DOCTORAL COURSE IN
ENGINEERING SCIENCE

CYCLE XXXVI

COORDINATOR Prof. Trillo Stefano

An Entanglement-Aware Digital Twin Ecosystem

Scientific/Disciplinary Sector (SDS) ING-INF/05

Candidate
Dott. Fogli Mattia

Supervisor
Prof. Stefanelli Cesare

Cosupervisor
Prof. Giannelli Carlo

Year 2020/2023

Acknowledgements

This thesis is the culmination of my Ph.D. at the University of Ferrara and it would not have been possible without the invaluable contributions of several individuals.

A special thanks to Prof. Cesare Stefanelli from the Department of Engineering, University of Ferrara and Prof. Carlo Giannelli from the Department of Mathematics and Computer Science, University of Ferrara for their guidance.

A special thanks to the members of North Atlantic Treaty Organization (NATO) Information Systems Technology (IST)-193 Research Task Group (RTG) for their inputs, contributions, and support within a stimulating and cooperative setting.

Finally, thanks to my family and friends, who have supported and encouraged me in ways that do not need elaboration.

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface. 8, 19, 25, 34, 51, 59, 61, 63, 66, 68, 69

AWS Amazon Web Services. 17, 21, 24, 26, 27, 70

BS Base Station. 40, 41

CDT Composed Digital Twin. 106, 112–116

CP Control Plane. 31, 33, 34, 37, 40, 41

CPS Cyber-Physical System. 1, 11

DNS Domain Name System. 52

DSL Domain Specific Language. 24–26

DT Digital Twin. 1–7, 11, 12, 14, 16, 17, 19–30, 51, 52, 54–72, 74, 77–79, 81–89, 91, 93, 95–101, 103, 105–109, 112–117, 119–121

DTDL Digital Twin Definition Language. 26

EI Eastbound Interface. 33

ERP Enterprise Resource Planning. 9, 10

FANET Flying Ad Hoc Network. 41

FE Forwarding Element. 32, 38

FIB Forwarding Information Base. 31

ForCES Forwarding and Control Element Separation. 33

FP Forwarding Plane. 31–34, 39, 40

HMI Human-Machine Interface. 8, 10, 53

IIoT Industrial Internet of Things. 8, 11, 79, 99, 105–107, 117

IIRA Industrial Internet Reference Architecture. 10

IoT Internet of Things. 1, 8, 11, 82, 83

IST Information Systems Technology. i

IT Information Technology. 7, 8, 10, 30, 50, 52, 55, 79, 81, 82

JSON JavaScript Object Notation. 17, 20, 23, 97

K3s Lightweight Kubernetes. 48

K8s Kubernetes. 48, 49

KubeEdge Kubernetes Native Edge Computing Framework. 48, 49

LC Local Controller. 37

MANET Mobile Ad Hoc Network. 40

MDT Microservices Digital Twin. 96, 103, 105, 109, 119

MEC Multi-access Edge Computing. 10, 19, 70, 89, 113, 114

MES Manufacturing Execution System. 9–11, 55

MP Management Plane. 33, 34

NASA National Aeronautics and Space Administration. 14

NAT Network Address Translation. 47

NATO North Atlantic Treaty Organization. i, 13

ND Network Device. 31–36, 38, 39

NI Northbound Interface. 34

NTK Need-To-Know. 13

ODTE Overall Digital Twin Entanglement. 1, 6, 83–86, 88, 91, 93, 97, 100–105, 113–115, 117, 119, 120

OEE Overall Equipment Effectiveness. 83

OPC UA Open Platform Communications United Architecture. 8

OS Operating System. 42–44

OT Operation Technology. 7–10, 12, 30, 50, 52, 79, 81, 82, 98

PLC Programmable Logic Controller. 8, 10, 55

PT Physical Twin. 5, 14, 20–28, 51, 56–62, 65, 67, 68, 70, 72, 74, 78, 79, 81–84, 86–88, 91, 93, 99, 100, 103, 105–107, 113, 114

QoE Quality of Experience. 82, 83

QoI Quality of Information. 82

QoS Quality of Service. 47, 51, 54, 55, 82, 83

RAMI 4.0 Reference Architecture Model Industrie 4.0. 10, 11

RC Root Controller. 37

REST Representational State Transfer. 8, 61

RF Radio Frequency. 12

RTG Research Task Group. i

SA Situation Awareness. 13

SDFN Software-Defined Flying Network. 39–41

SDMAN Software-Defined Mobile Ad hoc Network. 39, 40

SDN Software-Defined Networking. 6, 29, 30, 32–34, 36, 38, 54

SDT Serverless Digital Twin. 97, 103–105, 119

SDVN Software-Defined Vehicular Network. 39, 40

SDWMN Software-Defined Wireless Mesh Network. 39

SDWSN Software-Defined Wireless Sensor Network. 39

SenML Sensor Measurement Lists. 106

SI Southbound Interface. 33, 34

SPoF Single Point of Failure. 34, 36

TEN Tactical Edge Network. 13, 30

TN Tactical Network. 12–14, 48, 54

TOC Tactical Operations Center. 12, 13

UAV Unmanned Aerial Vehicle. 13

VM Virtual Machine. 42, 43, 51, 98

VMM Virtual Machine Monitor. 41, 42

WI Westbound Interface. 33

WLDT White Label Digital Twin. 96, 116

Contents

Chapter 1

Introduction

Digital Twins (DTs), whose overarching ambition is to serve as bridges between the physical and virtual worlds, have been gaining momentum in both academia and industry. Originally developed for the manufacturing sector, DTs nowadays play a crucial role in a plethora of different domains, ranging from the Internet of Things (IoT) to Cyber-Physical System (CPS).

This work proposes a vision of DTs that revolves around the concept of entanglement, which is what primarily distinguishes DTs from traditional software components. However, the limitations of existing platforms and metrics do not support this vision. To bridge this gap, not only does this work provide a novel perspective on what DTs are, but it also engineers, discusses possible implementations of, and experimentally evaluates an entanglement-aware DT ecosystem. In this ecosystem, DTs expose information about their entanglement, and the middleware orchestrates them accordingly.

Fundamentally, the strands of the entanglement-aware DT ecosystem are:

- A metric to measure the entanglement. In this regard, the original contribution is Overall Digital Twin Entanglement (ODTE)—a concise yet expressive metric to measure entanglement.

- Entanglement-aware DTs. This work delves into DT engineering with a focus on entanglement awareness. It originally discusses how to apply software design patterns for building DTs as microservices and a methodology for building DTs with serverless functions.

- Entanglement-aware middleware. The novel contribution primarily lies in the middleware orchestration capability of enforcing the desired quality of entanglement despite failures along the cloud-to-edge continuum.

As there is no consensus on a definition of DT, the following aims to investigate what DTs are and explore the meaning of entanglement in this context.

Intuitively, *a DT is a virtual representation of a physical object.* Imagine a DT of a self-driving car. Engineers could work on the virtual representation running on a computer, without actually putting any car on the road. Note that there are no constraints against creating virtual representations of intangible objects as well. In this regard, consider a DT of a patent. Such a DT would record all the information about an invention, making it accessible online for research and reference, just like the original patent document. Therefore, a DT may be tentatively defined as a virtual representation of an object, whether tangible or intangible.

However, *the virtual representation would probably differ from the actual representation.* This might happen intentionally or accidentally. It occurs intentionally when the design of the DT itself rules out the possibility of having an exact representation. In fact, a subset of the properties of the object is typically of application interest. Only such properties are therefore used to build the virtual representation. In some cases, instead, it is not possible to represent the properties of an object in the virtual space because, for example, of technological limitations. But even in those cases where an exact representation would be in principle possible, it might be eventually ruled out because it is irrationally expensive. Discrepancies between virtual and actual representations may occur accidentally when the design aims to provide an exact representation, but it turns out that this is not the case.

In light of the aforementioned considerations, what is the value of a DT, as tentatively defined so far? First, *a virtual representation is virtually accessible from anywhere and at any time*, as long as the code that describes it, the computer that runs the code, and the network that connects the computer to the Internet are functioning properly. This means that it is no longer required to be physically close to the object to observe it. In other words, a DT physically decouples the object from the observer. Moreover, *a virtual representation can augment the actual representation.* A virtual representation is augmented when it describes things that the actual representation does not. Let us consider a temperature sensor that detects the environment every minute and measures the temperature. A DT of such a sensor could not only represent the state of the sensor (i.e., the current measured temperature) in the virtual space, but also provide analytics derived from historical data (e.g., the maximum measured temperature). In this example, analytics are augmentation functions.

It might be argued that it is possible to physically decouple an object from the observer by simply connecting the object to the Internet, with the underlying assumption that the object has an adequate interface to expose information about itself. However, it is worth noting that the set of these objects is rather limited compared to the class of any existing object. For example, a building does not belong to that set because it lacks computational and communication capabilities. Another counterargument might be that the devices deployed in the building can be connected to the Internet and, therefore, provide information about the building. Although these devices can describe what they observe about the building (e.g., how many people are within the building), they fall short in providing explanations about it, let alone the emergent phenomena (e.g., why is there no one on Sunday and Saturday?). This is also where the tentative definition falls short. *A DT includes not only the virtual representation of the object, but also a model of the object.* To some extent, a DT has an explanatory power greater than that of the object it represents.

The model may be used to conjecture about what might happen or what might have happened. When the object goes offline unexpectedly (and consequently stops sharing information with the DT, making it unable to provide an up-to-date virtual representation), the DT can conjecture about the object offline status. For example, if the most recent update from the object indicated a low battery level, the DT could employ the model to calculate the energy consumption since then, thus deducing whether the object could have depleted its battery during that interval. The model can also be used to make predictions. In this regard, a typical example is predictive maintenance, which seeks to prevent equipment failures before they occur. As a result, *a DT decouples the object from the observer in both space and time.* As mentioned above, the former removes the requirement for physical proximity between the object and the observer. The latter refers to the capability of a DT to revert the virtual representation of the object to a past state, depict its current state, predict future states, and speculate about potential events (i.e., things that did not happen but might have or things that did happen but the outcome is still unknown).

But to what extent can a DT depict the current state of the object it represents in the virtual space? In practical terms, how can an observer be sure that the DT and the object are in the same state? Note that the DT and the object would likely never be in literally the same state. This is because, as mentioned, the virtual representation would probably differ from the actual representation, either intentionally or accidentally. Thus, the expression "the same state" is meant to refer to the alignment of the object properties as represented by the DT in the virtual space with those of the actual object at a specific point in time. But it is also physically impossible to transmit information instantaneously. Therefore, *there is an inevitable time lag between when the object changes state and when the DT mirrors it.* Note that even assuming instantaneous exchange of information, such a time lag would persist. In fact, state transition requires computation, which in turn takes time.

So far, it has only been considered a unidirectional communication flow—from the object to the DT. It is possible, though, that an object not only shares information about itself but also provides a means to receive commands. If so, *a DT would be able to take actions on the object, thus potentially transforming the physical space.* This also implies a bidirectional communication flow—from the object to the DT to build the virtual representation and, vice versa, from the DT to the object to change the physical space. A DT can send commands to an object either in response to an observer's request or through autonomous decision-making. Imagine that an observer (e.g., an application or a human being) makes a request that requires the DT to take action. For example, a technician who interacts with a DT of a production line might send a single request to switch production mode. Note that the observer might not even know the sets of underlying actions that the DT is pushing downwards, i.e., towards the object. Behind the scenes, the DT might translate that request into several low-level requests for the single components of the production line to make that switch happen. It is also possible that such a DT does not directly interact with the physical objects either. A composition of DTs might be in place, with a high-level DT, the one the technician interacts with, communicating with low-level DTs, each representing a single component of the production line.

This bidirectional communication pattern introduces an additional requirement. Ensuring that the observer and the DT are in the same state is not sufficient. For example, consider a light bulb that cycles on and off regularly, a DT representing it virtually, and an observer interacting with the DT to turn off the light bulb. If the DT is in the "off" state, is the light off because the DT command was executed correctly, or is it off due to its regular cycling? This is an instance of what may be defined as the entanglement problem in the context of DTs. In particular, *a DT and the object that the DT represents in the virtual space are entangled if (i) the object virtual representation mirrors the actual representation over time, and (ii) the object behavior aligns with the issued commands.* The mirroring between the virtual representation and the actual representation refers to the subset of object properties of application interest. For those DTs that do not issue commands, only the first condition affects the entanglement.

In conclusion, *a DT is a virtual entity entangled with an object, whether tangible or intangible, of which the DT provides a (augmented) representation in the virtual space, thus decoupling the object from the observer in space and time.* Since this work mainly deals with physical objects, the object that a DT represents will be referred to as a Physical Twin (PT) in the following.

The reminder of this thesis is structured as follows.

Chapter 2 introduces two cloud-to-edge continuum scenarios that can benefit from the adoption of DTs. The first scenario is Industry 4.0. In this context, several research efforts have already explored the role that DTs can play throughout the production cycle. The other scenario is coalition tactical operations. The rationale behind the inclusion of this scenario is that tactical environments pose unique challenges, thus revealing facets of what DTs can do or where DTs can be used that would remain largely unexplored otherwise.

Chapter 3 lists the identified requirements for an entanglement-aware DT ecosystem, which consists of DTs and the middleware to orchestrate them. Each requirement is examined from three angles: DTs, the middleware, and what are the limitations of existing platforms.

Chapter 4 discusses the foundational technologies that serve as the building blocks for the entanglement-aware DT ecosystem. Specifically, this chapter delves into three technologies—Software-Defined Networking (SDN), container orchestration, and chaos engineering. It also includes dedicated in-depth examinations aimed at demystifying the application of a technology beyond commonly accepted frameworks. For example, Kubernetes is the de facto industry standard container orchestration system, making its adoption in the context of Industry 4.0 self-evident. However, the challenge lies in demonstrating its effectiveness in a tactical environment, where the assumptions of modern industrial settings no longer apply.

Chapter 5 investigates how to engineer DTs. Specifically, it proposes how to apply software engineering design patterns to build DTs based on microservices. Furthermore, this chapter explores the development of DTs within the framework of serverless computing. In both cases, the design is inherently event driven. The primary distinction lies in the core abstractions, with containers being central in the former case and functions in the latter.

Chapter 6 introduces the entanglement-aware DT ecosystem. This chapter provides a technical perspective on the entanglement problem and proposes the ODTE metric to measure the quality of entanglement. Then, it details an architecture for entanglement-aware DTs and their life cycle. Lastly, it describes the architecture of the middleware to orchestrate such DTs. The key feature of this middleware is entanglement-aware orchestration along the cloud-to-edge continuum.

Chapter 7 offers an overview of the implemented components of the entanglement-aware DT ecosystem, covering both microservices and serverless implementations. This chapter then delves into the experimental evaluation of these implementations, which includes a performance assessment of the entanglement-aware DTs, an analysis of the overhead of the entire ecosystem, and an exploration of the effectiveness of the middleware orchestration capabilities along the cloud-to-edge continuum.

Chapter 8 provides conclusive remarks and discusses future work.

Chapter 2

Digital Twins in the Cloud-to-Edge Continuum

This chapter introduces two scenarios where DTs can unleash their potential. Although quite different, they both include the cloud-to-edge continuum in their modern settings. These scenarios are Industry 4.0 (see Section 2.1) and coalition tactical operations (see Section 2.2). Each scenario is first described in detail, followed by a discussion of the role DTs could play. As opposed to Industry 4.0, where there exists extensive research literature, DTs are largely unexplored in the context of coalition tactical operations. Consequently, Section 2.2 takes on a more speculative nature. The unique challenges that tactical environments pose make it possible to explore facets of DTs that would remain largely unexplored otherwise. Lastly, Section 2.3 provides a summary of the chapter.

Original contributions are as follows. Industry 4.0 is presented through the lenses of the Purdue model, considering the convergence between Operation Technology (OT) and Information Technology (IT), and offering an overview of its recent evolution towards the cloud-to-edge continuum. At the tactical edge, where coalition tactical operations occur, several assumptions that hold for enterprise environments no longer apply. The notion of *tactical cloud* is introduced, and the cloud-to-edge continuum is discussed accordingly. For each scenario, an analysis of the role of DTs is provided, considering instances where they have already proven effective and envisioning their potential for future advancements.

2.1 Industry 4.0

The spread of IoT within industrial environments has recently enabled easier monitoring and control of industrial equipment from remote locations, e.g., via Representational State Transfer (REST) or Open Platform Communications United Architecture (OPC UA), thus fostering the advent of the fourth industrial revolution. Initial attempts to implement the Industry 4.0 paradigm relied on ad hoc approaches, allowing technicians and industrial applications to directly interact with machines through their Application Programming Interfaces (APIs). This trend promoted the integration of OT, i.e., the part related to industrial machines and automation, and IT, i.e., the part related to data management and processing. However, this has raised several issues related to, among others, machine heterogeneity and proper management. In particular, industrial machines typically offer non-standard APIs, which require users to be familiar with machine-specific details. Since commands and information are sent and retrieved directly, this may not only lead to issuing contradictory, if not even inconsistent, commands, but also to querying machines too frequently. These issues become more apparent when considering and properly modeling the actual organization of modern industrial environments, which comprise the shop floor, plant, and enterprise levels.

The shop floor level primarily focuses on industrial automation. Its main components include industrial machines, Programmable Logic Controllers (PLCs), Human-Machine Interfaces (HMIs), and Industrial Internet of Things (IIoT) devices. Industrial machines tend to have extremely long lifetimes (between 10 and 15 years, if not even longer in some cases) and may implement different (proprietary) protocols. In addition, software upgrades may not always be possible, since manufacturers usually forbid software upgrades for safety reasons, or industrial machines may not support them at all. In contrast to industrial machines, IIoT devices are characterized by a substantially shorter lifetime, usually communicate via well-known protocols, and support monitoring and control capabilities while being low cost. It is worth mentioning, however, that IIoT devices also present complex chains of software dependencies (e.g., third-party libraries), thus making integrity mechanisms challenging to be guaranteed [1].

The plant level regards the management of manufacturing processes. The crit-

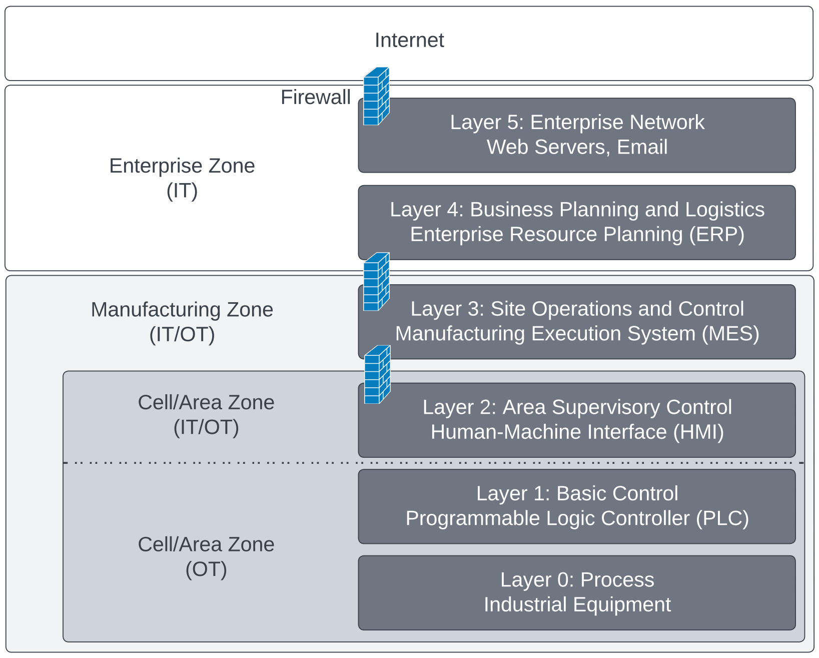

Internet
Firewall
Enterprise Zone
(IT)
Layer 5: Enterprise Network
Web Servers, Email
Layer 4: Business Planning and Logistics
Enterprise Resource Planning (ERP)
Manufacturing Zone
(IT/OT)
Layer 3: Site Operations and Control
Manufacturing Execution System (MES)
Cell/Area Zone
(IT/OT)
Layer 2: Area Supervisory Control
Human-Machine Interface (HMI)
Layer 1: Basic Control
Programmable Logic Controller (PLC)
Cell/Area Zone
(OT)
Layer 0: Process
Industrial Equipment

Figure 2.1: The Purdue model.

ical component is the Manufacturing Execution System (MES), which allows information to flow upstream and downstream between the shop floor level (where industrial machines produce goods) and the enterprise one (where managers make decisions). In particular, the MES receives instructions about how industrial machines should behave from operators, and then it transmits such instructions downwards, i.e., towards the shop floor.

The enterprise level is about making decisions on how to plan business operations. In this regard, decision-makers rely on the Enterprise Resource Planning (ERP), which collects information about supply chains, cash flows, customer orders, and production processes, to decide what, when, and how many products should be manufactured.

The most common network implementation of this logical structure is arguably the Purdue model [2], which organizes the industrial network into three zones and six layers (see Figure 2.1). The cell/area zone is the bottom one, comprises layers 0 to 2, and concerns OT. The shop floor components that craft goods belong to layers 0 and 1. Such layers rely on a time-sensitive network connecting industrial machines and PLCs, while devices that control crafting processes, e.g., HMIs, reside in layer 2. The manufacturing zone resides in the middle. It contains layer 3, which traditionally included only OT. With the convergence of OT and IT, it now encompasses both OT and IT, including those components that manage the manufacturing process as a whole, e.g., the MES. At the top there is the enterprise zone, which comprises layers 4 and 5. Such layers primarily provide IT-oriented functionalities and facilities, such as web servers, email servers, databases, and the ERP system, to name a few. Recently, industrial network implementations have evolved into multi-domain infrastructures, with some software components being deployed off-premises within the so-called cloud-to-edge continuum. In this context, the cloud-to-edge continuum refers to a distributed computing paradigm that spans from edge computing resources managed at the digital factory level, to Multi-access Edge Computing (MEC) resources deployed in close proximity to the digital factory but managed by telecommunication operators, and extending to data centers managed by cloud providers. Consequently, in the current landscape, the enterprise zone of the Purdue model now encompasses multiple heterogeneous domains, even potentially owned by different actors.

When Industry 4.0 started to emerge, Reference Architecture Model Industrie 4.0 (RAMI 4.0) aimed to align the Purdue model with modern industrial environments [3]. Compared to other emerging standards that mainly focused on how smart appliances and related data are managed, such as Industrial Internet Reference Architecture (IIRA) [4], RAMI 4.0 was a better fit for the larger scenario of the smart value chain, also properly handling the development, deployment, and maintenance of smart appliances [5]. In particular, RAMI 4.0 provides functional descriptions per component that illustrate how the (smart) product life cycle can interact cross-layer with any other component of the factory. This ranges from field and control devices to stations, work centers, and the enterprise as a whole. In other words, RAMI 4.0 envisions a more open architecture in comparison with the Purdue model, characterized by partial deperimetration, i.e., even manufacturing zone borders tend to be blurred, thus allowing industrial components and controllers to interact more freely. The ultimate goal is to maximize the flexibility of the system by integrating the factory environment with the external world.

RAMI 4.0 greatly fostered the discussion among academic and industrial researchers on the proper architecture on which Industry 4.0 applications should be based, with the notable positive effect of disseminating the broader concept of smart factory. However, its actual implementation is still far from being achieved (if it ever will be), since this high-level model has not always reflected actual requirements and capabilities of real-world industrial environments. In particular, the envisioned cross-layer interactions among factory components have been shown to be very complex to develop and manage. In fact, most industrial machines are configured to receive control messages from the control room (by the MES above all) and send back some information about their current state (e.g., number of crafted and faulty products). Sporadically, industrial machines also exchange messages with each other, e.g., machines in the same production line share information about the rate of crafted products. In any case, once industrial machines and companion control servers are deployed, their dynamic reconfiguration is not possible, e.g., requiring to stop production to reroute messages towards a different control server.

2.1.1 Digital Twins for Industry 4.0

Recently, the role of DTs has been re-analyzed and re-shaped both by the scientific and the industrial communities. The primary objective is to clearly identify their definitions and responsibilities, as well as to identify new challenges and opportunities among different application domains, in particular in relation to IoT and IIoT [6]–[9]. A shared reference architecture [10] has been proposed in the context of the Industrial Internet of Things Consortium, taking into account DT relationships, composition, and main services (e.g., prediction, maintenance, and safety). Such architecture also covers different production stages and use cases, in particular related to manufacturing [11] and product design [12]. DTs are increasingly being considered a part of CPS architectures, realising twin models of assets and machines [13], the computational modules of the physical components of CPSs [14], or within the RAMI 4.0 ecosystem as an important pattern for the manufacturing process and the administration shell [15].

