Architectural Patterns and Styles

Renan Johannsen de Paula
Venilton FalvoJr

SSC5944-1: Arquitetura de Software (2016)
Prof. Dr. Elisa Yumi Nakagawa

Contents

1. Introduction
2. Client-Server
3. Service-Oriented Architecture (SOA)
4. Representational State Transfer (REST)
5. Microservices
6. Appendix: Internet of Things (IoT)

Introduction

Introduction

An architectural **Style** is a **specialization** of **element and relation** types, together with a **set of constraints on how they can be used**.

On the other hand, an architectural **Pattern** expresses a fundamental structural organization schema for software systems. It **provides a set of predefined subsystems**, specifies **their responsibilities**, and includes **rules and guidelines for organizing the relationships** between them.

Clements et al (2011)

Introduction

In some cases, **architectural elements** are composed in ways that **solve particular problems**. The **compositions** have been found **useful** over time, and over many **different domains**, and so they have been **documented and disseminated**. These compositions of architectural elements, called architectural **Patterns**, **provide packaged strategies for solving some of the problems facing a system**.

(Bass et al, 2013)

Difference between Patterns and Styles

In Clements et al. (2011) you can find an extended discussion on the difference between an architectural pattern and an architectural style. It argues that a **Pattern is a context-problem-solution triple**; a **Style is simply a condensation** that **focuses** most heavily on the **solution** part.

(Bass et al, 2013)

Difference between Patterns and Styles

An essential part of an architecture **Pattern** is its **focus on the problem and context as well as how to solve the problem in that context**. An architecture **Style focuses on the architecture approach**, with more **lightweight guidance** on when a particular style may or may not be useful. Very informally, we can put it this way:

- Architecture **Pattern**: **{ problem, context } → architecture approach**;
- Architecture **Style**: **architecture approach**.

Clements et al (2011)

Why Patterns and Styles?

Although no fixed set of views is appropriate for every system, broad guidelines can help us gain a footing. Architects need to think about their software in three ways simultaneously:

1. How it is **structured** as a **set of implementation units**;
2. How it is **structured** as a **set of elements that have runtime behavior and interactions**;
3. How it relates to **nonsoftware structures** in its environment.

Clements et al (2011)

Client-Server

Client-Server

Client-Server style components interact by requesting services of other components. Requesters are termed **clients**, and service providers are termed **servers**, which **provide a set of services through one or more of their ports**.

Clients initiate interactions, **invoking services** as needed **from servers** and **waiting** for the **results of those requests**.

Clements et al (2011)

Client-Server: Constraints

- Clients are **connected** to servers **through request/response connectors**;
- **Server components can be clients to other servers**;
- **Specializations may impose restrictions**:
  - Numbers of attachments to a given port;
  - Allowed relations among servers.
- **Components may be arranged in tiers**.

Clements et al (2011)

Client-Server: Examples

Typical examples of systems in the client-server style include the following:

