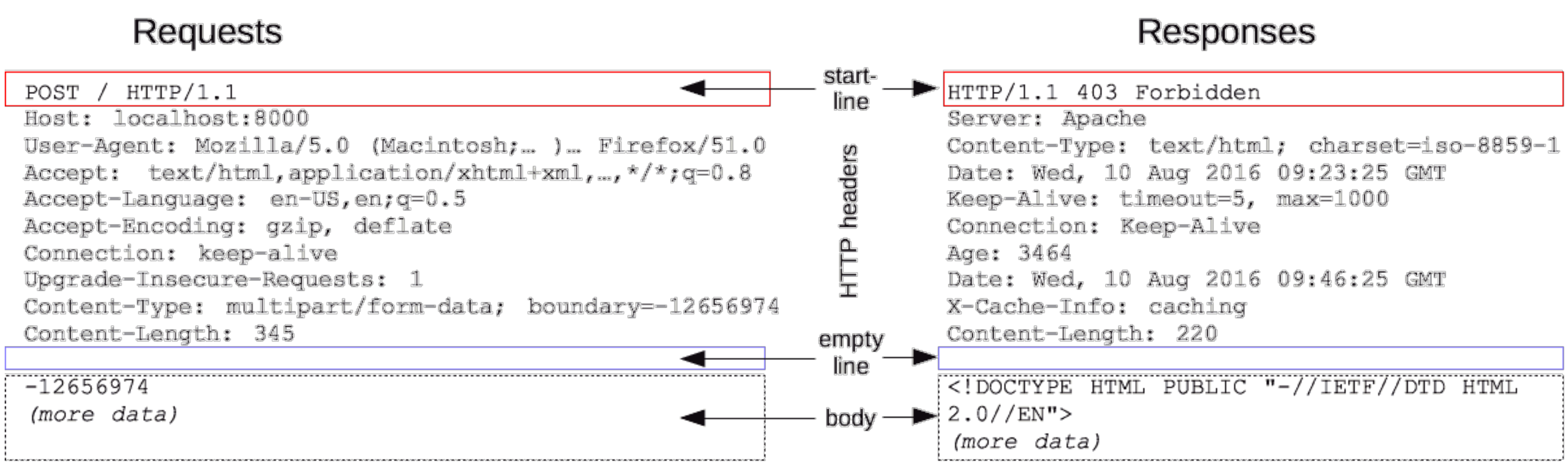
port
(443 is default
for https)

resource

HTTP Messages Betwen Clients and Servers



Parts: method, resource, status code, headers, body



https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

HTTP Methods (types of messages)

Types of request

- **POST**: create a new resource (request+response have body)
- **PUT**: update a resource (request+response have body, usually)
- **GET**: fetch a resource (response has body)
- **DELETE**: delete a resource
- others...

Canvas REST API example:

`GET https://canvas.wisc.edu/api/v1/conversations`
(see all Canvas conversations in JSON format)

`POST https://canvas.wisc.edu/api/v1/conversations`
(create new Canvas conversation)

https://canvas.instructure.com/doc/api/conversations.html

Outline

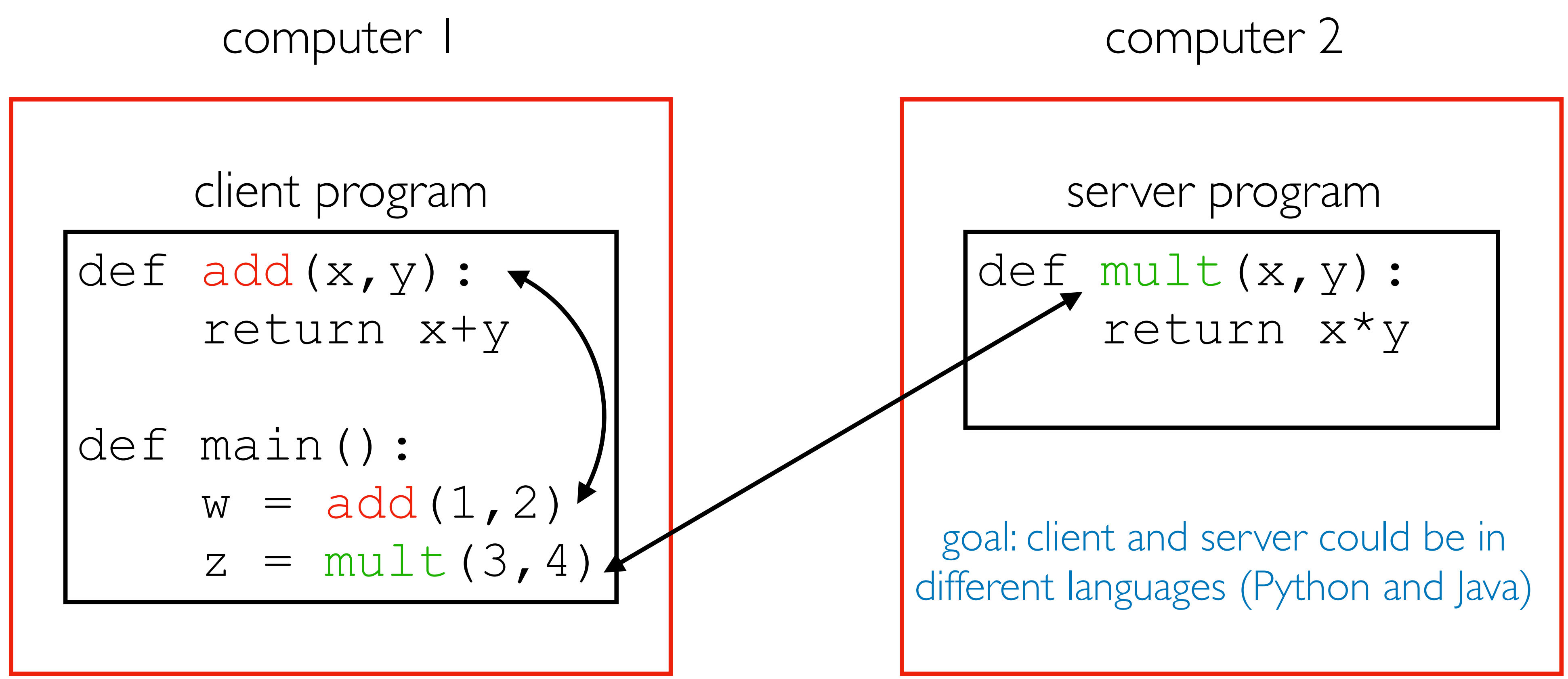
HTTP

gRPC

Docker Port Forwarding

Docker Compose

Remote Procedure Calls (RPCs)



