Enterprise Programmering 2

Lesson 10: MicroServices

Dr. Andrea Arcuri

Goals

- Learn what MicroServices are, and where/when you need to use them
- Understand the concept of *Load Balancing*
- Understand the role played by *API Gateways*

The Monolith

- Single enterprise application containing everything
  - Eg, single WAR deployed on a Wildfly/Glassfish server
  - Note: can still be divided in packages/modules, but the packaged “executable” will just be a single file (eg WAR or JAR)
- On non-trivial systems, can easily be more than 1 million lines of code
- Extremely common
  - Most enterprise systems developed until the 2010-2015 years are monoliths

Monolith Hell

- Lot of issues with monolith applications
- What happens when new developer joins the team?
  - Understanding 1 million lines of code will take time before becoming productive
- What if for some specific task you need a different technology?
  - Eg, Python, NodeJS, etc.
- How to scale if some functionality is highly used?
  - Need to deploy the whole monolith on many machines, even if you just need a small subset

Monolith Hell (cont.)

- What if you need to update/fix a single functionality?
  - Need to redeploy the whole monolith on all the machines
- What if a single functionality is buggy?
  - Might take down the whole monolith application
- What if your technology stack becomes obsolete?
  - Rewrite whole monolith is not viable
  - Adding new functionalities in a new technology stack might conflict with current stack in the monolith
- Etc. Etc.

MicroServices to the Rescue

- MicroServices are an architectural pattern to address some of the issues in monolith applications
- **No Silver Bullet**
  - MicroServices are not the answer to all problems, and they have their own set of issues
- **EXTREMELY** popular in industry in the last few years
- If you are going to work as a backend developer, most likely you will end up dealing with REST in a microservice architecture

Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

MicroServices in a Nutshell

- Divide your system in *independent* components, ie services
- Each component should be *compilable* and *deployable* on its own
  - Typically, but not necessarily, they are RESTful web services
- How many components?
  - “Two Pizza” rule: a team shouldn’t be bigger than what 2 pizzas could feed
    - Actual rule coming from Amazon, a pioneer in microservices
- Not uncommon having applications made by hundreds of components
- Communications should be programming/OS agnostic
  - Eg, JSON/XML over HTTP

Benefits: Easy To Understand

- A new engineer will start working on just one component
- Understanding a single component (e.g., a RESTful web service) is easier than trying to figure out how a whole monolith works
- Easier to test/debug, as can execute in isolation
    - Would still need to mock interactions though, eg with WireMock

Benefit: More Robust

- If one component is failing/buggy, can shut it down in isolation until fixed
- All the other hundreds of components will still be up and running
- Of course, functionalities will be reduced and some will be missing
  - Application should still work, although in a "degraded mode"
  - Make sure to avoid communications with missing service, eg *Circuit Breaker* with Hystrix

Benefit: Language Agnostic

- As components are independent, they can be written in different languages
  - Java, C#, Python, NodeJS, Ruby, etc.
- Less worries about the future
  - If in 10 years your technology stack dies, for new components can easily switch to a new language/framework
- Can easily experiment
  - Eg, for a new component you can try something different, like C#, Scala or NodeJS
  - Extremely important when evaluating new technologies/frameworks

Benefit: Scale on Demand

- Not all components will be used/access equally
- Some are just for functionalities that are seldom used
- Highly used components can be replicated/deployed on several servers
- Just need to deploy extra instances of components you need
  - Want more running instances on different machines of components that use more CPU
- This can be fully *automated*
  - Eg, reduce number of running instances of components that are seldom used

Benefit: Safer Deployment

- Can deploy components in isolation
  - Eg replace version X with version X+1
- If something goes wrong with X+1, you just need to rollback that single component
- Less risky then deploying a whole monolith…

No Silver Bullet

- In engineering, there is never a solution that fits all problems
- MicroServices have their own issues
- Lot of benefits, but *do not blindly follow hypes*
- Needed for *large* systems. For *small* systems, monolith can be a better solution
  - What a 3rd year student can do by his/her own or in a group of few students over a couple of months is by definition “small”…

Drawback: Computation Overhead

- Communications between different components are more expensive than in a monolith
  - Eg, HTTP over TCP
  - Even if running on same machine
- Lot of un/marshaling to/from JSON/XML
- A direct Java call in same JVM is far much cheaper…

Drawback: Complex setup

- No more 1 single WAR/JAR, you have (for example) 500 now…
- Can’t use simple script to deploy/start the whole application
  - Need special tools, e.g. *Kubernetes* or *Docker-Compose*

Drawback: Atomicity

- Some actions have to be atomic
  - Eg, sequences of operations should all pass or all fail
  - Eg, can only buy an item if still present in warehouse and credit transaction does not fail, and those can be implemented in different components
- In single application, easier to ensure atomicity
  - Eg, think of transactions to a database
  - Recall ACID: Atomicity, Consistency, Isolation and Durable
- In a distributed system (even if running on the same server machine), much harder to implement reliable atomicity

Drawback: Testing

- Yes, you can test components in isolation, but then have to mock away all inter-component interactions
  - Eg, using WireMock
- Starting, stopping and cleaning up 500 components is more difficult than a single monolith

The 12 Factor App

“The twelve-factor app is a methodology for building software-as-a-service apps” (https://12factor.net/)

