11

A Centralized Support Infrastructure (CSI) to Manage CPS Digital Twin, towards the Synchronization between CPS Deployed on the Shopfloor and Their Digital Representation

Diego Rovere[1], Paolo Pedrazzoli[2], Giovanni dal Maso[1], Marino Alge[2] and Michele Ciavotta[3]

[1]TTS srl, Italy
[2]Scuola Universitaria Professionale della Svizzera Italiana (SUPSI), The Institute of Systems and Technologies for Sustainable Production (ISTEPS), Galleria 2, Via Cantonale 2C, CH-6928 Manno, Switzerland
[3]Università degli Studi di Milano-Bicocca, Italy
E-mail: rovere@ttsnetwork.com; pedrazzoli@ttsnetwork.com; dalmaso@ttsnetwork.com; marino.alge@supsi.ch; michele.ciavotta@unimib.it

In order to support effective multi-disciplinary simulation tools in all phases of the factory life cycle, it is mandatory to ensure that the Digital Twin mirrors constantly and faithfully the state of the CPS. CPS nameplate values change over time due to situation and strain. Thereupon, this chapter describes the future CPS as equipped with special assets named Functional Models to be uploaded to CSI for synchronization and data analysis. Functional Models are essentially software routines that are run against data sent by the CPS. Such routines can regularly update CPS reference values, estimate indirect metrics, or train predictive models. Functional Models are fully managed (registered, executed, and monitored) by the CSI middleware.

11.1 Introduction

The main purpose of the CSI is to manage CPS Digital Twins (DTs) allowing the synchronization between CPS deployed on the shopfloor and their digital representation. In particular, during the whole factory life cycle, the CSI will provide services (via suitable API endpoints) to analyze the data streams coming from the shopfloor and to share simulation models and results among simulators.

In this chapter, we present the implementation of a distributed middleware developed within the frame of MAYA European project, tailored to enable scalable interoperability between enterprise applications and CPS with especial attention paid to simulation tools. The proposed platform strives for being the first solution based on both Microservices [1, 2] and Big Data [3] paradigms to empower shopfloor CPS along the whole plant life cycle and realize real-digital synchronization ensuring at the same time security and confidentiality of sensible factory data.

11.2 Terminology

Shopfloor CPS – With the expression "Shop-floor CPS" we refer to Digital-Mechatronic systems deployed at shopfloor level. They are physical entities that intervene in various ways in the manufacture of a certain product. For the scope of this chapter, Shopfloor CPS (referred to as Real CPS or simply CPS) can communicate to each other and with the CSI.

CPS Digital Twin (or just Digital Twin) – In the smart factory, each shopfloor CPS is mirrored by its virtual alter ego, called Digital Twin (DT). The Digital Twin is the semantic, functional, and simulation-ready representation of a CPS; it gathers together heterogeneous pieces of information. In particular, it can define, among other things, Shopfloor CPS performance specifications, Behavioral (simulation) Models, and Functional Models.

Digital Twin is a composite concept that is specified as follows:

CPS Prototype (or just Prototype) – Chapter 12 proposes a meta-model that paves the way to a semantic definition of CPS within the CSI. Following the Object-Oriented Programming (OOP) approach, we distinguish between a Prototype (or class) and its derived instances. A CPS prototype is a model that defines the structure and the associate semantic for a certain class of CPS.

A prototype defines fields representing both general characteristics of the represented CPS class and the state of a specific Shopfloor CPS.

CPS Instance – Once a shopfloor CPS is connected to the CSI platform, a set of processes are run to instantiate, starting from a CPS prototype, the Digital Twin. The Digital Twin is an instance of a specific CPS prototype. Therefore, a CPS instance can be defined as the computer-based representation (live object in memory or stored in a database) of its Digital Twin, which can be considered a more abstract concept even independent of this implementation within the CSI.

Behavioral Models – These are simulation models, linked to the semantic representation of a CPS (prototype and instance) and stored within the CSI. Each Digital Twin can feature behavioral models of different nature to enable the multi-disciplinary approach to simulation.

Functional Models – In layman's terms, functional models are pieces of software to be run on a compliant platform created to analyze data coming from the shopfloor. Data can enter a platform in the form of streams or imported from other sources (text files, excel, databases, etc.). The results of the analysis are used to enrich the Digital Twin implementing the real-to-digital synchronization. They can be used, for instance, to update license plate data of Digital Twins or to enable predictive maintenance specific on the considered CPS.