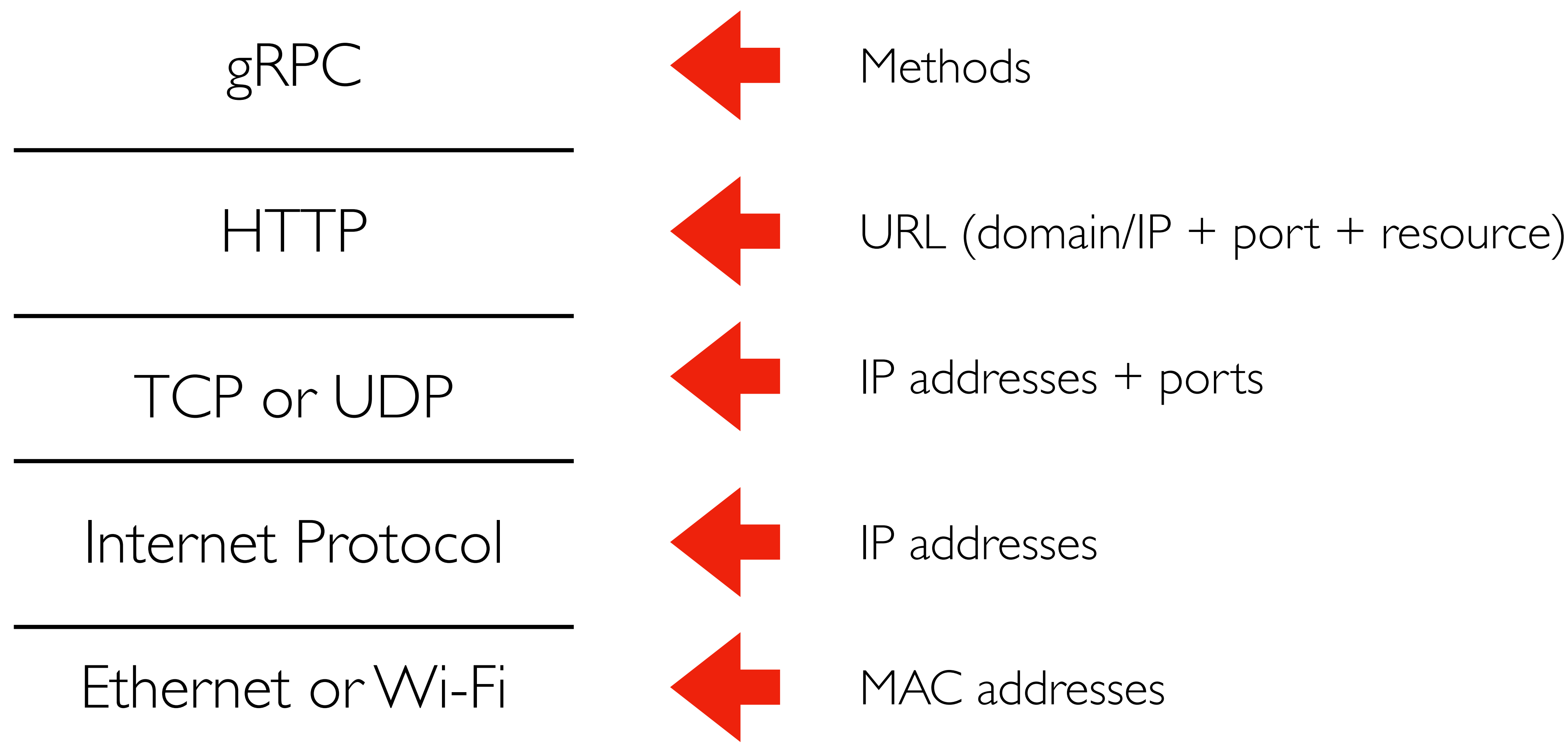
procedure = function

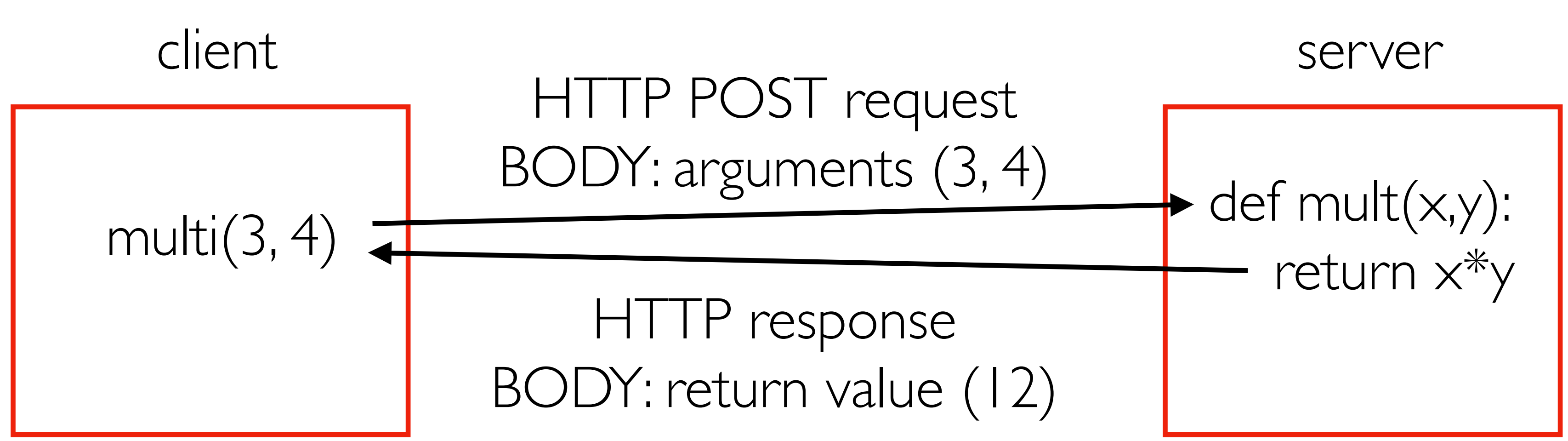
- **main** calling **add** is a regular procedure call
- **main** call **mult** is a remote procedure call

There are MANY tools to do RPCs

- Thrift (developed at Meta)
- gRPC (developed at Google) -- this semester

gRPC builds on HTTP





Serialization/deserialization (Protobufs)

How do we represent arguments and return values as bytes in a request/response body?