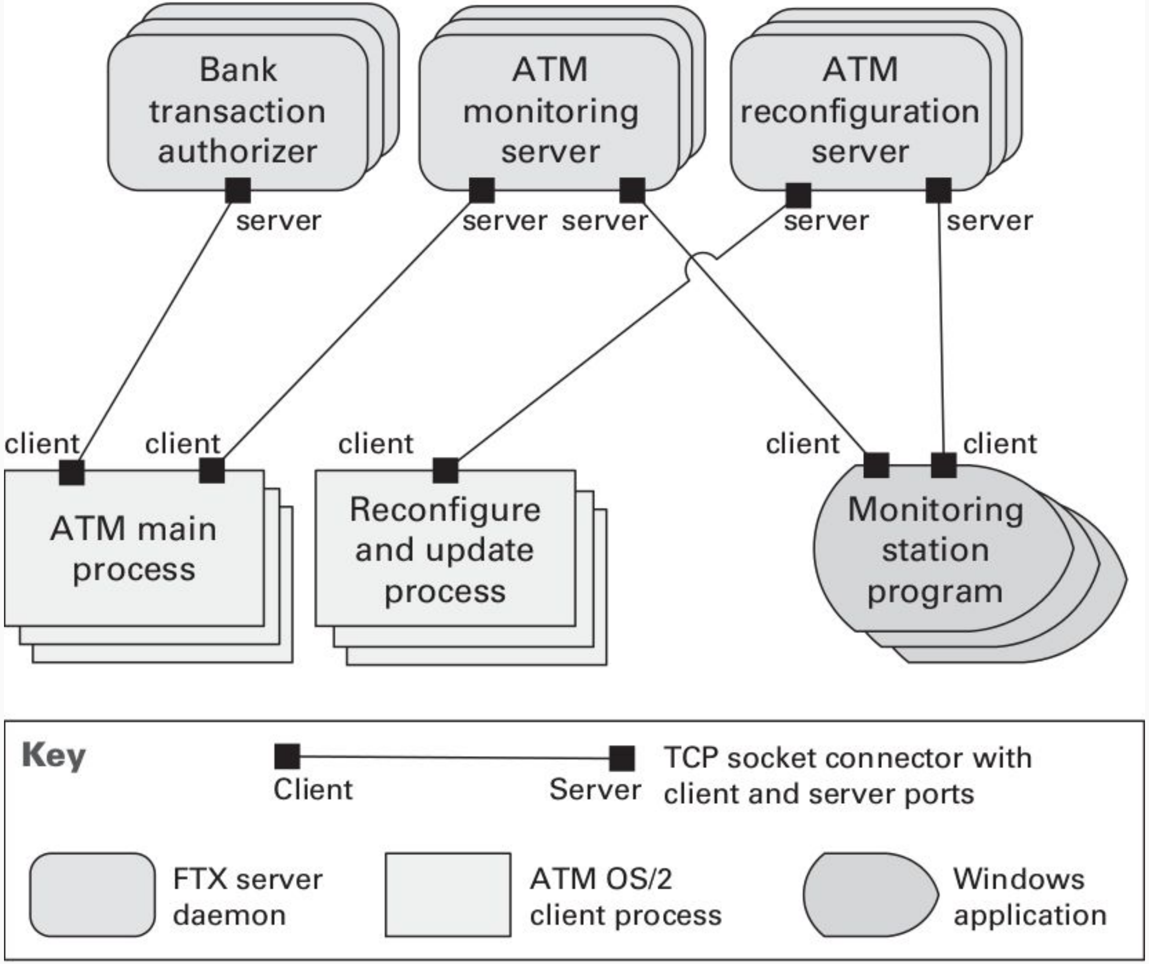
- Information systems running on **local networks**, where the clients are GUI applications (such as Visual Basic) and the server is a database management system (such as Oracle);
- **Web-based** applications where the clients run on Web browsers and the servers are components running on a Web server (such as Tomcat).

Clements et al (2011)

Client-Server: ATM Example

The Automated Teller Machine (**ATM**) **banking system** developed in the **early 1990s**. At that time, client-server architectures were the modern alternative to mainframe-based systems. (**J2EE and .NET application servers didn't exist and multitier was not yet described as a style**.)

Clements et al (2011)



Client-server architecture of an ATM banking system, using informal notation (Clements et al, 2011).

Service-Oriented Architecture (SOA)

SOA

Service-Oriented Architecture (**SOA**) **consist of a collection of distributed components that provide and/or consume services**. In this style, service **provider** components **and** service **consumer** components **can use different implementation languages and platforms**. **Services are largely standalone**.

Computation is achieved by a set of cooperating components that provide and/or consume services over a network.

Clements et al (2011)

SOA: Elements

- **Service providers**, which provide one or more services through published **interfaces**. Properties will vary with the implementation technology (such as EJB or ASP.NET) but may include performance, authorization constraints, availability, and cost;
- **Service consumers**, which **invoke services** directly or through an intermediary.