11.3 CSI Architecture

The overall CSI component diagram is shown in Figure 11.1: a relevant part of the platform consists of a microservice-based infrastructure devoted to administrative tasks related to Digital Twins and a Big Data deployment accountable for processing shopfloor data. Since the two portions of our middleware have different requirements, being also grounded on different technological solutions, in what follows, they are presented and discussed separately.

11.3.1 Microservice Platform

In a nutshell, the microservice architecture is the evolution of the classical Service Oriented Architecture (SOA) [4] in which the application is seen as a suite of small services, each devoted to a single activity. Within the CSI,

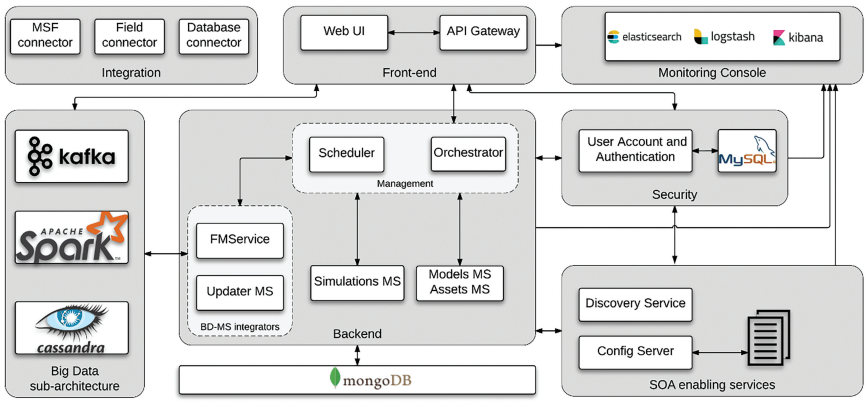

Figure 11.1 CSI Component Diagram.

each microservice exposes a small set of functionalities and runs in its own process, communicating with other services mainly via HTTP resource API or messages. Four groups of services can be identified and addressed in what follows.

11.3.1.1 Front-end services

Front-end services are designed to provide the CSI with a single and secure interface to the outer world. As a consequence, any other service can be accessed only through the front-end and only by trusted entities. The main services in this group are:

Web-based UI

The Web-based UI is a Web application for human–machine interaction; it provides a user friendly interface to register new CPS or to execute queries. Administration tools such as security management and platform monitoring are available as well.

API Gateway

The API Gateway, instead, is a service designed to provide dynamic and secure API routing, acting as a front door for the requests coming from authorized players, namely users via the Web UI and devices/CPS executing REST/WebSocket calls. In layman's terms, all the other platform services are accessible only through the gateway and only by trusted entities.

The gateway is based on Netflix Zuul[1] for dynamic routing, monitoring, and security, and Ribbon[2], a multi-protocol inter-process communication library that, in collaboration with Service Registry (see SOA enabling services), dispatches incoming requests applying load-balance policy. The API gateway, finally, offers an implementation of the Circuit Breaker[3] pattern impeding the system to get stuck in case the target back-end service fails to answer within a certain time.

11.3.1.2 Security and privacy

Security policies are enforced by the User Account and Authentication (UAA) service, which is in charge of the authentication and authorization tasks:

UAA Service

In a nutshell the main task of this service is to check users' (human operators, CPS or microservices) credentials to verify the identity and issuing a time-limited OAuth2 [13] token to authorize a subset of possible actions that depends on the particular role the user has been assigned to. Users' data, roles and permission are stored in a relational database: currently, MySQL[4] database is used to this end.

It is worth to notice that authentication and authorization is required not only for human users and CPS but also to establish a trustful collaboration between microservices avoiding malevolent and tampering actions.

11.3.1.3 SOA enabling services

SOA enabling services: this group of services has the task to support the microservice paradigm; it features:

Service Registry

This service provides a REST endpoint for service discovering. This service is designed to allow transparent and agnostic service communication and load balancing. Based on Netflix Eureka[5], it exposes APIs for service registration and for service querying, allowing the services to communicate without referring to specific IP addresses. This is especially important in the scenario in which services are replicated in order to handle a high workload.

