Università degli Studi di Roma “Tor Vergata”
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Microservices and Serveless Computing

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2018/19

Valeria Cardellini

References and resources

• J. Lewis, M. Fowler, “Microservices”,
  https://martinfowler.com/articles/microservices.html


• M. Roberts, “Serverless Architectures”,
  https://martinfowler.com/articles/serverless.html

• https://martinfowler.com/microservices/
• https://microservices.io
Microservices

- A “new” emerging architectural style for distributed applications that structures an application as a collection of loosely coupled services
- Not so new: derive from SOA
  - But with some significant differences
- Address how to build, manage, and evolve architectures out of small, self-contained units
  - *Modularization*: decompose app into a set of independently deployable services, that are loosely coupled and cooperating and can be rapidly deployed and scaled
  - Services equipped with dedicated memory persistence tools (e.g., databases)

Service Oriented Architecture

- **Service Oriented Architecture (SOA)**: paradigma architetturale per progettare sistemi sw distribuiti lascamente accoppiati
- Definizione OASIS: *paradigma* per l'organizzazione e l’utilizzazione di risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per *offrire, scoprire, interagire ed usare* le capacità di produrre gli effetti voluti in modo consistente con presupposti e aspettative misurabili
- Proprietà (secondo W3C, [http://www.w3.org/TR/ws-arch/](http://www.w3.org/TR/ws-arch/))
  - Vista logica
  - Orientamento ai messaggi e alla descrizione
  - Granularità dei servizi, orientamento alla rete, neutralità della piattaforma
Service Oriented Architecture (2)

- Tre entità che interagiscono tra loro
  - Service requestor o consumer: richiede l’esecuzione di un servizio
  - Service provider: implementa il servizio e lo rende disponibile
  - Service registry: offre un servizio di pubblicazione e di ricerca di servizi disponibili

Web services

- Web services: implementation of SOA
- W3C definition:
  - A web service is a software system designed to support interoperable machine-to-machine interaction over a network
  - It has an interface described in a machine-processable format (specifically WSDL)
  - Other systems interact with the web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP
Web services

• More than 60 related standards and specifications
  – Basic ones:
    • To describe: WSDL (Web Service Description Language)
    • To communicate: SOAP (Simple Object Access Protocol)
    • To register: UDDI (Universal Description, Discovery and Integration)
  – To define business processes: BPEL (Business Process Execution Language), BPMN (Business Process Model and Notation), …
  – To define the SLA: WSLA, …

• Many technologies
  – Among which ESB (Enterprise Service Bus), an integration platform that provides fundamental interaction and communication services for complex applications

SOA vs. microservices

• Heavyweight vs. lightweight technologies
  – SOA tends to rely strongly on heavyweight middleware (e.g., ESB), while microservices rely on lightweight technologies

• Protocol families
  – SOA is often associated with web services protocols
    • SOAP, WSDL, and WS-* family of standards
  – Microservices typically rely on REST and HTTP

• Views
  – SOA mostly viewed as integration solution
  – Microservices are typically applied to build individual software applications
Microservices: benefits

- Increase software agility
  - Each microservice becomes an independent unit of development, deployment, operations, versioning, and scaling
  - Exploit container-based virtualization

- Faster delivery

- Improved scalability

- Greater autonomy

Microservices and containers

- Microservices as ideal complementation of container-based virtualization
  - Package each service as a container image and deploy each service instance as a container
  - Manage each container at runtime (scaling and or migrating it)

- Pros:
  - Service instance scaling out/in by changing the number of container instances
  - Service instance isolation
  - Resource limits on service instance
  - Build and start rapidly

- Cons:
  - Need container orchestration to manage the multi-container app
Microservices and scalability

- How to improve service scalability?
- State is complex to manage and scale
- Stateless services scale better and faster than stateful services

Stateless vs. stateful service

- Stateful service: multiple instances of a scaled-out service need to synchronize their internal state to provide a unified behavior
- Issue: how can a stateful service that is scaled-out maintain a synchronized internal state?

Stateless vs. stateful service

- Stateless service: state is handled external of service to ease its scaling out and to make the application more tolerant to service failures.

Integration

- Let us consider two issues related to the integration of microservices:
  - Synchronous vs. asynchronous communication
  - Orchestration vs. choreography
Synchronous vs. asynchronous

• Should communication be synchronous or asynchronous?
  – Synchronous: request/response style of communication
  – Asynchronous: event-drive style of communication

• Synchronous communication
  – Typical RESTful design patterns

• Asynchronous communication
  – MOM systems are often used to realize microservice applications with an event-drive style of communication

Orchestration and choreography

• Web services (and microservices) can be combined to define a new added-value application in two ways:
  – Orchestration
  – Choreography

• **Orchestration**: centralized approach
  – A single centralized process (the orchestrator) that coordinates the interaction among different services
  – The orchestrator is responsible for invoking and combining the services, which can be unaware of the composition
Orchestration and choreography

- **Choreography**: decentralized approach
  - A global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints

- Orchestration is simpler but
  - Single point of failure
  - Tight coupling

---

Example: orchestration and choreography

- Example: customer creation
  - The process for creating a new customer

Microservices: orchestration and choreography

- In a microservice architecture choreography is preferred over orchestration
- Pros
  - Lower coupling
  - Increased flexibility and ease of changing
- Cons
  - Extra work to monitor and track the processes across system boundaries
Example of microservices application

- E-commerce application that takes orders from customers, verifies inventory and available credit, and ships them
- Components: user interface along with some backend services for checking credit, maintaining inventory and shipping orders

Some large-scale examples

- Netflix and Twitter: 500+ microservices (in 2015)
How to decompose?