1. *Codebase*

   One codebase tracked in revision control, many deploys

2. *Dependencies*

   Explicitly declare and isolate dependencies

3. *Config*

   Store config in the environment

4. *Backing services*

   Treat backing services as attached resources

5. *Build, release, run*

   Strictly separate build and run stages

6. *Processes*

   Execute the app as one or more stateless processes

7. *Port binding*

   Export services via port binding

8. *Concurrency*

   Scale out via the process model

9. *Disposability*

   Maximize robustness with fast startup and graceful shutdown

10. *Dev/prod parity*

    Keep development, staging, and production as similar as possible

11. *Logs*

    Treat logs as event streams

12. *Admin processes*

    Run admin/management tasks as one-off processes

Containers and Orchestration

A Single Component

- Typically, but **not** necessarily, a RESTful web service
- Language does not matter
- Issue when dealing with different languages
  - How to deploy, start/stop different components?
- How to guarantee that a component can run in different servers?
  - Even if Java is highly portable, still need to make sure same version of JRE is installed on all the servers
  - Subtle differences between OSs and internal configurations
- Need *Immutable Delivery*

Deploy Operating System (OS) Images

- Do not limit to just package a JAR or WAR file
- Create a whole image of an OS, including all needed software
  - Eg the version of JRE that you need
- Virtual Machines
  - Do not install the OS image on the server, but rather run it in a virtual box
  - Different tools enable this
    - Eg, *VirtualBox* from Oracle
    - Eg, *Parallels* if you need to run Windows on a Mac

Docker to the Rescue

- Virtualization technology
- Create OS images, on top of a predefined one
  - Eg a predefined image could be a Linux distribution with the latest version of JRE installed
  - Large catalog online of existing base images
- When building a component, instead of creating a JAR/WAR file, it will create a Docker image

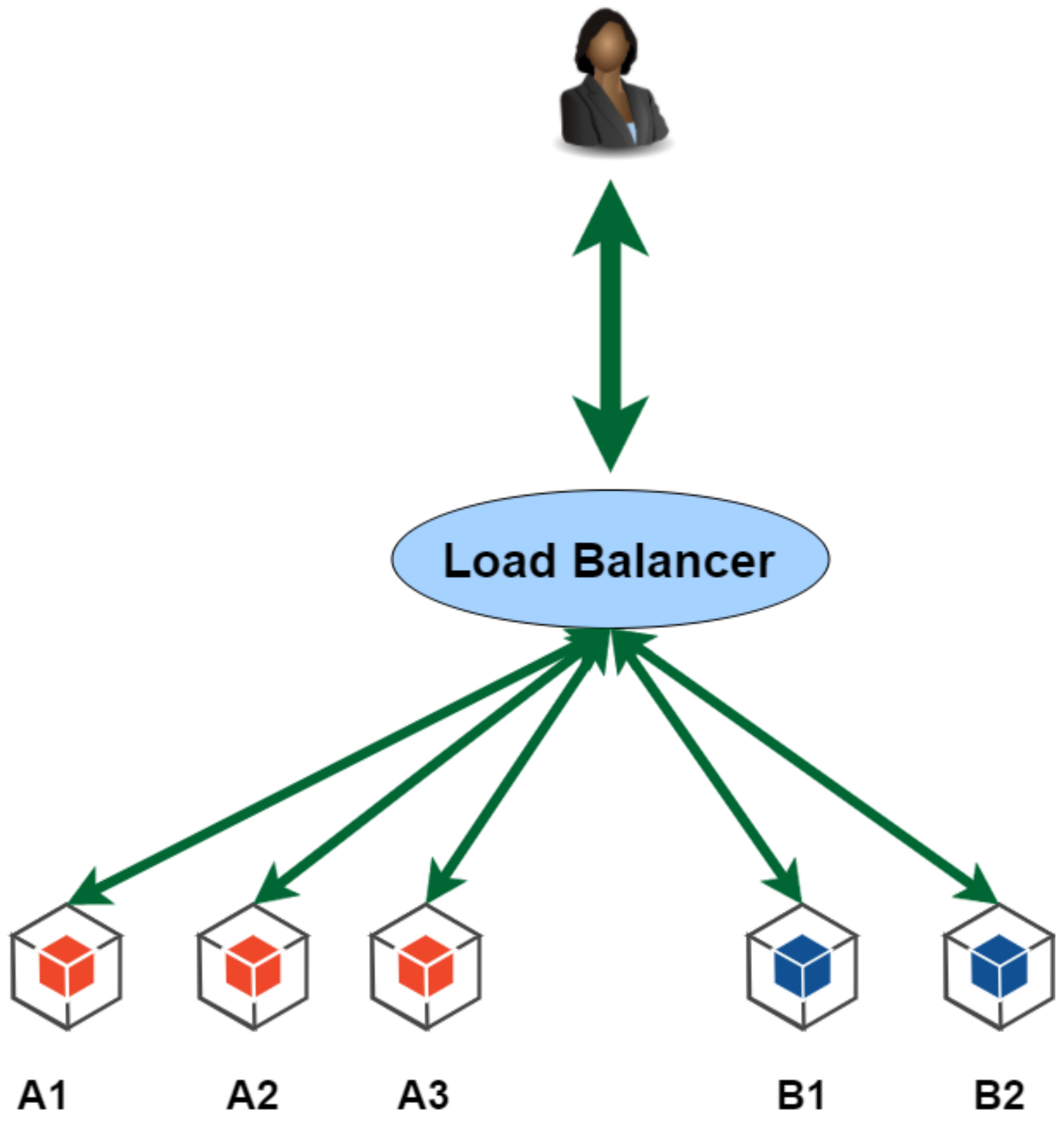
Orchestration

- You might have 100s of services, with Docker
- How to start all of them?
- How to stop them?
- How to automatically restart a service that crashed?
- How to automatically spin more instances of highly used services?
- How to automatically kill instances of seldom used services?
- Etc.