---

[1] https://github.com/Netflix/zuul/wiki

[2] https://github.com/Netflix/ribbon

[3] https://martinfowler.com/bliki/CircuitBreaker.html

[4] www.mysql.com

[5] https://github.com/Netflix/eureka/wiki

Configuration Server
The main task of this service is to store properties files in a centralized way for all the micro-services involved in the CSI. This is a task of paramount importance in many scenarios involving the overall life cycle of the platform. Among the benefits of having a configuration server, we mention here the ability to change the service runtime behavior in order to, for example, perform debugging and monitoring.

Monitoring Console
This macro-component with three services implements the so-called ELK stack (i.e., Elasticsearch, Logstash, and Kibana) to achieve log collection, analyzing, and monitoring services. In other words, logs from every microservice are collected, stored, processed, and presented in a graphical form to the CSI administrator. A query language is also provided to enable the administrator to interactively analyze the information coming from the platform.

11.3.1.4 Backend services

To this group belong those services that implement the Chapter 12 meta-data model and manage the creation, update, deletion, storage, retrieval, and query of CPS Digital Twins as well as simulation-related information. In particular, the CSI features the following services:

Orchestrator
The Orchestrator microservice coordinates and organizes other services' execution to create high-level composite business processes.

Scheduler
Service for the orchestration of recurring action. Example of those jobs are: importing data from external sources at regular intervals, updating CPS Prototypes and instances, removing from internal databases stale data, and sending emails enclosing a report on the system's healthy to administrators.

Models MS/Assets MS
Models and Assets microservices handle the persistence of Digital Twin information (their representation and assets, respectively) providing endpoints for CRUD operations. In the current version of the CSI, these two components are merged into a single service in order to streamline the access to MongoDB and avoid synchronization issues.

FMService
This service is able to communicate with the Big Data platform; its main task is to submit the Functional Models to Apache Spark, to monitor the execution, cancel, and list them.

Updater MS
This service is designed to interact with the Big Data platform (in particular with Apache Cassandra) to retrieve data generated by the Functional Models.

Simulations MS
This service is appointed to managing the persistence of simulation-related data within a suitable database.

11.3.2 Big Data Sub-Architecture

Big Data technologies are becoming innovation drivers in industry [5]. The CSI is required to handle unprecedented volumes of data generated by the digital representation of the factory in order to keep updated the CPS nameplate information. To this end, a data processing platform, specifically a Lambda architecture [6], has been implemented according to the best practices of the field. The Lambda Architecture was introduced as a generic, linearly scalable, and fault-tolerant data processing architecture. In particular, both data in rest and data in motion patterns are enforced by the platform, making it suitable for both stream and batch processing.

The Lambda Architecture encompasses three layers, namely batch, speed, and serving layers. The batch layer is appointed to the analysis of large quantities (but still finite) of data. A typical scenario is that wherefore the data ingested by the system are inserted in NoSQL Databases. Pre-computation is applied periodically on batches of data. The purpose is to offer the data a suitable aggregated form for different batch views. Note that the batch layer has a high processing latency because it is intended for historical data.

The speed layer is in charge of processing infinite streams of information. It is the purpose of the Speed Layer to offer a low latency, real-time data processing. The speed layer processes the input data as they are streamed in and it feeds the real-time views defined in the serving layer.

The Serving Layer has the main responsibility to offer a view on the results of the analysis. The layer responds to queries coming from external systems; in this particular case, the serving layer provides an interface that integrates with the rest of the CSI.

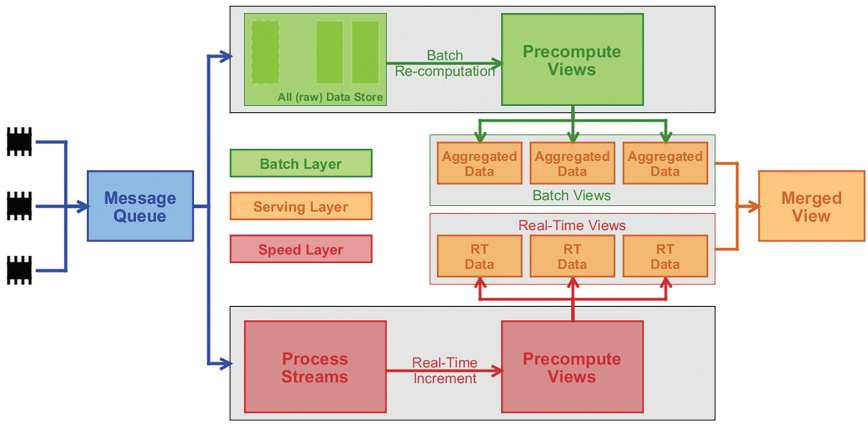

Figure 11.2 Lambda Architecture.