- How to decompose the application into microservices?
- Mostly an art, no winner strategy but rather a number of strategies:
  - Decompose by **business capability** and define services corresponding to business capabilities
    - E.g., Order Management is responsible for orders
  - Decompose by **domain-driven design (DDD) subdomain**
    - E.g., Order Management, Inventory, Product Catalogue, Delivery
  - Decompose by **use case** and define services that are responsible for particular actions
    - E.g., Shipping Service is responsible for shipping complete orders
  - Decompose by **nouns or resources** and define a service that is responsible for all operations on entities/resources of a given type
    - E.g., Account Service is responsible for managing user accounts

Microservice technologies timeline

From "Microservices: The Journey So Far and Challenges Ahead".
Generations: at the beginning

• 4 generations of microservice architectures

• 1st generation based on:
  - **Container-based virtualization**
  - **Service discovery** (e.g., Eureka, ZooKeeper and etcd)
    • Let services communicate with each other without explicitly referring to their network locations
  - **Monitoring** (e.g., Graphite, InfluxDB and Prometheus)
    • Enable runtime monitoring and analysis of the behavior of microservice resources at different levels of detail

Generations: container orchestration

• Then, **container orchestration**
  - E.g., Kubernetes, Apache Swarm
  - Automate container allocation and management tasks, abstracting away the underlying physical or virtual infrastructure from service developers
  - But failure-handling mechanisms still implemented in services source code

![Diagram of container orchestration](image)
Generations: service discovery and fault tolerance

- **2nd generation** based on discovery services and fault-tolerant communication libraries
  - E.g., Consul (service discovery), Finagle and Hystrix (communication)
  - Let services communicate more efficiently and reliably

Generations: sidecar and service mesh

- **3rd generation** based on sidecar (or service proxy or ambassador) technologies (e.g., Prana and Envoy)
  - Encapsulate communication-related features such as service discovery and use of protocol-specific and fault-tolerant communication libraries
  - Goal: to abstract them from service developers, improve swrisusability and provide homogeneous interface
Generations: serverless

- 4th generation based on serverless computing and FaaS (e.g., AWS Lambda) to further simplify microservice development and delivery

Serverless and FaaS

- Serverless computing
  - Cloud computing model which aims to abstract server management and low-level infrastructure decisions away from users
  - Users can develop, run and manage application code (i.e., functions), without no worry about provisioning or scaling computing resources
  - Runtime environment is fully managed by Cloud provider
  - Serverless: functions still run on “servers” somewhere but we don’t care

- Function as a Service (FaaS) often as synonym of serverless
  - Although some discussion
Serverless and FaaS

• Characteristics
  – Ephemeral compute resources (may only last for one invocation)
  – Automated (i.e., zero configuration) elasticity
  – True pay per use (i.e., pay only for consumed compute time)
  – Event-driven

• Major Cloud providers now offer FaaS
  – AWS Lambda
  – Azure Functions
  – Google Cloud Functions
  – IBM BlueMix OpenWhisk
  – Oracle tbd

FaaS: example

• The “Hello World” FaaS example from Google
  – HTTP request written in Node.js that displays “Hello World”
    or “Hello (name)” if you pass in a parameter

/**
 * HTTP Cloud Function.
 *
 * @param {Object} req Cloud Function request context.
 * @param {Object} res Cloud Function response context.
 */
exports.helloHttp = function helloHttp (req, res) {
  res.send(`Hello ${req.body.name || 'World'}!`);
};
FaaS: current limits

- Performance
  - Startup latency and cold starts
    - “The first time you deploy a function it may take several minutes as we need to provision the underlying infrastructure to support your functions. Subsequent deployments will be much faster.” (Google Cloud Functions)
- Language choice
- Resource limits
- Vendor lock-in

```{reduced flexibility}

Serverless platforms: OpenWhisk

- OpenWhisk [https://openwhisk.apache.org](https://openwhisk.apache.org)
- Open source, distribute serverless platform that executes functions in response to events at any scale
- Based on Docker containers
```
Serverless platforms: OpenWhisk

- Developers write functional logic (called **Actions**)
  - In any supported programming language
  - Can be dynamically scheduled and run in response to associated events (via **Triggers**) from external sources (**Feeds**) or from HTTP requests
- Functions can be combined into **compositions**