Container Cluster Manager Frameworks

- Open-source tools: eg *Kubernetes* and *Mesos*
- Allow you to easily deploy and monitor Docker containers on different servers
- *Kubernetes* created at Google, and used internally for their systems
- Note: we will not use such tools in this course, but you need to know about them
- We will use *Docker-Compose* to start a static set of Docker images, without automated scaling or failure restart handling

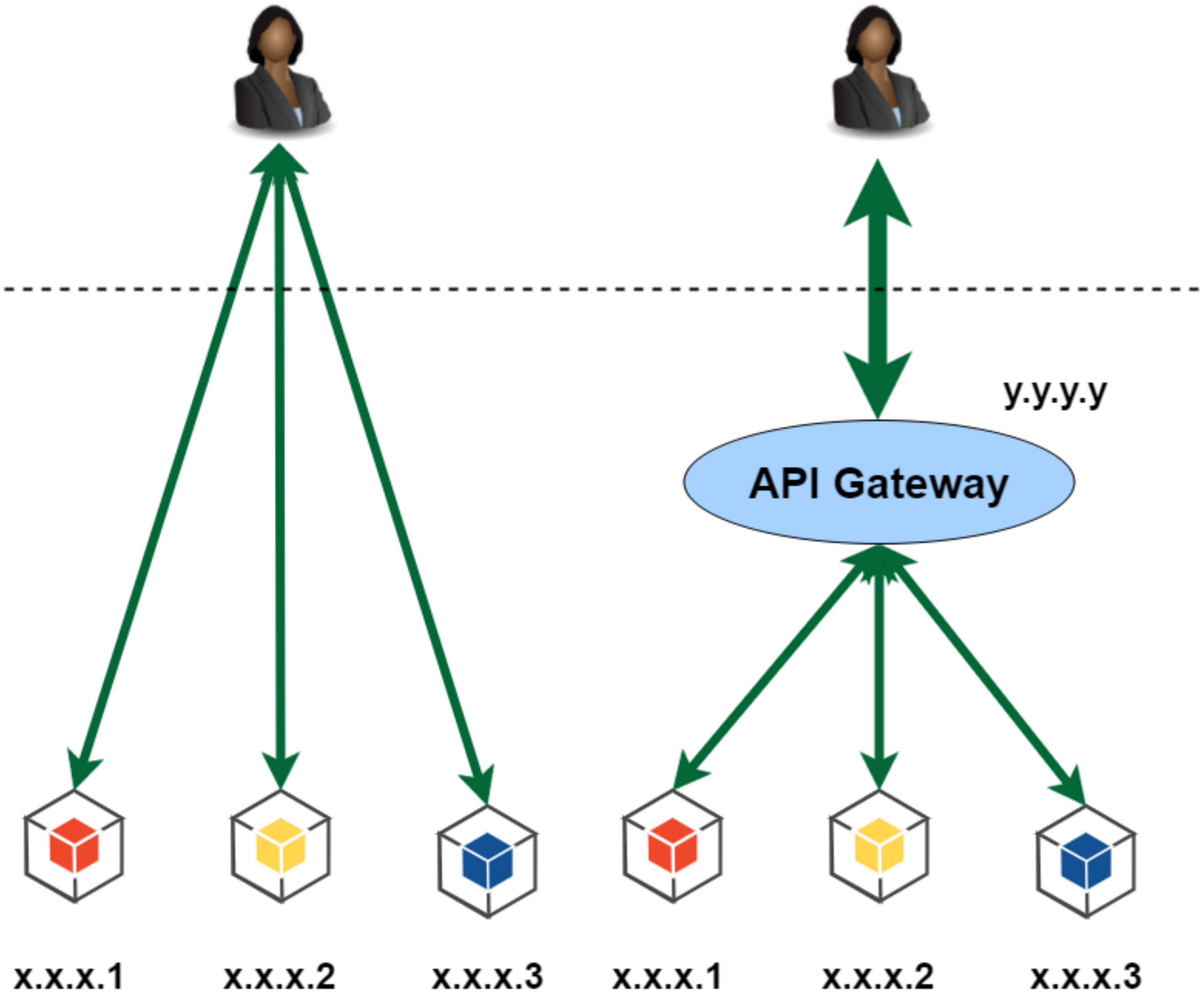
Some More Details…



Load Balancer
A1
A2
A3
B1
B2

Load Balancing

- Different technique, eg Round Robin
  - At each request, forward to next instance, and once all are asked once, next one is from the beginning as in a ring, ie 1-2-3-1-2-3-1-2-3-1-…
- ESSENTIAL that the communication protocol is *stateless*
  - 2 successive calls might end up in 2 different running instances of the same service
  - State has to be handled externally, eg in a database
  - Note: if in a web application you have state (and you run several instances) like *stateful* EJBs and session JSF beans, then need to configure load balancer to remember session mapping (eg, based on cookies). In a REST API, just avoid internal state.