DTs have already been proposed and investigated in many different industrial applications and contexts, such as job scheduling [16], resource management [17], network traffic prediction [18], anomaly detection [19], zero defect manufacturing [20], and structural health monitoring [21]. Since DTs had proved to be effective in a wide range of use cases, they became a critically important element of the smart factory infrastructure. However, the deployment of DTs is everything but trivial. In this regard, it is worth mentioning the heterogeneity of involved devices, ranging from simple vibration sensors to complex drilling/assembly tools, and the adoption of emerging standards for securing the OT domain, such as IEC 62443 for cybersecurity that pushes for network segregation [22].

2.2 Coalition Tactical Operations

Nowadays, ever-increasing processing and storage resources are available at all echelons, from operations centers to tactical units. However, tactical edge communications still suffer from scarce network resources. In fact, Tactical Networks (TNs) are wireless and ad-hoc, implying unreliable connectivity, limited bandwidth, and variable latency [23]. The absence of a fixed infrastructure, as in the civilian environment, implies that the nodes must implement routing functionalities to enable communication capabilities, and therefore the loss of connectivity of one node may affect connectivity to other nodes. Also, the hostility of the environment in which they must operate implies reduced performance due to high churn rate, Radio Frequency (RF) interferences, and poor connectivity. It is also necessary to take into account attacks from enemies that might cause network partitions as well as the movement of air, sea, and land troops affecting the topology of the network continuously. Additionally, modern military missions typically involve coalition operations, where heterogeneous mission partners (even belonging to different nations) cooperate in the field. As a result, the distribution of mission-critical information is more complicated than ever. On the one hand, the dynamic nature of the tactical environment frequently disrupts communications. On the other hand, individual resource sharing policies prevent mission partners from taking full advantage of available resources in situ.

Coalition tactical operations typically involve several operational domains, potentially commanded by distinct Tactical Operations Centers (TOCs) belonging to different nations [24]. As a result, the battlefield comprises a plethora of heterogeneous assets that need to communicate mission-critical information to each other as close as possible to real time to enable Situation Awareness (SA) dissemination [25], which is crucial to improve decision making. Such information distribution should adhere to the Need-To-Know (NTK) principle [26] that allows the sharing of sensitive information directly at the tactical edge, without having to resort to the traditional path of going up one command hierarchy before coming down a second hierarchy. Another crucial aspect of coalition tactical operations concerns the dissimilarity of computational capabilities (i.e., network, storage, and processing [27]) between the decision-making level (i.e., TOCs) and the tactical operational level (i.e., mobile dismounted units deployed in the tactical field). In fact, the former benefits from abundant resources, whereas the latter cannot take resources for granted. At the tactical operational level, Tactical Edge Networks (TENs) interconnect mobile units in self-organizing wireless multi-hop networks, enabling communications on the battlefield without relying on preexisting network infrastructures. Such interconnected units differ greatly in terms of computational capabilities, ranging from resource-rich aircraft, battleships, and vehicles to resource-constrained dismounted soldiers and Unmanned Aerial Vehicles (UAVs) [28].

Nations are starting to adopt cloud environments in military contexts. For example, in January 2021 the NATO has selected the company Thales to provide a defense cloud solution. This cloud solution should be deployable from the headquarters level down to a forward base level [29]. The tactical edge level, primarily consisting of dismounted soldiers and vehicles on the move, is not integrated into this cloud environment. Similarily, in September 2021, the European Commission started a project for a "Military multi-domain operations cloud." In the military context, a cloud is still a pool of general-purpose resources available on-demand to run services. This pool is also a location-agnostic environment where services can transparently migrate across available resources. However, either connectivity to the cloud or connectivity inside the cloud (or both) relies on TNs. In contrast to a general cloud, this kind of cloud is called a *tactical cloud*. For a tactical cloud, two scenarios need to be discussed. The first scenario deals with a tactical cloud in the mission infrastructure. Such a cloud consists of a small data center at the compound level or on a ship. As a result, the connectivity between cloud nodes is fast and reliable. Connectivity outside the tactical cloud is instead limited. The second scenario regards a tactical cloud spanning different platforms on the move, e.g., vehicles, ships, or aircraft moving in the theater. In this case, connectivity outside (cloud-to-cloud/other network) and inside the tactical cloud relies on TNs. Figure 2.2 illustrates cloud-to-cloud and cloud-to-edge communications in a federated cloud infrastructure between two nations. In particular, partner clouds are deployed at headquarters and at the level of platforms (e.g., vehicles, ships, and aircraft), whereas dismounted soldiers connect as edge nodes. In this context, instead, the cloud-to-edge continuum includes resources from the tactical edge to the headquarters, with tactical clouds potentially on the move and belonging to different nations.

2.2.1 Digital Twins for Coalition Tactical Operations

In the military field, DTs have primarily been investigated in the aviation and aerospace sectors [30]. Although examples in the literature are limited, DTs have proven effective in supporting the life cycle of costly, difficult, and risky products, such as rocket engines. In this context, DTs are typically viewed as passive entities. Glaessgen et al. [31], outlining the role of the DT paradigm for future National Aeronautics and Space Administration (NASA) and U.S. Air Force vehicles, defined DTs as "an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin." From this point of view, it is evident that DTs are not intended, for example, to send commands to their physical counterparts.

In coalition tactical operations, the potential role of DTs remains largely unexplored. Before delving into the issues that DTs could address, it is crucial to highlight the specific nuances of this scenario that might initially hinder their effectiveness. Connectivity issues at the tactical edge could render real-time operations almost impossible, especially when dealing with large volumes of data. A potential countermeasure could involve relocating the DT as close as possible to its PT, minimizing the raw data flowing over the network. Although computational resources are not as scarce as network ones, the potential demands of heavy models

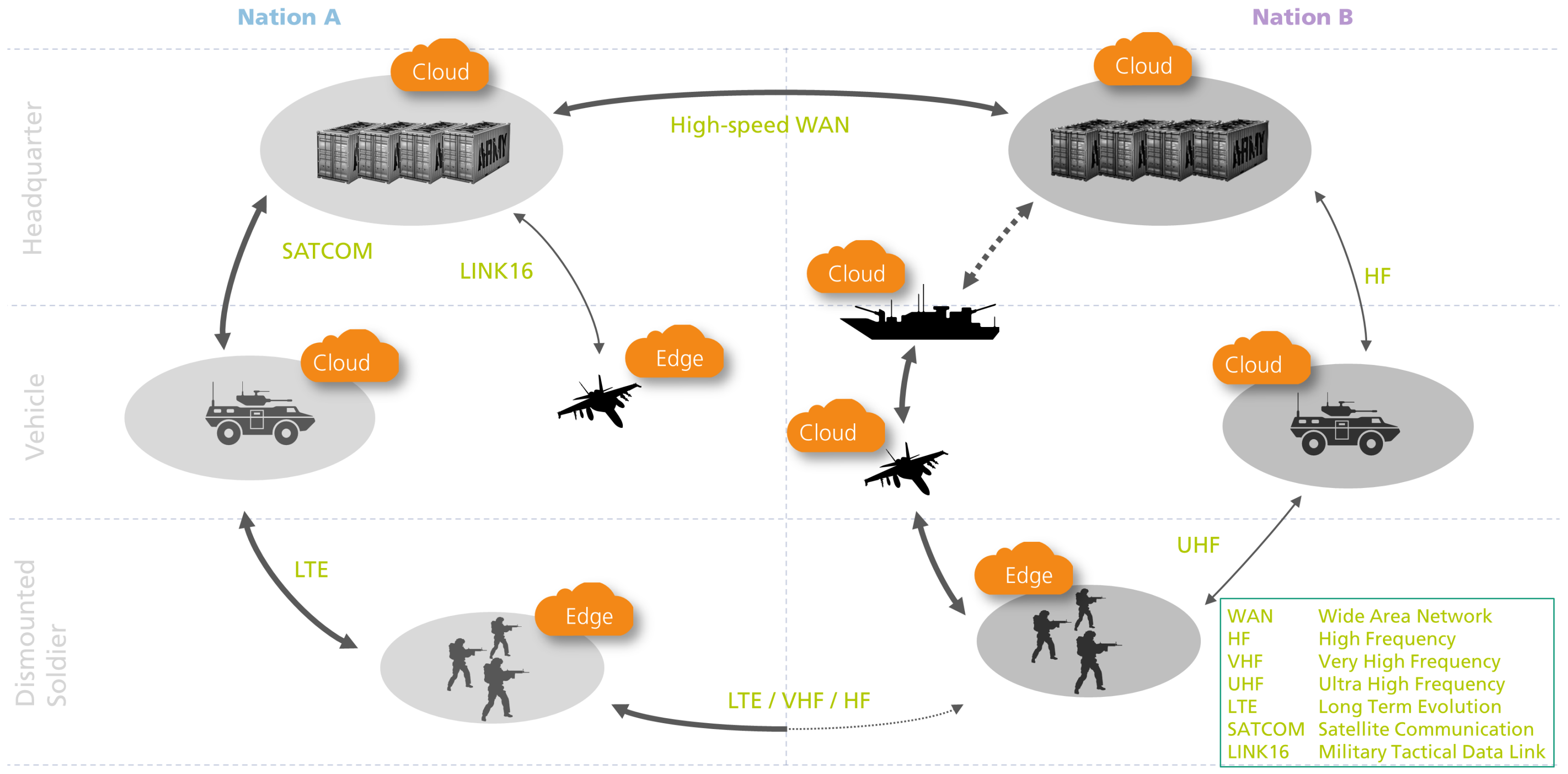


Figure 2.2: Schematic overview of a federated cloud infrastructure.

might not be met at the tactical edge. Another challenge is the transient availability of computational resources, with priorities and external factors (e.g., enemy attacks) dictating their fluctuation. Lastly, DTs would likely become targets of cyber-security attacks. This becomes even more evident when DTs are envisioned as active entities capable of directly influencing their counterparts.

In a federated cloud infrastructure, DTs would be services within tactical clouds. Within this heterogeneous domain, DTs could play a crucial role in federating underlying assets, serving as gateways for one nation to expose its assets to others in accordance with the owner's policies. Additionally, as DTs build virtual representations of the objects they are entangled with, they would enhance the observability, manageability, and accountability of the underlying assets. This is in addition to those tasks where DTs have already demonstrated effectiveness, such as predictive maintenance, anomaly detection, and resource management.

2.3 Chapter Summary

This chapter discussed two scenarios that incorporate the concept of the cloud-to-edge continuum in their modern settings. These scenarios are Industry 4.0 and coalition tactical operations. Each scenario was thoroughly described, followed by an analysis of the potential roles that DTs could fulfill.

Section 2.1 outlined the logical structure (i.e., shoop floor, plan, and enterprise levels), along with its most common network implementation (i.e., Purdue model), of modern industrial scenarios. The identified challenges range from heterogeneity to security. In this context, DTs have already proven to be effective in various use cases, such as job scheduling, anomaly detection, and zero defect manufacturing.

Section 2.2 explored the unique peculiarities of coalition tactical operations, such as scarce and transient resources, potential attacks by enemies, and the presence of heterogeneous operational domains. In this demanding scenario, DTs could prove beneficial not only at the federation level, but also in enhancing the observability, manageability, and accountability of the underlying assets.

Chapter 3

Entanglement-Aware Digital Twin Ecosystem: Requirements

To meet the need for cost-effective development and deployment, big tech companies have started to provide platforms to create and operate DTs, e.g. Microsoft Azure, Amazon Web Services (AWS), and Eclipse Ditto. Although they are feature rich and production grade platforms [32], they intend DTs as centralized, passive entities based on JavaScript Object Notation (JSON) files instead of an ecosystem of entities that coexist with an orchestration environment.

To bridge this gap, this chapter delineates the requirements for an entanglement-aware DT ecosystem. In this work, the word "ecosystem" encompasses both DTs and the orchestration middleware responsible for their effective management in the cloud-to-edge continuum. Figure 3.1 provides a schematic representation of the ecosystem. Each identified requirement is approached in three steps: (i) definition, (ii) exploration of how DTs and the middleware may fulfill the requirement, and (iii) examination of the limitations of existing platforms in supporting the requirement. Additionally, related work is presented throughout the discussion for each requirement, with an analysis of the gaps that need to be bridged in existing platforms. The identified requirements are cloud-to-edge mobility (see Section 3.1), variable load resilience (see Section 3.2), entanglement awareness (see Section 3.3), life cycle (see Section 3.4), declarative application description (see Section 3.5), and accountability (see Section 3.6). Lastly, Section 3.7 provides

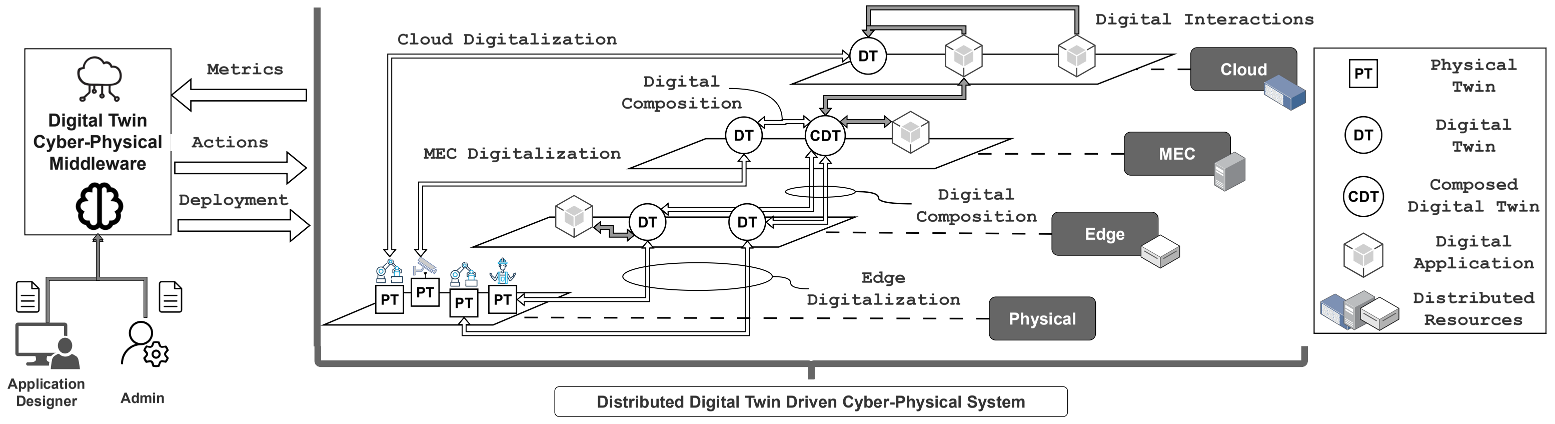


Figure 3.1: Ecosystem of DTs in the cloud-to-edge continuum.

a summary of the chapter.

The novelty of this chapter lies in the identification of the requirements for an entanglement-aware DT ecosystem. The discussion of such requirements makes it clear why they are essential for such an ecosystem and where the examined platforms fall short in supporting them. A key takeaway is that none of the examined platforms supports all the identified requirements, emphasizing the need for a tailored ecosystem.

3.1 Cloud-to-Edge Mobility

Definition: *A mobility-capable DT ecosystem supports mobility along the cloud-to-edge continuum and allows individual containerized DTs to be transparently migrated to the hosting domains that best fit the application constraints.*

Computing and communication resources can be owned by different providers and located in different domains, such as edge on-premises (e.g., digital factories), MEC (e.g., telco networks), or in the cloud (e.g., big tech companies). Each solution has benefits and drawbacks: public clouds offer lower investment costs but less predictable performance w.r.t. edge solutions, whereas edge solutions provide tenants with full control and likely the highest performance in exchange for higher maintenance costs [33]–[36]. The capability of transparently moving DTs along the cloud-to-edge continuum is crucial to fulfilling entanglement requirements. A DT can be dynamically (re)deployed to the location that best suits its needs, either closer to its physical counterpart or to where more resources are available.

Digital Twins: Nowadays, containerization is the key to efficiently supporting mobility. As such, DTs should be packaged with all the necessary configuration files, libraries, and dependencies to run across different environments reliably. This approach leads to conceive DTs as containerized microservices, using specific APIs to communicate with their peers, applications, objects, and orchestration services. Once containerized, DTs can be easily deployed on any hosting platform (thereby supporting mobility) and replicated to meet demand. Containerization is also likely to facilitate their adoption, promoting automation and standardization [37]–[40].

Middleware: The orchestration middleware migrates DT containers, optimizes the use of resources, replicates containers under excessive load (see Section 3.2), while maintaining them monitored and healthy. More specifically, it should support the following: *(i) DT mobility*: if required, a DT should be offloaded from the current location and moved to a new location; *(ii) DT service continuity*: if a DT moves to another location, the application associated with that DT should continue to run properly; *(iii) mobility of the PT state*: historical data regarding the PT state should support mobility and possibly be migrated along with the DT. Relocation procedures should minimize total migration time [41]–[43].

Limitations of existing platforms: Mobility is not supported on available commercial platforms. In particular, Azure and AWS model DTs as JSON entities capable of receiving data from PTs via a set of connectors. As such, all DTs reside on cloud nodes and cannot be moved to different hosting domains or even change tenant. Eclipse Ditto, instead, provides a library for writing DTs that could potentially help developers support mobility (i.e., Ditto DTs could be containerized), but there is no native support for such a feature and everything is delegated to the developers.

3.2 Variable Load Resilience

Definition: *A DT ecosystem resilient to variable loads supports DT replication, admission control, and resource allocation mechanisms for incoming tasks, e.g., an application requesting to observe an object.*

Applications (especially those rooted in low-latency high-bandwidth scenarios) might impose variable loads, thus possibly requiring a variable amount of resources over time. Two key factors drive the overall load on a distributed system such as an ecosystem of DTs: *(i)* the number of requests to be accomplished, and *(ii)* the complexity of those requests. To deal with peaks in the number of requests, replicas of a DT may be spawned, limiting their number according to the available resources (admission control). Concerning requests complexity, a DT model may require non-negligible resources. For example, a deployment domain without a sufficient number of GPUs or CPUs may negatively impact the responsiveness of a DT model or even completely prevent it from working. Because of this, mechanisms for allocating resources when and where they are needed should be supported (resource allocation). As for cloud-to-edge mobility (see Section 3.1), the capability to be resilient against variable loads plays a significant role in meeting entanglement requirements. For example, a DT that lacks the necessary resources is unlikely to remain consistently entangled over time. It is important to note that resource allocation cannot only be predetermined (fire-and-forget strategy). In fact, a DT whose resources are not dynamically scaled up and down to accommodate fluctuating volumes of incoming requests would also likely become disentangled.

Digital Twins: Handling variable loads implies horizontal replication of DTs. Replicas of the same DT, all associated with the same PT, should behave consistently, i.e., they should represent the same status and behavior of the PT. However, multiple replicas requiring independent synchronization inevitably increase the load on the PT. To avoid this effect, replicas may be organized in a hierarchical fashion. A primary DT directly synchronizes with the PT while it interacts with other secondary DTs, which do not interact directly with the PT. This scheme has the advantage of a centralized implementation for the primary DT, but introduces a delay in updating the secondary DTs [44], [45].

Middleware: Almasan et al. [46] recently discussed how networks of DTs can be integrated with two typical mechanisms of distributed systems: admission control and resource allocation. The admission control system maintains the network of DTs performance and availability when some DTs experience high load. When admission control is enabled, it sorts requests by priority, giving preference to higher priority operations. In particular, a tenant experiencing a high load should not degrade the performance or availability of other tenants running on the same host. In case of a positive decision from the admission control, the resource allocation mechanism verifies and, if necessary, adjusts the resources requested (see Section 3.5) according to those which are available.

Limitations of existing platforms: Microsoft Azure and AWS support some form of replication. Specifically, the data structures representing DTs are updated via serverless functions running in the cloud that can be replicated and thus support variable loads. However, the actual concept of a DT replica, intended as an active component, does not exist. On the contrary, a DT implemented with Eclipse Ditto could potentially be containerized and replicated. However, there is no native support for this feature.

3.3 Entanglement Awareness

Definition: *An entanglement-aware DT ecosystem exposes the quality of entanglement and is capable of actions aimed at safeguarding the constraints defined by applications.*

In a practical application, engineering a DT that exactly reflects the PT is difficult for a number of reasons. Nevertheless, applications are often designed and implemented in light of specific assumptions, such as the DT replies in less than 100 ms, or the PT sends updates every 200 ms [16]. Because of this, providing applications with a metric describing how well the DT is rendering its PT w.r.t. the expected performance is key. Recent works [47], [48] proposed approaches to quantify the quality of entanglement taking into account both the freshness of the data collected from the PT and the ratio between the amount of collected and required data.

Digital Twins: The need for independent communications, possibly using different protocols and timings, to *(i)* communicate with the physical counterpart(s), *(ii)* communicate with applications, and *(iii)* exchange commands, configurations, and metrics with the orchestration middleware calls for a modular internal architecture. In this regard, we conceive DTs as multi-component, multi-container entities, supposed to be pluggable and reconfigurable at runtime. Isolation and extensibility can be supported by making use of micro-kernel designs (at the component level) and subcontainers associated with different scheduling priorities and resources (at the container level) [49].

Middleware: DTs providing well-defined entanglement interfaces enable the orchestration middleware to perceive the ecosystem and plan actions to reach the service levels required by the applications. Thus, the middleware should be aware of the communication interfaces provided by DTs and use them to collect contextual data, analyze them w.r.t. application constraints, and take actions accordingly. For example, the middleware might improve the quality of entanglement of a DT by reconfiguring its communication protocols, assigning to it more resources for speeding up its internal model, spawning a replica, or migrating it closer to its PT.

Limitations of existing platforms: Commercial platforms do not embed any form of entanglement support. In fact, they only provide connectors to receive data from PTs and store them as JSON data without providing any further assistance. Developers can build entanglement-aware functionalities, for example, by enriching PT data with timestamps, but without relying on any systematic support.

3.4 Life Cycle

Definition: *A DT ecosystem should be fully aware of the cyber-physical nature of the DTs (compared to general purpose containerized software) and should support their complete life cycle: deployment, entanglement, updates (for model augmentation), and reconfiguration (for enforcing application constraints).*

Conceiving DTs as an orchestrated ecosystem acting as a medium for cyber-physical applications implies several changes w.r.t. plain microservices. Firstly, DTs should support a runtime environment (i.e., expose contextual metrics, receive commands, etc.) and enforce adaptation. Indeed, they should support a synergic decision making process in which decisions at the orchestration level are refined at the DT level and viceversa. Secondly, the orchestration middleware should be aware of the internal status of the DTs (i.e., unbound, bound, entangled, disentangled, etc.) and support their life cycle [50]. Finally, due to a possibly large number of DTs under management, the orchestration middleware should minimize human interventions and promote the automation of frequent operations, such as updates and reconfigurations [51].

Digital Twins: Containerized DTs should be reliable and dependable components preventing catastrophic failures. As such, they should adopt modular designs that allow internal modules and communication interfaces to work independently. For example, separate interfaces can be used for communicating with the physical, digital, and control (i.e. the middleware) layers independently. Other modules can be used for managing the DT model, augmentation functions, the storage of the PT history, and the entanglement. Furthermore, PTs and applications might receive updates over time due to software/security issues or changing requirements. As such, DTs should support updates (via the control interface) to reflect those changes (e.g., supporting a new network protocol introduced in the PT). Finally, since DTs might be subjected to changing operating conditions, they should support dynamic reconfigurations (via the control interface).

Middleware: The orchestration middleware cannot be a standard orchestration system (i.e., Kubernetes) but, instead, it requires specific features that account for this scenario. As such, it should receive data and send commands to/from the control interfaces of DTs, and be aware of the network topology, resources, configurations, and application constraints. In this manner, it can compare the status of the DT ecosystem with the application requirements, possibly planning adaptive actions.

Limitations of existing platforms: None of the available commercial platforms support these features. In fact, Microsoft Azure and AWS conceive DTs as centralized passive entities that do not send/receive data and commands, and do not require reconfigurations and updates. It would be possible to build containerized DTs using Eclipse Ditto, but without any systematic support from the library.