Serialization: various types (ints, strs, lists, etc) to **bytes** ("wire format")
Deserialization: **bytes** to various types

Challenge 1: every language has different types and we want cross-languages calls

gRPC uses Google's Protocol Buffers provide a uniform type system.

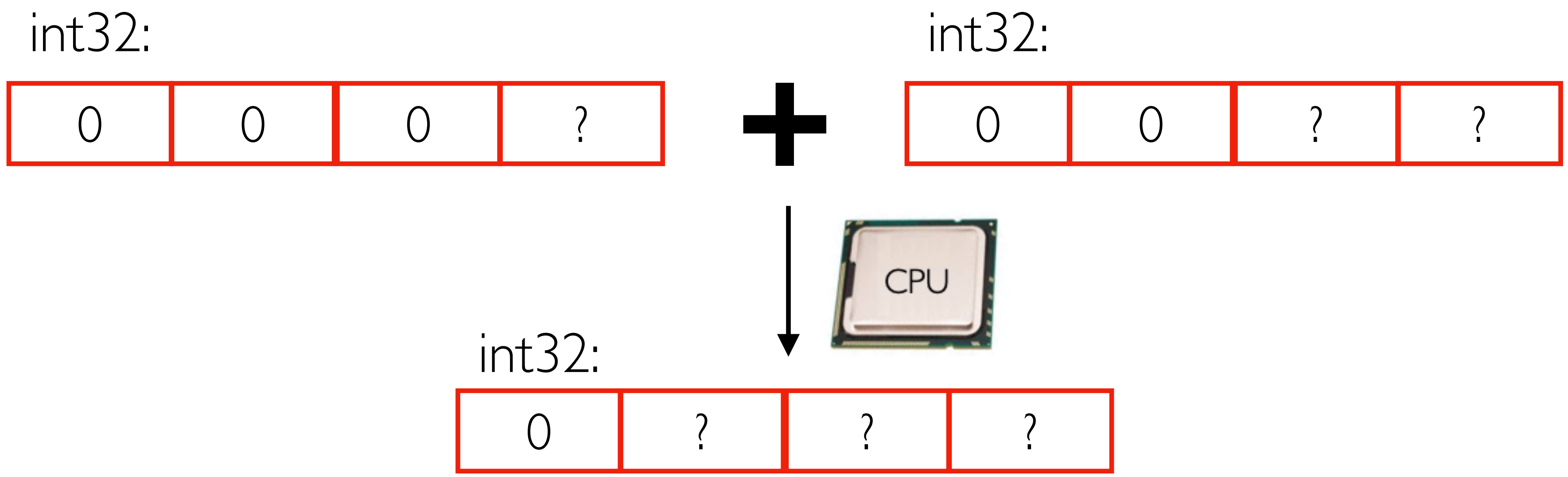
Challenge 2: different CPUs order bytes differently

cpu A int32:	byte 1	byte 2	byte 3	byte 4
cpu B int32:	byte 4	byte 3	byte 2	byte 1

.proto	C++	Java	Python
double	double	double	float
float	float	float	float
int32	int32	int	int
int64	int64	long	int
uint32	uint32	int	int
uint64	uint64	long	int
sint32	int32	int	int
sint64	int64	long	int
bool	bool	boolean	bool
string	string	String	str
bytes	string	ByteString	bytes

https://protobuf.dev/programming-guides/proto/

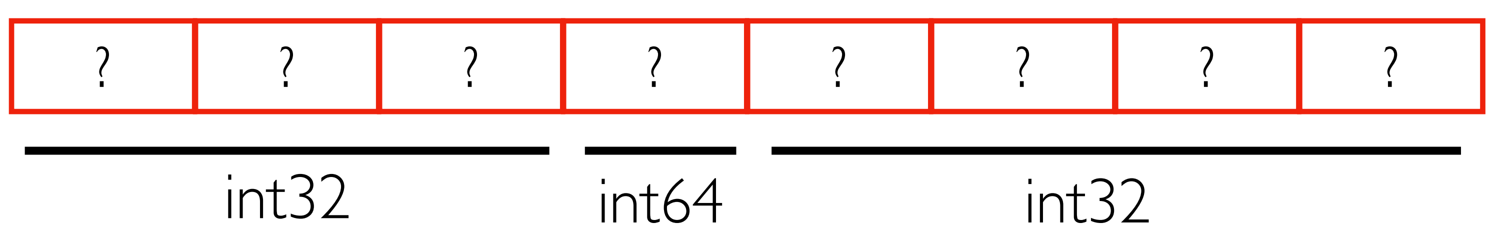
Variable-Length Encoding



For **computational efficiency**, int32's use 4 bytes during computation. Also helps w/ offsets.

For **space efficiency**, smaller numbers in int32s user fewer bytes (4 bytes is max).
This reduces network traffic.

Example nums in a protobuf:



Demos...