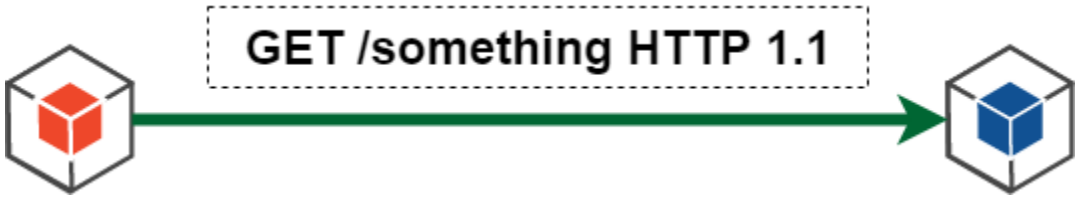
y.y.y.y
API Gateway
x.x.x.1
x.x.x.2
x.x.x.3
x.x.x.1
x.x.x.2
x.x.x.3

API Gateway

- Client might need to interact with hundreds of services
- Keeping track of them in the client is far too complex, and expose internal details of the microservice system (which might change)
- One single entry point, which will forward to the right REST service
- Positive: much easier to write clients, less coupling
- Negative: one point of failure, possible bottleneck

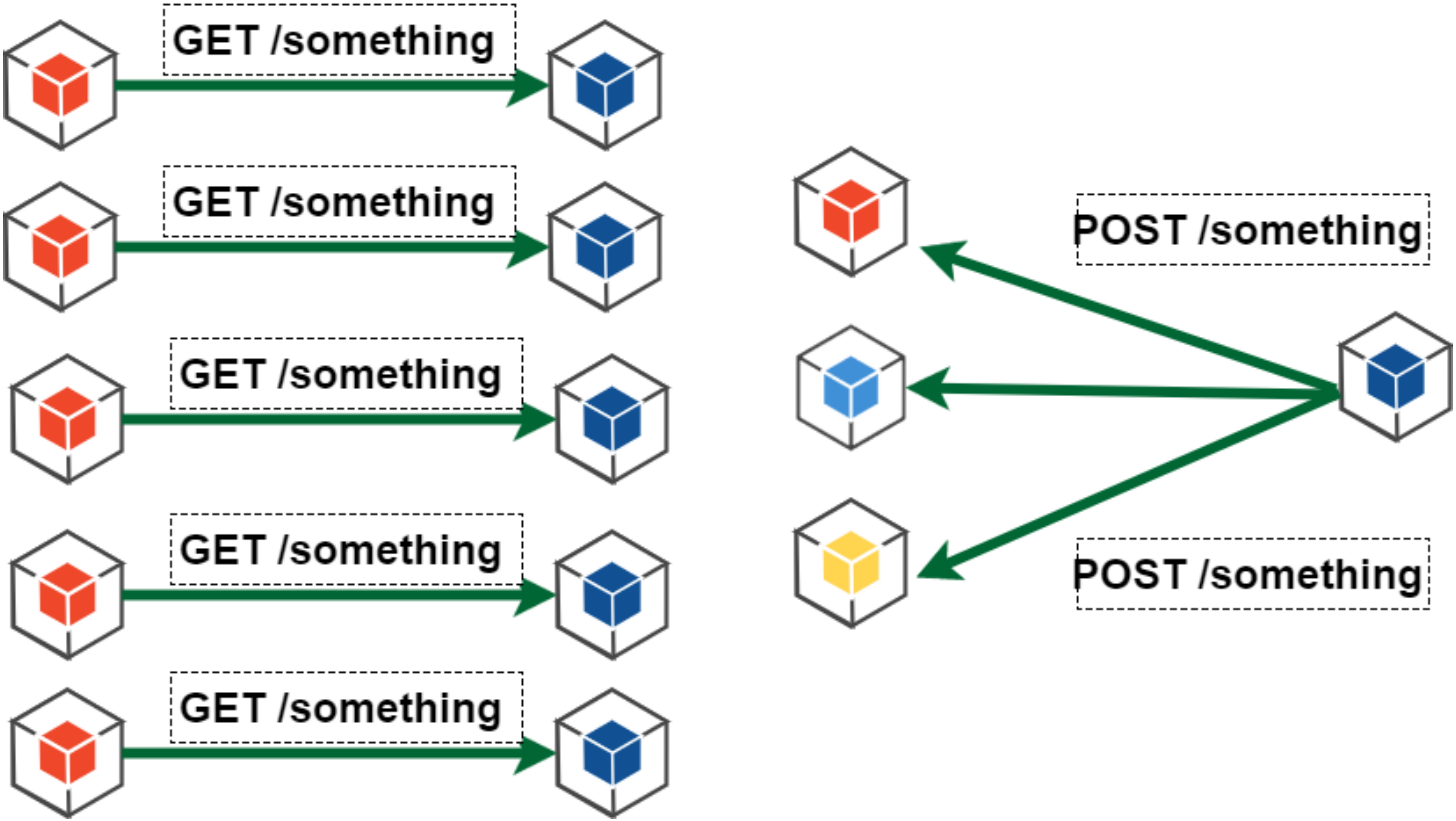
Inter-Service Communications

- When service X needs something from service Y, if REST service, can just do a HTTP call to it
  - Y provides the information, and X just asks for it directly
  - Y is passive, it is X that starts the communication