Designing and setting up a Big Data environment, here in the form of the Lambda Architecture (Figure 11.2), is a complex task that starts with doing some structural decisions. In what follows, some high-level considerations about the technological choices made are presented:

11.3.2.1 Batch layer

The field of Big Data is bursting with literally hundreds of tools and frameworks, each with specific characteristics; however, recently, some new solutions have appeared on the market that natively extend MapReduce [7] paradigm reduce, and, among other things, provide a more flexible and complete programming paradigm paving the way to the realization of new and more complex algorithms.

The solution selected to implement this layer, Apache Spark [8], claims to be up to 100× faster than Hadoop on memory and up to 10× faster on disk. This is mainly due to a particular distributed, in memory data structure called Resilient Distributed Datasets (RDD). Shortly, Apache Spark attracted the interest of important players and gathered a vast community of contributors, only to mention a few: Intel, IBM, Yahoo!, Databricks, Cloudera, Netflix, Alibaba, and UC Berkely. Moreover, Spark implements both map-reduce and streaming paradigm, features out-of-the-box an SQL-like language for automatic generation of jobs, and supports several programming languages (Java, Scala, Python, and R).

11.3.2.2 Stream processing engine

If the batch processing engine enables the analysis of large historical data (often referred to as Data at Rest), then the stream processing engine is the component of the Lambda Architecture that is in charge of continuously manipulating the incoming data in quasi real-time fashion (i.e., the Data in Motion scenario). Recently, stream processing has increased in popularity. Only within the Apache Foundation, we identified several tools supporting different flavors of stream processing. Among them is Spark Streaming [9], the tool used to implement this layer.

Spark Streaming relies on Spark core to implement micro-batching stream processing. This means that the elements of the incoming streams are grouped together in small batches and then manipulated. As a consequence, Spark shows a higher latency (about 1 second). Spark Streaming is a valid alternative owing to the rich API, the large set of libraries, and its stability.

Spark can work in standalone mode featuring on its own resource manager or it can rely on external resource managers (as YARN). Other resource managers exist (e.g. Apache Mesos), but they are related more to cluster management than on Big Data. Nonetheless, Spark can be executed over both YARN and Mesos.

11.3.2.3 All data store

A central role in the Lambda Architecture is played by the All Data Store, which is the service in charge of storing and retrieving the historical data to be analyzed. Depending on the type of data entering the system, this element of the platform can be realized in different ways. In MAYA, we decided to implement it through a NoSQL database particularly suitable for fast updates, Apache Cassandra [10]. It is the most representative champion of the column-oriented group. It is a distributed, linear scalable solution capable of ensuring high volumes of data. Cassandra is widely adopted (it is the most used column-oriented database) and features an SQL-like query language named CQL (Cassandra Query Language) along with a Thrift[6] interface. As far as stream views are concerned, Cassandra has been successfully used to handle time series for IoT and Big Data.

11.3.2.4 Message queueing system

In a typical Big Data scenario, data flows coming from different sources continuously enter the system; the most used integration paradigm to handle

[6] https://thrift.apache.org/

data flows consists in setting up a proper message queue. A message queue is a middleware implementing a publisher/subscriber pattern to decouple producers and consumers by means of an asynchronous communications protocol. Message queues can be analyzed under several points of view, in particular policies regarding Durability, Security, Filtering, Purging, Routing, and Acknowledgment, and message protocols (as AMQP, STOMP, MQTT) must be carefully considered.

Message queue systems are not a novelty and many proprietary as well as open-source solutions have appeared on the market in the last years. Among the open-source ones, there is Apache Kafka [11]. A preliminary analysis seems to demonstrate that Kafka is the most widely used in big players' production environments as, for instance, in LinkedIn, Yahoo!, Twitter, Netflix, Spotify, Uber, Pinterest, PayPal, Cisco, and Coursera among the others. Kafka is written in Java and originally developed at LinkedIn; it provides a distributed and persistent message passing system with a variety of policies. It relies on Apache Zookeeper [12] to maintain the state across the cluster. Kafka has been tested to provide close to 200,000 messages/second for writes and 3 million messages/second for reads, which is an order of magnitude more that its alternatives.

