# Unit 4. Service Deployment. Docker

Network Information System Technologies

# Index

# 1. Goals

- Describe aspects to have into account when deploying distributed applications.
- Show the problems derived from dependencies, and how they can be addressed.
- Discuss deployment and its problems in a generic system.
- Know some deployment tools.
- Manage a specific deployment example.

# Index

# 2. Concept of deployment

- **Deployment:** Activities that allow a software system to be ready for its use.

  ⇒ Activities related with the installation, activation, upgrade and removal of components or the whole system.

# 2.1. Deployment of a distributed application

- A distributed application is a collection of heterogeneous components spread over a computer network
  - Many components, built by different developers
  - Those components may change in a fast and independent way (e.g., new versions, etc.)
- Application components should cooperate → there are dependences among them
- Those nodes may be heterogeneous (different hardware, operating systems, etc.), but each component has its own execution requirements
- There may be additional security requirements (privacy, authentication, etc.)

# 2.2. Deployment example

Let us assume the broker architecture developed in the second lab project

- It is composed of 3 autonomous agents(client, broker, worker)
  - They may have been developed in diferent programming languages, by diferent programmers
- Variable amount of instances in each component (e.g., several clients and several workers). Each instance...
  - ...may be started/stopped/restarted independently of the rest of instances
  - ...may fail on its own, independently of the others
  - ...has its own location (i.e., is placed in a given node)

# 2.2. Deployment example

- Dependences and requirements
  - Each client has to...
    - Know the broker location (IP address and frontend port)
    - Have a unique identity
  - Each worker has to...
    - Know the broker location (IP address and backend port)
    - Have a unique identity
  - All components have a given set of development requirements
    - JavaScript as their programming language
    - NodeJS as the JavaScript interpreter
    - ZeroMQ as the communication library

# 2.2. Deployment example: Manual deployment

- Copy the source code of each component in those computers where each instance should be run
  - We should guarantee that in those nodes the software base (NodeJS, ZeroMQ...) is correctly installed in the proper version
- Launch every instance of each component in the appropriate order (broker → workers → clients)
  - In the command line needed for starting each component, the appropriate arguments should be stated (e.g., IP address and port number to refer to the broker, agent identity, etc.)