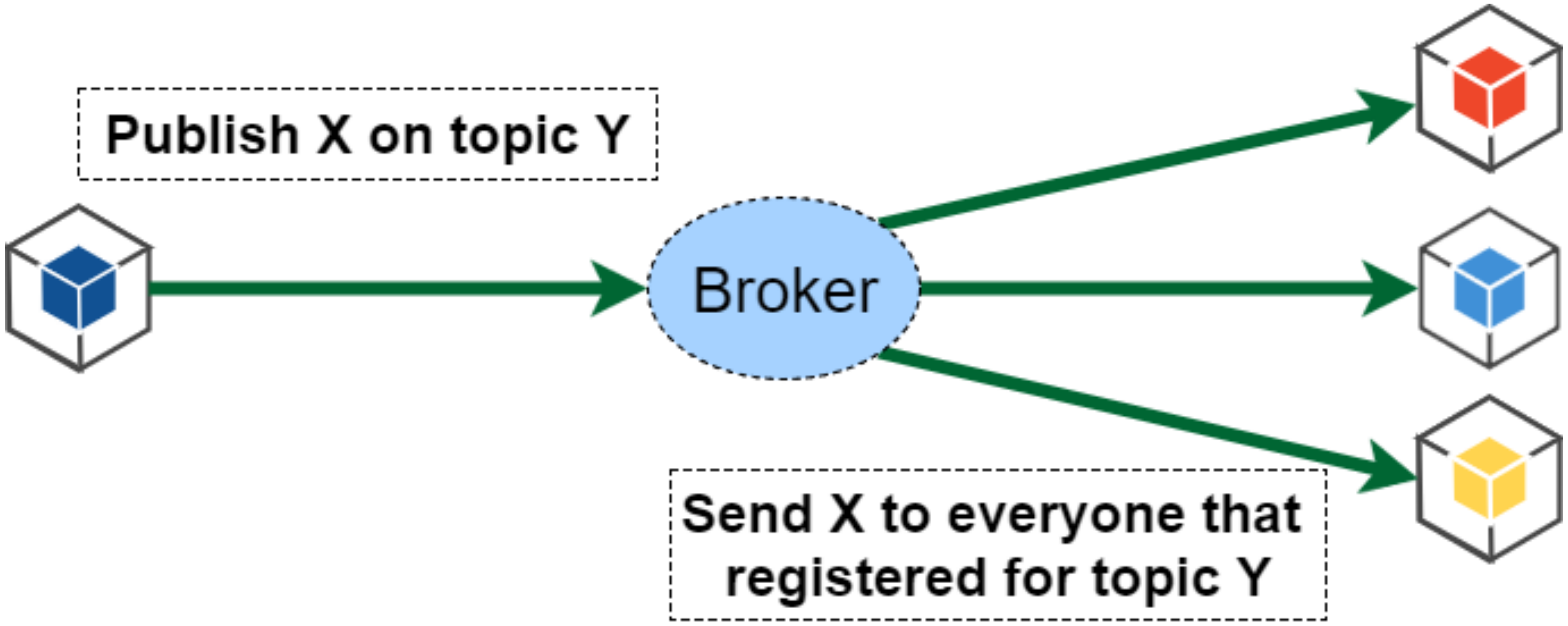
Inter-Service (cont.)

- What if X and Z are waiting for some events to happen in Y?
  - Y a service representing a game
  - X is service showing stats/info of a user
  - Z is service representing a “score board”
- Using REST API, I have two options
- 1) X and Z do continuous pulls (ie GET), eg every 10 seconds, to see if any change in Y (eg, has a new game finished?)
  - Very bad, highly inefficient
- 2) Y starts communication, and sends (ie POST) data to X and Z
  - Not scalable, Y has to know about all possible services interested in its data


GET /something
GET /something
GET /something
GET /something
GET /something
POST /something
POST /something

Message Broker

- A *broker* (which will be a running process) will receive/forward messages
- A service that wants to *publish* some information, will create a *topic* on the broker, and then send messages to it
  - this is independent from HTTP, using a specific protocol defined by the broker
- Clients will register with the broker for one or more topics, and then will *asynchronously* receive all messages sent to those topics
- Think about sending an email to a mailing list…
- Broker can guarantee delivery: messages can be saved to disk, and clients can receive messages sent *before* they contacted the broker (useful if some clients had to restart, or previous network issues)



Eg, Y is “New Game/Match Is Finished”, and X is the detailed info of such game/match, eg the ID, who won, etc.

Message-Oriented Middleware (MOM)

- Different broker tools, in different programming languages
  - ActiveMQ, RabbitMQ, Qpid, SonicMQ, etc.
- Different protocols as well
  - OpenWire, Stomp, AMQP, etc.
  - A broker can support several protocols, and translate/bridge one to the others
- Advanced Message Queuing Protocol (AMQP)
  - Language agnostic, can connect Java to NodeJS and C#
  - Very (most?) popular MOM
  - Another popular one is Kafka, but that is technically just a distributed log system

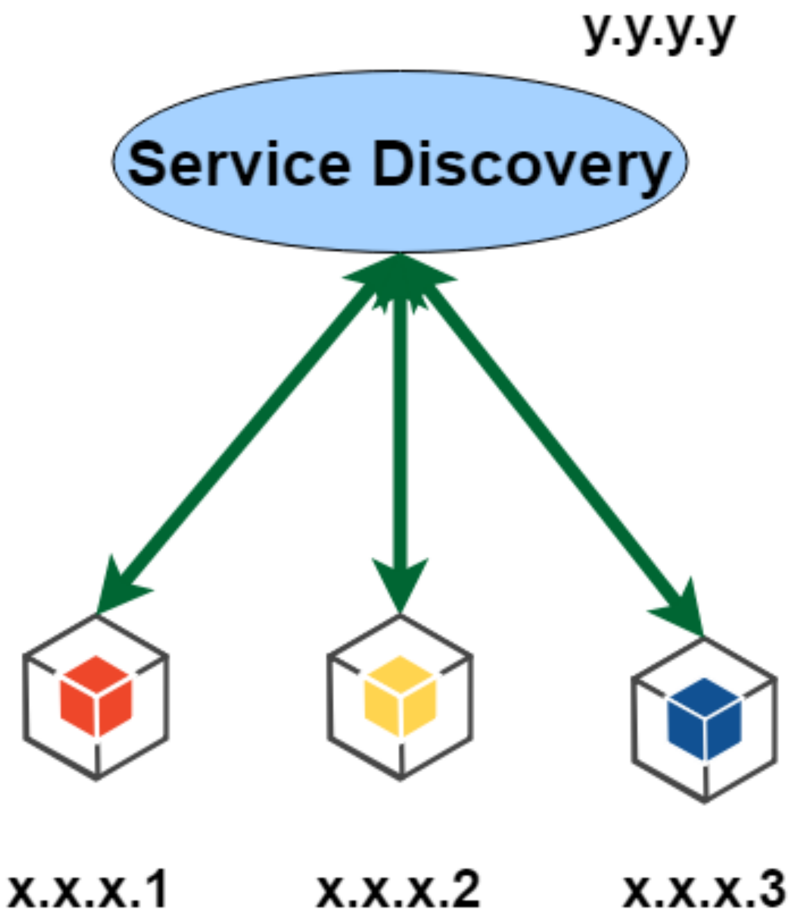
RabbitMQ

- Written in Erlang
- Implementing AMQP
- It is the MOM we will use in this course
- We will start it with Docker
- We will look at its details in a later class, not here