11.3.2.5 Serving layer

This layer provides a low-latency storage system for both batch and speed layers. The goal of this layer is to provide an engine able to ingest different types of workloads and query them showing a unified view of data. The rationale is that the outcomes of the different computations must be suitably handled to later be further processed. In particular, batch views will contain steady, structured, and versioned data, whereas stream views will contain time-related data. Within the CSI, we have adopted the following flexible approach: in case Batch activities are required, the serving layer is implemented by means of Apache Cassandra NoSQL database, otherwise Apache Kafka is exploited. Notice that it is not uncommon to use a persistent and distributed message system as serving layer as, for example, in ORYX2[7], where precisely Kafka is used.

11.3.3 Integration Services

Technically, these services do not belong to the CSI at the moment, but we envision their development in the following phases of the project with the aim

[7] http://oryx.io/

of streamlining the interaction processes with external tools and databases; in particular, at the moment of writing we foresee the following services:

MSF Connector
This component passes the CPS id, the simulation model in AutomationML format, and the simulation types requested by the user. The MSF sends in return the simulation results per each simulation type requested.

Field Connector
This service serves to bridge the gap between the communication layer and the field in case of CPS non-compliant with the CSI. In particular, it will create suitable WebSocket channels for data streams coming from the field and root those data to the Big Data platform inside the CSI.

DB Importer
Database Importers will be in charge of importing valuable data from external databases to enable the execution of Functional Models on those data.

11.4 Real-to-Digital Synchronization Scenario

Several usage scenarios are possible to be executed within the CSI. Nonetheless, we propose the following as a reference use case, as it involves a good part of CSI components and functionalities. The objective is to use it as a reading key to better understand the relationships among the CSI and how they are reflected into the architecture. The considered scenario concerns the automated processing of data streams coming from CPS and can be described as follows:

1. A human operator registers a new CPS. This action can be performed via the graphical UI or by means of available REST [13] endpoints;
2. The CPS logs in on the CSI, its digital identity is verified and the Digital Twin is activated;
3. The Functional Model featured by the Digital Twin (if any) is set up, scheduled, and executed;
4. WebSocket channel is established between the CPS and CSI. The CPS starts sending data to the platform;
5. The Functional Model periodically generates updates for a subset of attributes of the corresponding Digital Twin;
6. The CPS disconnects from the CSI and consequently the related Functional Models is halted and dismissed.

Figure 11.3 describes in UML the main actions carried out by the CPS and by the CSI in the scenario at hand. In particular, the CPS connects by logging in on the platform, at that point it is associated to a WebSocket endpoints and it can start sending data up to the CSI. The CSI, on the other hand, launches the execution of the Functional Model associated with the CPS.

A deeper insight is gained by means of Figure 11.4; in it, the interactions among services within the CSI are highlighted. It is clear, in fact, that the CPS connects with the CSI via the API ***Gateway***. In the current version

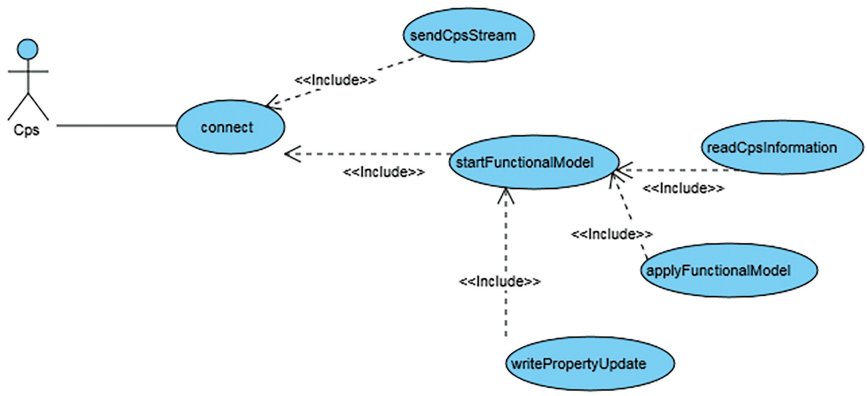

Figure 11.3 CPS connection.