# Index

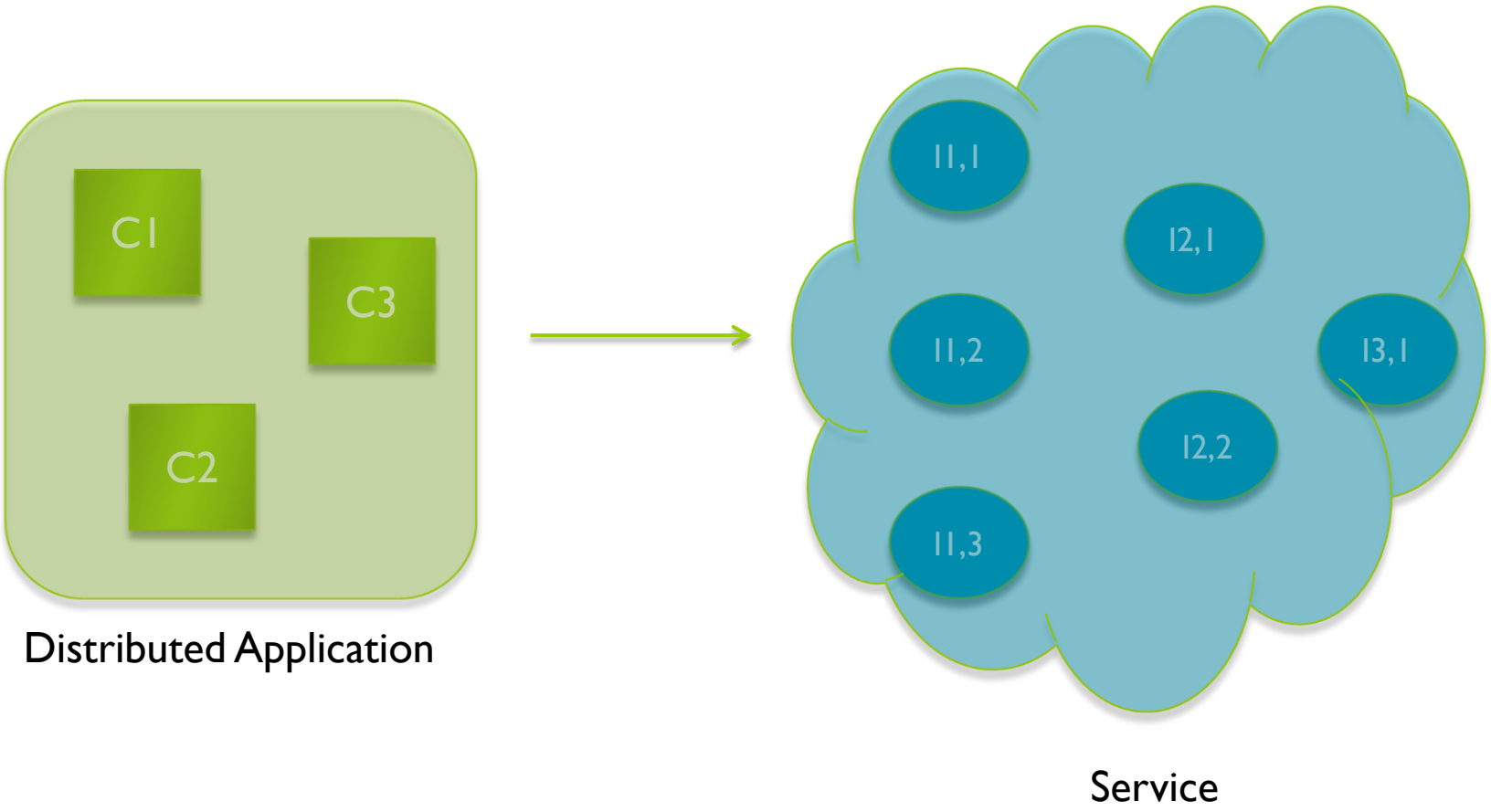
# 3. Service deployment

- We develop distributed applications in order to provide **services** (functionality) to remote clients
  - Application + Deployment = Service
- Every service sets a SLA (Service Level Agreement)
  - Functional definition (what it does)
  - Throughput (maximum req/sec, response time...)
  - Availability (percentage of time that the service remains active)
    - Although there are **ephemeral** services (i.e., short-lived, available for a short time interval), we focus on continuously available (i.e., **persistent**) services. For instance: gmail, Dropbox...
- **Service deployment** = installation, activation, upgrade and adaptation of the service.

# 3. Service deployment

- Installation and activation.- Software execution
  - Software dependence resolution (e.g., needed libraries)
  - Software configuration (compatible versions)
  - Determine the amount of instances of each component and their distribution in different nodes
  - Agent dependence resolution (e.g., port numbers)
  - Set the appropriate component start order
- Deactivation.- Stop the software system in an ordered way
- Upgrade.- Replace components (newer versions)
- Adaptation.- (With service continuity) after:
  - Failure/recovery of an agent
  - Agent configuration changes
  - Scaling (reaction to workload changes)

# 2. Services




Distributed Application

Service

# Index

# 4. Deployment automation

A large scale deployment cannot be manually managed → An automation tool is needed

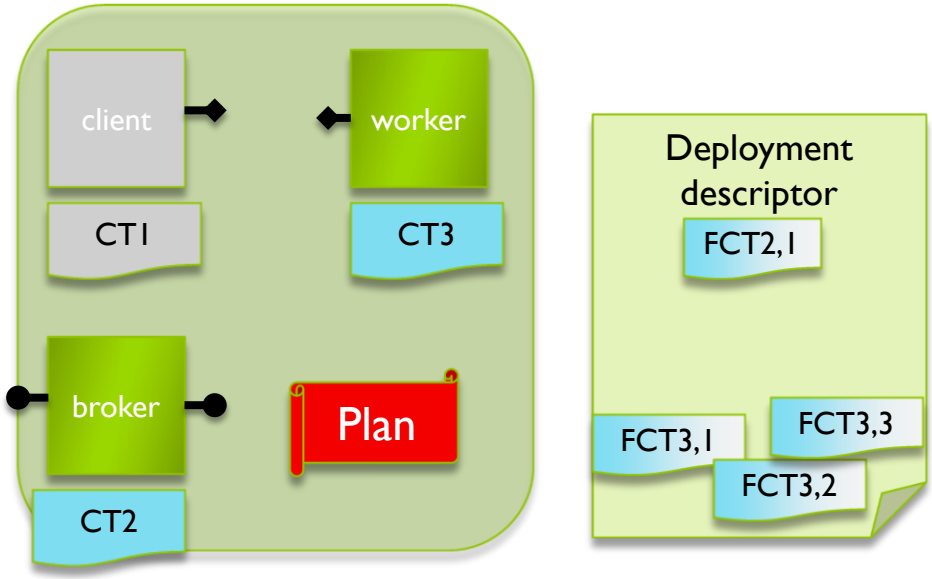
- Configuration of every component
  - File with a list of configuration parameters and dependency descriptions
  - The tool creates a specific configuration por every component instance
- Global configuration plan
  - Inter-component connection plan (list of public endpoints, list of dependences)
  - The location of each instance is decidec
  - Dependency resolution or binding (endpoints of interacting agents are connected, including dependences on external services)

# 4. Deployment automation: Example

- Several client instances
  - Arguments: frontendURL, id
  - Dependency on a *broker*
- One broker instance
  - Arguments: frontendPort, backendPort
- Several worker instances
  - Arguments: backendURL, id
  - Dependency on a *broker*
- Plan
  - Their start order is: *broker*, *workers*, *clients*
  - The broker endpoints are: frontend (external) and backend (internal)
  - There are no dependencies on other external services