Outline

HTTP

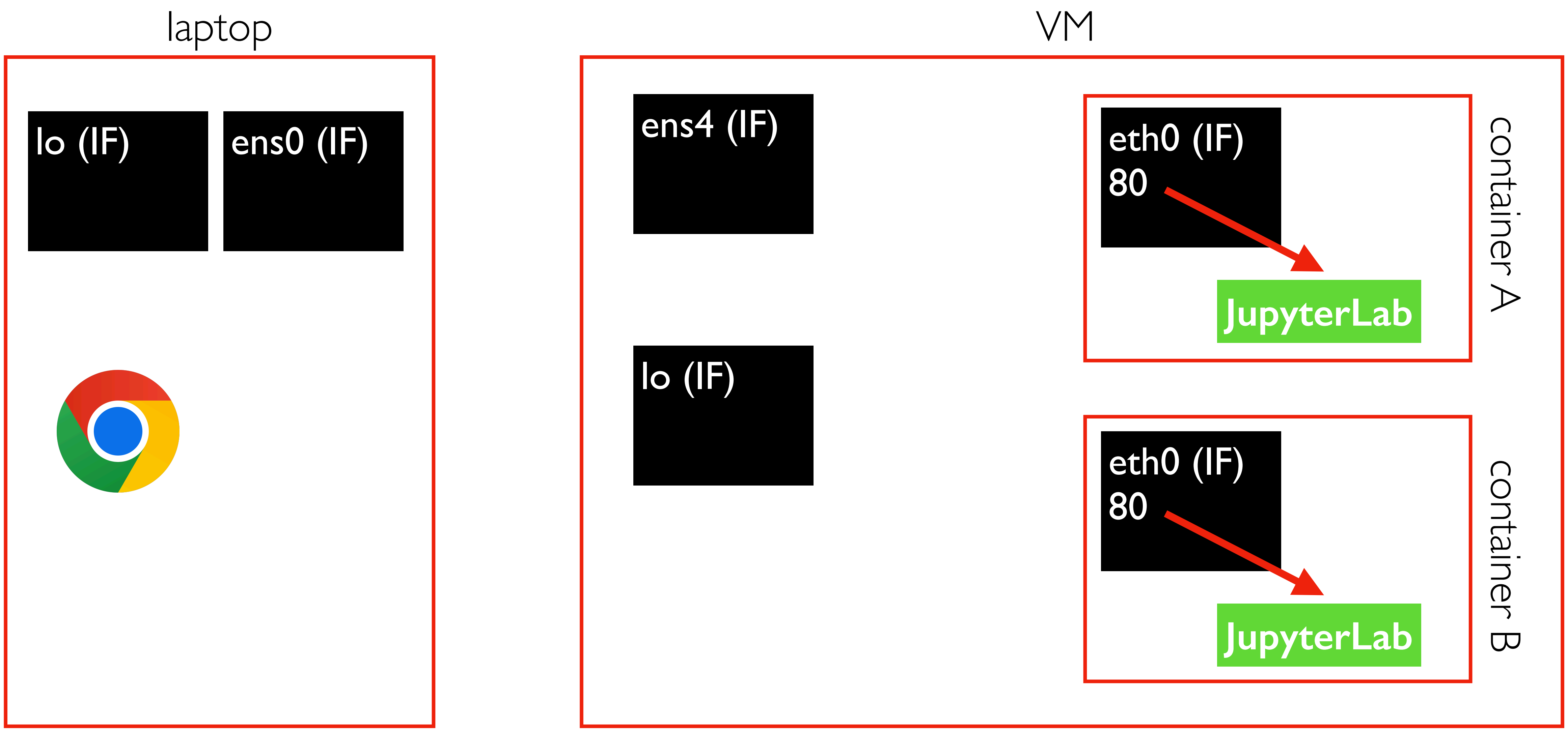
gRPC

Docker Port Forwarding

Docker Compose

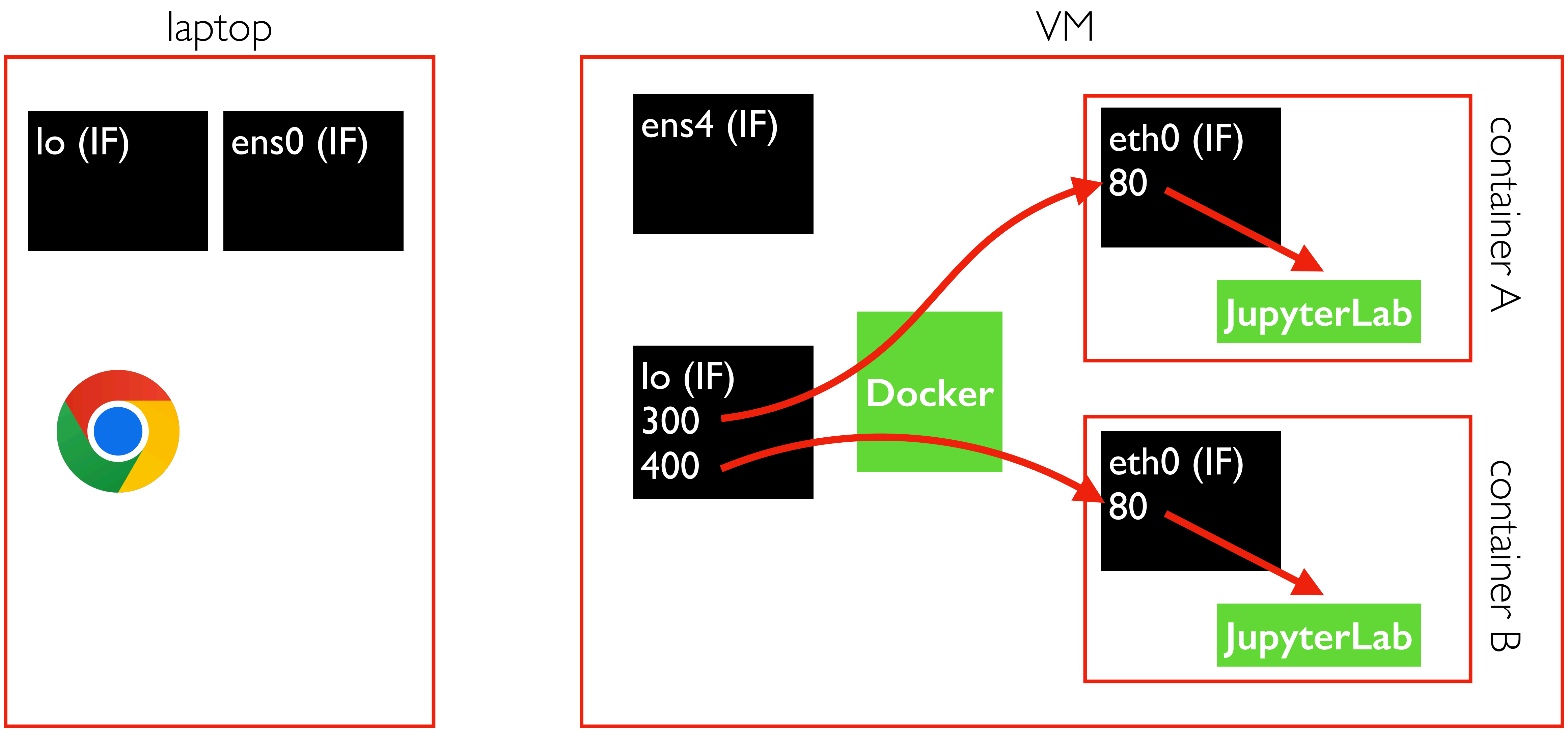
Interfaces (IF) and Ports

both containers have
a virtual port 80