Clements et al (2011)

SOA: Elements

- Simple Object Access Protocol (**SOAP**) **connector**, which uses the SOAP protocol for synchronous communication between Web services, typically over **HTTP**. Ports of components that use SOAP are often **described in WSDL**;
- REpresentational State Transfer (**REST**) **connector**, which relies on the basic request/response operations of the **HTTP** protocol.

Clements et al (2011)

SOA: Constraints

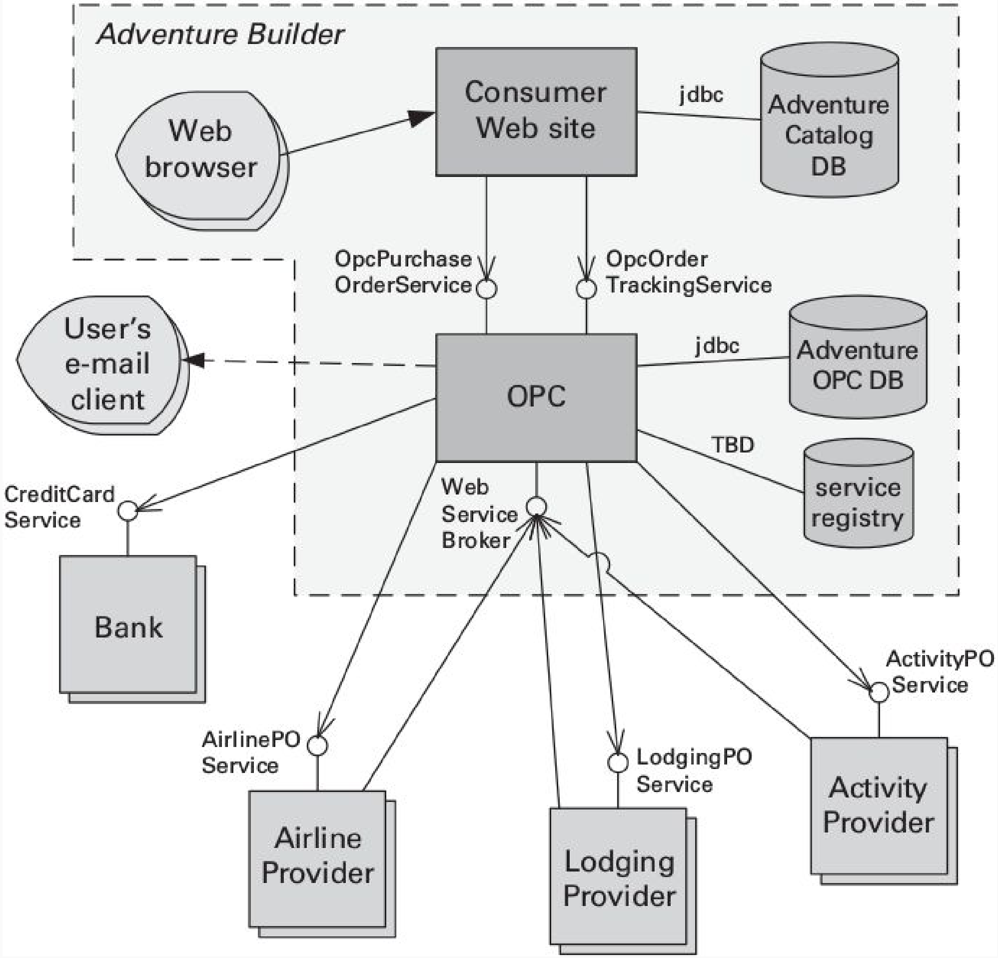
- Service consumers are connected to service providers, but **intermediary components** (such as **ESB**, registry, or BPEL server) **may be used**;
- ESBs lead to a hub-and-spoke topology;
- **Service providers may also be service consumers**;
- **Specific SOA patterns impose additional constraints**.