# 4. Deployment automation: Example

Distributed application

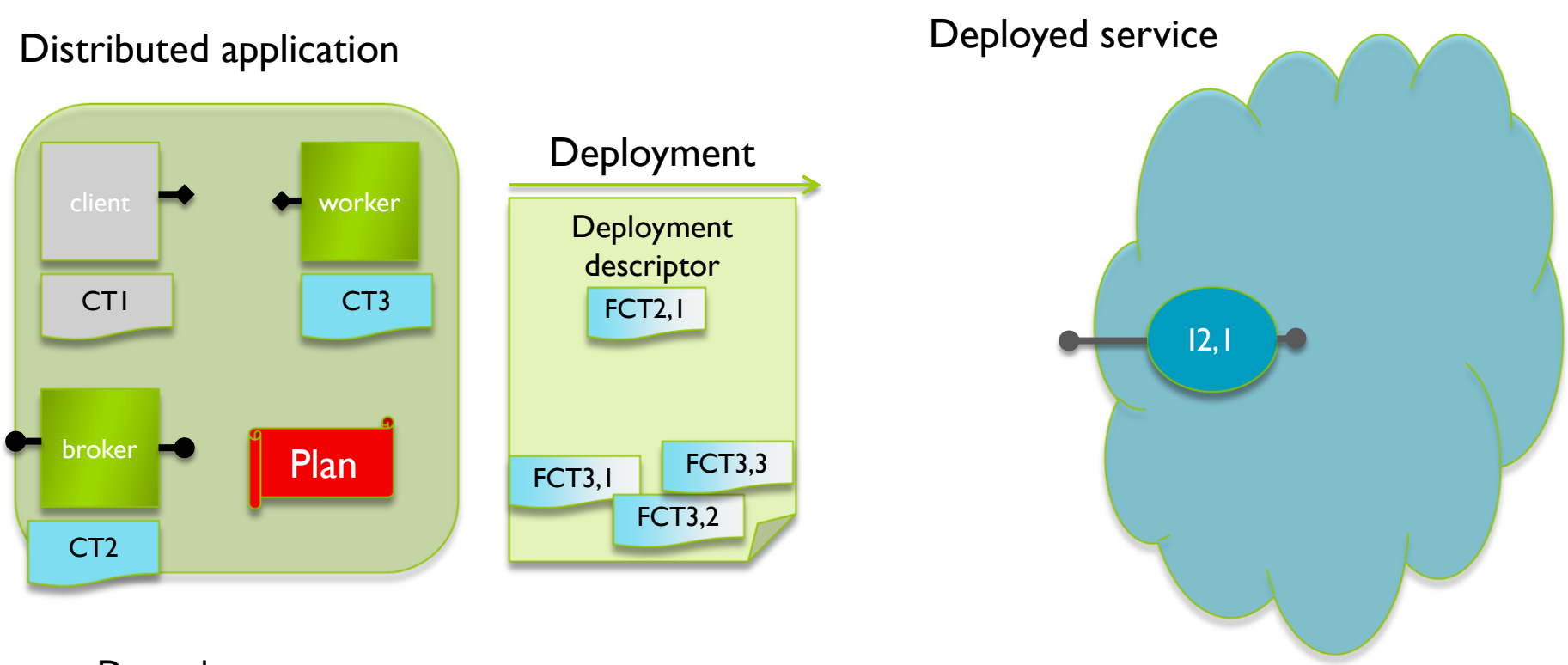



- Dependency
- Endpoint
- Bound dependency
- Bound endpoint
- Exposed service endpoint

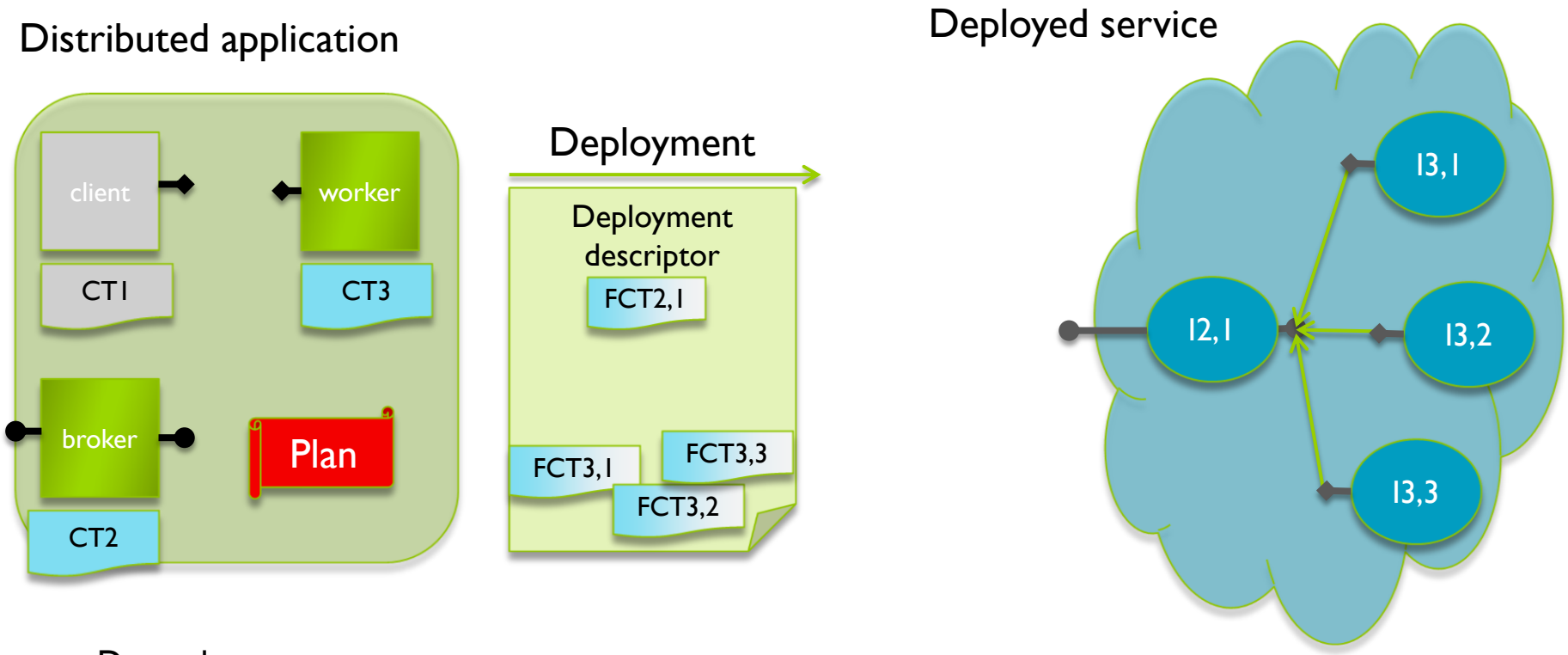
CTi — Configuration template for component i