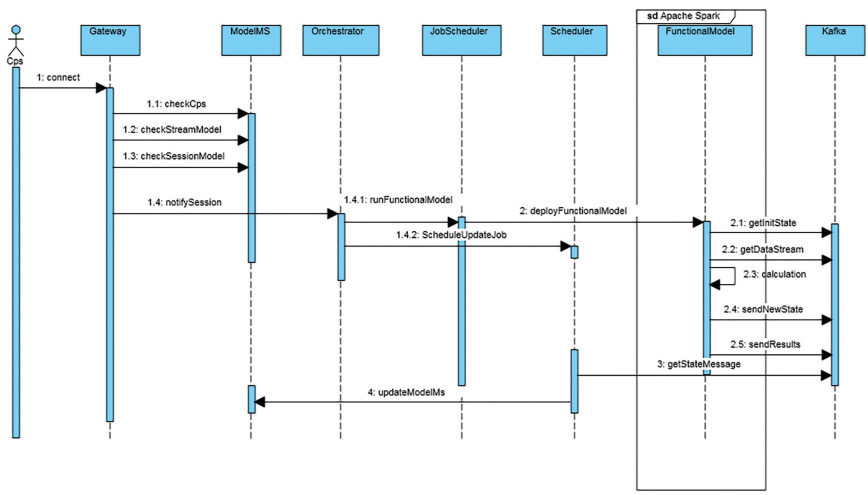

Figure 11.4 Sequence diagram.

of the CSI, the Gateway is in charge of checking whether the CPS asking for being attended is legit (it must have been created within the platform beforehand). To do this, the Gateway interrogates the ***Models MS*** service. The Gateway then creates a WebSocket endpoint for the CPS, redirects the incoming workload to Kafka, and notifies the ***Orchestrator***. This, in turn, is in charge of running the Functional model(s) associated with the CPS. The Functional models are executed within the Big Data platform (in Apache Kafka cluster) and in particular they use Kafka not only as source of data but also as the endpoint where to post the results of the computation. Meanwhile the Orchestrator has scheduled a recurrent job on the ***Scheduler*** that picks up the updated from the output Kafka topic and uses them to update the nameplated values of the CPS Digital Twin.

During the whole process, the Security is present in the form of SSL connection, CPS log in via OAuth2, and service-to-service authorization and authentication. We outlined the real-to-digital synchronization in Figure 11.5, wherein the reader can spot the presence of all the players present in the sequence diagram plus the UAA Service in charge of the authentication and authorization tasks. The actions performed by this service are pervasive and would have made the sequence diagram unintelligible.

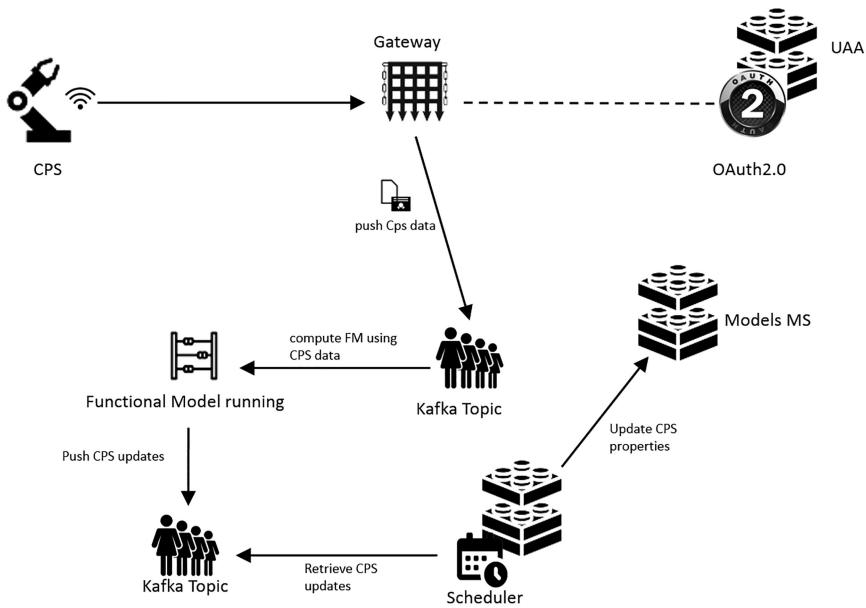

Figure 11.5 Outline of the Real-to-digital synchronization.

11.5 Enabling Technologies

CSI aims at being the first reference middleware for smart factories based on a composite Microservices/Big Data approach paying particular attention to security concerns. In the following paragraphs, we examine the reasons behind the technical choices made.

11.5.1 Microservices

The Microservices approach proposes to have numerous small code bases managed by small teams instead of having a giant code base that eventually every developer touch with the result of making more complex, slow, and painful the process of delivering a new version of the system.