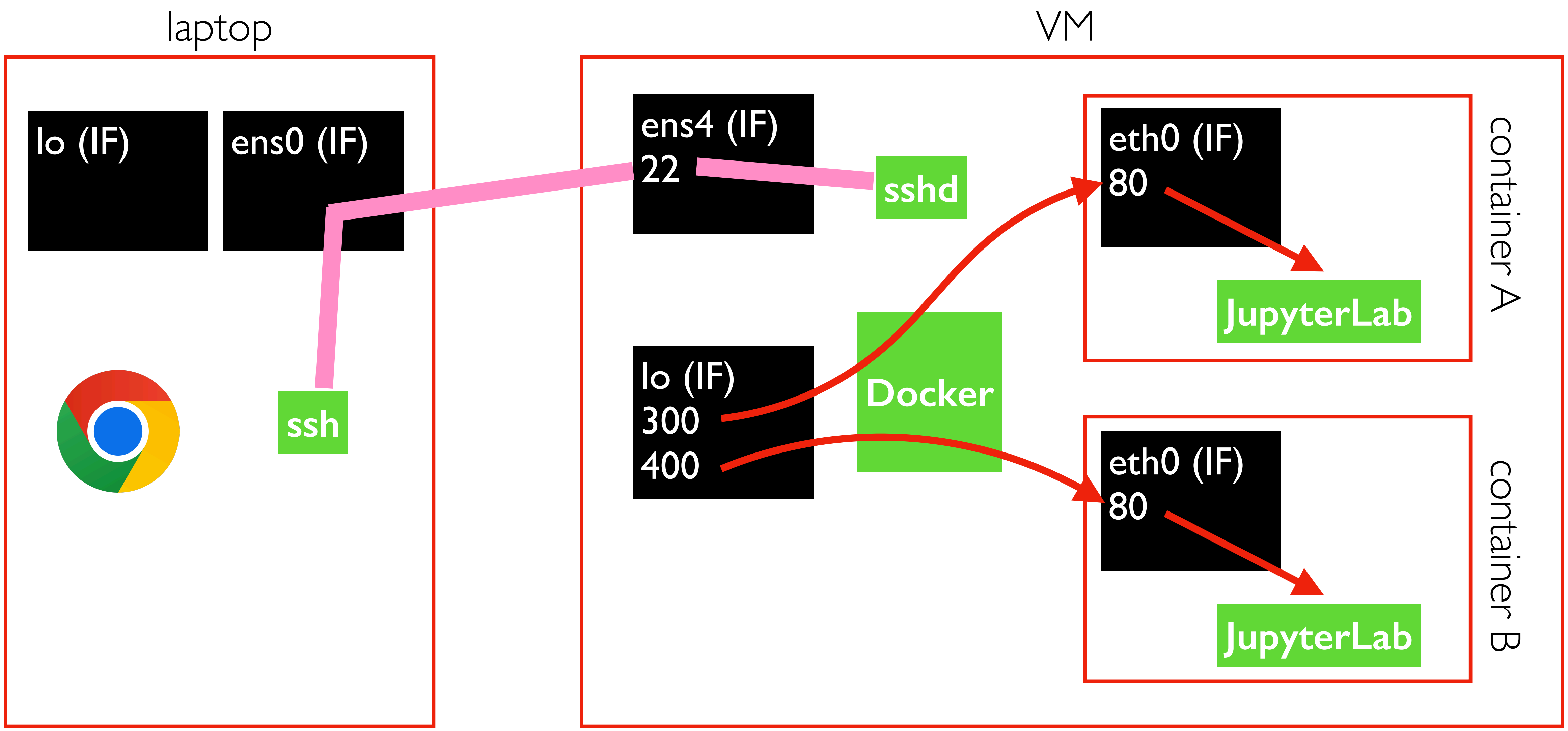
docker run -d myimg
docker run -d myimg

Interfaces (IF) and Ports



docker run -d **-p** 127.0.0.1:300:80 myimg
docker run -d **-p** 127.0.0.1:400:80 myimg