Clements et al (2011)

SOA: Adventure Builder 2010 Example

This system was taken from the example software architecture document accompanying this book online, at wiki.sei.cmu.edu/sad. The **Adventure Builder system** (Adventure Builder 2010) **interacts via SOAP** Web services with several other **external service providers**. Note that the external providers **can be mainframe, Java, or .NET** — the nature of these external components is transparent because the SOAP connector provides the necessary **interoperability**.

Clements et al (2011)



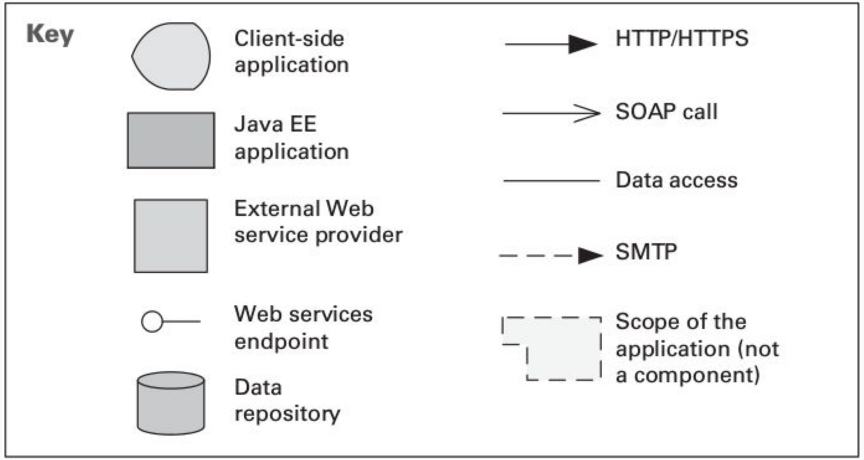


Diagram of the SOA view for the Adventure Builder 2010, using informal notation (Clements et al, 2011).

REpresentational State Transfer (REST)

REST

The **RE**presentational **S**tate **T**ransfer (REST) is the software architectural style of the **World Wide Web**. REST is an architectural style for **distributed hypermedia systems**, describing the **software engineering principles** guiding REST and the **interaction constraints** chosen to retain those principles, while **contrasting them to the constraints of other architectural styles**.

Clements et al (2011) and Fielding (2000)

REST

Since 1994, the REST has been **used to guide the design and development of the architecture for the modern Web**.

Was done as part of the Internet Engineering Task Force (**IETF**) and World Wide Web Consortium (**W3C**) efforts to define the architectural standards for the Web: Hypertext Transfer Protocol (**HTTP**), Uniform Resource Identifier (**URI**), and Hyper Text Markup Language (**HTML**).

Fielding (2000)