Service Discovery

- How does service X know the IP address of Y, if X wants to communicate with Y (eg, a REST call)?
  - Hardcoding the IP address of Y in X is not a viable option…
- *Service Registry:* a process/component that keeps track of the IP addresses of all running services
  - X will ask the service registry for the IP address of Y
  - IP address should not be hardcoded
- Different approaches for communications with registry and IP registration

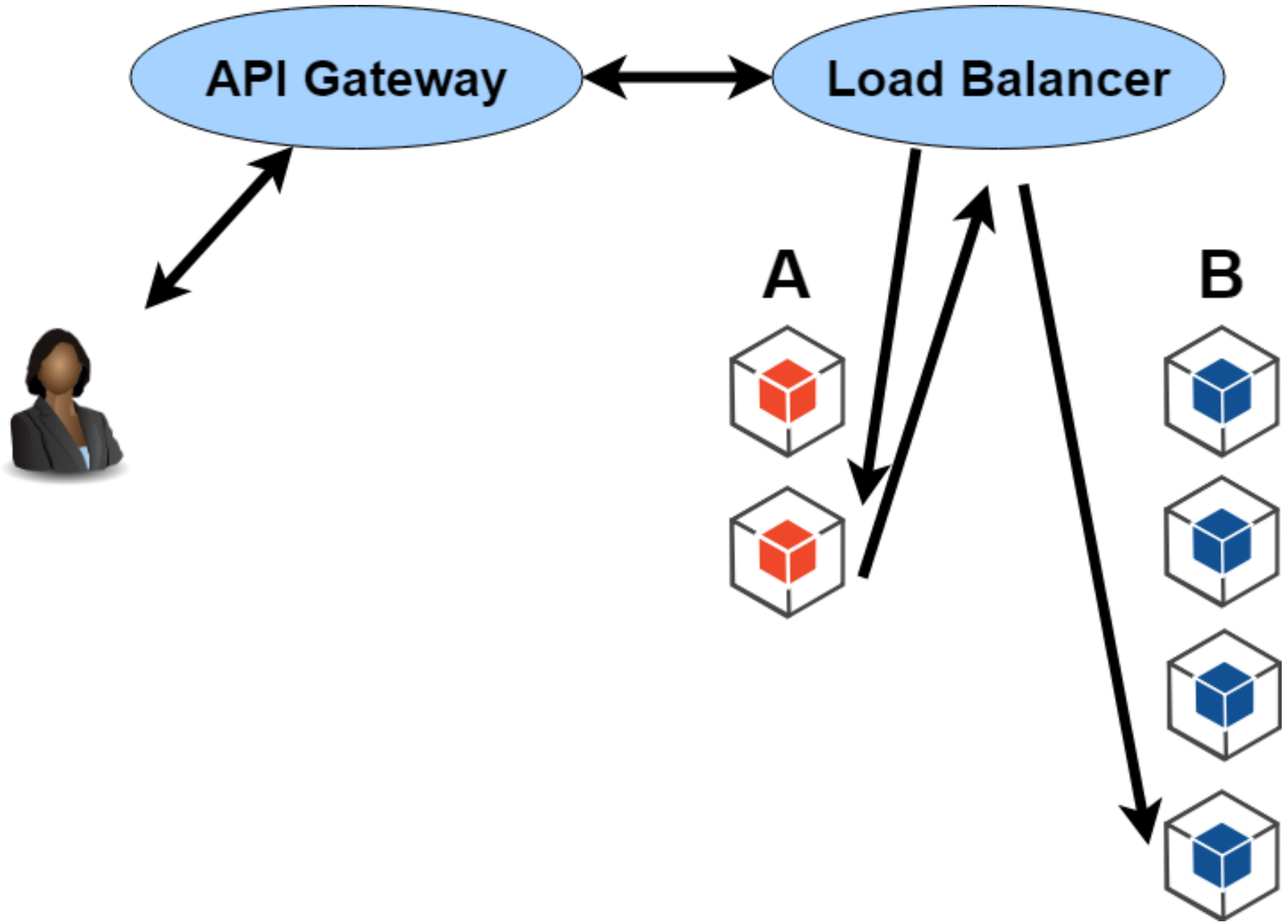
- The Service Discovery (SD) will be a running process
- All services will need to know the IP address or hostname of SD
- Services will have a name, e.g. *A*, *B* and *C*
- When service *A* starts, it will contact SD and states that it is running on given IP address
- It will then receive the IP addresses of all other current registered services
- Note: if a service is replicated, there will be different IP addresses for the same service name
  - this is also one of the reasons why we are not using DNS here
- If services leave or join, SD will inform all registered services about it, ie at each topology change
- To know if service is still reachable, need to send an heartbeat every N seconds (eg 30s)