3.5 Declarative Application Description

Definition: *A DT ecosystem that supports application descriptions provides a declarative Domain Specific Language (DSL) for describing cyber-physical applications, thus enabling a clear separation of concerns between development and* *operations.*

DSLs are alternatives to general purpose languages (e.g., Java, Python, etc.). While the latter tend to be more complete, they can be time-consuming when performing domain-specific actions. A DSL reduces these issues with a simpler grammar that lends itself to the specific application domain. Developers can adopt a DSL to describe applications in terms of DTs, PTs, resources, constraints, etc. For example, the description of an application that requires five DTs deployed on edge nodes and associated to specific PTs, supporting replication, requiring one GPU each, and providing updates every 150 ms. In addition, they can offload complexity from the design and development of the application core by defining complex objects, such as composite DTs or pipelines in a human-readable fashion. For example, instead of coding a function that computes the average power consumption of a set of industrial robots within a DT (which implies additional coding, testing, and integration activities), a DSL configuration file could be used to describe the need to deploy an additional DT dedicated to receiving values and computing their average.

Digital Twins: A cyber-physical application is a comprehensive construct that unifies DTs and their PTs. Fundamentally, a cyber-physical application must contain at least one physical entity and one digital entity. Each PT has a unique identifier and is associated with metadata about the properties of application interest. Likewise, each DT has its own unique identifier, a source for deployment (e.g., the container image to be used), and a type indicating if it is simple or composed. Each DT is associated with one or more objects, carries specific deployment requirements, and provides details about its own deployment specifications.

Regarding composition, DTs should provide APIs and communication schemes for managing other DTs as if they were PTs in a hierarchical fashion. Each change in one DT that is part of a composition scheme (i.e., an observed DT) is propagated towards the upper levels of the composition scheme. The communication scheme to be used is strictly tied to the quality of representation expected by applications because keeping a composition of DTs highly entangled might require significant networking resources, possibly disrupting other services. To save bandwidth in case the composition of DTs is not observed (i.e., used) by applications, DTs might choose not to propagate updates coming from PTs to upper layers [52].

Middleware: The orchestration middleware should be able to parse DSL descriptions and enact the required actions during both deployment and operations. Firstly, during deployment, the orchestration middleware should fetch DTs and deploy them according to the specified resources (e.g., memory, disk, number of CPUs or GPUs, connectivity, etc.), and constraints (e.g., entanglement, mobility boundaries, etc.). Secondly, as described above in this section, during operations, the orchestration middleware should monitor DT metrics and plan actions aimed at safeguarding application constraints.

Limitations of existing platforms: While Eclipse Ditto and AWS do not offer any DSL, Microsoft Azure provides users with the ability to define custom DTs in self-defined terms. This capability is provided through user-provided models and represented in the Digital Twin Definition Language (DTDL) [53]. DTDL models have names and contain elements, such as properties, telemetry, and relationships, that describe what these types of entity do. However, given the passive nature of Azure DTs, DTDL can only be used to describe the DT model, thus not shaping the whole life cycle of a DT.

3.6 Accountability

Definition: *An accountable DT ecosystem gathers information, analyzes it, and takes appropriate measures based on actual data. It is also capable of producing audit trails that can be inspected when problems occur.*

A DT ecosystem integrates loosely coupled DTs into one cohesive system supporting applications expected to provide both functionally correct results and acceptable performance levels in accordance with application constraints (e.g., quality of entanglement). However, identifying the source of a failure in a DT system can be difficult: DTs can be complex, having many execution branches and invoking services from other DTs, their PTs, or even the execution node/runtime environment [54], [55].

Digital Twins: Key aspects of accountability at the DT level are: *(i) tracing and monitoring*: DTs should expose metrics and tracing information that allow the orchestration middleware to monitor their status (including the status of their host nodes) and performance. In this context, DTs are also required to maintain the status of their associated PTs, at least those associated with relevant events/decisions/actions; *(ii) logging and auditing*: DTs should log their decisions and actions (together with associated events and data). These logs should be stored in a trusted location to enable further analytics.

Middleware: The orchestration middleware should periodically collect, aggregate, and analyze DT logs, detect different types of faults, and support management algorithms for handling them whenever possible. In practical terms, accountability can be reached by keeping track of the ownership of DTs and PTs, monitoring their status and metrics, and using tracing techniques to identify which DTs are involved in each event, decision, or action.

Limitations of existing platforms: Microsoft Azure and AWS are cloud services and thus accountable by design (trusted logging is supported at the platform level). However, they do not take decisions and do not enforce actions: they receive data from tailored connectors and store them. On the contrary, a DT ecosystem should be able of taking actions (e.g., replicating, migrating, and reconfiguring DTs) for satisfying application constraints and thus should rely on an accountable decision-making process.

3.7 Chapter Summary

This chapter discussed the identified requirements for an entanglement-aware DT ecosystem, i.e., cloud-to-edge mobility, variable load resilience, entanglement awareness, life cycle, declarative application description, and accountability. The key takeaways are the following.

DTs should be conceived as containerized entities. This facilitates mobility along the cloud-to-edge continuum (see Section 3.1) and variable load resilience (see Section 3.2). For example, a containerized DT can be easily migrated to a different location to optimize resource allocation or a new replica spawned to deal with traffic surges.

What primarily sets DTs apart from traditional software components is their cyber-physical nature, with entanglement being the prominent characteristic (refer to Section 3.3). Consequently, DTs should provide information about their entanglement (as discussed in Section 3.4), and the middleware should effectively consume this information. For instance, a DT might lose its entanglement due to slow communication links with the PT or insufficient resources. In the former case, the middleware might facilitate migration to a different location, requiring cloud-to-edge mobility. In the latter, it could scale up resources, thus necessitating variable load resilience.

The need for a declarative application description arises from the cyber-physical nature of DTs. Section 3.5 introduced the concept of a cyber-physical application, which serves as a comprehensive construct for defining the relationship between DTs and PTs within an application context.

In conclusion, Section 3.6 highlighted the significance of accountability. Given the potential for various issues to arise, it is crucial to meticulously monitor events and the corresponding measures taken to address them, enabling comprehensive analysis.

Chapter 4

Entanglement-Aware Digital Twin Ecosystem: Foundational Technologies

This chapter delves into the foundational technologies for building an entanglement-aware DT ecosystem that meets the requirements delineated in Chapter 3. These technologies are SDN (see Section 4.1), container orchestration (see Section 4.2), and chaos engineering (see Section 4.3). Each technology is first discussed from an historical point of view, thus delineating its trajectory and current state of the art. While a technology may be prevalent in one domain, it might not hold the same status in another domain, and in some cases, it may even be considered impractical. If so, an in-depth discussion that debunks the misconception follows. In conclusion, Section 4.4 provides a summary of the chapter.

The reasons why these technologies are foundational for the entanglement-aware DT ecosystem are as follows. First, SDN provides high-level abstractions that make networks programmable, enabling fine-grained network management. This feature is critical to optimize network performance and flexibility. Second, the concept of DTs as containerized entities, as envisioned in the requirements outlined in Chapter 3, requires an underlying container orchestration system to ensure efficient management and deployment of these entities. The middleware plays a key role in this architecture, logically employing both SDN and container orchestration technologies. By leveraging the abstractions offered by these technologies, the middleware can effectively orchestrate both networking and computing resources. This orchestration is carried out with the overarching objective of maintaining the entanglement requirements over time. The entire ecosystem thus forms a distributed system, characterized by a complex network of interdependencies that may not become fully apparent until they are encountered in the production environment. It is in this context that certain system failures can only be detected and, more importantly, thoroughly understood. This is where chaos engineering comes in. Chaos engineering provides a systematic approach to identifying vulnerabilities and enhancing the resilience of the system, thereby playing a crucial role in the continuous improvement and robustness of the entanglement-aware DT ecosystem.

The original contributions of this chapter primarily lie in identifying the foundational technologies for building an entanglement-aware DT ecosystem and debunking some common misconceptions about their adoption in the scenarios identified in Chapter 2. The first misconception pertains to SDN, which traditionally focused on fixed wired environments but has now extended its principles to wireless ad hoc scenarios. Section 4.1.4 offers an in-depth discussion of software-defined wireless ad hoc networks, commonly found in TENs. In the domain of container orchestration systems, Kubernetes stands as the industry standard. However, Kubernetes relies on assumptions that might no longer be valid at the tactical edge. Section 4.2.3 presents an empirical study on the performance of various Kubernetes distributions under challenging network conditions. Finally, chaos engineering originated in the realm of IT systems. Section 4.3.2 explores the application of its principles in the OT domain, thus making it actionable for resilience assessment of DTs in modern industrial settings.

4.1 Software-Defined Networking

4.1.1 History

In the 1990s, active networking emerged as a set of strategies to overcome relevant network issues, such as network ossification, which is due to vertically-integrated purpose-built Network Devices (NDs) consisting of tightly coupled hardware and proprietary software, and proliferation of middleboxes (e.g., firewalls, network address translators, and load balancers). Active networks aimed to go beyond traditional packet networks by envisioning packets carrying software executed by routers/switches for manipulating packets' content.

According to [56], the metaphor of active networking was pursued through two distinct approaches, either programmable switches or capsules. The former relies on an out-of-band channel for management and an in-band channel for data transfer. Users inject programs into routers/switches through the out-of-band channel, and, in turn, routers/switches inspect data packet headers and deliver packets to programs accordingly. The latter conceives every packet (or capsule) as a piece of software. Such a so-called capsule, along its path, may pass through several routers/switches, which execute its content in an isolated environment. On the one hand, active networks offered several significant intellectual contributions towards programmable networks. On the other hand, as [57] points out, since the active networking movement missed the "killer" application justifying such a network redesign, active networks did not see widespread deployment.

If compared with active networks, subsequent research efforts adopted a more pragmatic strategy. Instead of proposing radical paradigm shifts, e.g., programmable switches and capsules, the networking research community focused on decoupling the Forwarding Plane (FP) and Control Plane (CP), moving the control logic outside NDs, i.e., outside the core network. The FP forwards packets according to a data structure called Forwarding Information Base (FIB), while the CP controls the FP by updating the FIB through a device-specific interface. Developed at Stanford in the mid-2000s, OpenFlow [58] is the most significant implementation of an open interface between the CP and FP. The research task group behind OpenFlow was interested in enabling researchers to perform experiments in their everyday campus networks, given that researchers had no practical ways to test their ideas in real-world scenarios. Exploiting a common set of features shared by Ethernet routers and switches across different network equipment vendors, OpenFlow provides a programmatic interface for manipulating the FIB. OpenFlow-enabled routers/switches (henceforth defined as OpenFlow switches) associate an action to each flow entry and establish a secure channel with a remote controller. Sim-

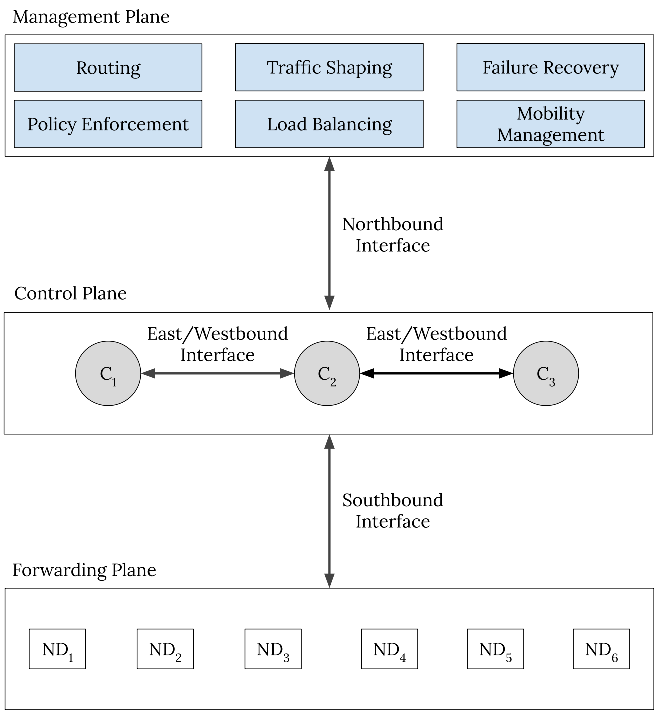


Figure 4.1: Layering and fundamental abstractions.

ilar to a traditional operating system, such a controller abstracts the underlying resources, i.e., NDs, and offers network-wide abstractions to network applications, which exploit such abstractions as building blocks to perform high-level network control and management tasks.

4.1.2 Definition and Layering

While there is no agreement on a standard architecture design [59], [60], the SDN paradigm mainly leads to a 3-tier network architecture (see Figure 4.1).

First, the control logic is removed from NDs, which become mere Forwarding Elements (FEs) that make flow-based (rather than destination-based) forwarding decisions, where a flow is defined by rules matching a set of packet header values. Accordingly, the FP (see Figure 4.1-down), also called ”data plane”, consists of wireless or wired interconnected NDs, available both in hardware (e.g., Arista 7150 Series [61], NoviSwitch 2000 Series, and Centec V350 Series [62]) and software (e.g., Open vSwitch [63], BOFUSS [64]). To mitigate the network infrastructure heterogeneity, the FP exposes its resources through abstraction models, such as OpenFlow Switch Specification [65], Forwarding and Control Element Separation (ForCES) [66], YANG [67], and Management Information Base for Simple Network Management Protocol [68].

Then, the control logic is moved in an out-of-network software entity, i.e., the SDN controller in charge of dictating the behavior of the entire network infrastructure. Thus, the CP (see Figure 4.1-center) abstracts the underlying network infrastructure and, based on its network-wide view, provides high-level abstractions to network applications. In particular, the CP is a logically (but not necessarily physically) centralized entity that comprises one or more SDN controllers.

Finally, software applications running on top of such a controller can program the whole network. Consequently, the Management Plane (MP) (see Figure 4.1-up) consists of network applications that dictate the network behavior, performing sophisticated tasks, such as routing, load balancing, policy enforcement, mobility management, traffic shaping, and failure recovery, among others. In turn, the CP is responsible for translating such high-level requests coming from the MP in low-level instructions for the FP. Summing up, network applications dictate "what to do", SDN controllers define "how to do", and NDs behave accordingly.

In addition, Southbound Interfaces (SIs) connect FP and CP by providing low-level communication mechanisms to the SDN controller so that it can remotely manage NDs in a vendor-agnostic fashion. Control-oriented SIs may be distinguished from configuration-oriented ones, where the former support the SDN controller in controlling traffic flows, while the latter in configuring NDs. Although OpenFlow is the most widely adopted control-oriented SI, alternatives are ForCES [69], Protocol-Oblivious Forwarding [70], OpenState [71], and OpFlex [72]. Instead, examples of configuration-oriented SIs are ForCES [69], Network Configuration Protocol [73], Simple Network Management Protocol [74], and Open vSwitch Database Management Protocol [75].

Eastbound Interfaces (EIs) enable controller-to-controller communications, while Westbound Interfaces (WIs) link the SDN domain with traditional networks [76]. Examples of EIs are SDNi [77], Apache ZooKeeper [78], Advanced Message Queuing Protocol [79], and WheelFS [80]. Instead, examples of WIs are Border Gateway Protocol [81] and Path Computation Element Communication Protocol [82].

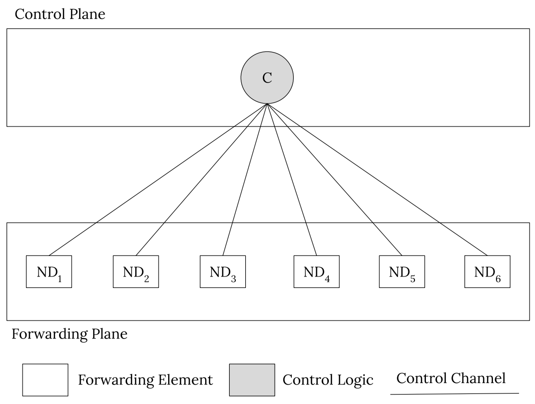
Northbound Interfaces (NIs) connect CP and MP by providing high-level services to network applications, easing network programmability. In contrast to SIs, where OpenFlow represents a de facto standard, a widely accepted NI is still missing. Therefore, the portability of network applications across heterogeneous SDN controllers is heavily limited. According to [59], [83], NIs range from ad hoc APIs, RESTful APIs, and APIs based on programming languages (e.g., Frenetic [84], Nettle [85], Procera [86], and Pyretic [87]).

By advocating for a clear FP and CP decoupling, SDN moves control logic away from NDs to controllers. This leads to a logically centralized CP responsible for monitoring, controlling, and configuring the FP. Control architectures, i.e., how controllers interact with each other, may be categorized in centralized, distributed, and federated (plus hybrid as a cross-category attribute). In addition, distributed control architectures may be further specialized in flat (each controller acts as a peer) and hierarchical (controllers are hierarchically organized). The rest of this section discusses design patterns (see Figure 4.2 and Figure 4.3) of such control architectures and provides notable examples of how well-known SDN controllers fit this taxonomy.

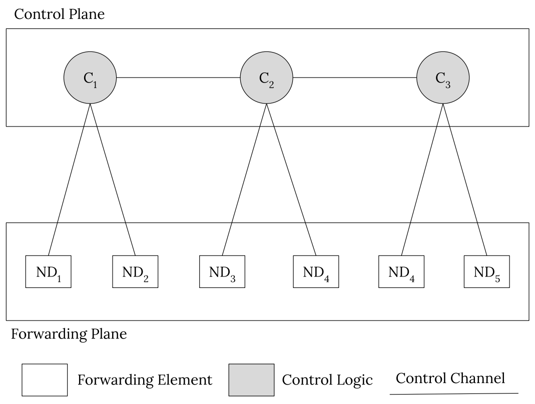
4.1.3 Control Architecture

Centralized

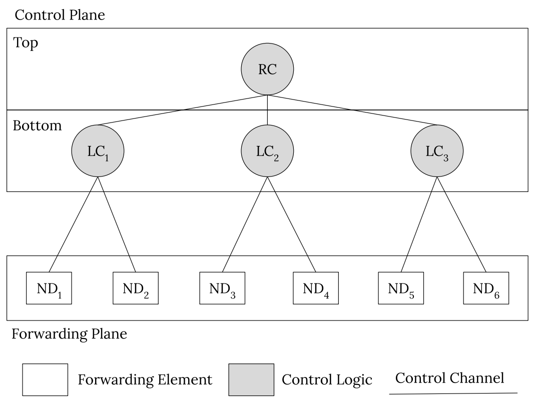
As Figure 4.2a shows, a centralized control architecture consists of a single controller responsible for every ND. Although a single controller represents the optimal design choice in terms of simplicity, it intrinsically implies scalability and resiliency issues [88]. Regarding scalability, the incoming requests may overwhelm the controller as the network grows. In fact, when a packet goes through a ND without matching any flow rule, the ND sends a flow request to the controller, which replies with the generated flow rule. Therefore, centralized control architectures, where a single controller takes over the flow setup process for the whole network, may easily result in bottlenecks, additional delays, and thus limited network scalability. In this regard, a workaround is the adoption of proactive strategies, which inject flow rules into NDs in a proactive fashion, mitigating the controller load consequently. About resiliency, a single controller represents a Single Point of Failure (SPoF),



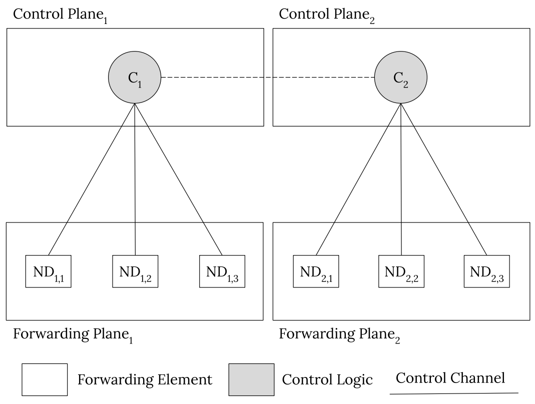
(a) A centralized control architecture.



(b) A flat control architecture.



(c) A hierarchical control architecture.



(d) A federated control architecture.

Figure 4.2: Control architectures for software-defined networks.

potentially breaking the network when NDs cannot serve incoming flows due to the controller's unreachability. Hybrid strategies (see Section 4.1.3) can overcome such a lack of resiliency by (temporarily) falling back to traditional routing protocols on NDs that cannot reach the controller.

Examples of controllers based on centralized control architectures are NOX [89], Maestro [90], [91], Beacon [92], NOX-MT [93], Floodlight [94], POX [95], Ryu [96], POF Controller [97], Meridian [98], and Rosemary [99].

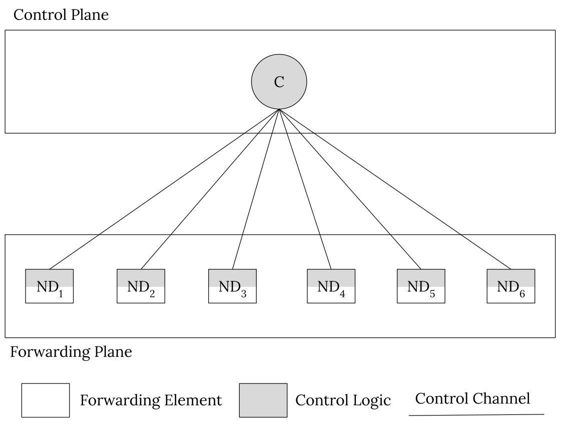


Figure 4.3: A hybrid centralized control architecture.

Distributed

To address the scalability and resiliency issues raised by centralized control architectures, the SDN community started to propose distributed control architectures in which multiple controllers interact with each other to manage the network infrastructure. On the one hand, multiple controllers can improve overall performance (by fairly sharing network load), resiliency (by avoiding a SPoF), and scalability (by on-demand deploying additional controllers if necessary). On the other hand, such controllers must accomplish coordination and synchronization tasks to maintain a global and (eventually) consistent network-wide view [100]–[102]. In addition, distributed architectures make extremely challenging the controller placement problem [103], which consists in optimizing the number of deployed controllers and their placement in the network topology.

The taxonomy further distinguishes distributed control architectures in flat and hierarchical. As Figure 4.2b shows, flat control architectures involve multiple controllers managing non-overlapping subsets of NDs. Note that controllers based on flat architectures interact with each other to share complete knowledge of the whole network, while each controller directly manages only a subset of NDs. One of the first published proposals designing a flat control architecture is HyperFlow [104], which proposes multiple controllers sharing the same consistent network-wide view, i.e., the state is fully available in each controller. HyperFlow is implemented as an application running on top of NOX. Other examples of controllers based on flat control architectures are Onix [105], ONOS [106], and OpenDayLight [107].

As Figure 4.2c shows, hierarchical control architectures assume a multi-layer CP. For example, Kandoo [108] proposes a two-layer hierarchical CP, where the bottom layer consists of multiple Local Controllers (LCs) that do not communicate with each other, while the top one comprises a single Root Controller (RC). LCs hold local-domain network views and handle frequent and small (mice) flows, while the RC takes over infrequent and huge (elephant) flows by taking advantage of its network-wide view. Given that mice flows are more than elephant ones, LCs substantially reduce the burden of the RC. Other examples of controllers based on hierarchical control architectures are Orion [109], B4 [110], and Espresso [111].

In conclusion, it is worth stressing the main difference between flat and hierarchical control architectures. In flat control architectures, all controllers share the same network-wide view ensuring optimal decision-making but leading to non-trivial issues in guaranteeing state consistency. In hierarchical control architectures, LCs, whose visibility is limited to a local network domain, provide regular updates to RCs, which aggregate such information. Therefore, only RCs (i.e., a subset of controllers) hold such a network-wide view, whereas LCs may make suboptimal decisions.

Federated

As Figure 4.2d shows, a federated control architecture assumes loosely coupled controllers (managing non-overlapping network domains) that do not share a global network-wide view. Instead, each controller shares a summary of its own state (i.e., typically a high-level abstraction of its local network view) or requests specific information to remote controllers for enabling end-to-end services. Since controllers do not share the whole picture of what they know, federated control architectures naturally fit inter-organizational scenarios, in contrast to distributed architectures that best suit intra-organizational ones. Note that the absence of a global network-wide view reduces complexity since there are no consistency strategies in place, decreases control network traffic because controllers do not share their whole state, and improves privacy thanks to the sovereignty of controllers over information sharing. An example of federated control architecture is DISCO [112].

Hybrid

The peculiarity of hybrid control architectures is to maintain a certain degree of control logic within NDs, as Figure 4.3 shows. Therefore, such hybrid solutions do not strictly follow the SDN canonical approach, which advocates for NDs acting as mere FEs. In this taxonomy, "hybrid" is an orthogonal attribute to centralized, flat, hierarchical, and federated.