FCTi,j — Filled configuration template for instance j of component i

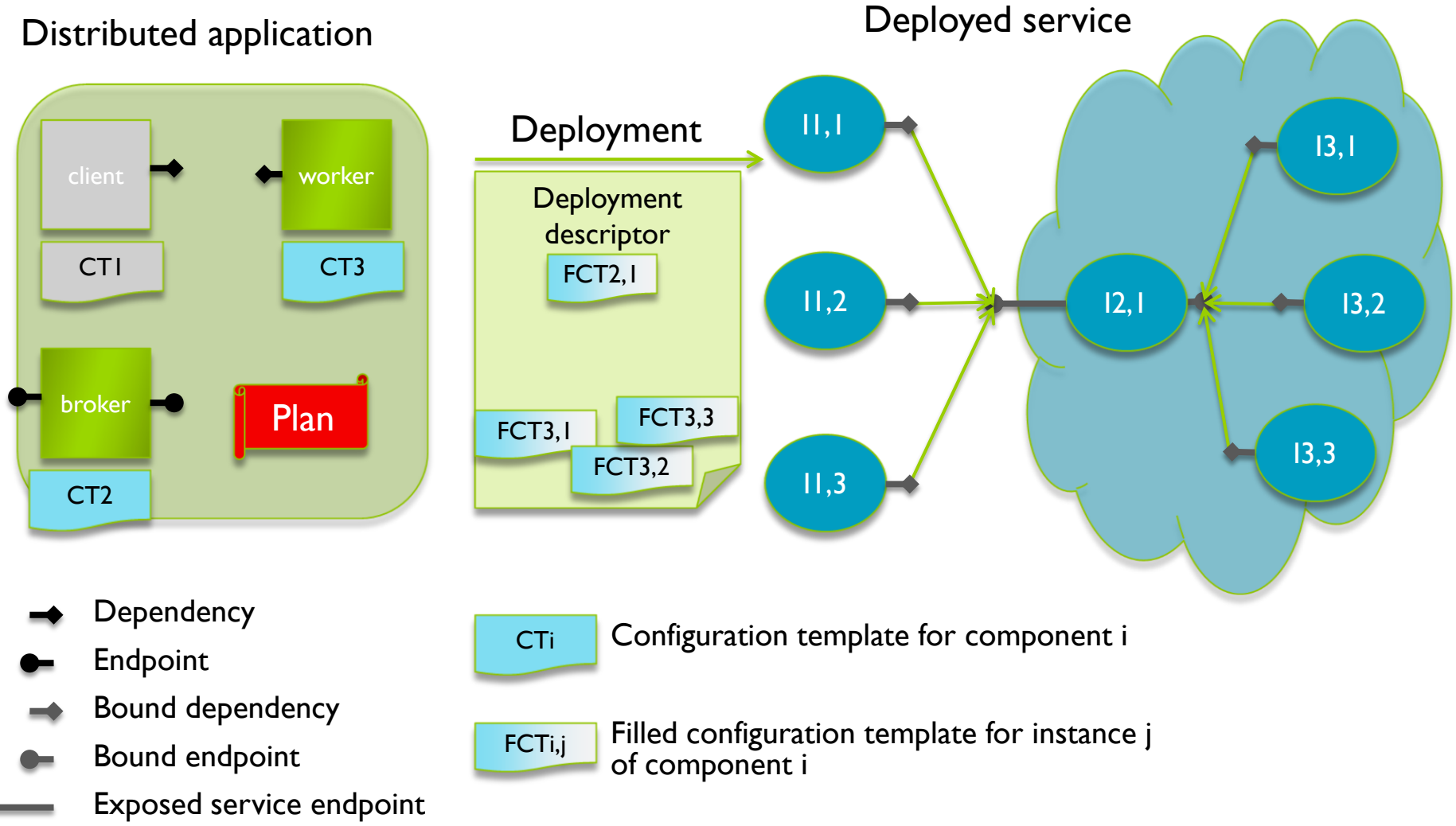
# 4. Deployment automation: Example



# 4. Deployment automation: Example



# 4. Deployment automation: Example



# 4. Deployment automation

Dependency resolution. Options:

1. The program code defines how dependences must be resolved
   - E.g., reading data from a configuration file, or receiving data from a socket
   - Low level
2. **Dependency injection** (recommended)
   - The program code exposes local names for relevant interfaces
   - The container fills those variables with object instances
     - This generates a graph with all service component instances
     - The edges of the graph are links dependency-endpoint
       - As it has been shown in the previous slides

# Index

# 5.1. Deployment in the cloud: IaaS

- Based on hardware virtualisation
  - Virtual machines in different "sizes" (i.e., hardware characteristics)
  - Flexibility in resource allocations
- However, there are some deployment limitations
  - Allocation decisions cannot be automated (low level)
    - Number of instances per component, location, VM type...
  - No choice on networking characteristics (latency, bandwidth...)
  - Inaccurate failure models
    - Not really independent failure modes
    - Limited help in recovering failed instances