Interfaces (IF) and Ports

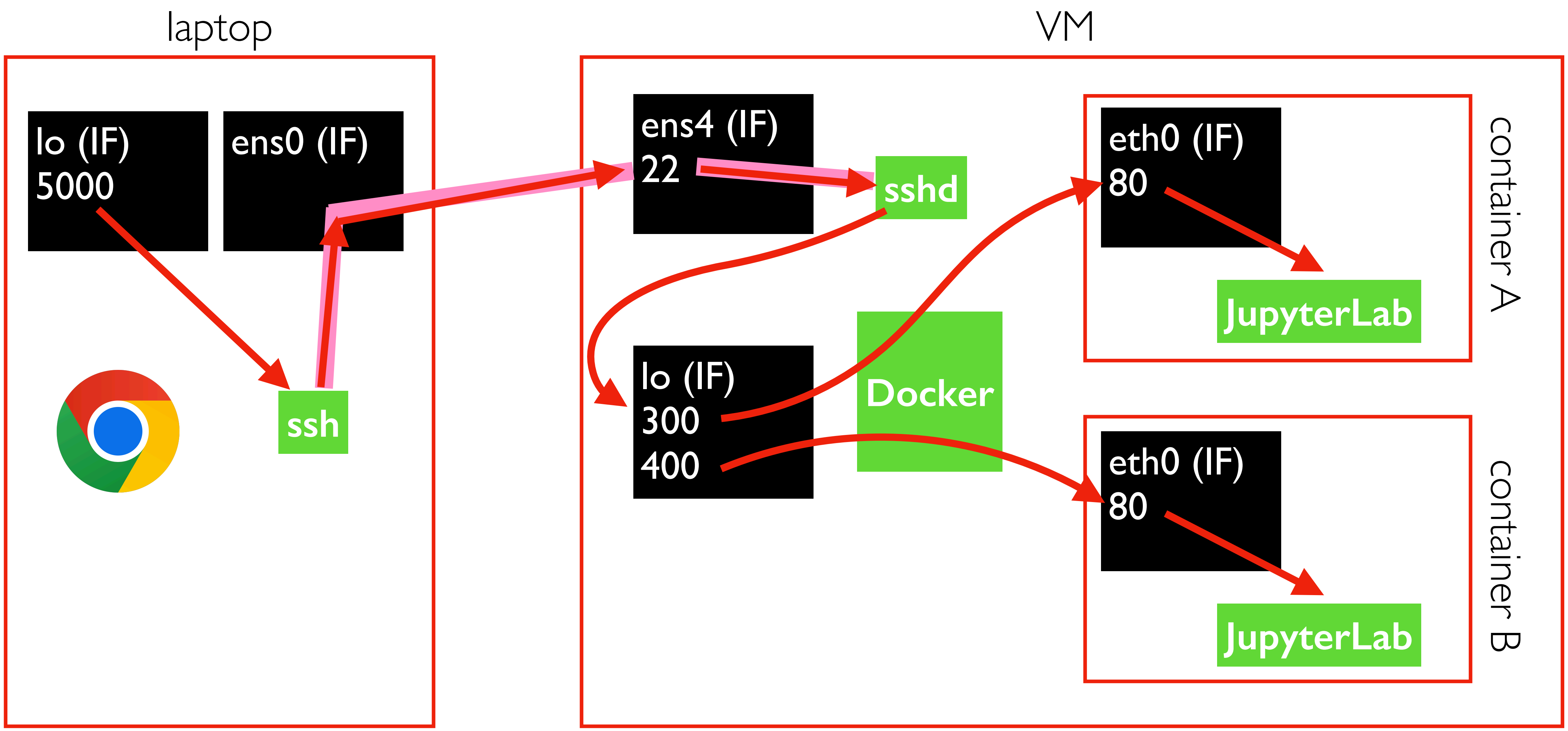


ssh USER@VM

docker run -d **-p** 127.0.0.1:300:80 myimg
docker run -d **-p** 127.0.0.1:400:80 myimg

the SSH connection can be used to send comands and/or forward network traffic

Interfaces (IF) and Ports

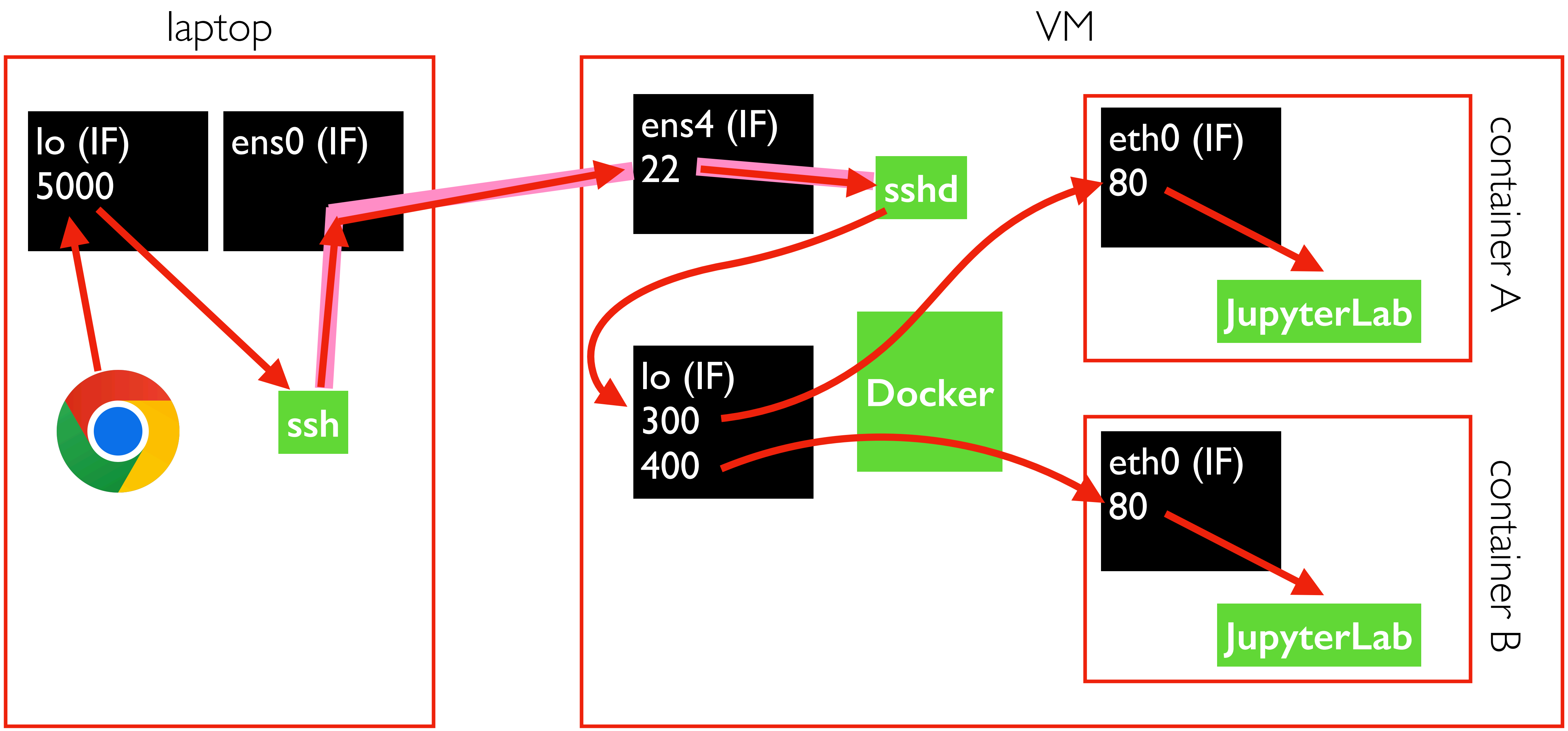


ssh USER@VM **-L** localhost:5000:localhost:300

docker run -d **-p** 127.0.0.1:300:80 myimg
docker run -d **-p** 127.0.0.1:400:80 myimg