As mentioned in Section 4.1.3, a typical scenario involving hybrid control architectures is when NDs reenable their local control logic (e.g., by falling back to traditional routing protocols) to face emergency circumstances (e.g., the controller's unreachability). More radical hybrid approaches take advantage of hybridization not only when unexpected events occur but on a regular basis, e.g., the controller computes forwarding weights and disseminates them to NDs that take into account such weights but make (semi-)autonomous decisions. Another example is the case of in-network processing, where NDs locally process network traffic by dropping, delaying, or aggregating packets. In particular, in-network processing, also known as in-network computing, has been proposed to support the execution on networking devices of software modules that typically run on hosts [113], [114].

Let us note that hybridization positively impacts not only on resiliency, e.g., NDs keep operating successfully despite controller failures, but also on scalability, e.g., in-network processing allows reducing the packets traversing the network.

4.1.4 Software-Defined Wireless Ad Hoc Networks

Figure 4.4 schematically depicts how centralized, flat, and hierarchical control architectures fit resiliency, scalability, and simplicity requirements. The figure omits federated control architectures because they may be considered as compositions of different architectures, where each federated domain may implement a distinct design pattern. In this taxonomy, hybridization is a cross-category attribute, and thus it can be applied to each control architecture. For example, centralized control architectures best fit simplicity with little resiliency and scalabilty. However, by adopting an hybrid strategy, centralized control architectures can improve resiliency (e.g., fallback routing mechanisms) as well as scalability (e.g., in-network processing) at the cost of simplicity. Similar considerations can be done for hi-

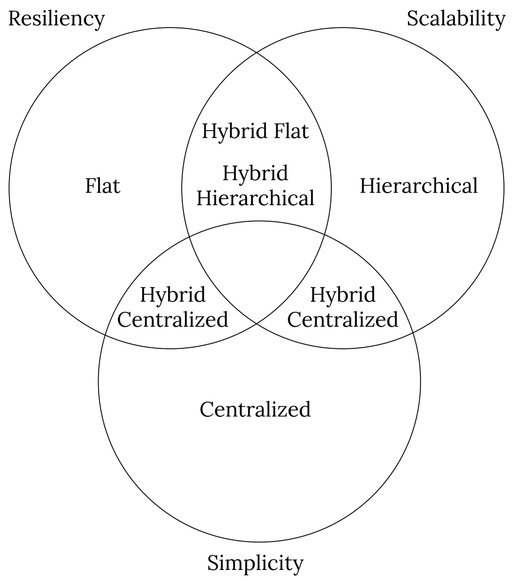


Figure 4.4: Correlation between design patterns and complexity.

erarchical and flat architectures: hybrid strategies provide better resiliency or scalability, while increasing complexity.

Table 4.1 lists the surveyed literature by outlining the role of hybridization in designing control architectures for Software-Defined Wireless Sensor Networks (SDWSNs), Software-Defined Wireless Mesh Networks (SDWMNs), Software-Defined Mobile Ad hoc Networks (SDMANs), Software-Defined Vehicular Networks (SD-VNs), and Software-Defined Flying Networks (SDFNs). For the sake of clarity, Figure 4.5 depicts a slice of Table 4.1 graphically, i.e., the role of hybridization in correlation with the mobility degree.

In particular, the figure shows that wireless ad hoc scenarios involving relatively stable NDs, such as SDWSNs and SDWMNs, do not recur to hybridization extensively. In this regard, it is worth remarking that the FP of SDWSNs includes primarily static sensor nodes. Moreover, the FP of SDWMNs usually consists of only mesh routers (i.e., static elements) [120]–[123], while mesh clients (i.e., dynamic elements) are included rarely [124]. Since relatively stable network topologies usually prevent the need to maintain control logic outside the controller(s), the role of hybridization is rather marginal. A notable exception regards SDWSNs, where [115], [118] exploit hybrid architectures to enable in-network processing for

Table 4.1: The role of hybridization in software-defined wireless ad hoc networks.

SDN	Proposal	Architecture	Network Devices	Network Topology	Hybridization Strategy	Objective
	[115]	Centralized	Sensors	Relatively stable	In-network processing	Scalability
	[116]	Centralized	Sensors	Relatively stable	Fallback routing mechanism	Resiliency
SDWSN	[117]	Flat	Sensors	Relatively stable	-	-
	[118]	Flat	Sensors	Relatively stable	In-network processing	Scalability
	[119]	Hierarchical	Sensors	Relatively stable	-	-
	[120]	Centralized	Mesh routers	Relatively stable	-	-
	[121]	Centralized	Mesh routers	Relatively stable	Fallback routing mechanism	Resiliency
SDWMN	[122]	Flat	Mesh routers	Relatively stable	-	-
	[123]	Flat	Mesh routers	Relatively stable	-	-
	[124]	Federated	Mesh routers and clients	Dynamic	Dynamic controller election	Resiliency
	[125]	Centralized	Mobile nodes	Dynamic	Dynamic controller election	Resiliency
SDMAN	[126]	Centralized	Mobile nodes	Dynamic	Backup forwarding rules	Resiliency
	[127]	Flat	Mobile nodes	Dynamic	Distributed routing protocol	Resiliency
	[26]	Hierarchical	Mobile nodes	Dynamic	Fallback routing mechanism	Resiliency
	[128]	Centralized	Vehicles	Highly dynamic	Distributed routing decision-making	Scalability
	[129]	Centralized	Vehicles and RSUs	Highly dynamic	Fallback routing mechanism	Resiliency
SDVN	[130]	Hierarchical	Vehicles and RSUs	Highly dynamic	Fallback routing mechanism	Resiliency
	[131]	Centralized	Vehicles, RSUs, and BSs	Highly dynamic	-	-
	[132]	Flat	Vehicles, RSUs, and BSs	Highly dynamic	-	-
	[133]	Hierarchical	Vehicles, RSUs, and BSs	Highly dynamic	Dynamic controller election	Resiliency
	[134]	Centralized	UAVs	Extremely dynamic	-	-
	[135]	Centralized	UAVs	Extremely dynamic	Dynamic controller election	Resiliency
SDFN	[136]	Flat	UAVs	Extremely dynamic	-	-
	[137]	Hierarchical	UAVs	Extremely dynamic	-	-
	[138]	Hierarchical	Aircraft	Extremely dynamic	-	-

improving scalability.

As the mobility degree grows, so does the role of hybridization. In fact, SDMANs systematically take advantage of hybridization. Since Mobile Ad Hoc Networks (MANETs) are infrastructure-less multi-hop wireless ad hoc networks where nodes move unpredictably, resiliency is critical. Accordingly, [26], [125]–[127] (i.e., the full spectrum of the proposals surveyed for this scenario) implement hybridization strategies for improving resiliency. Note that even flat control architectures recur to hybridization in SDMANs, notwithstanding such architectures are the most resilient by nature (see Figure 4.4).

Although SDVNs are characterized by a higher mobility degree than SDMANs (dynamic vs. highly dynamic), the role of hybridization is not as much as pervasive in the former in comparison with the latter. However, it is worth noting that several proposals for SDVNs include Base Stations (BSs) as part of the FP to benefit from better coverage of fixed network infrastructure [131]–[133]. SDVNs mainly exploit hybridization to ensure information circulation among vehicles when the CP is unreachable, keeping the ability to cope with emergency circumstances (e.g., road accidents) timely.

Lastly, the role of hybridization seems negligible in SDFNs at first glance. However, it is worth noting that [134], [136] place the CP on BSs, which are

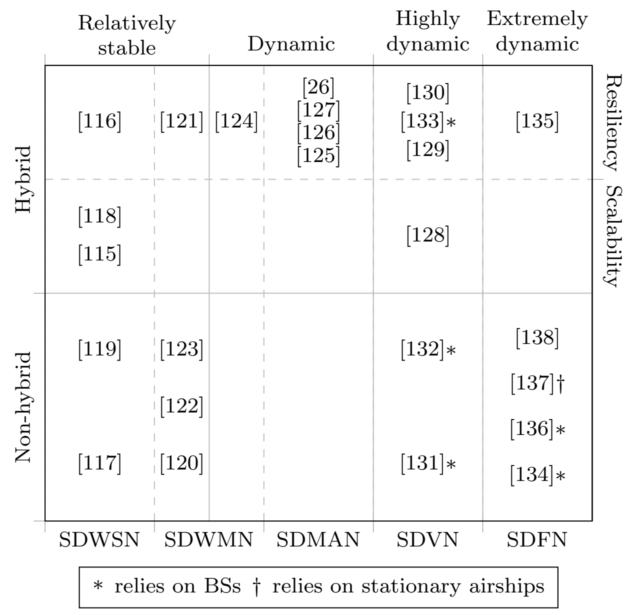


Figure 4.5: Correlation between mobility and hybridization.

not formally part of Flying Ad Hoc Networks (FANETs). Similarly, [137] places the CP on stationary airships, which, as BSs, are stable. This explains why the proposals for SDFNs quantitatively recur to hybridization less than the others, although SDFNs have the highest mobility degree.

4.2 Container Orchestration

4.2.1 History

Virtualization may be defined as the act of creating the virtual version of a resource, where, in computer science, a resource may be either hardware or software. IBM started to work on virtualization in the early 1960s by concurrently launching two projects: SIMMON and CP-40. The latter was the precursor of the VM/370, nowadays known as z/VM. The heart of VM/370 architecture is the Virtual Machine Monitor (VMM) running directly on the bare hardware. In general, a VMM is a piece of software that behaves as follows [139]: any program running under VMM should show the same behavior as if it ran on the original machine directly except for differences due to available resources and timing dependencies,

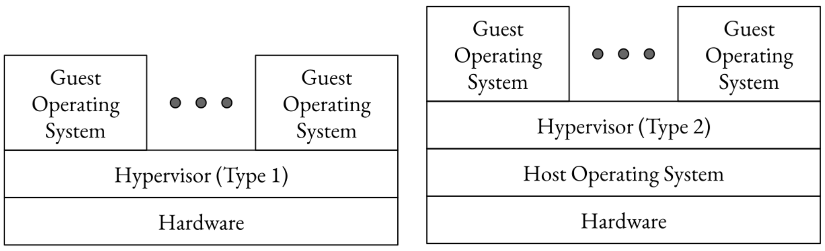


Figure 4.6: Comparison between hypervisors of type 1 and type 2.

efficiency, and control of the hardware resources. Virtual Machines (VMs) may be considered as environments created and orchestrated by the VMM. A VM can run every Operating System (OS) with the only and obvious requirement that the OS must be compatible with the underlying virtualized hardware layer. Nowadays, VMMs are also known as hypervisors, and they can fall into two categories: type 1 and type 2. The difference between them is that hypervisors of type 1 run directly on the hardware layer, whereas hypervisors of type 2 exploit the advantages that the underlying OS makes available [140], as shown in Figure 4.6.

From a high-level view, they denote two different approaches: hardware-level and OS-level virtualization. The strategies mentioned above vary in the point at which they draw the virtualization boundary. The virtualization boundary may be defined as the abstraction level at which the virtualized part of the system is separated from the virtualizing infrastructure [141]. Type 1 hypervisors represent a decoupling level between guest OSs and the underlying hardware, meaning that this class of hypervisors can control how guest OSs use the hardware resources [142] (the virtualization boundary is at the hardware interface level). On the other hand, type 2 hypervisors by running on top of the host OS may be threatened like any other applications (the virtualization boundary is at the OS interface level). In both cases, virtualization allows encapsulation referring to the ability of a VMM to capture the VM software's state and remap it, implying a chief degree of portability. Therefore, VMs can be stopped and resumed as well as moved between different physical machines. Migrating VMs is easier than migrating processes running directly on the host OS. The former case requires only the movement of memory and disk images, while the latter requires taking into account all the

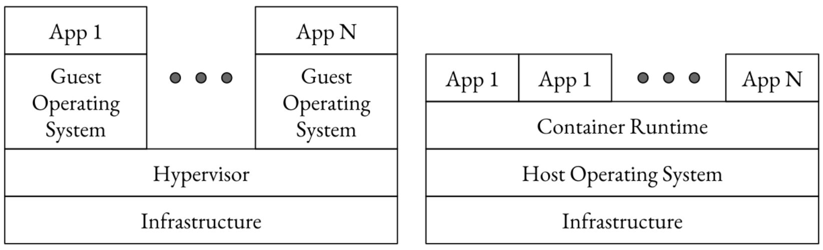


Figure 4.7: Comparison between VMs and containers.

information usually managed from an OS to handle processes. Another significant benefit is the strong isolation level provided by virtualization. Each VM may be considered as a sandbox; what happened in a sandbox remains isolated from the rest without affecting the overall system. Type 1 hypervisors provide a more robust level of isolation than those of type 2 due to the fact that a hypervisor is more straightforward than an OS; it requires fewer lines of code, and the presence of bugs is consequently reduced. From a security and fault-tolerance perspective, isolation implies an excellent advantage.

Containerization is an OS-level virtualization technique. A container may be defined as an object that isolates a set of resources of the host, for the application or system running in it [143]. Containers represent a lightweight alternative as compared to hypervisor-based virtualization techniques. Hypervisors abstract hardware; therefore, they naturally introduce a certain degree of overhead in terms of hardware and device drivers [144]. In contrast, containers avoid that overhead by acting at the OS level by isolating processes. Each container runs on top of the host OS and shares the underlying host OS kernel with the others, as shown in Figure 4.7.

The concept of containerization is decades old, since it emerged in 1979 when `chroot` was implemented in Unix V7. `chroot` is a system call that changes the root directory of the current running process and its children [145]. That process cannot access files and commands outside that designated directory tree. Given that `chroot` acts by changing an ingredient in the pathname resolution and nothing else, it does not aspire to solve security issues, nor to provide a fully sandboxing mechanism for processes as well as to restrict filesystem system calls [146]. Despite its limitations, `chroot` was the first step towards the fundamental concept of process isolation. The `namespaces` feature was implemented in the Linux kernel in 2002. A *namespace* wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource [147]. Currently, there are available seven namespace types: `cgroup_namespaces`, `ipc_namespaces`, `network_namespaces`, `mount_namespaces`, `pid_namespaces`, `user_namespaces`, and `uts_namespaces`. For example, the namespace type `network_namespaces` virtualizes the network stack by providing isolation of the system resources associated with networking [148]. `namespaces` represent a remarkable breakthrough towards the effective implementation of the concept of processes isolation. In 2006, Google released `cgroups` (Control Groups), a feature implemented in the Linux kernel to manage, restrict, and audit groups of processes [149]. From a technical perspective, it provides a mechanism for aggregating/partitioning sets of tasks (processes), and all their future children, into hierarchical groups (each of them called *cgroup*) with specialized behavior [150]. A *cgroup* binds a collection of processes with a set of parameters involving one or more *subsystems*. A *subsystem*, also known as a resource controller, represents a kernel module that modifies the behavior of the processes in a *cgroup* [151]. For example, resource controllers can limit the memory available for a *cgroup* as well as account the CPU time used by a *cgroup*. Two years later, in 2008, `lxc` (Linux Containers) emerged. `lxc` is a low-level Linux container runtime [143] or, in other words, an OS-level virtualization technique to manage multiple containers by utilizing a single Linux kernel. `lxc` provides resource management through `cgroups` and resource isolation via `namespaces`.

4.2.2 Kubernetes

Containers have transformed the application creation process, affecting development, testing, and distribution. A monolithic application can be broken down into several loosely coupled containerized modules, where each of them follows different released policies without affecting the overall working system. The starting monolithic application may be distributed across multiple cloud providers in different regions of the world. Several development teams may work only on a specific module to best meet their know-how and then distribute their work through container images. A container image is autarchic by definition; it is self-sufficient, given that it knows and contains everything needed to work correctly. However, along with the benefits introduced by containerization, new challenges arise.

Kubernetes is an open-source platform that provides basic mechanisms for deployment, maintenance, and scaling to manage containerized applications in a cluster [152]. Objects are persistent entities in the Kubernetes system representing the state of the cluster. Each object is composed of two sections: *spec* and *status*, where the former represents the desired state for the object while the latter the actual state of the object. In other words, an object represents a record of intent in the sense that Kubernetes will continuously work to ensure that the desired state matches the actual state [153]. Pods are the smallest deployable object in the Kubernetes objects model and represent units of deployment. Each Pod consists of a group of containers tightly coupled, storage resources, unique network IP, and policies governing how it should behave [154]. A Pod may be defined as a group of containers with shared namespaces and volumes, indicating that containers involved in a Pod hold the same IP, must coordinate for utilizing network resources (such as network ports), and can communicate through inter-process communications (for example using `localhost` network interface). Deployment is the high-level object to manage a replicated application where each replica is a Pod running in the Kubernetes cluster [155]. Deployments permit to declare the number of the desired replica (Pods) of a specific application, and Kubernetes will take care that the desired number corresponds to the real number. Opposed to Deployments, Pods are not durable entities. Indeed, Pods are mortal, and when they die (due to schedule failures, node failures, or other evictions), they are not resurrected. Therefore, users do not create Pods (low-level objects) directly but declare Deployments (high-level objects), which, in turn, can create and destroy Pods dynamically. Given that each Pod has its own IP, the Pods' ephemerality would cause serious reliability problems because a set of Pods may provide functionalities to another set of Pods, but the IP range available may change over time. Services are the objects that define how to access sets of Pods by exposing them as network services [155]. In particular, a Service keeps tracking the IPs of its targeted set of Pods. Consequently, different sets of Pods should not communi-

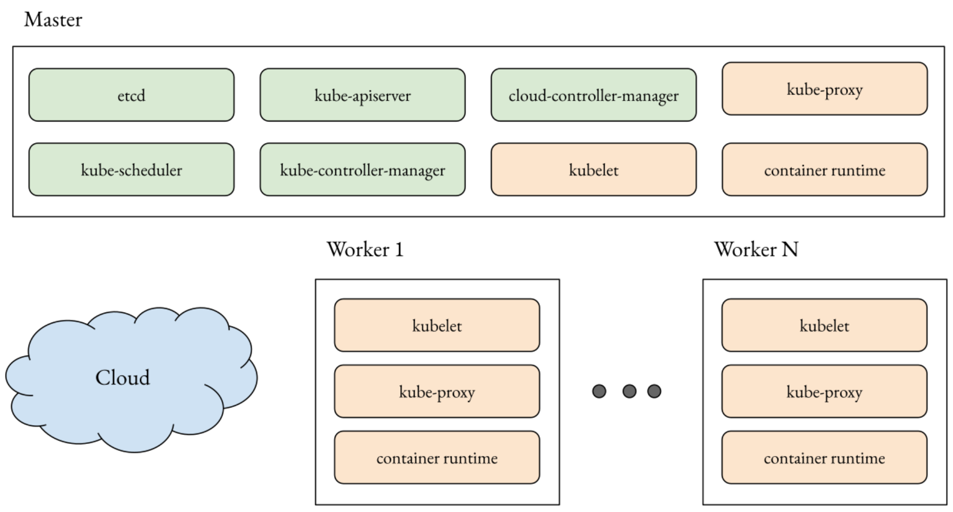


Figure 4.8: Kubernetes cluster architecture.

cate directly, but through Services representing an essential decoupling level in the Kubernetes ecosystem. Services as well as Deployments use labels to target Pods. Kubernetes provides a labeling mechanism where labels are key/value pairs, and they can be attached to objects to select only specific subsets of them [156].

The brain of a Kubernetes cluster is the control plane, and it consists of a set of components that can run in a single master node as well as across multiple masters to support high-availability clusters. In the most straightforward Kubernetes cluster architecture, a cluster is composed of a set of workers and a master, as shown in Figure 4.8.

The components forming the control plane are [157]:

- `etcd`. It may be defined as a distributed reliable key-value store for the most critical data of a distributed system [158]. A Kubernetes cluster stores its persistent state (objects) in an instance of `etcd`;

- `kube-apiserver`. It represents the front-end for the control plane by acting as a gateway for the Kubernetes cluster, meaning that every component interacts with the others through the `kube-apiserver`. It exposes the Kubernetes API and validates the objects received before storing them in `etcd`;

- `kube-controller-manager`. It runs controller processes that are the core abstraction of the Kubernetes ecosystem. Controllers may be defined as control loops that attempt to move the current cluster state towards the desired state. For example, the Deployment controller ensures that the actual state of each Deployment object matches its desired state. Controllers retrieve objects through the `kube-apiserver`;

- `kube-scheduler`. It watches for unscheduled Pods and binds them to nodes according to constraints, such as available resources, Quality of Service (QoS) requirements, and affinity as well as anti-affinity specifications. For example, when a user creates a new Deployment through `kube-apiserver`, `kube-scheduler` (that is a controller) notices a change and decides where places those new Pods;

- `cloud-controller-manager`. It runs controllers interacting with the underlying cloud providers. `cloud-controller-manager` avoids dependencies between Kubernetes and cloud providers so that they can evolve independently.

Each node involved in a Kubernetes cluster runs `kubelet`, `kube-proxy`, and a container runtime [157]. `kubelet` is the most important controller in the Kubernetes ecosystem, since it represents the container execution layer. Also, it ensures that Pods scheduled in its node are running and healthy. `kube-proxy` is a network proxy that implements the Service abstraction by maintaining network rules to allow network communication inside and outside of a Kubernetes cluster. A container runtime is the software layer responsible for running containers (e.g., Docker).

Networking represents a fundamental part of the Kubernetes ecosystem. There are four different levels of networking in Kubernetes: Container-To-Container, Pod-To-Pod, Pod-To-Service, and External-To-Service [159]. Each networking level must not violate the following three constraints: all Pods can communicate among them without using Network Address Translation (NAT), agents on a node (e.g., `kubelet`) can communicate with all Pods on that node, and each Pod sees for itself the same IP that other Pods sees for it. Pods are abstractions that solve the Container-to-Container networking level according to the constraints,

Table 4.2: Network scenarios

Scenario	Bandwidth (Mbps)	Latency (s)	Packet Loss (%)
LAN	No constraints	No delay	No loss
BEST	4	0.2	5
AVG	2	0.4	10
WORST	0.65	3	15

Table 4.3: Performance results

Distribution	BEST	AVG
K8s	$\leq$ 11 workers 20 replicas per worker	$\leq$ 5 workers 20 replicas per worker
K3s	$\leq$ 11 workers 20 replicas per worker	$\leq$ 5 workers 20 replicas per worker
KubeEdge	$\leq$ 17 workers 20 replicas per worker	$\leq$ 11 workers 20 replicas per worker

given that containers involved in a Pod share the network namespace and can communicate through the `localhost` network interface. Kubernetes follows an IP-per-Pod model, in which each Pod has its own IP, meaning that Pod-To-Pod communication is allowed through real IPs, either the Pods are in the same host or different machines. Each Pod has its own Ethernet device (`eth0`), different Pod's namespaces are linked by using virtual Ethernet devices (`veth`), and the communication among Pods on the same host is enabled by attaching each `veth` to a network bridge (`cbr0`).

4.2.3 Kubernetes in Tactical Networks

Fogli et al. [160], [161] and Kudla et al. [162] explored how various Kubernetes distributions perform in TNs. The rationale behind these studies is that since Kubernetes has been designed for enterprise environments, it might take for granted resources that are not available in TNs. Specifically, they captured the overhead introduced by three Kubernetes distributions, i.e., Kubernetes (K8s) [163], Lightweight Kubernetes (K3s) [164], and Kubernetes Native Edge Computing Framework (KubeEdge) [165], while performing orchestration-related tasks, i.e., cloud initialization and application deployment. Table 4.2 details the network scenarios (i.e., LAN, BEST, AVG, and WORST) affected by incrementally degraded network conditions used to evaluate performance. These scenarios vary along three dimensions, i.e., bandwidth, latency, and packet loss. It is worth mentioning that the available bandwidth in each scenario is distributed among nodes equally, which means that each node consumes at most a fraction of the configured bandwidth.

The performance results showed that the deployment of a container orchestration solution under disadvantaged network conditions is feasible under specific circumstances. Such circumstances depend on a combination of factors, i.e., cloud size (how many participating nodes), cloud workload (how many running applications), network parameters (bandwidth, packet loss, and latency), and the container orchestration solution itself. In this regard, Table 4.3 provides insights on how to size a cloud and its workload based on a given network scenario and a Kubernetes distribution. A key takeaway is that a single giant cloud is not feasible. A further outcome is that narrowband networks do not fit the requirements demanded by the Kubernetes distributions experimentally evaluated. In fact, such networks consist of long-range communications links primarily intended for voice and with limited data support, providing bandwidth of an order of magnitude less than the WORST scenario. Lastly, with regard to Kubernetes distributions, KubeEdge proved to be the most promising technology. It is worth mentioning that KubeEdge is not a solution per se, but it requires a K8s-based cluster (even a single control plane node) as a prerequisite.