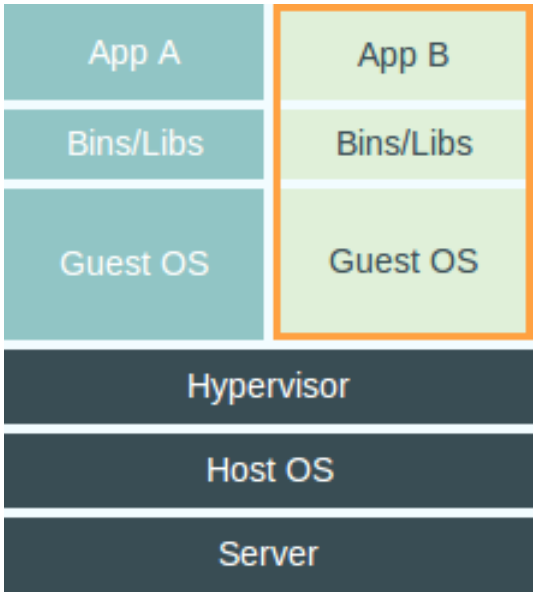
# 5.2. Deployment in the cloud: PaaS

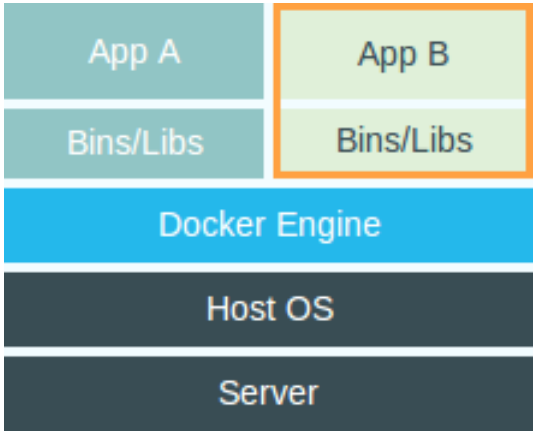
- SLA as the central element → SLA parameters for every component
  - Trying to achive deployment automation
    - Deployment plans derived from the SLA
    - Upgrading and configuration plans
  - Current state
    - Still with a limited automation (initial deployment and autoscaling of simple services are automated, but service upgrading complying with its SLA is not completely supported yet)
    - Microsoft Azure is one of the best providers in this regard

# Index

# 6. Containers

- **Provisioning =** booking the infrastructure needed by a distributed application.
  - Resources needed for component intercommunication.
  - Specific resources needed by each instance (CPU+ main memory + secondary storage). Alternatives:
    - Instances on **VM** → OS + Libraries
    - Instances on **containers** (light version of a VM) → Libraries
      - The host OS is used



# 6. Containers

Let us assume the usage of containers instead of VMs

- Lower flexibility
  - Software in each instance must be compatible with the host OS
  - Container isolation is not perfect
- Uses fewer resources
  - E.g., let us assume 100 instances of a component that needs 900MB (OS) + 100MB (libraries + program)
    - With VMs: 100*(900MB + 100MB) = 100GB
    - With containers: 900MB + 100*100MB = 10.9GB
  - Space and time (e.g., in order to install the image) are saved!
- Easier deployment (automation tools with configuration files)
- May be applied in many scenarios

# 6. Containers: Docker

- Its Dockerfile configuration file automates the deployment of every instance
- It supports a version control system (Git)
- Besides the native file system, it defines a read-only file system that may be shared among containers
- Allows cooperation among developers through a public repository

NOTE.- We assume that the host OS is Linux (although there is a Docker version for Windows)

# Index

# 7. Docker

The Docker ecosystem consists of 3 components:

1. **Images** (builder component) Read-only templates taken as a basis for container instantiation.
   - E.g., we may build an image that provides Linux+NodeJS+zmq, called **centos-zmq**
2. **Containers** (executor component) Created from images. They maintain all items needed for executing an instance.
   - E.g., in order to run our second lab project, each instance may be run in a container created from the **centos-zmq** image
3. **Repository** (distributor component) Global store where images can be pushed and pulled (hub.docker.com)
   - E.g., we may push the **centos-zmq** image there, and everyone may pull and use it for generating containers

# 7.1. Docker: Image

- In the repository, there are predefined images for different Linux distributions (e.g., image **Centos:7.4.1708**)
- New image = baseImage + instructions. Examples:
  - `Centos:7.4.1708` + instr. to install node → `centos-nodejs`
  - `centos-nodejs` + instr. to installzmq → `centos-zmq`
  - `centos-zmq` + … → build images for each component (*client*, *broker*, *worker*)
- Docker uses console commands
- General structure: **docker** `action options arguments`
  - Information on local images: **`docker images`**
  - Information on a given image: **`docker history`** `imageName`

# 7.2. Docker. Container

- Create and start a container from an image

```
docker run options image initialProgram
```

- E.g., `docker run -i -t centos bash` downloads the image `centos`, creates a container, allocates a file system, allocate a networking interface and an internal IP addrees, and runs `bash`
  - Options `-i -t` choose the interactive mode (the host console remains open and connected to the container)
- Now, we may modify the container state using command in an interactive session at the console
- Finally, we may create a new image saving the current state of the container

```
docker commit containerName imageName
```