the SSH connection can be used to send comands and/or forward network traffic

Interfaces (IF) and Ports



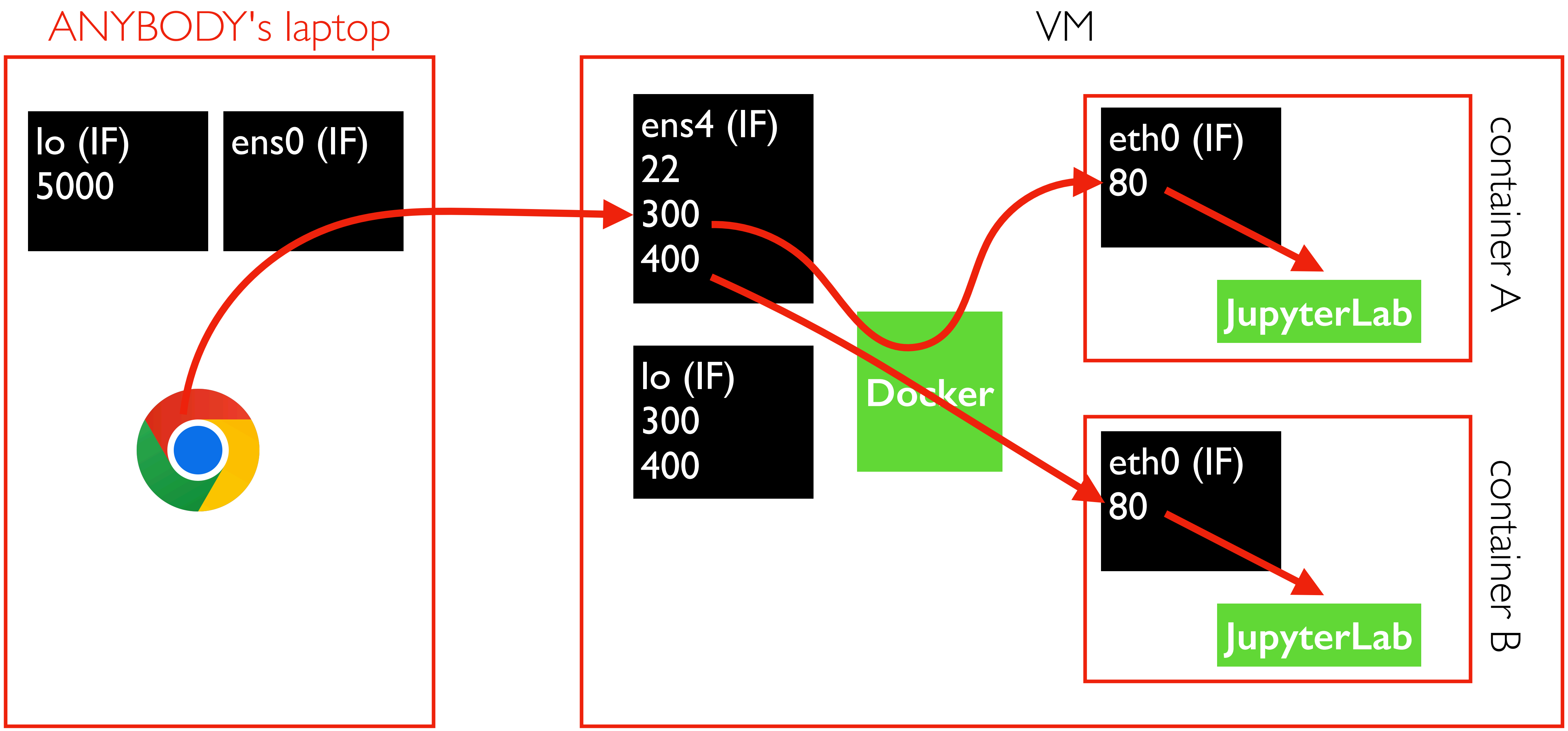
ssh USER@VM **-L** localhost:5000:localhost:300

docker run -d **-p** 127.0.0.1:300:80 myimg
docker run -d **-p** 127.0.0.1:400:80 myimg

http://localhost:5000/lab (in browser)