4.3 Chaos Engineering

4.3.1 History

The traditional approach to software testing typically takes place during the development phase. The rationale is to build resilient software before it is released in production. Therefore, the objective is to avoid failures in production. Yet a distributed system is, in fact, a complex system whose dependencies might not even be fully understood by those who have engineered it. As a result, the production environment is the only place where some failures can be discovered and, above all, understood.

To improve the resilience of distributed systems, over twenty years ago Amazon launched GameDay—a program whose overarching ambition was to make the company's systems, software, and people more resilient by purposely injecting failures in production on a pre-planned day [166]. More recently, Netflix has proposed chaos engineering [167] and released some tools to implement it [168]. Specifically, [167] defines chaos engineering as the discipline of "experimenting on a distributed system to build confidence in its capability to withstand turbulent conditions in production."

What makes chaos engineering fundamentally different in comparison to the traditional approach to software testing is how it looks at failures. The rationale is as follows. Any system will break regardless of any prior testing when released in production. This happens because many failures, which are usually unpredictable by nature, can only happen in production. The only countermeasure, therefore, is to try to deliberately break the system in production in a controlled manner (thus preventing a complete system crash) and learn how to make it more resilient accordingly. Note that this is not meant to diminish the usefulness of software testing in development, which remains the basis for building resilient software systems.

4.3.2 Resilience Assessment in Industrial Scenarios

Although chaos engineering was initially proposed for the testing of complex and distributed IT services, its application may bring benefits to other application domains that also require high reliability and fault tolerance. In the OT domain, chaos engineering can help improve the resilience of the OT ecosystem by executing what-if experiments to explore their impact on different configuration deployments. For instance, such experiments can help identify the minimum set of resources, i.e., computing and network devices, that would guarantee the desired working conditions even in case of undesirable and unexpected faults.

Notable examples of real-world faults include hardware faults (both at the computing and at the network appliance level), network issues (such as increased communication latency, or limited bandwidth lower than the actual traffic throughput generated by machines), and application (partial) unavailability (e.g., a crashed microservice). The combination of those faults leads to the concept of profile, i.e., a set of faults to inject into the target running system that would make the target system run outside its steady state, and thus likely causing undesirable effects that will expose undiscovered software vulnerabilities and architectural weaknesses.

Fogli et al. [169] identified three main chaos engineering profiles for DTs, each targeting a different possible issue category related to network, computing performance, and worker/node availability. Chaos engineering profiles may be regarded as dynamic components that start with a baseline "intensity" and increase it over time to put more stress on the system under testing, e.g., by gradually increasing the latency of a communication link.

By delving into finer details, the **network profile** targets packet dispatching issues of the industrial environment. In this regard, it is important to evaluate both the resilience of the DT network and the network connectivity with the PT. For example, an increase in communication latency might break the entanglement between the DT and its physical counterpart, or it might cause inconsistencies in the case of latency sensitive applications. Chaos engineering actions to stress the network resources might increase network link latency and/or packet loss, or might terminate a whole network appliance, e.g., a network switch.

The **computing performance profile** considers issues of worker nodes executing the DT, e.g., by simulating that required computing resources are only partially available. This can be implemented by saturating the computational resources available for the worker nodes. For example, the execution of faulty processes designed to be CPU and memory consuming would reduce the amount of CPU time and/or memory the DT can leverage, thus likely degrading its performance. Note that here are several other methods to affect the performance of DTs, such as acting on the niceness (in the case of UNIX nodes) to assign a lower priority to the processes associated to the DT. Finally, in the case of virtualized worker nodes, e.g., VMs, another approach is to interact with the virtualization platform API to dynamically decrease the amount of reserved computing resources.

From a resilience perspective, the computing performance profile and the network profile are crucial to estimate the performance of different deployments, e.g., to identify the minimum set of resources capable of providing a target QoS level.

This factor is important in OT environments, since computing and network resources at the different layers usually have very heterogeneous characteristics. Moreover, the intensity of these profiles is an important element to consider. Chaos engineering adopters should, based on their expertise, select a reasonable configuration and intensity that would reflect real-life failures and malfunctions.

The **node/worker availability profile** describes the possibility of having multiple failures caused by, e.g., software and hardware issues. Such experiments can be implemented by terminating one or more worker nodes hosting the DT. Specifically, the termination process can be limited to the worker executing the DT, to other workers the DT interacts with, or both. This profile allows verifying the correctness of the deployment configuration, i.e., if the computing resources were opportunely replicated to be resilient to such failures. Although this profile is easy to implement, its application should be tailored to each given scenario. In fact, this profile is very effective in testing the performance of highly distributed and cloud-based IT applications, which can leverage a plethora of computing resources from different cloud vendors. However, it could be less effective when dealing with OT applications, which are generally not characterized by a large number of computing devices compared to datacenters.

Table 4.4 provides a list of possible fault actions that chaos engineering adopters can use to implement profiles. Specifically, the table divides such actions into multiple categories to define the action target, e.g., the application protocol, the worker node, or the network. **Application protocol** fault actions target application components at multiple levels. For example, these actions can modify Domain Name System (DNS) configurations to simulate naming and service lookup errors or can introduce errors at the web sever level (in the case of web-based applications), such as additional random termination of requests. Differently, **software component** fault actions can terminate or inject failures into a single software component, e.g., a microservice running on a worker node, while **host computing** fault actions execute a process on a target worker node requiring a considerable amount of CPU and/or memory. **Host network** fault actions can be injected to create network failures at the host level. Such actions allow chaos engineering adopters to introduce packet delay and corruption, partitions, and bandwidth shaping in a very fine-grained manner. For example, Linux Netfilter provides a set of compo-

Table 4.4: List of fault actions for chaos engineering

Category	Action	Description
Application Protocol	DNS failure	Returns a DNS error
Application Protocol	DNS random response	Returns a random IP address
Application Protocol	HTTP abort	Aborts a given request/response
Application Protocol	HTTP delay	Delays a given request/response
Application Protocol	HTTP replace	Replaces one or more sections of a given req./resp.
Application Protocol	HTTP patch	Adds content to one or more sections of a given req./resp.
Software Component	Kill	Kills the container
Software Component	Failure	Makes the container unavailable for a given time
Host Computing	Memory Stress test	Consumes a given amount of memory
Host Computing	CPU Stress test	Consumes a given amount of CPU
Host Network	Packet delay	Delays packets for a given time
Host Network	Packet loss	Drops packets with a given probability
Host Network	Packet duplication	Duplicates packets with a given probability
Host Network	Packet corruption	Corrupts packets with a given probability
Host Network	Packet reordering	Reorders packets with a given probability
Host Network	Traffic shaping	Limits the bit rate
Host Network	Partition	Creates a network partition
Host Network	Port	Occupies a given port
Host I/O	System call delay	Delays file system calls for a given time
Host I/O	System call failure	Returns for file system calls an error with a given probab.
Host I/O	Attribute override	Overrides a file system attributes with a given probab.
Host I/O	Read/write mistake	Makes mistakes while reading/writing with a given probab.
Host Configuration	Time	Shifts the host's time of a given amount
Host Configuration	Shutdown	Turns off the host
Host Configuration	Unmount	Unmounts attached disks or partitions
Host Configuration	Remove files	Removes host configuration files

nents to change all these configuration parameters at run-time. **Host I/O** fault actions inject failures such as execution delays or errors while executing system calls. Such actions could be used to invalidate the readings coming from external devices, e.g., a HMI or a sensor, and to test how the faulty data would affect the reliability of the system. Finally, other relevant fault actions are related to **host configuration** errors, e.g., a time synchronization misalignment in a distributed application. These actions can be very helpful for testing the resilience of applications that rely on accurate timing. Additionally, simulating storage failures by unmounting attached network or physical disks might represent useful actions reflecting faults that are likely to happen in a real-world scenario.

With regard to the implementation of the chaos engineering profiles, a network profile may be implemented by applying some actions of the host network category. To do so, a first plausible step might consist of modifying the steady-state network latency by introducing an additional packet delay, e.g., an increased delay of about 50 ms to all (or to a subset of) packets leaving the host network interface. A computing performance profile, instead, may be implemented by applying the stress test actions of the host computing category to consume memory and CPU resources. This profile would work well in the case of applications whose components are replicated across multiple worker nodes. In fact, by targeting one or more replicas with stress test actions, it is possible to evaluate how the application would respond in the case of a non-optimal number of replicas available, e.g., by detecting a decreased QoS for a given DT.

4.4 Chapter Summary

This chapter identified the foundational technologies, i.e., SDN, container orchestration, and chaos engineering, for building an entanglement-aware DT ecosystem.

Section 4.1 began by tracing the historical evolution of SDN and presenting the well-accepted 3-tier network architecture. Then, it introduced a taxonomy for control architectures. Lastly, the section explored software-defined wireless ad hoc networks, uncovering the significance of hybridization in designing control architectures for such networks. The discussion on software-defined wireless ad hoc networks aimed to dispel the misconception that SDN is limited to fixed and wired environments, thus justifying its applicability in dynamic scenarios, such as coalition tactical operations.

Section 4.2 delved into containerization, an OS-level virtualization technique, and provided an overview of Kubernetes, the industry-standard container orchestration system. Given that Kubernetes is not inherently designed for resilience under challenging network conditions, as it typically assumes enterprise-like environments, the section explored the performance of various Kubernetes distributions in emulated TNs.

Finally, Section 4.3 explained the rationale behind chaos engineering—the discipline of "experimenting on a distributed system to build confidence in its capability to withstand turbulent conditions in production [167]." The section then elucidated the relevance of chaos engineering to DTs and described how to apply this concept in OT environments.

Chapter 5

Digital Twin Engineering

In the realm of DTs, a trend gaining momentum is to adopt approaches and technologies typical of web- and cloud-based IT environments, such as virtual machines and containers. In fact, they represent valuable approaches to make dynamic management of DTs easier and to ensure the required QoS while also improving their reusability and composability [170], [171]. Containerization is already a mature technology, widely used in cloud computing, and its adoption is currently gaining speed also in industrial environments [172], [173], e.g., to develop and deploy MESs and PLCs as microservices [174]. The state-of-the-art literature either only proposes high-level guidelines, e.g., stressing the importance of microservices architectures but without detailing how to design DTs accordingly, or focuses on specific scenarios, e.g., by proposing vertical solutions for specific markets.

To bridge this gap, Section 5.1 analyzes the most relevant DT properties introduced by Minerva et al. [44] and the engineering requirements to build DTs with those properties. Then, Section 5.2 proposes how to apply software engineering design patterns to translate such properties into actionable tools to build context-aware, adaptive, and autonomous DTs based on microservices. Subsequently, Section 5.3 discusses how to engineer DTs in the context of serverless computing, along with the benefits of this approach. Lastly, Section 5.4 concludes the chapter. In contrast to Chapter 3, which identifies the requirements for an entanglement-aware DT ecosystem at the platform level, this chapter delves into

DT internals. Specifically, it focuses on the engineering aspects of DTs, discussing two different approaches to build DTs, both compliant with the proposed ecosystem.

5.1 Digital Twin Properties

In the context of DT engineering, the relevant properties are reflection (see Section 5.1.1), persistency (see Section 5.1.2), memorization (see Section 5.1.3), augmentation (see Section 5.1.4), composability (see Section 5.1.5), replication (see Section 5.1.6), and accountability/manageability (see Section 5.1.7). The discussion around each property follows this format. First, a definition of the property is provided. An investigation is then conducted to determine the requirements necessary to engineer a DT in line with the definition. Lastly, the impact of a DT that exhibits such a property is explored.

5.1.1 Reflection

Definition: The reflection property describes a DT as an entity that mirrors a PT. Changes that occur to the PT should be reflected in the DT and viceversa.

Engineering: **(R1)** The DT should be able to discover the PT and handle communications and interactions according to the supported protocols and data formats. For example, a DT supporting a specific type of machine should autonomously search for supported entities and either establish a permanent connection (i.e., enabling reflection) or ease its configuration process. **(R2)** The DT should be aware of the quality of reflection that it provides to applications. This notion, generally identified as entanglement, can promote adaptive behaviors aimed at providing determined service levels, such as tuning communication protocols, throttling external requests, or even migrating the DT on the basis of internal or environmental conditions.

Impact: The availability of DTs capable of delivering adaptive reflection represents a fundamental enabler towards their use as autonomous entities instead of passive digitalized replicas. DTs should be in charge of tasks concerning reflection, such as discovering physical counterparts, identifying properties of interest, and maintaining the desired level of entanglement according to internal and environmental conditions. Additionally, DTs should autonomously detect and possibly react to potential issues (e.g. adapting the networking configuration for increasing or decreasing entanglement). Possibly, DTs may also notify external observers about misalignments with physical counterparts. It is worth noting that the autonomous discovery of compatible PTs has cascading benefits to other requirements. For example, a DT representing thermal-related features of a smart building might search and connect to all compatible devices of the building, or their associated DTs, without time-consuming manual interventions.

5.1.2 Persistency

Definition: The persistency property defines a DT as an entity that is always available. Its availability exceeds the actual existence of the PT. A DT may be available before creation, during malfunctions and crashes, and after the end of life of the PT.

Engineering: **(R3)** The DT should be resilient and therefore organized in decoupled and independent components, represented in Figure 5.1, so that a localized fault does not compromise the entire container. **(R4)** The DT should be highly available, i.e., it must support replication in response to failures, both internal (e.g., the DT fails) or environmental (e.g., the node running the DT container fails). **(R5)** To minimize the effects of such events, DTs should support autonomous reconfiguration. In fact, configurations should be remotely stored and retrieved when needed. By doing so, replicas, instead of restarting with the same configuration of a failed container, can possibly retrieve an alternative version.

Impact: The adoption of DTs decouples applications from the complexity and fragmentation of the physical layer. This results in an implicit agreement between these two levels. Applications rely on DTs to interact with PTs and any disruption in their functioning might represent a critical issue (e.g., the control room that

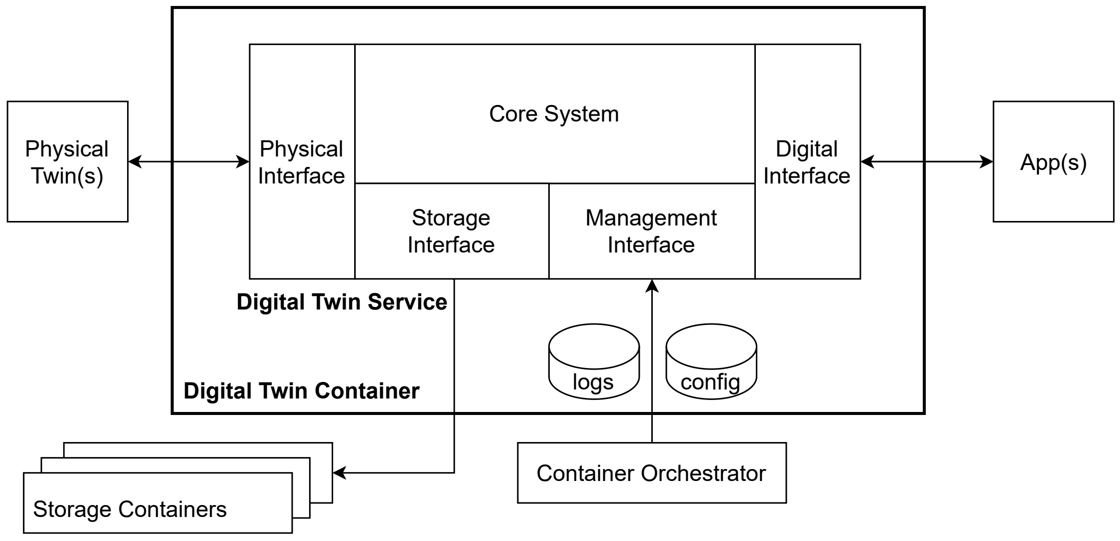


Figure 5.1: Schematic representation of a DT.

suddenly stops receiving telemetry from deployed robots).

5.1.3 Memorization

Definition: The memorization property defines a DT as an entity that stores status changes and events that involved the PT. A DT represents the status of the PT over time and space.

Engineering: **(R6)** The DT should be able to maintain the current state of the PT internally, acting as a cache between the PT and applications. Indeed, to improve autonomy and minimize response times to applications, the current state of the PT should be held within the DT itself, without using external storage services. **(R7)** The DT should manage the entire history of states and events involving the PT in a context-aware fashion. Furthermore, the DT should manage different loads of requests from applications. Thus, the DT should support different replication strategies for its storage services and autonomously select the most suitable one.

Impact: Observing and efficiently interacting with a PT not only at its current state but also through the navigation of its historical data via a uniform interface separates responsibilities and has the potential to significantly simplify the design of applications. Memorization can also be used to support context awareness and adaptation directly in DTs via machine learning algorithms capable of predicting future states from past states. Furthermore, the resulting outputs (i.e., the predicted future states) can be memorized within the same data structure, allowing forward navigation in the predicted "future" of the DT.

5.1.4 Augmentation

Definition: The augmentation property defines a DT as an entity that can offer functions that the PT does not provide by means of APIs. Augmentation can add new functionalities that the PT does not support or provide access to data in particular formats.

Engineering: **(R8)** The DT should be expandable (adaptive) with additional functionalities by supporting dynamic configuration. For example, a complex DT supporting multiple PTs or functionalities could be deployed with different configurations on different nodes depending on their resources. The configuration should be updated, without requiring manual interventions, whenever the DT is migrated to a node with different capabilities. **(R9)** The DT should support software updates. At the most basic level, both PTs and applications might receive updates over time, thus requiring changes in the DT to keep it functional. Furthermore, updates enable the addition of augmentation functions to the DT itself without the need of a redeployment.

Impact: The possibility of easily and dynamically augmenting the capabilities provided by a DT with respect to the original physical counterpart represents the fundamental characteristic of DTs and the reason why they should be seen and modeled as active software components with independent behavior. Through dynamic augmentation, it is possible to extend interoperability without changes in the PT or without requiring the redeployment of the DT (e.g., to support new protocols or data formats). It can also allow to introduce intelligent and cognitive functionalities directly in the DT, optimizing both digital and physical layers.

5.1.5 Composability

Definition: The composability property defines a DT as an entity that supports the correlation of different elementary DTs into complex organizations and provides views on the aggregated DT and individual components.

Definition: **(R10)** The DT should be able to manage other DTs as if they were PTs. Each change in any DT that is part of a composition scheme (i.e., an observed DT) is communicated to an observing DT. In this way, as soon as one DT detects a change in its PT, the change is propagated towards the upper levels of the composition scheme. Alternatively, in the case the composed DT is not observed by any application, the lower levels might choose not to communicate the changes to save bandwidth. The same principles should also be applied to commands that can be propagated from the composed DT to the underlying DTs to modify and actuate the physical counterparts.

Impact: The communication scheme used for composition is strictly tied to reflection, entanglement, and adaptive capabilities. In fact, being a distributed communication scheme, it might require remarkable network resources to guarantee an acceptable quality of entanglement. To avoid network overload, DTs participating in a composition scheme should coordinate to dynamically select a suitable trade-off between entanglement and networking resources.

5.1.6 Replication

Definition: The replication property defines a DT as an entity that can be replicated to serve the needs of different applications. Replicas of the same PT must behave consistently, i.e., they cannot have a different status and cannot exhibit different behaviors.

Engineering: **(R11)** The DT, also leveraging the container orchestrator, should support replication to deal with variable requests from applications. However, as soon as replicas are spawned, two different communication schemes become possible: peer-to-peer and master-slave. The former implies that all replicas of the same DT communicate directly with the PT. The latter implies that the group of replicas elects a master responsible for managing the PT while all the others, behaving as slaves, communicate with the master to receive updates about the state of the PT. As a consequence, the DT should be aware of the internal and environmental conditions to autonomously select the most suitable approach.

Impact: The DT leverages its awareness of the computational environment to autonomously select the most suitable replication scheme. As an example, when dealing with constrained PTs, the master-slave approach might be preferred. On the other hand, when powerful PTs are involved, the peer-to-peer approach might reduce communication overhead among replicated DTs and avoid all the intricacies of distributed master-election protocols. The same flexibility can be also exploited to handle different visibility and responsibilities levels in order to segregate the DT authorized and in charge to communicate with the PT (master) and additional DTs (slaves) responsible, for example, for specific interaction with applications and unauthorized to directly interact with the physical layer.

5.1.7 Accountability/Manageability

Definition: The accountability property defines a DT as an entity that allows one to determine its status and activities and to optimize its execution in the framework in which it operates. It should also provide information about the usage of the PT.

Engineering: (R12) The DT should be observable. Indeed, the DT should not only be aware of its state but also make it available via standard interfaces (e.g., REST APIs, event-driven communication patterns). Additionally, the whole history of the events concerning the DT (i.e., execution logs) should be exposed in a similar fashion.

Impact: Observability pushes adaptation outside the DT itself. For example, the orchestrator might detect a DT running on limited resources and respond by either migrating it to another node or spawning a replica for guaranteeing the required quality of entanglement. Additionally, the availability of the history of events related to the PT enables long-term analytics based on machine learning algorithms, such as failure prediction or anomaly detection.

5.2 Design Patterns for Digital Twins as Microservices

This section explores which software engineering design patterns help meet the requirements identified while discussing reflection, persistency, memorization, augmentation, composability, replication, and accountability/manageability (see Section 5.1). These design patterns serve as the fundamental building blocks for the development of DTs as microservices. In the following, the explored design patterns are divided into three categories, i.e., those for building single-node, single-container services (see Section 5.2.1), those for single-node, multi-container services (see Section 5.2.2), and those for multi-node services (see Section 5.2.3).

5.2.1 Single-Node, Single-Container Services

The *microkernel pattern* consists of two types of components: a core system and plug-in modules (i.e., the physical, digital, management, and storage interfaces) as shown in Figure 5.2. The core system, usually containing only the minimal functionalities required to make the system operational, manages the state, configuration, and behavior of the PT (R6). The plug-in modules, instead, are independent components enhancing or extending the core system with additional capabilities without the need of redeployments (R9). As an example, the storage plug-in provides a clear boundary between the management of the current state of the PT that happens inside the DT core and past and predicted states, which are, instead, externalized (R7). Generally, plug-in modules should be independent from each other and can be connected to the core in a number of different ways, from point-to-point binding (i.e, the core accepts an object instance of a plug-in) to messaging. This kind of architecture provides decoupled operations and prevents generalized failures (R3). Indeed, the failure of one plug-in does not determine the failure of the whole container. The *asynchronous queuing pattern* would further

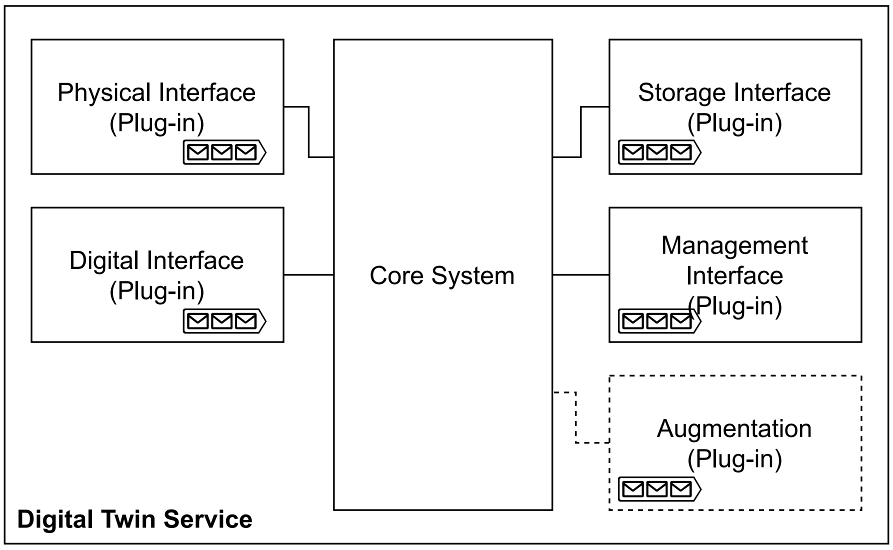

Physical Interface (Plug-in)
Digital Interface (Plug-in)
Core System
Storage Interface (Plug-in)
Management Interface (Plug-in)
Augmentation (Plug-in)
Digital Twin Service

Figure 5.2: Microkernel pattern.

improve isolation and decoupling. The introduction of asynchronous queues allows for extremely evolvable and resilient architectures. In fact, protection mechanisms against bursts or sustained rates of excessive requests can be transparently embedded within the queue themselves, thus simplifying the development of the other components.