REST: Constraints

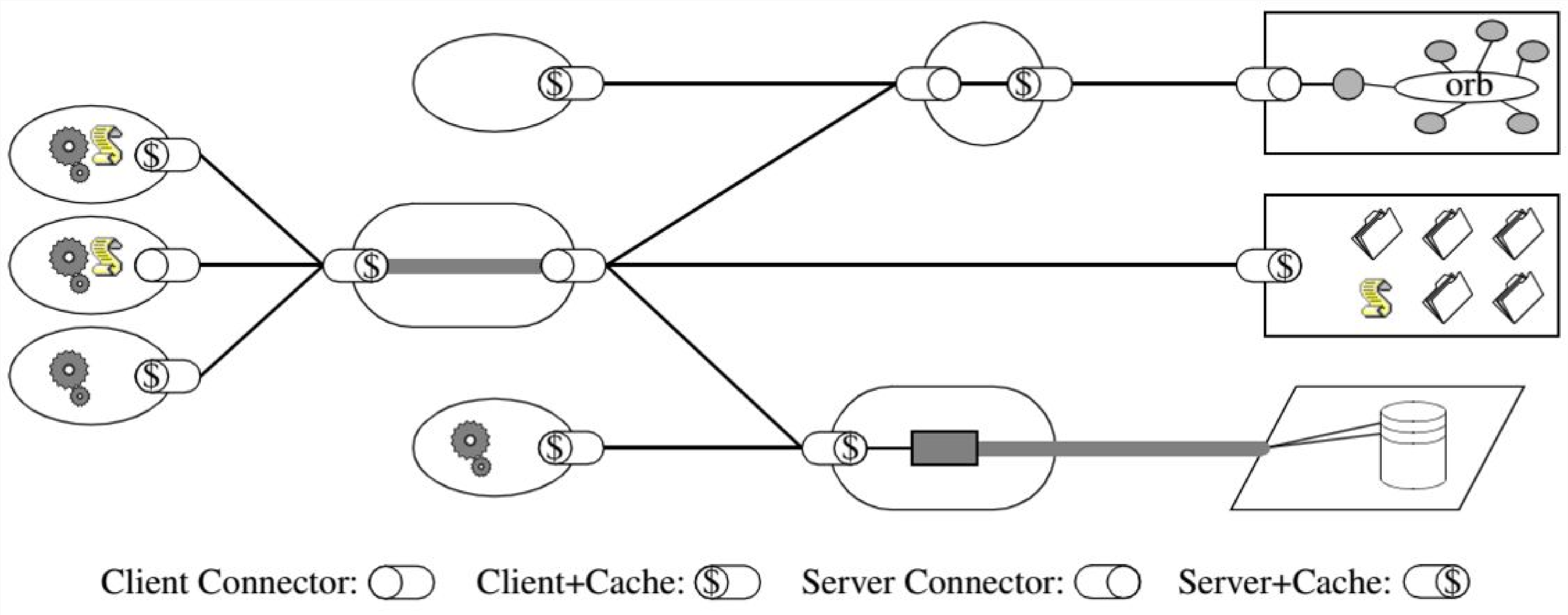
1. **Null Style**; plus
2. Client-Server = **Client-Server**; plus
3. Stateless = Client-**Stateless**-Server; plus
4. Cache = Client-**Cache**-Stateless-Server; plus
5. Uniform Interface = **Uniform**-Client-Cache-Stateless-Server; plus
6. Layered System = Uniform-**Layered**-Client-Cache-Stateless-Server; plus
7. Code-On-Demand = **REST**.

Fielding (2000)

REST: RESTful

Applications conforming to the REST constraints can be called **RESTful**. RESTful systems typically communicate over **HTTP** with the same **Methods** (GET, POST, PUT, DELETE etc) that browsers use to retrieve web pages and to send data to remote servers. REST systems interface with external systems as web **resources identified by URI**, for example, which can be operated upon using standard methods such as `GET /people/1` or `DELETE /people/1`.

Fielding (2000)



REST Style (Fielding, 2000)

Microservices

Microservices: Core concepts

Perhaps the most important concept to understand with this pattern is the **notion of a service component**.

Service components contain one or more modules (e.g., Java classes) that represent either a **single purpose function** (e.g., providing the weather for a specific city or town) or an **independent portion of a large business application** (e.g., stock trade placement or determining auto insurance rates).

Richards (2015)

Microservices: Separately deployed units

Applications built using the micro services architecture pattern are generally more robust, provide better **scalability**, and can more easily support continuous delivery.