In a nutshell, the microservice architecture is the evolution of the classical Service-Oriented Architecture (SOA), in which the application is seen as a suite of small services, each devoted to as single activity. Each microservice exposes an atomic functionality of the system and runs in its own process, communicating with other services via HTTP resource API (REST) or messages.

The adoption of the microservice paradigm provides several benefits, as well as presents inconveniences and new challenges. Among the benefits of this architectural style, the following must be enumerated:

Agility – Microservices fit into the Agile/DevOps development methodology [2], enabling business to start small and innovate fast by iterating on their core products without affording substantial downtimes. A minimal version of an application, in fact, can be created in shorter time reducing time-to-market and up-front investment costs, and providing an advantage with respect to competitors. Future versions of the application can be realized by seamlessly adding new microservices.

Isolation and Resilience – Resiliency is the ability of self-recovery after a failure. A failure in a monolithic application can be a catastrophic event, as the whole platform must recover completely. In a microservice platform, instead, each service can fail and heal independently with a possibly reduced impact on the overall platform's functionalities. Resilience is strongly dependent on compartmentalization and containment of failure, namely Isolation. Microservices can be easily containerized and deployed as single process, reducing thus the probability of cascade-fail of the overall application. Isolation, moreover, enables reactive service scaling and independent monitoring, debugging, and testing.

Elasticity – A platform can be subject to variable workloads especially on seasonal basis. Elasticity is the ability to respond to workload changes provisioning or dismissing computational power. This is usually translated into scaling up and down services. This process can be particularly painful and costly in case of on premise software; easier and automated in case of cloud-based applications. Nonetheless, microservices allows for a finer grain approach, in which services in distress (e.g., that are not meeting their Quality of Service) can be identified and singularly scaled taking full advantage of cloud computing since it requires the provisioning of just the right amount of resources. This approach can lead to substantial savings in the cloud that usually implements pay-per-use provisioning policies.

As far as the challenges and drawbacks derived by the choice of adopting microservices are concerned, we mention here:

Management of Distributed Data – As each microservice might have its private database, it is difficult to implement business transactions that maintain data consistency across multiple databases.

Higher Complexity of the Resulting System – Proliferation of small services could translate into a tangle Web of relationships among them. Experienced teams must be put together to deal in the best possible way with microservice platforms.

11.5.2 Cloud Ready Architecture: The Choice of Docker

Containerization services (among which the most known is definitely Docker [14]) and microservice are two closely related yet different aspects of the same phenomenon; although containerization is not essential to realize microservice architectures, it is certainly true that it enables microservices to fully realize their potential; Docker's *agility*, *isolation*, and *portability*, in fact, powered the rise and success of the microservice pattern while the latter gathered an ever-increasing interest around containers. It can be safely said that there are now two faces of the same coin and have made the fortune of each other.

At this point, it is important to answer to the simple question: what is a containerization system? A containerization system (hereinafter, we will use Docker and containerization system interchangeably) is a para-virtualization platform that exploits isolation features of Linux kernel, as namespaces and *cgroups* (recently also Windows' ones), to create a secure and isolate environment for the execution of a process. Each process running in a container has

access to its own file system and libraries, but it shares with other containers the underpinning kernel.

This approach is defined para-virtualization because, unlike virtualization systems that emulate hardware to execute whole virtual machines to run atop, there is no need to emulate anything. Moreover, Docker do not depend on specific virtualization technologies and, therefore, it can run wherever a Linux kernel available. The overall approach results to be lightweight with respect to more traditional hypervisor-based virtualization platform allowing for a better exploitation of the available resources and for the creation of faster and more reactive applications. In the light of these considerations, it should be clear how Docker fits perfectly for microservices, as it isolates containers to one process and makes it simple and fast to handle the full life cycle of these services.

The current version of the CSI is provided with a set of scripts for automatic creation of Docker images for each of the services involved in the platform. Deployment scripts, which rely on a tool called Docker-compose, are provided as well to streamline the deployment on a local testbed. Nonetheless, a similar approach can be used to execute the platform on the most important Clouds (e.g. Amazon ECS, Azure Container Service).

11.5.3 Lambda Architecture

A very important subset of CSI functionalities consists in the capability to handle unprecedented volume generated by the digital representation of the factory. To this end, a Big Data platform has been integrated with the microservice one. The phrase Big Data usually refers to a large research area that encompasses several facets. In this work, in particular, we refer to Big Data architectures. The following benefits deserve to be enumerates:

Simple but Reliable – The CSI Big Data platform has been implemented employing a reduced number of tools; all of them are considered state of the art, are used in production by hundreds of companies worldwide, and are backed by large communities and big Information and Communications Technologies players.