5.2.2 Single-Node, Multi-Container Services

Imagine a DT that supports dynamic configuration and exposes its configuration via a standard API. One approach could be to add a specific plug-in to the microkernel architecture described above, while another approach could be to break up the DT into two separate containers: one running the DT itself (i.e., core and basic plugins) and the other running a dynamic configuration daemon. While the former is perfectly legitimate, the latter case has notable advantages. Containers, in fact, establish boundaries around resources (e.g., 8 GB of memory, 6 cores), teams (e.g., one team owns one container image), and concerns (e.g., this image provides dynamic configuration). As an example, using multiple containers allows to assign them different priorities and resource requirements, e.g., ensuring that the configuration daemon uses computing resources only when the DT is offloaded.

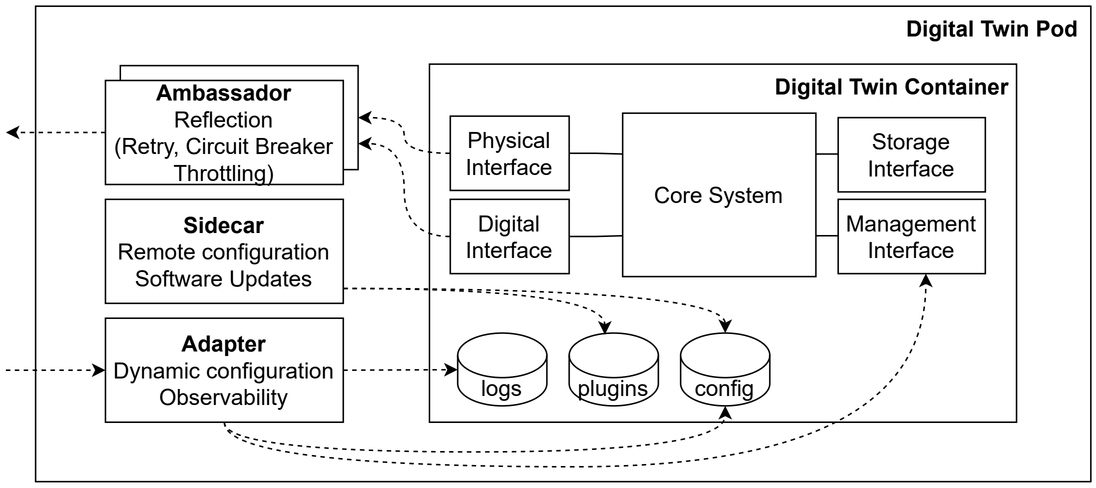


Figure 5.3: DT conceived as a multi-container entity.

In addition, containers represent a relatively small and focused piece of code managed by a single team and usually can be updated, tested, and deployed more easily than complex, monolithic services. Containers can also be easily reused across multiple teams and applications, often leading to high-quality implementations since they are built once and used in different contexts. This is the rationale behind conceiving DTs as multi-container entities, namely *pods* (a term introduced in Kubernetes). The three patterns discussed here, represented in Figure 5.3, propose to deploy the DT container along with a secondary container responsible for different tasks. In addition to being scheduled on the same machine, the two containers are assumed to have access to shared resources, such as the filesystem and network interfaces.

The *sidecar pattern* considers two containers: the application container (i.e., the DT container) and the sidecar container, augmenting the application usually without accepting or establishing network connections on its behalf. In its simplest form, a sidecar container can be used to add functionalities to a container that might otherwise be difficult to improve. In more articulated cases, sidecars can be used to engineer multi-container services which are inherently more robust and scalable than those structured in a single container. Remote configuration (R5), requiring DTs to store and retrieve their configurations from a remote server, can be implemented with a sidecar container monitoring the configuration files of the

DT. The sidecar is responsible for maintaining aligned local and remote configurations. If the remote configuration differs from the local one, it downloads the new configuration and notifies the DT to reconfigure itself using the updated files. Similarly, software updates (R9) can also be implemented using a sidecar container. As an example, it is possible to use a containerized daemon that periodically downloads changes from a git repository, updates the local code of the DT (e.g., the folder containing plug-ins), and triggers the DT to restart itself. As a consequence, pushing updates to a git repository results in the updated code being deployed to the running DT in a simple yet reusable fashion.

The *ambassador pattern* uses an ambassador container to act as a broker between the application container and external services. Similarly to sidecars, ambassadors are paired to the primary container and scheduled on the same node. Requirements concerning adaptive reflection (R2) can be implemented using both the ambassador and the microkernel patterns in a complementary fashion. For example, they could be either implemented within the communication plug-ins of the DT (i.e., physical/digital interface plug-ins) or delegated to a specialized daemon running within the ambassador. In the latter case, the communication plug-ins of the DT act as basic network proxies and delegate external connections entirely to the ambassador container. As an intermediate solution, an ambassador could be used to improve a DT providing only basic reflection capabilities with more advanced properties, such as autonomous switching among different communication protocols (R2) or automatic search for compatible PTs (R1). In addition, an ambassador can be used to provide the communication interfaces of the DT with additional layers of protection against failures of other services. For example, protection patterns, such as *throttling*, *circuit breaker*, or *retry* (R3), can be easily implemented within ambassadors without the need of modifying and redeploying the DT container.

Ambassador containers are not limited to function with digital or physical interfaces. Indeed, they can be used for brokering any connection to external services. The storage interface, for example, is designed to store and retrieve past and future states of the PT. As expressed in (R7), the DT should be capable of dealing with high request loads from applications asking for past or future states of the PT as well as with large bodies of data representing its entire history. These

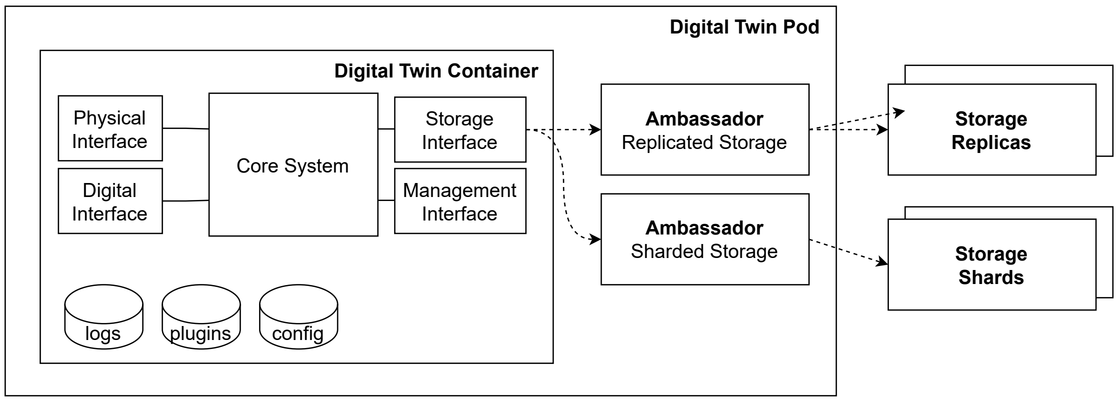


Figure 5.4: Ambassador pattern.

functionalities can be implemented within ambassador containers, as depicted in Figure 5.4. In case of high load of requests, data can be statelessly replicated so that each replica manages the whole history of the DT, thus scaling up the number of manageable requests. Alternatively, in the case of large bodies of data, sharding (i.e., partitioning based on content, for example, each shard contains one year of history) can be applied. This approach does not necessarily unload the storage services (i.e., all requests might ask for the same shard) but, instead, allows for scaling up the size of DT's past and future states. In Figure 5.4, the two cases are managed by separate ambassadors. Nevertheless, it would be possible to implement both replication strategies in a single container, capable of choosing the most suitable one.

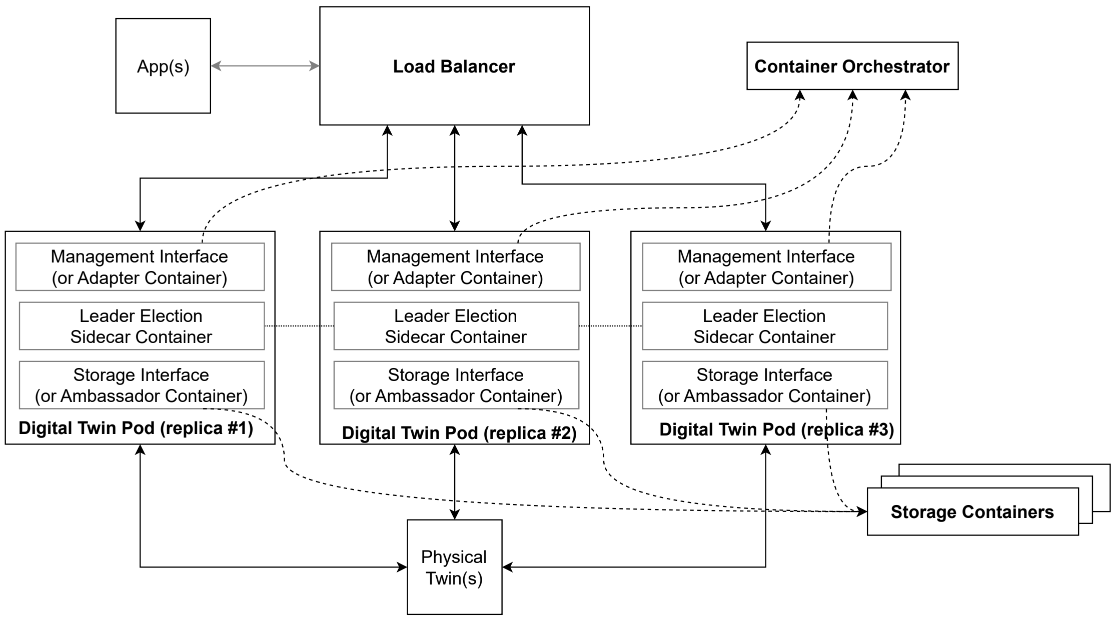
The *adapter pattern* is used to modify the interface of the primary container so that it conforms to a predefined interface. For example, an adapter might ensure that an application implements a consistent monitoring interface (i.e., all logs saved using the same format). The observability (R12) and dynamic configuration (R8) requirements can be implemented using this approach. Indeed, instead of modifying the DT core or adding plug-ins, a dedicated daemon could be run inside an adapter container. Regarding the observability requirement (R12), a daemon could monitor the logs produced within the DT container and expose them via standard APIs. Accordingly, also the internal state of the DT (exposed via its management interface) can be monitored and exposed to external services, such as the container orchestrator using the same interface. Large factories running massive deployments could greatly benefit from this containerized approach. For example, it could be possible to deploy any kind of DT, possibly produced by different vendors, and then make them uniformly observable by adding a properly crafted adapter to their pod. Dynamic configuration (R8) shares many similarities with remote configuration (R5). A daemon containerized as an adapter can read the configuration files of the DT and expose them via a standard interface. Whenever users or external services apply changes to the configuration, the daemon updates the configuration files within the DT filesystem and signals the DT to reload it.

5.2.3 Multi-Node Services

The persistency requirement (R4) relates to the availability of software components. Indeed, DTs have to be restarted if their container fails or hangs. If the node running the DT fails, the container has to be migrated and restarted on another node. The implementation of this feature is based on a properly implemented management interface exposing the internal state of the DT. For example, the interface has to provide http endpoints specifying if the container is either ready for execution or actually serving requests. By querying this interface, the container orchestrator can take autonomous decision on whether the DT have to be either restarted or migrated to another node. Another approach is based on the *Singleton pattern*. The singleton pattern, despite the different flavours it assumes in different contexts, generally implies that only one instance (of an object, a process, a container, etc.) should exist at any given time for the sake of maintaining integrity and consistency. In the context of containerized services, this pattern implies the use of a load balancer managing only one replica of a service. Since only one instance is running, that instance owns the access right to all the resources (i.e., in this case the PT) without the need for electing a master replica. This simplifies implementation and deployment, but introduces disadvantages in terms of reliability since, in case of issues, software updates, or migrations, a little downtime is required for reverting to a functioning state. Frequently, however, its simplicity outweighs the reliability trade-off.

The replication requirement (R11) can be similarly implemented by making use of the *load balancer pattern* prescribing the use of a load balancer for splitting requests among a pool of replicas. The pool can be monitored via the management interfaces of the replicas and, depending on environmental conditions, can be enlarged, shrunk, or migrated to different network locations. It is worth noting that stateless replicas are advisable, since requests can be routed to any replica regardless of their content or their state. In stateless replication, in fact, each replica is aware of the entire state which, in our context, comprises both the PT and the set of containers storing its future and past representations. Despite the clear benefits in terms of reliability, this approach has some potential issues. For example, the concurrent access of all replicas might overload the PT, thus degrading the quality of entanglement. To prevent this drawback, replicas can adopt a master-slave strategy. The master DT is the single owner of the PT, while slave DTs lose the right of direct access and interact with the PT only via the master DT. In other words, the master enacts a *proxy pattern* between the slaves and the PT, thus reducing its load. However, the master-slave approach requires to implement a master election algorithm usually based on distributed consensus algorithms, such as Paxos or RAFT [175]. Luckily, there are a number of distributed key-value stores embedding such consensus algorithms without the need for complex implementations within the DT itself. These systems, which can be packaged in a sidecar container as depicted in Figure 5.5, provide a replicated and reliable data store comprising the primitives necessary to build election abstractions out-of-the-box.

The composability requirement (R10) prescribes that DTs should receive updates and possibly send commands to a group of peers instead of a single PT. A bare implementation of this feature might require only slight changes to the physical and digital interfaces of the DT to support groups of devices instead of a single one. Indeed, each command directed to a PT could be sent to a group of PTs or other DTs and each update directed to an application can be sent to a group of applications or other DTs. However, the mere fact of receiving updates or sending commands to a group of peers does not make a composed DT but more a proxy between applications and the physical environment. What makes composability meaningful is providing applications with composed APIs representing, in a synthetic way, a complex underlying reality. As an example, a composed DT



App(s)
Load Balancer
Container Orchestrator
Management Interface (or Adapter Container)
Leader Election Sidecar Container
Storage Interface (or Ambassador Container)
Digital Twin Pod (replica #1)
Management Interface (or Adapter Container)
Leader Election Sidecar Container
Storage Interface (or Ambassador Container)
Digital Twin Pod (replica #2)
Management Interface (or Adapter Container)
Leader Election Sidecar Container
Storage Interface (or Ambassador Container)
Digital Twin Pod (replica #3)
Storage Containers
Physical Twin(s)

Figure 5.5: Load balancer pattern.

representing a smart-building should provide APIs for querying the average temperature or the presence of fire in the entire building, rather than bare access to a list of sensors. This problem is often tackled in software engineering with the *API gateway pattern*. This pattern has been in fact proposed to aggregate multiple requests, often directed to different microservices, into a single one. That is, an application attached to a composed DT sends a single request that is decomposed in simpler requests and dispatched to the involved DTs. The received replies are then aggregated and presented to the application as one single response. In addition to providing a unified synthetic representation of a complex system, this pattern is also useful in reducing chattiness among involved components.

5.3 Serverless Digital Twins

This section explores DTs in the context of serverless computing. Specifically, Section 5.3.1 highlights the benefits of serverless computing, while Section 5.3.2 introduces an original proposal to design DTs using serverless functions.

5.3.1 Benefits

The adoption of a serverless design for DTs implies substantial benefits from the engineering and operations perspective. The serverless architecture removes all responsibility of server system maintenance, configuration, scalability tracking, and management from application owners and puts it on the service provider. This can reduce the investment required in operations and also frees up developers to create and expand their applications without being constrained by server capacity. Moreover, application development built on a serverless system scales better regardless of how high or low the usage is at any given time. As such, application developers spend more time on design, coding, and testing instead of code deployment and release management [176].

Developers can also code using a variety of languages. AWS Lambda functions can be written in Java, Go, PowerShell, Node.js JavaScript, C#, Python, and Ruby. Google Cloud Functions support Node.js JavaScript, Python, and Go, and allow for unlimited execution time for functions. Microsoft Azure Functions support a wider range of languages, including Bash, Batch, C#, F#, Java, JavaScript (Node.js), PHP, PowerShell, Python, and TypeScript, and have pricing similar to Lambda. Google Cloud Run supports any language that can run in a container, while AWS Lambda Layers allow developers to pull in additional code written in other languages.

The serverless design also avoids the need to upload code to servers or to do any back-end configuration to release working code. Since serverless applications are not monolithic stacks, there is no need to make changes to the whole application; instead, developers can update the application more granularly, one function at a time. This enables quick updates, patches, or adding new features with reduced efforts. Furthermore, since the application is not hosted on an origin server, its code can be run from anywhere along the cloud-to-edge continuum. As such, it is possible (transparently from the perspective of DTs) to run functions on cloud datacenters, MEC, or edge on-premises nodes, depending on application requirements and environmental situations [177].

In addition, designing DTs as serverless software components enables the reuse of fundamental building blocks (e.g., the model describing the PT, augmentation functions, physical/digital interfaces, etc.) across different domains, tenants, and applications, thus lowering the technical barriers to adopt these technologies. Along the same line, [178] has recently discussed the case for building a shared catalog of reusable DT models that might greatly benefit from the adoption of serverless functions.

Moreover, instead of spawning a replica of an entire DT (i.e., a new pod for containers built as microservices), serverless infrastructures can be used to offload computation from the DT. The groundwork for serverless computing has been laid with the PyWren [179], Lithops [180], and funcX [181] libraries. They showed that it is possible to create a data processing system that inherits the elasticity and simplicity of the serverless model, using stateless functions with remote storage. They do not provide the best parallel performance but offer some significant advantages if compared with a standalone server node. By using this approach, complex DT models requiring, e.g., the repeated execution of the same function on different data, are a good fit for serverless infrastructures, which can run many short tasks in a highly elastic way, i.e., by acquiring thousands of resources very quickly. Indeed, a DT capable of updating its internal model making use of an event-based chain of functions can be easily parallelized, thus enabling shorter computation delays and, consequently, an improvement of the quality of entanglement. As a result, a serverless DT would be able to handle an unusually high number of requests (or unusually complex models) as well as process a single request from a single application. On the contrary, a traditionally structured DT with a fixed amount of resources can be overwhelmed by a sudden increase in the number of requests or in their complexity.

Cloud and serverless datacenters have a significant impact on the world's total energy consumption (about 1 to 2.5% total energy consumption), although it has been estimated that half of this energy is consumed by idle servers [182]. Despite this 50% waste could be reclaimed by the serverless paradigm (involving the execution of short-lived functions), ensuring adequate management of energy efficiency in such systems remains a crucial challenge. Indeed, the proliferation of sensing capabilities is likely to further push the current trend of developing cyber-physical applications orchestrated along the cloud-to-edge continuum and making use of serverless infrastructures. Various approaches can be used to limit power consumption: power capping of serverless deployments, scheduling strategies to make more effective the use of physical resources where serverless functions are hosted, and mechanisms to minimize cold start times that can have significant power consumption requirements [182]. In addition, the inherent event-driven nature of function invocation enables easy coupling with dynamic resumption, such as Wake-on-Lan, and fast-booting technologies, such as Coreboot or Jumpstart [183], in conjunction with delay-tolerant function invocations. Libraries supporting serverless platforms, such as PyWren or funcX, allow for more fine-grained power capping. In fact, such libraries could target specific subcomponents that might not need to run at full speed, and better characterize the resource requirements of its functions, thus enabling improved execution density via adaptive resource sharing among multitenant functions.

5.3.2 Digital Twins as Function Chains

Figures 5.6 and 5.7 depict a function chain triggered when a status update is received from the PT, and another function chain triggered by an action request from an application, respectively. Specifically, updates that reach the *physical interface* involve a processing pipeline comprising five main steps (see Figure 5.6):

1. *physicalEventHandler*: any change of the state of the PT S_{PT} is received as a raw physical event (e.g., a MQTT message) and normalized to a standard event ev_{PT};

2. *shadowingHandler*: given a physical event ev_{PT} and a model M, a candidate for the new DT state $S^{'}_{DT}$ is computed by means of a shadowing function;

3. *augmentationFunction*: given a candidate state $S^{'}_{DT}$, a set of (possibly parallel) augmentation functions is used to produce a richer candidate for the DT state $S^{''}_{DT}$, which consists of more properties and relationships;

4. *twinHandler*: given a possibly augmented candidate state $S^{''}_{DT}$, the new DT state S_{DT} is consolidated and a digital event ev_{DT} is computed after the entanglement and the DT life cycle state are updated. Note that the *twinHandler* step has been designed as the composition of three smaller steps, each managing a single responsibility;

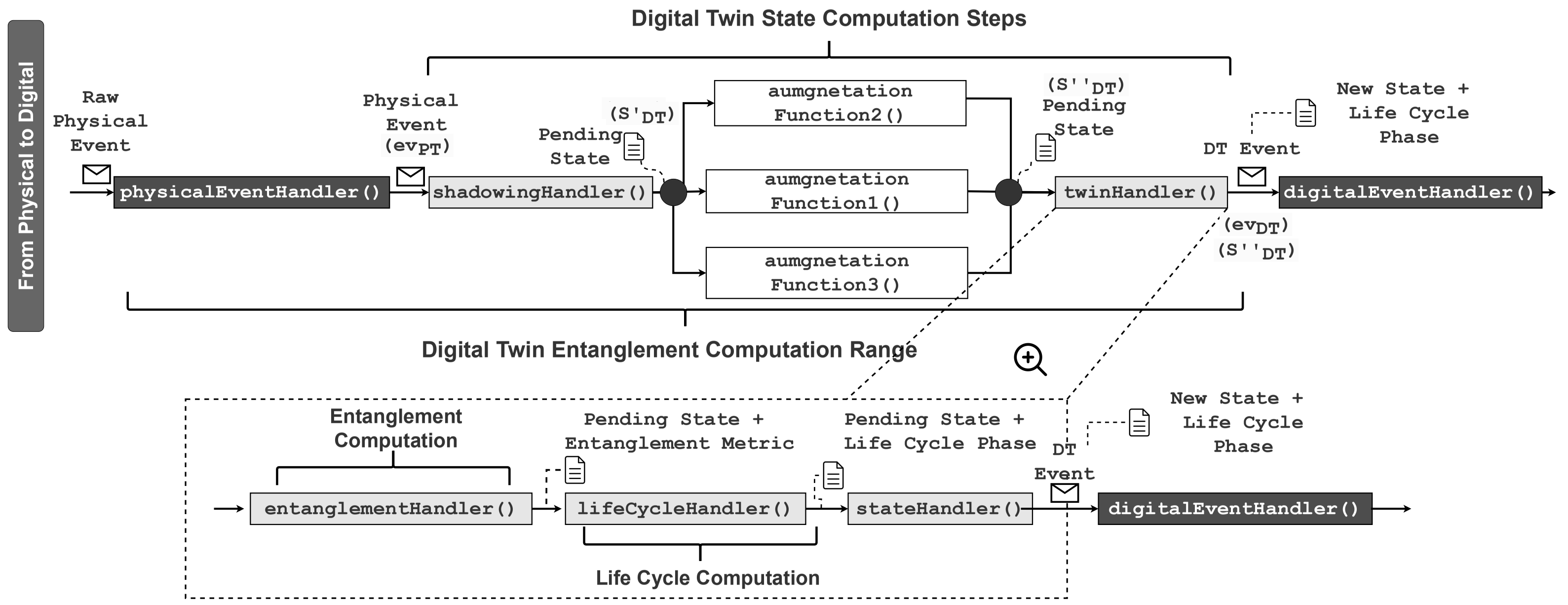


Figure 5.6: From PT to DT.

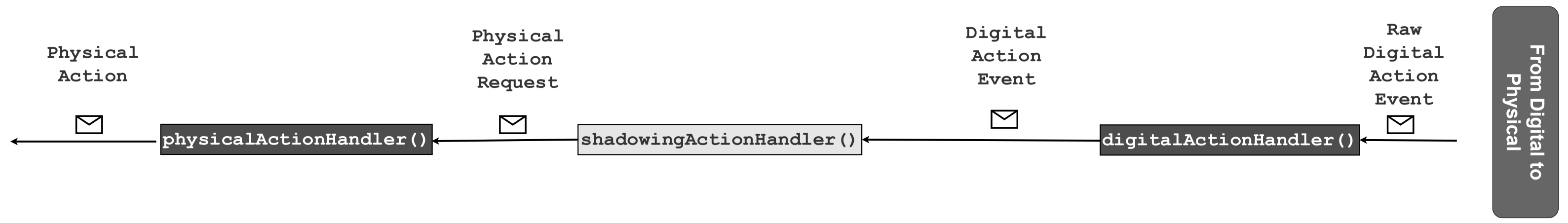


Figure 5.7: From DT to PT.