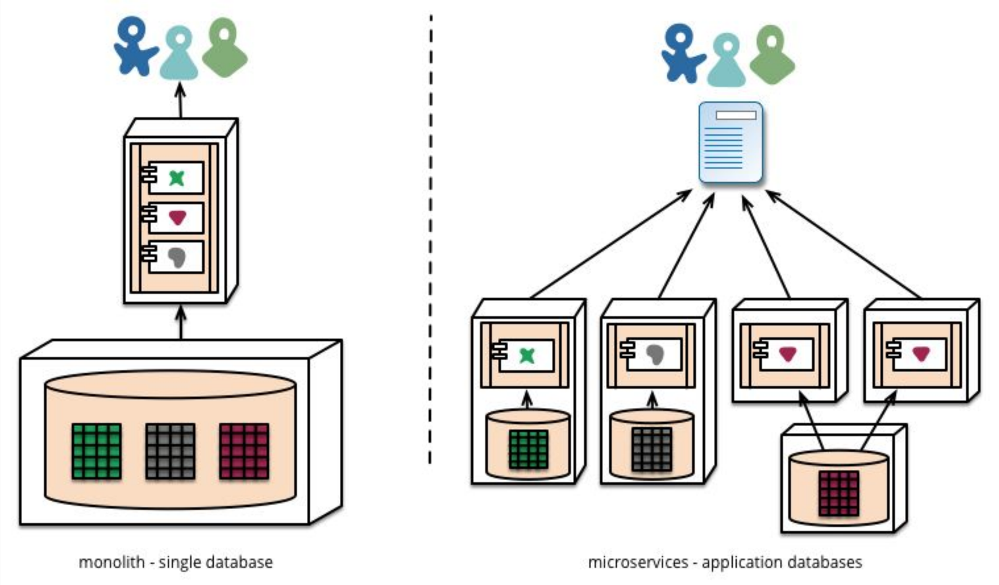
Richards (2015)

Microservices: Distributed architecture

Another key concept within the microservices architecture pattern is that it is a **distributed architecture**, meaning that all the components within the architecture are **fully decoupled** from one another and accessed through some sort of remote access protocol (e.g., JMS, AMQP, REST, SOAP, RMI, etc.). The distributed nature of this architecture pattern is how it achieves some of its **superior scalability** and **deployment characteristics**.

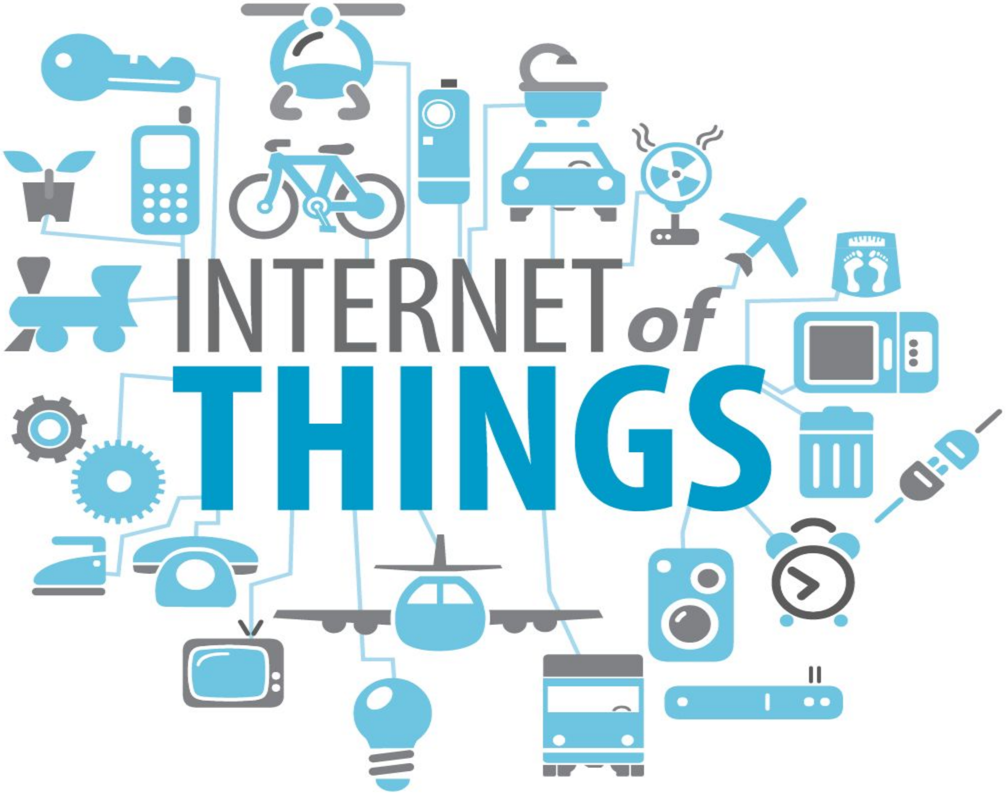
Richards (2015)

Microservices: Example



Martin Fowler (2014)

Internet of Things (IoT)



The IoT has become the major disruptive technology changing software — and society (Weyrich and Ebert, 2016).