yay! You can connect to JupyterLab
inside a container running on your VM

Interfaces (IF) and Ports



docker run -d **-p** 300:80 myimg

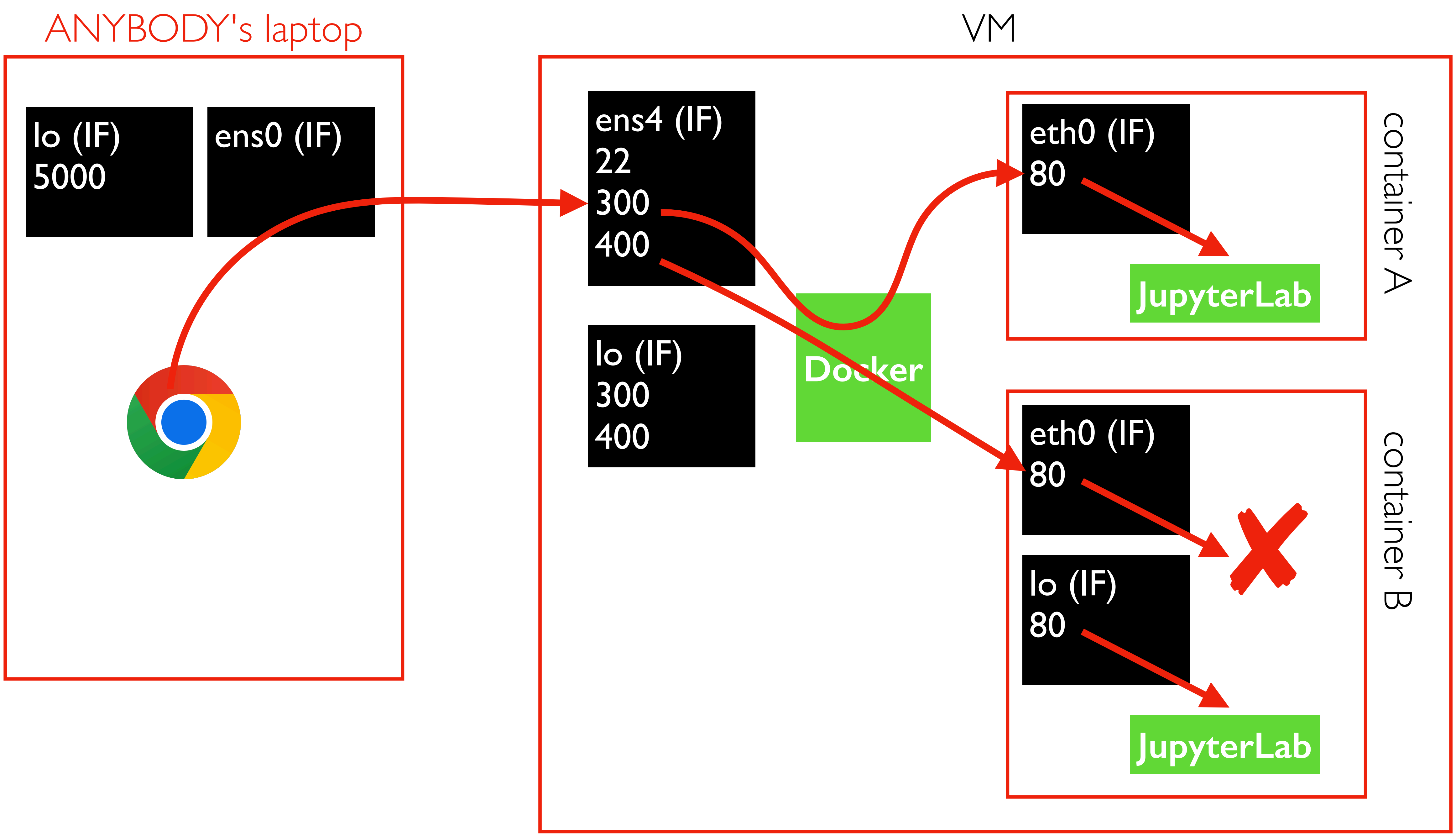
docker run -d **-p** 0.0.0.0:300:80 myimg

Careful, default is to listen on all ports!

Other security:

- firewall (block port 300)
- password (in JupyterLab)

Interfaces (IF) and Ports



Port forwarding does not go to loopback inside container

- don't use localhost or 127.0.0.1!
- easiest: use 0.0.0.0 (for all)

Outline

HTTP

gRPC

Docker Port Forwarding

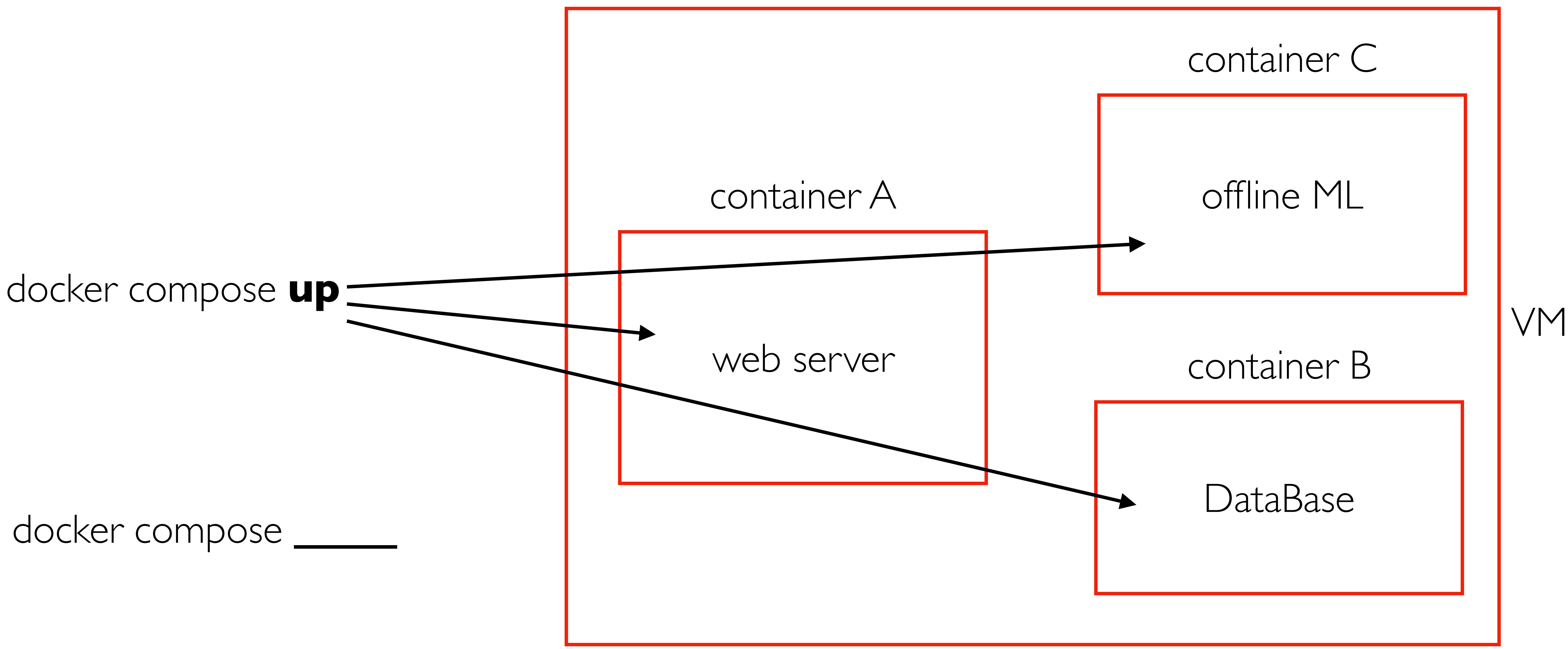
Docker Compose

Container Orchestration

Orchestration lets you deploy many cooperating containers across a cluster of Docker workers.

Kubernetes is the most well known.

Docker compose is a simpler tool that lets you deploy cooperating containers to a single worker.



Demos...