5. *digitalEventHandler*: digital event ev_{DT} is sent to listeners (e.g., external applications) via the *digital interface.*

Instead, action requests trigger a function chain that propagates on the opposite direction. This case is simpler than the previous one, typically not involving augmentation, and is based on a sequence of three main steps (see Figure 5.7):

1. *digitalActionHandler*: any request from applications is received as a raw digital action event and normalized to a standard event ev_{DT};

2. *shadowingActionHandler*: a new action request a_{PT} for the PT is generated by means of a shadowing function and propagated towards the PT;

3. *physicalActionHandler*: the action request a_{PT} is applied to the PT, determining a change of the PT state S_{PT}.

5.4 Chapter Summary

This chapter focused on DT engineering. In particular, it discussed how to build DTs through two different approaches—microservices and serverless.

Among the characteristic properties of DTs introduced by Minerva et al. [44], Section 5.1 extrapolated those relevant in the context of DT engineering: reflection, persistency, memorization, augmentation, composability, replication, and accountability/manageability. Each property was accurately defined, examined from an engineering perspective, and analyzed in terms of the potential impact upon successful implementation.

Section 5.2 explained which software engineering design patterns help meet the requirements identified while discussing the properties identified in Section 5.1, thus providing actionable tools to build context-aware, adaptive, and autonomous DTs based on microservices. The design patterns discussed in this section encompass microkernel, asynchronous queuing, sidecar, ambassador, adapter, Singleton, load balancer, proxy, and API gateway.

Lastly, Section 5.3 delved into DT engineering within the context of serverless computing, addressing both the advantages and design considerations. The serverless design originally proposed in this section relies on function chains capable of managing communication flows from physical to digital and vice versa.

Chapter 6

Entanglement-Aware Digital Twin Ecosystem

This chapter describes how to build an entanglement-aware DT ecosystem, where, as the name suggests, entanglement awareness is the most fundamental characteristic. This aligns with the definition of DT—a virtual entity entangled with an object, whether tangible or intangible, of which the DT provides a (augmented) representation in the virtual space, thus decoupling the object from the observer in space and time.

The novel contributions of this chapter consist of (i) a metric to measure the entanglement, (ii) entanglement-aware DTs, and (iii) an entanglement-aware middleware. The metric serves as a means for DTs to quantify entanglement with their physical counterparts, allowing them to be aware of the quality of entanglement over time. Entanglement-aware DTs share this information with the middleware, which then consumes it and orchestrates the DTs accordingly. The entanglement-aware DT ecosystem not only satisfies the requirements delineated in Chapter 3, overcoming the limitations of existing platforms, but also includes a metric specifically designed for measuring entanglement, thus fully supporting the proposed vision of DTs centered around the concept of entanglement.

The remainder of the chapter is structured as follows. First, Section 6.1 investigates the problem of entanglement in the context of DTs and proposes a metric to measure it. Then, Sections 6.2 and 6.3 elaborate on the architectural aspects

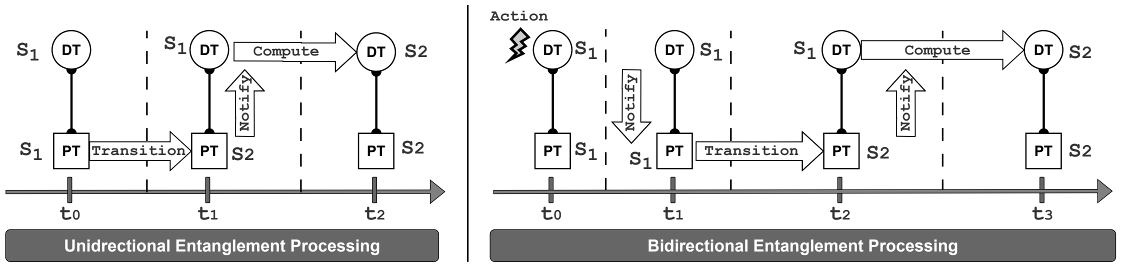

S1
DT
PT
Transition
S2
Compute
Notify
t0
t1
t2
t3
Unidrectional Entanglement Processing
Action
Bidirectional Entanglement Processing

Figure 6.1: Synchronization process for uni/bidirectional entanglement.

of entanglement-aware DTs and the middleware, respectively. The objective is to provide high-level architectural abstractions, without implementation-specific details, which will be discussed in Chapter 7. Lastly, Section 6.4 provides conclusive remarks.

6.1 The Role of Entanglement in Digital Twins

From a general perspective, the interactions between DTs and PTs can unfold in two key ways: *(a)* a state change in the PT is to be communicated to the DT; *(b)* a request to the DT from an application is to be communicated to the PT and then a state change confirmation is to be sent back to the DT and the application.

Figure 6.1-left schematically depicts the synchronization flow required to keep the DT and PT states aligned (denoted as S_i^{PT} and S_i^{DT}) when the PT detects a state change. At the beginning S^{PT} and S^{DT} are aligned in version 1 (t_0). When a new physical event (e.g., a change in the environment) occurs, it triggers a variation of the physical state (changed to S_2^{PT}) and generates a state update toward the DT. At this point, there is a misalignment between the two counterparts since the physical variation has not yet been reflected on the DT (t_1). Only when the DT receives the state update and computes its new state S_2^{DT} the two counterparts are properly synchronized (t_2). In this first scenario, the entanglement is unidirectional and directly reflects the time shift between the state of the physical entity and its digitalized replica.

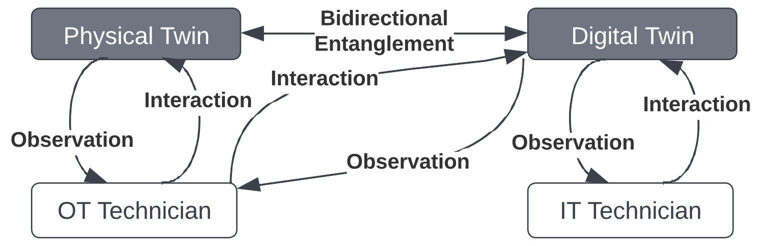
Figure 6.1-right, instead, represents a scenario in which an action is performed on the DT (e.g., from an IIoT application) and is to be propagated to the PT. It is worth noting that an action issued on the DT—aiming at modifying the state of the PT—should be intended as another form of state synchronization. When the DT receives the action, it notifies the PT about the request and waits for its state transition (from S_1^{PT} to S_2^{PT}). Then, once the state change on the PT is confirmed, the state of the DT is updated as well (from S_1^{DT} to S_2^{DT}). In this second scenario, the entanglement is even more relevant since a bidirectional exchange of information is required.

6.1.1 Entanglement: An Industrial Perspective

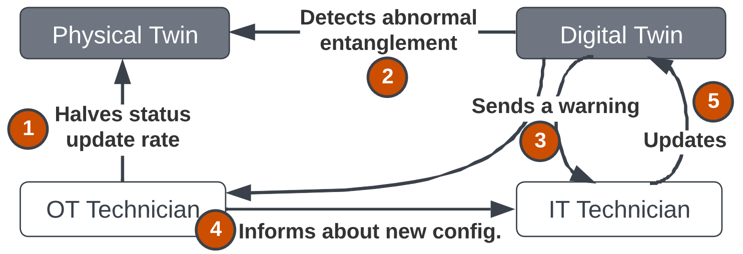
The following illustrative scenarios point out the main factors that may affect entanglement. Such scenarios illustrate the interactions between IT and OT in IIoT environments and how these interactions affect the entanglement. Specifically, each scenario involves an OT technician working on a PT and an IT technician working on a DT. For instance, the former might be a shop floor operator operating on a production line, while the latter might be a software architect deploying a DT as a microservice through a container orchestration system.

Baseline

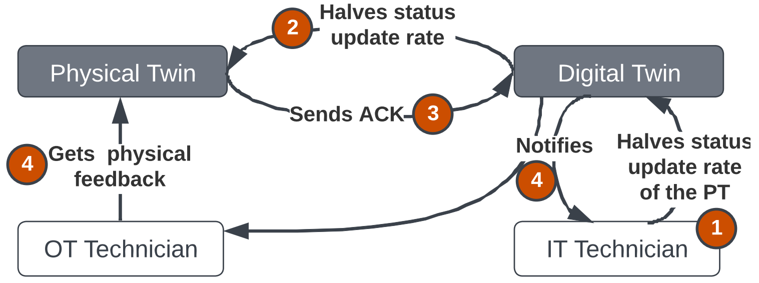
The baseline scenario describes the interactions between IT and OT in a DT-based industrial environment (see Figure 6.2a). Under the baseline scenario, it is assumed that these interactions do not disrupt the current entanglement characterizing the communication relationship between the PT and the DT. The OT technician interacts with the PT (e.g., a production line) to craft goods and observes the PT status to oversee what is going on. Additionally, the OT technician may request a simulation to the DT and, based on the simulation results, decide on the subsequent commands to send to the PT. The IT technician, instead, only interacts with the DT. Such interactions may relate, for example, to the deployment of the DT. It is worth remarking that different layers of the Purdue model provide different network performances, thus influencing the entanglement. Therefore, the IT technician should plan the DT deployment carefully and re-plan it dynamically according to the network conditions. As the number of DTs grows, so does the complexity of making effective decisions about their deployment. Thus, quantifying the entanglement in a concise yet expressive way becomes even more critical.



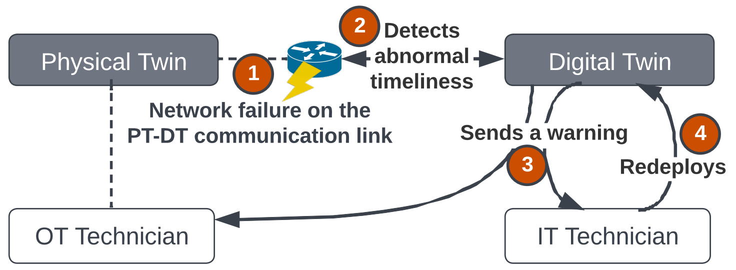
(a) Baseline.


Physical Twin
Detects abnormal entanglement
2
Digital Twin
1
Halves status update rate
Sends a warning
3
5
Updates
OT Technician
4
Informs about new config.
IT Technician

(b) Physical reconfiguration.



(c) Digital reconfiguration.



(d) Anomaly detection (network).

Figure 6.2: Illustrative industrial scenarios.

Physical Reconfiguration

The physical reconfiguration scenario sketches the case where an action of the OT technician on the PT disrupts the entanglement (see Figure 6.2b). For instance, the OT technician may change the configuration of the PT, which may result in a different status update rate, say halving the status updates per second. In turn, the DT detects an abnormal entanglement because it still expects double the status updates it is actually receiving. As soon as the DT detects that the entanglement got disrupted, it notifies the OT/IT technicians. At this time, the IT technician can only infer that something is not going as expected. Therefore, the OT technician, whose initial interaction caused the misalignment between the PT and the DT, should notify the IT technician about the change they made to the PT. Then, the IT technician can update the configuration of the DT accordingly, thus bringing the system back to a steady-state phase.

Digital Reconfiguration

The digital reconfiguration scenario sketches the case where an action of the IT technician on the DT disrupts the entanglement (see Figure 6.2c). At first glance, this scenario might seem symmetrical to the physical reconfiguration one, but it is not. In particular, the IT technician uses the DT to change the configuration of the PT. For example, the IT technician may halve the status update rate of the PT through the DT. As soon as the DT receives the instructions issued by the IT technician, it sets the PT accordingly. At this time, the DT must wait until the PT reports a status update reflecting a status change meeting the request(s) of the DT. Then, the DT can notify the OT/IT technicians back. Note that the OT technician might have already noticed that the PT changed status because of physical feedback from the PT, e.g., a robot part of the production line where the OT technician is operating changed position.

Anomaly Detection

Under the physical and digital reconfiguration scenarios, an intentional action triggered the course of action affecting the entanglement. In contrast, the anomaly detection scenario is about things that could go wrong unpredictably (see Figure 6.2d). In particular, this scenario takes into account anomalies striking either the PT (e.g., crash of the production line), the environment (e.g., poor network connectivity between the PT and the DT), or the DT (e.g., hardware fault of the server hosting the DT). Let us assume an outburst of latency upon the communication link that connects the PT and the DT. The DT can detect such an anomaly by looking at the timeliness of the received status updates from the PT. If we instead assume a crash of the PT, the DT can detect that something is not as expected because of a drop in the status update rate. Note that the OT technician may also detect the crash of the PT through physical feedback, e.g., the production line stops working. The recovery phase is started by the actor that detects the anomaly first. In the former case, the DT would start the recovery phase by notifying the IT technician, who might decide, e.g., to redeploy the DT somewhere else or fix the network. In the latter case, the OT technician would start the recovery phase, e.g., by fixing the PT. The outcome of the recovery phase is to bring the system back to a steady-state phase.

6.1.2 Limitations of Existing Metrics

The standard set of performance indicators for measuring network link performance (required to keep DTs and PTs entangled) is usually indicated with the general term of QoS. Traditionally, QoS focuses on network characteristics, such as latency, jitter, and packet loss, all of which, even if relevant, are not able to capture application-specific nuances. For instance, an increase of one second in network latency might be irrelevant for an application requiring updates every minute while dramatic for another one monitoring near real-time phenomena and requiring at least ten updates per second. To avoid such issues, alternative measures have been developed, e.g., Quality of Experience (QoE) [184] or Quality of Information (QoI) [185], with the goal of evaluating the performance of applications instead of the network links they rely upon.

In the field of IoT systems, various attempts have been made to measure application QoE via both subjective and objective means. Concerning the subjective family, recent works [186]–[189] propose QoE metrics in different contexts. However, these approaches do not assess how the application QoE relates to the individual components of IoT applications and models it through human evaluations. These approaches are not suitable for evaluating the quality of entanglement, since it is a product of a machine-to-machine process. Concerning the objective family, Li et al. [190] aimed to ensure QoE through existing QoS metrics. They proposed a regression model between QoE and QoS indicators (after extracting their principal components). Their results show that, in case of the absence of human feedback, QoE can be derived from QoS parameters. [191] proposes a QoE model for a communication app. They identified five key factors impacting QoE (i.e., integrality, retainability, availability, usability, and instantaneousness) and measured them. The final QoE value is a composition of these five measures, normalized between 0 and 1.

Existing QoE definitions focus on evaluating application quality from the lens of their users (e.g., video-conferencing) and are not suited for unsupervised use cases (i.e., applications where human feedback is unavailable). An analysis of the existing literature led Fizza et al. [184] to conclude that measuring QoE of applications where human involvement or feedback is not readily available can be approximated by observing four key features of collected data: *timeliness* (i.e., how fresh the collected data are for actually making decisions), *completeness* (i.e., the ratio of the amount of collected data to the total amount of required data), *accuracy* (i.e., the precision of the collected data), and *usefulness* (i.e., how useful the collected data are for the application).

6.1.3 Overall Digital Twin Entanglement

The main goal of the originally proposed ODTE metric is to measure in a concise yet expressive way the interactions involving state updates between DTs and PTs. Similarly to the traditional Overall Equipment Effectiveness (OEE) metric, designed to measure production effectiveness, ODTE is conceived as a multiplication of factors resulting in a number between 0 and 1. The factors involved—*timeliness* and *completeness*—have been suggested by Fizza et al. [184] for measuring the QoE of applications where data features can be captured while human feedback is unavailable. While we represent *timeliness* (T) as a single factor, *completeness* is represented with two subfactors: *reliability* (R), i.e., the ratio of the received state updates to the expected ones, and *availability* (A), i.e., the expected up-time of the PT from the perspective of the DT. Accordingly, ODTE is defined as:

$$ODTE = T \times R \times A \tag{6.1}$$

To quantify the timeliness of a state update, the DT needs to track the rate of incoming status updates over time, the elapsed time between when the PT produces a given update and when the DT receives it, and how long the DT takes to change its state (based on the received update). A suitable way to model this phenomenon is by making use of histograms. The DT may use a histogram to sample observations about the timeliness of the received updates. In this case, an observation o_i may be defined as follows:

$$o_i^{uni} = t_i^{DT} - t_i^{PT} + t_i^{exec} \tag{6.2}$$

where

- t_i^{DT} is the time at which the DT received the ith update;
- t_i^{PT} is the time at which the PT had produced the ith update;
- t_i^{exec} is the time the DT took to change state as a result of the ith update.

It is worth noting that o_i^{uni} only works for unidirectional entanglement. In the case of bidirectional entanglement, instead, an observation o_i may be modeled as follows:

$$o_i^{bi} = t_i^{PT'} - t_i^{DT'} + o_i^{uni} \tag{6.3}$$

where

- $t_i^{PT'}$ is the time at which the PT received the command from the DT;
- $t_i^{DT'}$ is the time at which the DT had issued the command.

Timeliness T can now be expressed as a quantile over a time window:

$$T(\varphi, t, O) \tag{6.4}$$

where

- $0 \leq \varphi \leq 1$ is the quantile;
- t is a time window (e.g., last 5 minutes);
- O is the set of observations about the received updates.

For example, $T(0.99, now - 5m, O) = 0.100$ means that 99% of the observations had timeliness of at most 100 ms over the last 5 minutes. For computing a normalized metric, such as ODTE, it is useful to express the timeliness as a percentage instead of in seconds. Thus, Equation (6.4) may also be defined as:

$$T'(T_d, t, O) \tag{6.5}$$

where T_d is the desired timeliness.

Equation (6.5) expresses the timeliness as a percentage and encapsulates any application-specific detail within the DT itself. It is reasonable, in fact, to assume that a DT is aware of the desired timeliness (T_d) of its physical counterpart. For example, if T_d is set to 200 ms (i.e., anything lower than 200 ms meets the requirement), $T'(200ms, 5m, O) = 0.999$ means that 99.9% of the updates had the desired timeliness. By doing so, anyone (or anything) monitoring the DT can understand if the timeliness of state updates respects the entanglement requirements without any a priori, application-specific knowledge.

Timeliness itself does not account for those updates that are never received by the DT, which, instead, are taken into account by the completeness factor. As stated above, completeness consists of two subfactors, namely R and A. Firstly, R measures the reliability of an entity expressed as the ratio of the received state updates to the expected ones within a specified time frame. Formally:

$$R(t, O) = \frac{u_{measured}(t, O)}{u_{expected}(t)} \tag{6.6}$$

where

- $u_{measured}(t, O)$ is the per-second average rate of the received updates based on the set of observations O over the time window t;
- $u_{expected}(t)$ is the *minimum* per-second average rate of the expected updates over the time window t. If $u_{measured}(t, O) > u_{expected}(t)$, then $R(t, O) = 1$.

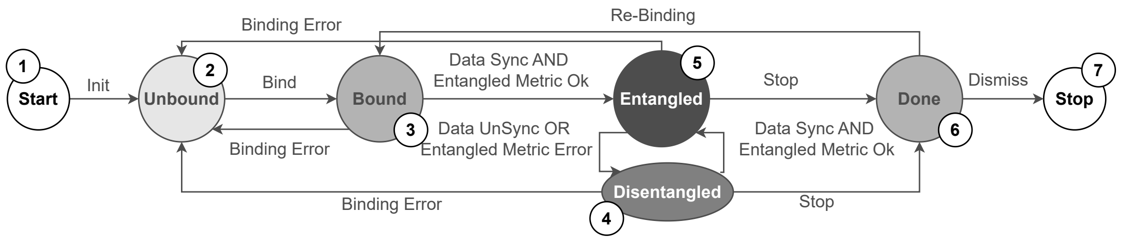


Figure 6.3: Life cycle.

For example, $R(now - 5m, O) = 0.5$ indicates that the DT received half of the expected updates within the last 5 minutes. Secondly, A measures the availability of the PT over a specified time frame. For example, $A(now - 5m, O) = 0.5$ means that the PT was active only half of the expected time over the last 5 minutes. Putting the three components together, the ODTE is defined as:

$$ODTE = T'(T_d, t, O) \times R(t, O) \times A(t, O) \tag{6.7}$$

From an operational viewpoint, the DT should be responsible for quantifying its own ODTE to provide either human operators or applications with a representation of its entanglement. It would also be possible to compute the ODTE outside the DT, e.g., by third parties services querying a time-series database containing T, R, and A.

6.2 Entanglement-Aware Digital Twins: Life Cycle and Architecture

Entanglement-aware DTs should monitor entanglement with their physical counterparts and evaluate it according to design principles, the context where they operate, and the application requirements. Following this principle, the life cycle proposed in [50] modeling the behavior over time of a DT-PT duality can be extended as follows (see Figure 6.3). Upon its start, the DT is *Unbound* and ready to bind with the PT. Once the binding is completed (a network channel with the PT is established and the DT is ready to initiate the digitization process), the DT moves to the *Bound* state and the quality of entanglement starts to be measured.

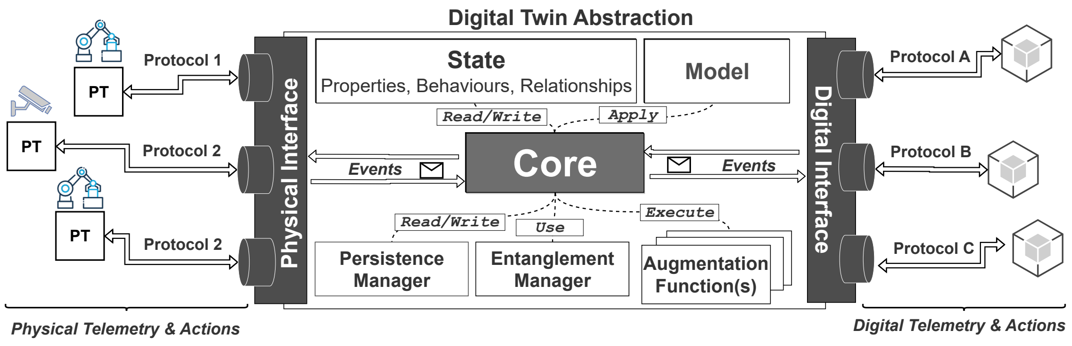


Figure 6.4: Entanglement-aware DT architecture.

If binding errors occur, the state reverts back to *Unbound* and the DT tries to recover the channel. Networking or computational resource issues involving the DT-PT synchronization and degrading the level of entanglement below a target threshold bring the DT into the *Disentangled* state. In this state, the DT becomes unable to provide its intended functionality. From the *Disentangled* state, the DT can transition to either the *Unbound* or *Done* state in case of an error during the binding procedure or if it is explicitly stopped by the middleware. Upon successful error recovery, the DT reverts back to the *Entangled* state. In the *Done* state, the DT remains accessible to external applications as a software component detached from the PT, retaining its memory and exposing collected historical data, events, and metrics together with the last DT state until it is dismissed, by transitioning to the *Stop* state.

Figure 6.4 depicts the architecture of an entanglement-aware DT. This architecture is built on top of state-of-the-art principles [44], [49], [50], aligns with the requirements described in Chapter 3, and is event-driven in nature (thus working for both microservices and serverless DT implementations).

From a technical point of view, the *Digital Twin Model* is responsible for determining how and when changes in the physical world should be mapped into the digital replica, as well as propagating inputs and actions to the PT. The model closely works with the *Digital Twin State* component, storing *attributes* (e.g., physical properties), *behaviors* (e.g., actions that can be performed on the DT), and *relationships* (e.g., modeling how PTs are linked in the physical space). The interaction with the physical and digital layers builds upon the *Physical* and *Digital*

Interfaces, each composed of different *Adapters* (which implement protocols and data formats). The model receives inputs from the physical layer. Such inputs are reflected in the digital representation either immediately or after various transformations to align them with the DT model (e.g., resampling signals, changing metric units). Given that modifying the functionalities of physical objects might be costly and complex, a physical asset can be functionally expanded through its DT, using a collection of *Augmentation Modules* introducing additional attributes, behaviors, or relationships. All internal DT modules are supported by a *Storage & Persistence* component that handles the memorization and retrieval of past DT states and relevant events.