IoT History

The **term "Internet of Things"** (IoT) was first used **in 1999 by Kevin Ashton**, who worked on a standard for **tagging objects using RFID** for logistics applications. However, the idea of ubiquitous computing goes back to the late 1980s.

Since then, researchers have worked on many systems focusing on **tags** and **sensors**, **middleware** and **cloud technologies**, and **communication networks**.

Weyrich and Ebert (2016)

What is IoT?

The Internet of Things (**IoT**) is about **innovative functionality** and **better productivity** by **seamlessly connecting devices**.

Will boost a tremendous amount of innovation, efficiency, and quality. **Connecting production, medical, automotive, or transportation** systems with IT systems and business-critical information will provide tremendous value to organizations. Major IT companies such as Cisco and SAP have predicted billions of networked devices and a universe of IT-based business services, with **expectations of a trillion-dollar**.

Weyrich and Ebert (2016)

IoT: Marketplace

A highly visible milestone was reached when the **IETF released IPv6**, the **protocol that enables the IoT**. Recently, the IoT has received a boost from commercial engagement and work on reference architectures driven by the major industries:

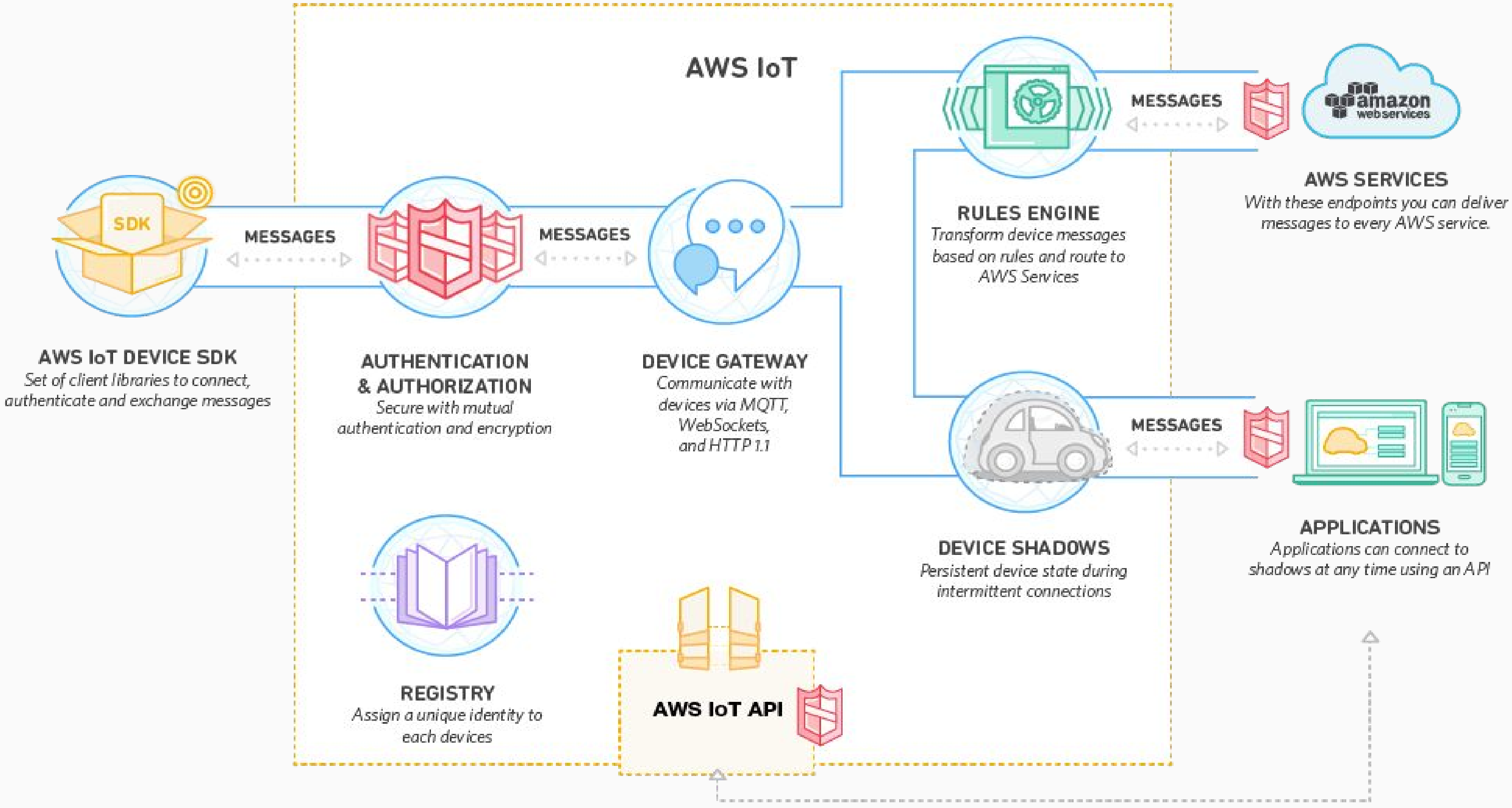
- **Google** has announced **Brillo** as an **OS for IoT devices in smart homes**;
- Devices are commercially available for machine-to-machine (**M2M**) communication standards such as **Bluetooth**, **ZigBee** and **low-power Wi-Fi**.