Client-Side Load Balancer

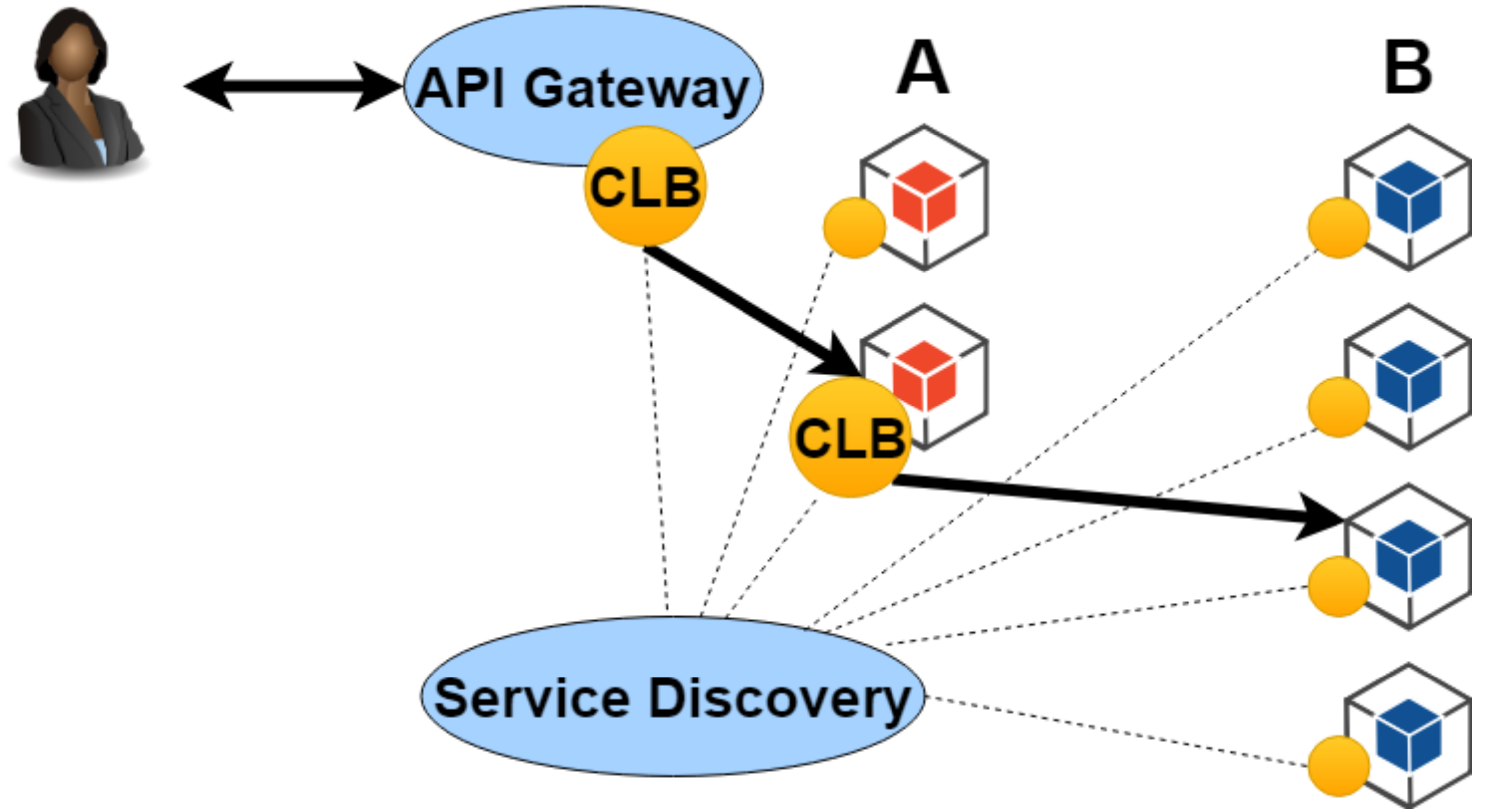
- A single load-balancer (LB) for all communications would be a major bottleneck
- Still need it for the API Gateway, but what about service-to-service communications?
- Client-side LB: not centralized, each service (including Gateway) is responsible to do the LB on each request
- But to do that, need to know the IP addresses of replicas of each single service… easy, ask the Service Discovery!

Single Load Balancer: Inefficient



- User makes a request, going through Gateway
- Needs to be routed to 1 of the 2 instances of service *A*
- To complete the request, *A* needs to get data from *B*, and so make a request to 1 of the 4 instances of *B*
- If always doing routing through Load Balancer, it becomes a bottleneck

With Client-Side Load Balancers

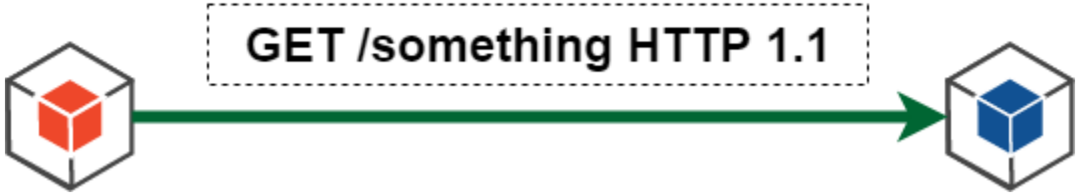


- Same scenario as before
- But, now, the instance of *A* (let's call it *A1*) asked by the Gateway knows IP addresses of all instances of *B*
- *A1* will decide to which of the 4 instances of *B* to ask to, each time choosing a different one (if Round-Robin)

Netflix Stack

- Netflix uses a lot of microservices
- Released many of their tools as open-source (most of them written in Java)
- Spring Cloud has direct support for such tools
- *Eureka*: for service discovery
- *Ribbon*: for client-side load balancer
- *Zuul*: for API Gateway… however this is deprecated in Spring Cloud, and we will use the new *Spring Cloud Gateway*

One-to-one communication, what if server is down?