Multi-paradigm and General Purpose – Batch and Stream processing as well as ad-hoc queries are supported and can run concurrently. Moreover, the unified execution model, coupled with a large set of libraries, permits the execution of complex and heterogeneous tasks (as machine learning, data filtering, ETL, etc.).

Robust and Fault Tolerant – In case of failure, the data processing is automatically rescheduled and restarted on the remaining resources.

Multi-tenant and Scalable – In MAYA, this means that several Functional Models can run in parallel sharing computational resources. Furthermore, in case more resources are provisioned and the platform will start to exploit them without downtimes.

The downside of this approach is that it is fundamentally and technologically different for the rest of the platform and required quite an integration work. For this reason, the main elements of the CSI Big Data architecture had to be interfaced with expressly created microservices (as FMserver and Updates MS, see Section 4.1.4 for more details). Finally, Big Data solutions generally require steep learning curves to be fully exploited being moreover really resource eager.

11.5.4 Security and Privacy

Security and privacy issues assume paramount importance in Industrial IoT. Here, we enforce those aspects since the earliest stages of the design, focusing on suitable Privacy-Enhancing Technologies (PETs) that encompass Authentication, Authorization, and Encryption mechanisms.

More in detail, authentication is the process of confirming the identity of an actor in order to avoid possibly malicious accesses to the system resources and services. Authentication can be defined as the set of actions a software system has to implement in order to grant the actor the permissions to execute an operation on one or more resources.

Specifically, seeking for more flexibility, we implemented a role-based access control model that permits the authentication process to depend on the actor's role. Suitable authentication/authorization mechanisms (based on the Oauth2 protocol) have been developed for human operators, and services and CPS.

Securing communication is the third piece of this security and privacy puzzle, as no trustworthy authentication and authorization mechanism can be built without the previous establishment of a secure channel. For this reason, the CSI committed to employ modern encryption mechanisms (e.g. SSL and TLS) for the communication and data storage as well.

11.6 Conclusions

This document presented the Centralized Support Infrastructure built within the H2020 MAYA project: an IoT middleware designed to support simulation

in smart factories. To the best of our knowledge, it represents the first example of Microservice platform for manufacturing. Since security and privacy are sensitive subjects for the industry, special attention has been paid on their enforcement from the earliest phases of the project. The proposed platform has been here described in detail in connection with CPS and simulators. Lastly, the overall architecture has been discussed along with benefits and challenges.

Acknowledgements

The work hereby described has been achieved within the EU-H2020 project MAYA, which has received funding from the European Union's Horizon 2020 research and innovation program, under grant agreement No. 678556.

References

[1] N. Dragoni et al., "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, Springer Berlin Heidelberg, 2017.

[2] S. Newman, *Building microservices*. " O'Reilly Media, Inc.," 2015.

[3] J. Manyika et al., "Big data: The next frontier for innovation, competition, and productivity," 2011.

[4] S. Newman, *Building microservices*. " O'Reilly Media, Inc.," 2015.

[5] C. Yang, W. Shen, and X. Wang, "Applications of Internet of Things in manufacturing," in *Proceedings of the 2016 IEEE 20th International Conference on Computer Supported Cooperative Work in Design, CSCWD 2016*, pp. 670–675, 2016.

[6] R. Drath, A. Luder, J. Peschke, and L. Hundt, "AutomationML-the glue for seamless automation engineering," in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pp. 616–623, 2008.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. OSDI - Symp. Oper. Syst. Des. Implement.*, pp. 137–149, 2004.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark?: Cluster Computing with Working Sets," *HotCloud'10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput.*, p. 10, 2010.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," *Sosp*, no. 1, pp. 423–438, 2013.

[10] A. Lakshman and P. Malik, "Cassandra," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35, 2010.

[11] J. Kreps and L. Corp, "Kafka: a Distributed Messaging System for Log Processing," *ACM SIGMOD Work. Netw. Meets Databases*, p. 6, 2011.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX Annual Technical Conference*, vol. 8, p. 11, 2010.

[13] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2. pp. 407–416, 2002.

[14] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using Docker technology," in *Conference Proceedings - IEEE SOUTHEASTCON*, 2016, July 2016.