# 7.2. Docker. Command groups

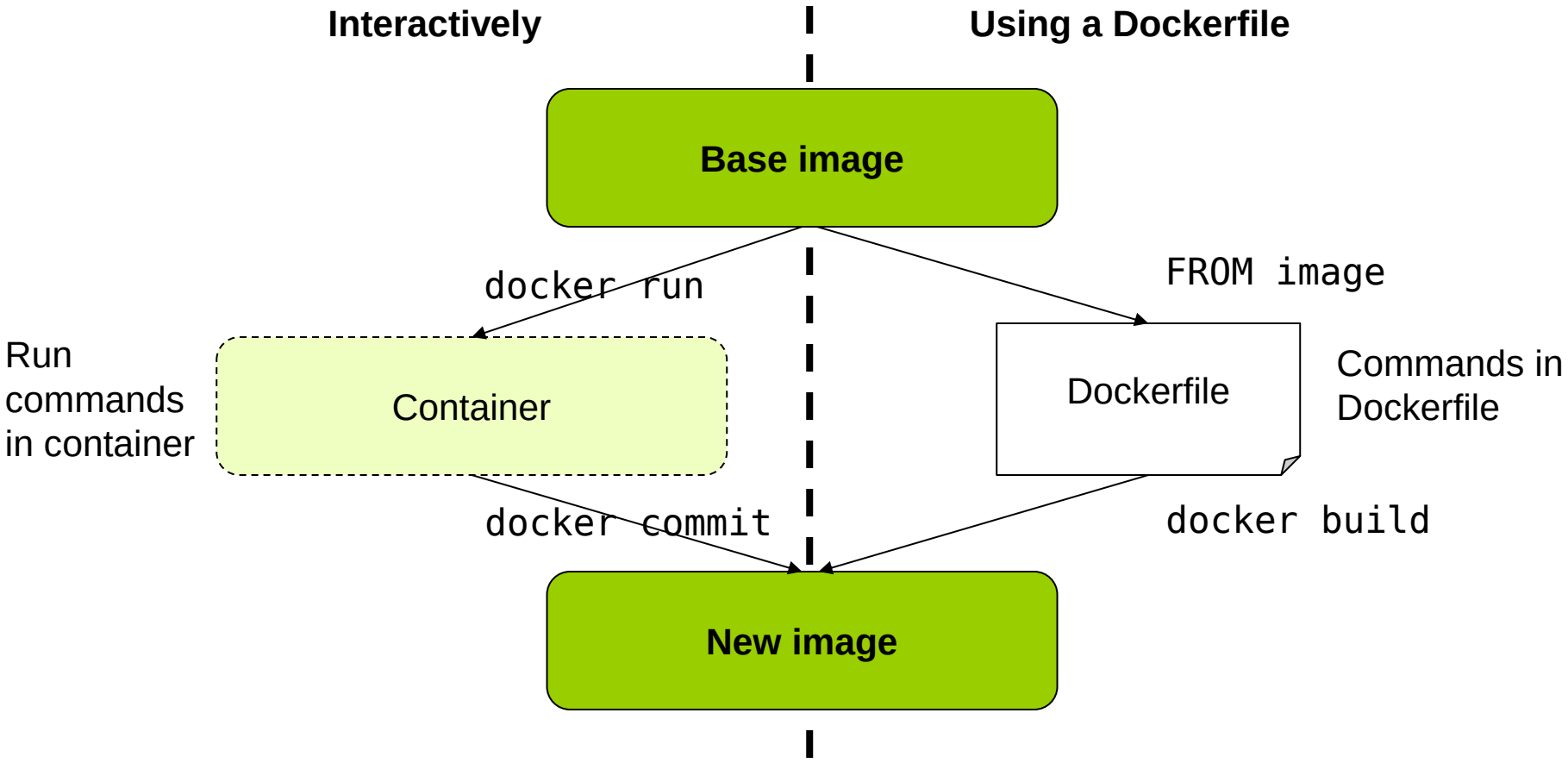
| Group | Description |
|---|---|
| **config** | Configuration management |
| **container** | Operations with containers |
| **context** | Contexts for distributed deployment (k8s, …) |
| **image** | Image management |
| **network** | Network management |
| **service** | Service (i.e. multiple containers that run the same image) management in distributed deployments (e.g. with docker-compose or docker swarm) |
| **system** | Global management |
| **volume** | Secondary storage management |

# 7.3. Docker. Main commands

1. Life cycle management (in the container group)
   - docker **commit | run | start | stop**
2. Informative (multiple groups)
   - docker **logs | ps | info | images | history**
3. Repository access (image group)
   - docker **pull | push**
4. Other (container group)
   - docker **cp | export**

# Index

# 8. Image creation



# 8. Image creation

- Two alternatives for image creation:
  - Interactively
    - Choose a base image and start a container

      ```
      docker run -i -t baseImage initialProgram
      ```

    - Modify interactively the container state
    - Save the current state as a new image:

      ```
      docker commit containerID newImageName
      ```

  - Create a new image from the instructions saved in a Dockerfile text file

    ```
    docker build -t newImageName pathToDockerfile
    ```

# 8.1. Interactive image creation. Example

- Image that may run NodeJS programs with 'zeromq'.
  - Step 1: Launch Docker using an interactive CentOS image (running bash):

```
$ docker run -i -t centos:7.4.1708 bash
```

  - Step 2: Run these commands in the container:

```
$ curl --silent --location https://rpm.nodesource.com/setup_10.x | bash -
$ yum install -y nodejs
$ yum install -y epel-release
$ yum install -y zeromq-devel
$ yum install -y make python gcc-c++
$ npm install zeromq@4
$ exit
```

  - From the host console, get the name and ID of that container: `docker ps -a`
  - Create the new image:

```
docker commit containerID newImageName
```