- If destination is down, all next messages to it are wasted until the server is up again
- If client tries several times to connect, then you end up flooding and congesting the network with pointless messages
- Would be better to wait a bit, before trying to reconnect to the destination
- If messages are saved, and resent when destination is up, you do not want to send all the stored messages at the same time (otherwise destination could go down again)

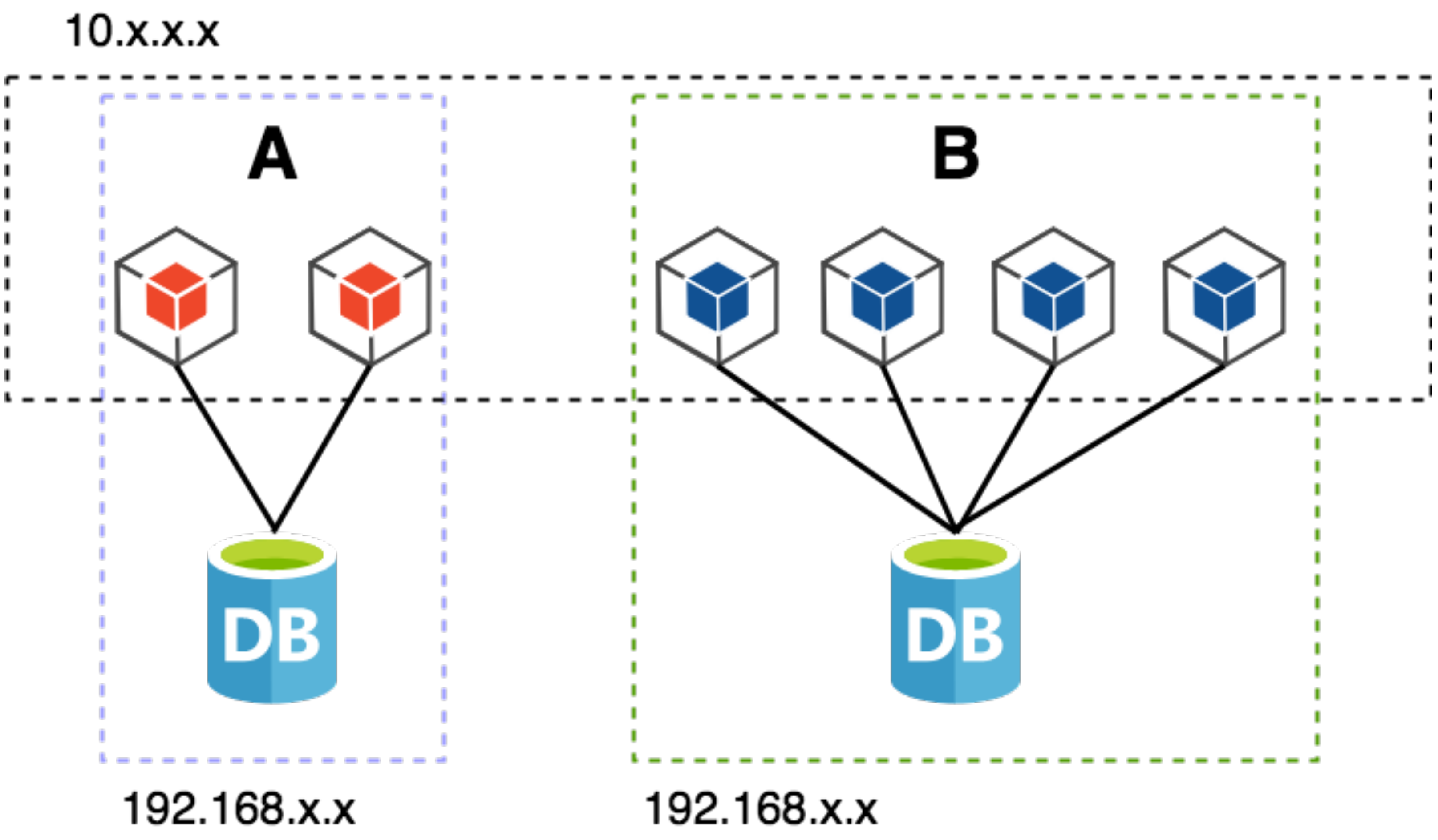
Circuit Breaker

- If too many connections to a server fail, stop ALL future attempt at connecting
- Can use a library (eg *Netflix Hystrix*) to wrap each call to external services
- Once the circuit breaker is on after several failures, it will periodically check if the server comes up again. If so, all communications are restored

Databases

- Services like REST APIs should be *stateless*
  - Can restart/stop at any time, and scale horizontally by replicated instances
- *State* saved in databases, running in separated processes
- How many databases?
- Services need to evolve (and be updated) *independently*… so each should have their own databases
- Replica instances of a same service will use the same database

Security: Different Virtual Networks



- Instances of *A* should only access own db, and not the one of *B* instances
- For security, could have separated VNs
- *A* can speak with *B* though, as on a same VN
- *Multihoming*: access to 2 or more networks

Conclusion

- MicroServices are extremely important and common in industry
- Aimed at *large systems*, that need to be maintained for years
  - think of Amazon, Netflix, Google, etc.
- **No Silver Bullet**
  - For small systems, monoliths are better
- Challenge: understand the benefits of microservices when all your school projects are actually “tiny” (even your BSc project)

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/microservice/discovery/***
- **advanced/microservice/gateway/***
- Study *Microservices From Design to Deployment*.