Weyrich and Ebert (2016)

IoT: Marketplace

- **Microsoft** has announced that **Windows 10** will support embedded systems for widespread microcontrollers such as **Raspberry Pi 2** and 3;
- **Samsung** and **other companies** have announced a **new generation of chips** for **smart devices**;
- Many implementation reports have described **networked microcontrollers serving as hubs for sensors, actuators, and tagging**.
- **Amazon** released in 2015 your cloud service for IoT, the **AWS IoT**.

Weyrich and Ebert (2016)


AWS IoT
SDK
MESSAGES
MESSAGES
MESSAGES
MESSAGES
amazon webservices
AWS SERVICES
With these endpoints you can deliver messages to every AWS service.
RULES ENGINE
Transform device messages based on rules and route to AWS Services
AWS IoT DEVICE SDK
Set of client libraries to connect, authenticate and exchange messages
AUTHENTICATION & AUTHORIZATION
Secure with mutual authentication and encryption
DEVICE GATEWAY
Communicate with devices via MQTT, WebSockets, and HTTP 1.1
DEVICE SHADOWS
Persistent device state during intermittent connections
APPLICATIONS
Applications can connect to shadows at any time using an API
REGISTRY
Assign a unique identity to each devices
AWS IoT API

References

References

1. **Bass** L, Clements P, and Kazman R. 2013. **Software Architecture in Practice**. Addison-Wesley, Boston, MA.
2. **Clements** P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Merson P, Nord R, and Stafford J. 2010. **Documenting Software Architectures: Views and Beyond**. Addison-Wesley, Boston, MA.
3. **Fielding** R. 2000. **Architectural Styles and the Design of Network-Based Software Architectures**. Ph.D. Dissertation. University of California, Irvine, CA.

References

4. **Richards** M. 2015. **Software Architectural Patterns**. O'Reilly Media, Sebastopol, CA.
5. **Fowler**, M. **Microservices: A definition of this new architectural term**. 2014. Available in: http://martinfowler.com/articles/microservices.html.
6. **Weyrich** M and **Ebert** C. 2016. **Reference Architectures for the Internet of Things**. IEEE Software, vol. 33, no. 1, pp. 112-116.

Thank you!

renanjohannsen@gmail.com

falvojr@gmail.com