# 8.2. Image creation with Dockerfile. Example

1. This Dockerfile is written to disk:

```
FROM centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_10.x | bash -
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN npm install zeromq@4
```

2. Change the current directory to that where the Dockerfile is placed
3. Run this command:
   - `docker build –t tsr1718/centos-zmq .`

# 8.3. Docker. Dockerfile

- Each line starts with a command (in uppercase, by convention)
- The first instruction (in the first line) must be

  `FROM imageBase`
- `RUN command` runs that command in the container shell
- `ADD source destination`
  - Copy files from a source (URL, directory or file) to a path in the container
  - If source is a directory, then it copies all its contents. If it is a compressed file, then it is extracted in the destination
- `COPY source destination` is equal to `ADD`, but it does not extract compressed files
- `EXPOSE port` states the port number where the container listens to incoming requests

# 8.3. Docker. Dockerfile

- `WORKDIR path` specifies the container working directory for subsequent `RUN`, `CMD`, or `ENTRYPOINT` commands
- `ENV variable value` assigns value to an environment variable that may be used by the programs to be run in the container
- `CMD command arg1 arg2 ...` provides default values for the command and arguments to be run in the container
- `ENTRYPOINT command arg1 arg2 ...` runs that command when the container is started (and the container is terminated when that command ends)

There must be a single `CMD` or `ENTRYPOINT` in the Dockerfile (otherwise, only the last one is used)

# Index

# 9. Multiple components in a node. Example

- Example: Service that consists of a "broker" and a "worker" that may be replicated as many times as needed.
  - Almost identical to that seen in part 6.3 of lab 2.

```
// ROUTER-ROUTER request-reply broker in NodeJS
const zmq = require('zeromq')
let cli=[], req=[], workers=[]

let args = process.argv.slice(2)
let fePortNbr = args[0] || 9998
let bePortNbr = args[1] || 9999

let sc = zmq.socket('router') // frontend
let sw = zmq.socket('router') // backend

sc.bind('tcp://*:'+fePortNbr)
sw.bind('tcp://*:'+bePortNbr)
```

```
sc.on('message',(c,sep,m)=> {
  if (workers.length==0) {
    cli.push(c); req.push(m)
  } else {
    sw.send([workers.shift(),'',c,'',m])
  }
})

sw.on('message',(w,sep,c,sep2,r)=> {
  if (c=='') {workers.push(w); return}
  if (cli.length>0) {
    sw.send([w,'',
      cli.shift(),'',req.shift()])
  } else {
    workers.push(w)
  }
  sc.send([c,'',r])
})
```

# 9. Multiple components in a node. Example

- Worker and client programs.
  - Similar to those in lab 2, with command-line arguments and default values.

```
// worker in NodeJS with URL & id arguments
const zmq = require('zeromq')
let req = zmq.socket('req')

let args = process.argv.slice(2)
let backendURL = args[0] ||
'tcp://localhost:9999'

let myID = args[1] ||
'WID'+parseInt(Math.random()*5000)
let replyText = args[2] || 'resp'

req.identity = myID
req.connect(backendURL)
req.on('message', (c,sep,msg)=> {
  setTimeout(()=> {
    req.send([c,'',replyText])
  }, 1000)
})
req.send(['','',''])
```

```
// client in NodeJS with URL & id arguments
const zmq = require('zeromq')
let req = zmq.socket('req')

let args = process.argv.slice(2)
let brokerURL = args[0] ||
'tcp://localhost:9998'

let myID = args[1] ||
'CID'+parseInt(Math.random()*10000)
let myMsg = args[2] || 'Hola'

req.identity = myID
req.connect(brokerURL)
req.on('message', (msg)=> {
  console.log('resp: '+msg)
  process.exit(0);
})
req.send(myMsg)
```

# 9. Multiple components in a node. Example

- With multiple components, some dependencies arise:
  - Clients need to know the frontend URL (IP address and port)
  - Workers need to know the backend URL (IP address and port)
- But we do not know the IP address until the broker container is started
- Manual alternative
  - Once the broker container is started, we get its IP address
  - We manually update those values in the client and worker Dockerfiles in order to generate their correct images
  - Start then workers and clients
- Automation:
  - A deployment plan is defined = Decription of components, properties and relations
  - We use a tool that completes the deployment reading that plan.

# 9.1. Manual method: Broker

- In a directory, we copy broker.js and this Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./broker.js broker.js
EXPOSE 9998 9999
CMD node broker
```

  - Ports to be used: frontend=9998, backend=9999
- Build the image: **docker build –t broker .**
- Start the broker container: **docker run –d broker**
- Find out its IP address
  - **docker ps –a** in order to the container ID
  - **docker inspect ID** in order to get its IP address

# 9.1. Manual method: Worker

- In another directory, we copy worker.js and this Dockerfile (assuming that the broker IP address is a.b.c.d)

```
FROM tsr1718/centos-zmq
COPY ./worker.js worker.js
CMD node worker tcp://a.b.c.d:9999
```

  - Take a look at the previous slide on how to get the a.b.c.d address
- In this directory, run: `docker build -t worker .`
- Start a worker with `docker run -d worker`
  - We may start as many instances as needed

# 9.1. Manual method: Local client

- In another directory, we copy client.js and write this Dockerfile:

```
FROM tsr1718/centos-zmq
COPY ./client.js client.js
CMD node client tcp://a.b.c.d:9998
```

  - In that directory, we run: `docker build -t client .`
  - Start a client with: `docker run -d client`
    - We may start as many instances as needed

# 9.1. Manual method: Remote client

- We start the broker with `docker run -p 8000:9998 -d broker`
  - Option `-p hostPort:containerPort` allows external accesses to the frontend port (through port 8000 in the host)
  - Option `-d` starts the container in background mode
- Clients may be started directly, without containers. They must connect to `tcp://hostIPaddress:8000`

# 9.2. Automated method. Example

- Create broker subdir., copy broker.js and this Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./broker.js broker.js
EXPOSE 9998 9999
CMD node broker
```

- Create worker subdir., copy worker.js and this Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./worker.js worker.js
CMD node worker $BROKER_URL
```

- Create client subdir., copy client.js and this Dockerfile

```
FROM tsr1718/centos-zmq
COPY ./client.js client.js
CMD node client $BROKER_URL
```

# 9.2. Automated method. Example

- Copy the text on the right to a `docker-compose.yml` file
- This command...

```
docker-compose up -d
```

  - builds all images
  - starts one instance of each one in the correct order
- We may start n instances of service X with

```
docker-compose up -d --scale X=n
```

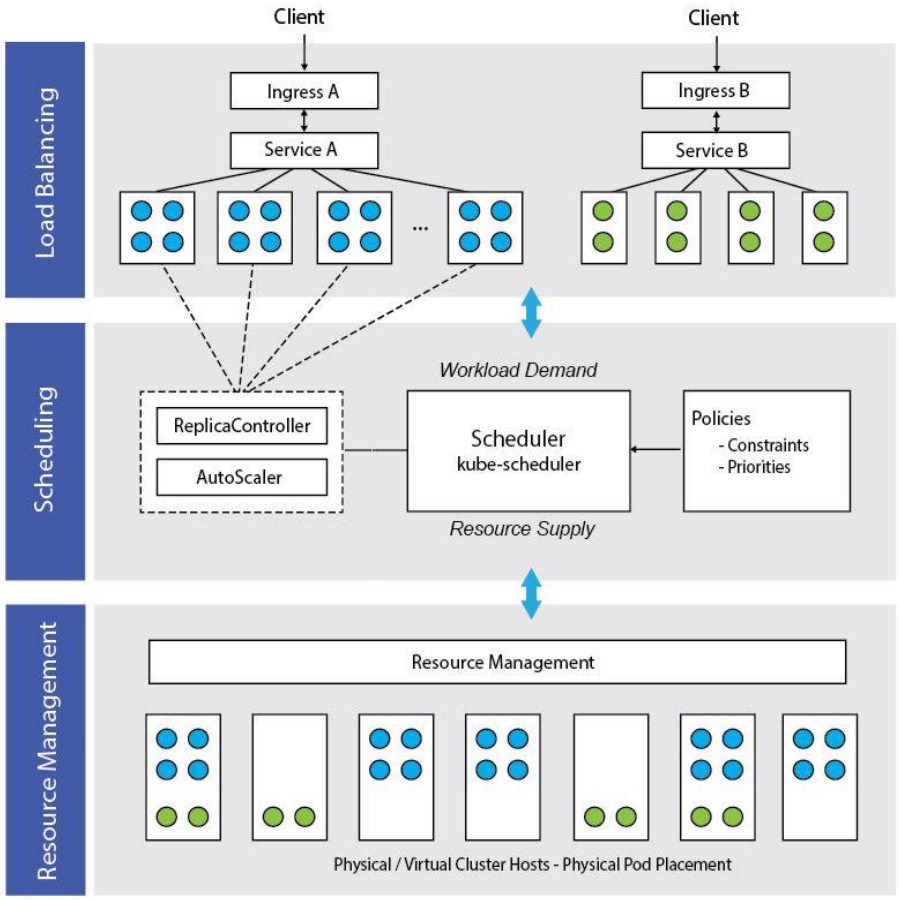
- There are other commands
  - Read the guide of this unit

```
version: '2'
services:
    cli:
        image: client
        build: ./client/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9998
    bro:
        image: broker
        build: ./broker/
        expose:
            - "9998"
            - "9999"
    wor:
        image: worker
        build: ./worker/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9999
```

# Index

# 10. Multiple components in different nodes

- Docker-compose only manages component in a single host
- If we need multiple hosts → the best option is **kubernetes**
  - It is a container orchestrator, but it does not depend on Docker
    - The guide for this unit provides a general description of its elements



# 10. Multiple components in different nodes

Kubernetes. Main elements:

- **Cluster** and node (real or virtual)
- **Pod**: smallest deployment unit
  - It includes containers that share a *namespace* and storage volumes
- **Replication controllers**: manage the life cycle of a group of pods,
  - ensuring that a specified number of instances is running,
    - scaling, replicating and recovering pods
- **Deployment controllers:** upgrade the distributed application
- **Service**: defines a set of pods and their public access
- **Secrets** (credentials management)
- **Volumes** (persistence)

# Index

# 11. Learning Results

- After this unit, the student must be able to
  - Know in some detail the salient aspects to be considered in the deployment of distributed applications
  - Understand the problems posed by dependencies, and the ways to address them
  - Use Docker in order to deploy distributed services

# Index

# 12. References

- Inversion of Control/Dependency injection
  - http://martinfowler.com/articles/injection.html
  - http://www.springsource.org/
- www.docker.com (Official Docker website)
  - docs.docker.com/userguide/ (**Official documentation**)
  - docs.docker.com/compose/ (Compose)
- github.com/wsargent/docker-cheat-sheet (Summary of Docker)
- http://kubernetes.io
- 12factor.net/ ("*The twelve-factor app*" methodology)