The DT also integrates an *Entanglement Manager* responsible for monitoring the quality of entanglement. This module plays a role in ensuring that the DT maintains an up-to-date representation of its PT, thereby enabling analysis, prediction, and decision-making. It also adjusts the DT state and generates contextual metrics. As previously introduced, the traffic volume to maintain the DT-PT duality can significantly vary across deployments and scenarios, ranging from real-time interactions (high volume) to batch processing (low volume). For this reason, it is key to use a metric for measuring the quality of entanglement (e.g., ODTE, see Section 6.1.3) that is decoupled from the specific use case in which the DT is used.

DTs also include a *Monitoring & Management Interface* (illustrated at the bottom of Figure 6.4) as a tool that allows human operators and the middleware to accommodate (time-varying) application requirements. Furthermore, the interface exposes DT contextual metrics (e.g., quality of entanglement and internal life cycle), providing insights into the performance and effectiveness of the DT.

6.3 Entanglement-Aware Middleware: Architecture

The first objective of the entanglement-aware middleware is to manage the execution of DTs while ensuring compliance with cyber-physical application requirements. This includes selecting the most suitable configuration and deployment strategy based on the current context. The middleware proactively monitors the quality of entanglement, facilitates optimal deployment execution, and plan countermeasures against performance degradation. To achieve these objectives, the middleware is structured around two main components: the *Core Node* and the *Worker Nodes* (on the left and right parts of Figure 6.5, respectively). The Core Node acts as the control plane and manages the operations of the distributed DTs. Worker Nodes can be deployed on different network layers, such as edge on-premises, MEC, and cloud, and execute the DTs. They run dedicated agents that facilitate communication with the core and manage the execution of deployed DTs.

The Core Node has a structured design with internal components and external interfaces, is dedicated to DTs management, and its use is intended for stakeholders, application designers, and platform administrators. It communicates with Worker Nodes via the *Monitoring Interface* (to collect contextual data from DTs) and the *Orchestration Interface* (to control DTs). Additionally, through the *Management Interface*, stakeholders can provide input, specify application requirements, quality of entanglement, and interact with the ecosystem. Finally, the *Reporting Interface* is to visualize and interact with the ecosystem. It allows the visualization of running applications with their DTs, inspecting resource utilization, accessing logs, and monitoring the health of the system.

The *Middleware Knowledge* component stores configurations, events, and actions executed by the middleware and consists of three subcomponents: the *DT Repository*, which contains DT artifacts, the *Infrastructure Knowledge*, which stores specifications and configurations of the cloud-to-edge continuum infrastructure, and the *Application Repository*, which stores cyber-physical application descriptions (including deployment specifications). The *Orchestrator* component manages DT orchestration strategies. If not specified by the application, it identifies the most suitable deployment configuration across the continuum. Then, it monitors contextual information (reading data from the Data Storage component) and plans actions to keep the quality of entanglement above the target. The *Data Storage* component represents a structured and multi-functional persistency layer that stores metrics, logs, and events. This component ensures consistent and up-to-date information for effective decision making and provides historical records of platform activities, thus enabling analysis and auditing. This component manages

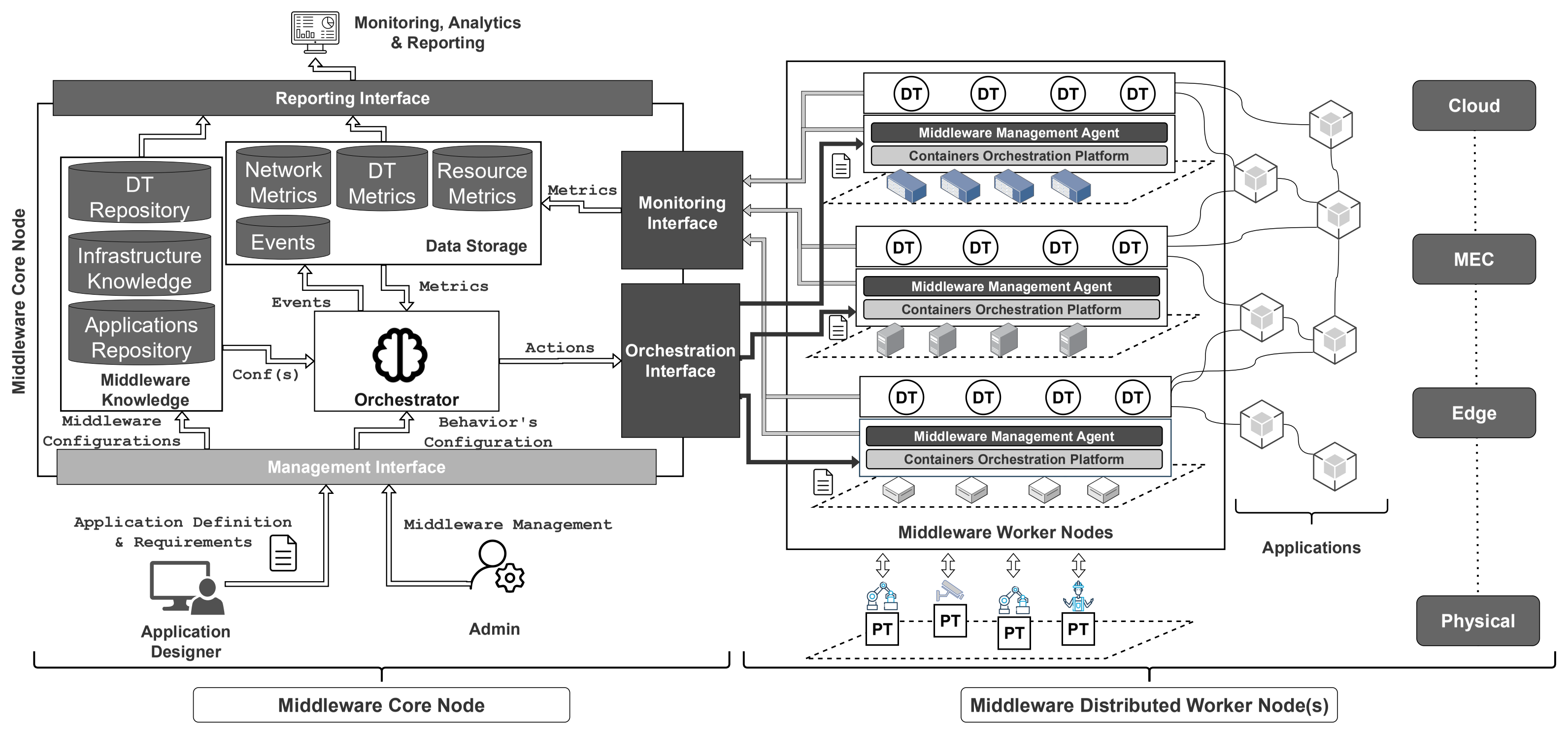


Figure 6.5: Entanglement-aware middleware architecture.

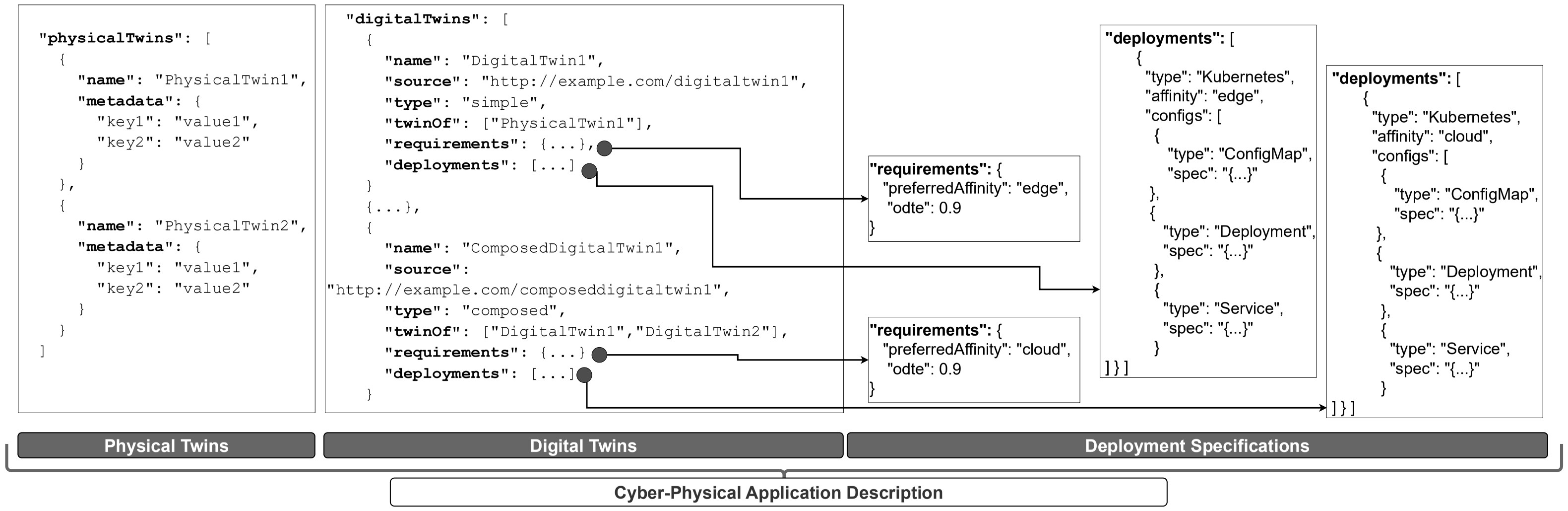
information related to *Network Metrics*, *Resource Metrics*, *DT Metrics*, and the *Event History*, which collects all orchestration-related events.

Figure 6.6 provides an illustrative example of a cyber-physical application description. In contrast to traditional applications, cyber-physical ones encompass both physical and digital layers. The physical facet of such applications consists of one or more PTs, identified by a unique identifier and described by a set of metadata (in the form of key-value pairs) capturing the relevant features of the physical object. It is worth noting that physical object metadata might vary depending on what is relevant in the context of a cyber-physical application. The digital facet, instead, consists of one or more DTs characterized by an unique identifier, a source (a reference to the container image to be executed), a type (either simple or composed), what it twins, a set of requirements (e.g., ODTE threshold), and a list of allowed deployments configurations (described below). A DT is simple if it twins one ore more PTs, while composed if it represents the status of other DTs. For example, the DT representing a digital factory is likely to be the composition of several underlying DTs, both simple (e.g., industrial machines) and composed (e.g., production lines).

DTs are deployed according to their preferred locations as long as the quality of entanglement is above the threshold set. When the ODTE value goes below the threshold, the DT becomes *Disentangled* (see Figure 6.3) and alternative deployments are provided as fallback strategies. A Deployment (see Figure 6.6, on the right) includes a type, an affinity, and a set of configuration files (needed to implement it). For example, if the type is "Kubernetes," the configuration files are expected to be Kubernetes objects, such as Deployment, Service, and ConfigMap. Another possibility would be to use Helm charts instead of raw Kubernetes objects. In that case, the type would be "Helm." The affinity specifies for which location the deployment is targeted along the cloud-to-edge continuum.

6.4 Chapter Summary

This chapter proposed an entanglement-aware DT ecosystem. Entanglement is what distinguishes DTs primarily from traditional software components, but there are no existing metrics available that are suitable to measure it.



```
"physicalTwins": [
  {
    "name": "PhysicalTwin1",
    "metadata": {
      "key1": "value1",
      "key2": "value2"
    }
  },
  {
    "name": "PhysicalTwin2",
    "metadata": {
      "key1": "value1",
      "key2": "value2"
    }
  }
]
```

```
"digitalTwins": [
  {
    "name": "DigitalTwin1",
    "source": "http://example.com/digitaltwin1",
    "type": "simple",
    "twinOf": ["PhysicalTwin1"],
    "requirements": {...},
    "deployments": [...]
  }
  {...},
  {
    "name": "ComposedDigitalTwin1",
    "source":
"http://example.com/composeddigitaltwin1",
    "type": "composed",
    "twinOf": ["DigitalTwin1","DigitalTwin2"],
    "requirements": {...}
    "deployments": [...]
  }
```

```
"requirements": {
  "preferredAffinity": "edge",
  "odte": 0.9
}
```

```
"requirements": {
  "preferredAffinity": "cloud",
  "odte": 0.9
}
```

```
"deployments": [
    {
    "type": "Kubernetes",
    "affinity": "edge",
    "configs": [
     {
       "type": "ConfigMap",
       "spec": "{...}"
     },
     {
       "type": "Deployment",
       "spec": "{...}"
     },
     {
       "type": "Service",
       "spec": "{...}"
     }
]}]
```

```
"deployments": [
    {
    "type": "Kubernetes",
    "affinity": "cloud",
    "configs": [
     {
       "type": "ConfigMap",
       "spec": "{...}"
     },
     {
       "type": "Deployment",
       "spec": "{...}"
     },
     {
       "type": "Service",
       "spec": "{...}"
     }
]}]
```

Physical Twins
Digital Twins
Deployment Specifications
Cyber-Physical Application Description

Figure 6.6: Example of a cyber-physical application description.

To bridge this gap, Section 6.1 explored the entanglement in the context of DTs. Section 6.1.1 elaborated on four illustrative industrial scenarios (i.e., baseline, physical reconfiguration, digital reconfiguration, and anomaly detection) to identify factors that can affect entanglement. Then, Section 6.1.2 discussed the limitations of existing metrics in representing entanglement. Lastly, Section 6.1.3 proposed ODTE—a concise yet expressive metric to measure entanglement.

ODTE was the first building block towards the entanglement-aware DT ecosystem. Section 6.2 detailed the life cycle and architecture of an entanglement-aware DT. The life cycle involves states to identify whether the DT is *Bound* or *Unbound* with the PT, where *Bound* means that the DT is ready to initiate the digitalization process, and, once *Bound*, whether the DT is *Entangled* or *Disentangled*. The proposed architecture aligns with the requirements described in Chapter 3 and its event-driven nature makes it suitable for microservices and serverless DT implementations.

Finally, Section 6.3 elaborated on the entanglement-aware middleware. The primary objective of the middleware is to manage the execution of DTs while ensuring compliance with cyber-physical application requirements. The section also provides an illustrative example of a cyber-physical application description, which serves as a comprehensive construct that unifies DTs and their PTs. The middleware's decision-making is intrinsically tied to the requirements specified in the cyber-physical application description, such as the preferred DT deployment along the cloud-to-edge continuum, or under which ODTE value a DT is no longer considered entangled.

Chapter 7

Experimentation

This chapter provides insights into the implementation of the entanglement-aware DT ecosystem. In this regard, Section 7.1 explores the proof-of-concept implementation of the ecosystem. Entanglement-aware DTs have been realized through two distinct approaches: microservices and serverless, offering a comprehensive representation of the concepts discussed in Chapter 5. These implementations adhere to the architectural guidelines laid out in Section 6.2. The entanglement-aware middleware has been implemented in the microservices flavor only. This is because serverless implementations of entanglement-aware DTs are heavily tied to the serverless framework of the cloud provider.

The chapter evaluates the performance results and overhead of the implemented entanglement-aware DT ecosystem (see Section 7.1). The notable outcome of this experimentation is not only the demonstration of the feasibility of such an ecosystem but also its effectiveness in enforcing entanglement. The accompanying analysis also highlights the benefits and drawbacks of microservices and serverless implementations. In particular, Section 7.2 details the testbed used for experimentation. Then, Section 7.3 analyzes the results of the performed experiments. This analysis (i) demonstrates that entanglement-aware DTs can measure the entanglement using the ODTE metric, (ii) quantifies the resource overhead of the entanglement-aware DT ecosystem, and (iii) shows that the entanglement-aware middleware can maintain the desired quality of entanglement over time, even in the presence of failures. Lastly, Section 7.4 provides the summary of the chapter.

7.1 Entanglement-Aware Digital Twin Ecosystem: Implementations

This section provides an overview of the implementations of the entanglement-aware DT ecosystem. The microservices implementation is detailed in Section 7.1.1, and the serverless implementation is covered in Section 7.1.2.

7.1.1 Microservices Implementation

Entanglement-Aware Digital Twins

The Microservices Digital Twin (MDT) was implemented through the White Label Digital Twin (WLDT) library, a modular Java stack based on a shared multithread engine to implement DT behavior and define its mirroring procedures, data processing, and interaction with external applications [192]. The library was extended to support cyber-physical entanglement and, more in general, the requirements described in Chapter 3 and the architectural specifications of Section 6.2. A management interface was added to allow dynamic control and re-configuration of a target DT, and existing metrics management systems were updated to expose life cycle and entanglement core metrics, thus matching the interoperability requirements with the middleware monitoring interface. The container images of the implemented DT were hosted in a dedicated container registry (i.e., DT Repository).

Entanglement-Aware Middleware

The Orchestrator (see Figure 6.5) was implemented as a module written in Go, which is a programming language that provides built-in support for concurrency through goroutines (i.e., lightweight threads of execution) and channels (the way goroutines exchange messages and synchronize their operations), scalability, high performance, and efficiency. The primary objective of the Orchestrator is to keep the quality of entanglement within the cyber-physical application constraints. The Management Interface was implemented as a RESTful API that offers endpoints to create, update, and delete cyber-physical applications. The OpenAPI Generator was used to generate the web server stub programmatically. Cyber-physical application definitions were stored as JSON files in a dedicated key-value store (i.e., Applications Repository) built on top of Etcd, a strongly consistent, distributed key-value store that organizes data hierarchically into directories. Once deployed, DTs expose metrics such as their life cycle state and the ODTE measure. Prometheus was used to collect such metrics periodically, store them in a real-time time-series database (i.e., DT Metrics), and query the database to extrapolate aggregated insights. Dedicated clients were integrated to make the Orchestrator interact with Etcd, Prometheus, and Kubernetes. Specifically, this implementation relies on (i) Etcd to be notified whenever an application definition is created, updated, or deleted, (ii) Prometheus to know whenever a DT becomes Disentangled, and (iii) Kubernetes to enforce orchestration decisions (e.g., a new deployment when a cyber-physical application is created or an alternative deployment when a DT becomes Disentangled).

7.1.2 Serverless Implementation

Entanglement-Aware Digital Twins

The Serverless Digital Twin (SDT) was implemented as an Azure Function App using Python. Such a Function App relies on Durable Functions, an extension for dealing with stateful applications in a serverless compute environment. To realize the event-driven functions illustrated in Figure 5.6, the function chaining pattern was used. This pattern refers to the ability to chain multiple functions together in a sequence, where each function output serves as the input to the next function in the chain. This pattern is achieved by using an orchestration function in charge of defining the sequence of function calls (known as the function chain). When the orchestrator function is triggered, it then calls multiple functions one after the other, passing the output of one function as the input to the next function in the chain. In this implementation, the orchestrator function is triggered whenever a new status update is published to the IoT Hub—a managed service hosted on the cloud acting as a central message hub for bidirectional communication from the edge to the cloud and vice versa.

7.2 Testbed Setup

The vision of DTs as microservices requires an underlying container-orchestration system. To this purpose, Kubernetes was used—the de facto industry standard container-orchestration system currently. As the number of DTs grows, so does the complexity of the system, which requires a scalable technology stack that adequately supports the deployment of DTs as microservices. This calls for the adoption of a so-called service mesh, an infrastructure layer that keeps the management, observability, and security of the whole environment practical at scale. Service meshes typically provide these capabilities (almost) transparently, i.e., with no (or few) service code changes. In this regard, Istio was adopted, which is arguably the most popular solution. Istio extends Kubernetes to establish a programmable, application-aware network [193].

In addition to the technology stack layer providing scalable deployment of DTs, there is also the need to support the monitoring of the environment, in particular to check that everything works as desired. Let us note that the monitoring phase is paramount for the OT domain, since DTs have to fulfill strict requirements, e.g., strong entanglement. In this regard, Prometheus was chosen to scrape metrics from DTs, store such metrics in a real-time time-series database, and query the database to extrapolate aggregate insights [194].

Since the introduction of the Symian Army toolkit by Netflix in 2011 [167], several tools have been released that can help software developers and administrators to discover unexpected vulnerabilities of software systems running in a production environment. In fact, the application of Chaos Engineering well suits complex microservice-based applications where the failure of a single microservice could undermine the execution of the whole distributed application. Chaos Mesh is a cloud-native Chaos Engineering platform for Kubernetes that allows injecting a broad spectrum of faults into a target [195].

The testbed consisted of a Kubernetes cluster of four nodes, each within its own VM. A single node acted as the master (i.e., the node running the control plane) while the others joined as workers (i.e., regular cluster nodes). Each VM was equipped with 2 vCPU and 8 GB of RAM, each running the Ubuntu operating system. The entire software stack was automatically installed in a configurable and reproducible fashion. In this regard, Ansible was used, a well-known configuration management tool that allows a control node to configure a set of target nodes over SSH [196]. Through Ansible, the Kubernetes cluster was built (i.e., Kubernetes along with the ancillary software required by Kubernetes to run successfully, such as cri-o [197] and Flannel [198]) as well as Istio, Prometheus, and Chaos Mesh deployed. The project developed to configure the testbed has been made publicly available on GitHub [199].

7.3 Experimental Evaluation

This section outlines the conducted experiments and evaluates their outcomes. Section 7.3.1 presents experiments designed to showcase the entanglement awareness of the implemented DTs. These experiments consider different scenarios, configurations, and deployment strategies, all of which impact the entanglement in varying ways. Then, Section 7.3.2 quantifies the overhead of the implemented entanglement-aware DT ecosystem, assessing network, CPU, and memory resource utilization. Finally, Section 7.3.3 illustrates the middleware's orchestration capabilities in preserving the entanglement of a cyber-physical application, even in the presence of injected failures.

7.3.1 Entanglement-Aware Digital Twins: Performance Results

Microservices Implementation

Through Kubernetes, a DT (the implementation described in Section 7.1.1), a PT, and a message broker were deployed as containerized applications. The PT, which mimics the behavior of an IIoT device, was also implemented in Java and, as the DT, exposes proper interfaces to make its behavior configurable at run-time. The message broker was Mosquitto, which supports the MQTT protocol. The PT sent status updates to the DT as MQTT messages, i.e., publishing on a specific topic to which the DT was subscribed. Analogously, the DT issued commands to the PT publishing on another topic to which the PT was subscribed.

Table 7.1: Experiments based on the Purdue model layers

Purdue Model	**Network conditions**	**Latency $\pm$ Jitter**	**Loss**
	Regular (R)	2.5 $ms \pm$ 2.5 ms	-
Layer 0/1	Deteriorated (D)	5 $ms \pm$ 5 ms	5 %
	Critical (C)	12.5 $ms \pm$ 12.5 ms	15 %
	R	12.5 $ms \pm$ 7.5 ms	-
Layer 2	D	25 $ms \pm$ 15 ms	5 %
	C	50 $ms \pm$ 30 ms	15 %
	R	35 $ms \pm$ 15 ms	-
Layer 3	D	70 $ms \pm$ 30 ms	5 %
	C	175 $ms \pm$ 75 ms	15 %

We first focused on the timeliness factor of the ODTE metric in the context of an industrial environment based on the Purdue model. In this regard, Table 7.1 details the network conditions we injected while performing the experiments. More specifically, we set the regular (R) network conditions potentially affecting each layer of the Purdue model. For example, a DT deployed at layer 0/1 under regular network conditions experienced a one-way latency of 2.5 $ms \pm 2.5$ ms (with a correlation between consecutive packets of 25%) and no packet loss. We also defined plausible deteriorated (D) and critical (C) network conditions for each layer. These experiments were conducted over a time window of 5 minutes.

We then explored the responsiveness of the ODTE metric under the illustrative scenarios described in Section 6.1.1. The physical reconfiguration scenario was emulated by halving the status update rate sent by the PT to the DT, i.e., from 1 to 0.5 status updates per second. Then, we instantiated the digital reconfiguration scenario by forcing the DT to calculate 17.5K prime numbers while performing a state transition. Lastly, we produced two instances of the anomaly detection scenario to investigate the responsiveness of the ODTE metric to latency (i.e., 50 $ms \pm 50$ ms) and the combined effect of latency (as before) and packet loss (i.e., 10%). We performed these experiments in three phases (5 minutes each) over a time window of 15 minutes overall. The first phase resembled the baseline scenario, the second put into action a given scenario, and the third consisted of rolling back what had been injected to reproduce the scenario (thus bringing the system back to the baseline).

Figure 7.1 shows the results that we collected from the experiments that focus on timeliness. The results are expressed in percentiles, i.e., 90th, 95th, 98th, 99th, and 99.9th, and computed based on the metrics Prometheus scraped over a 5-

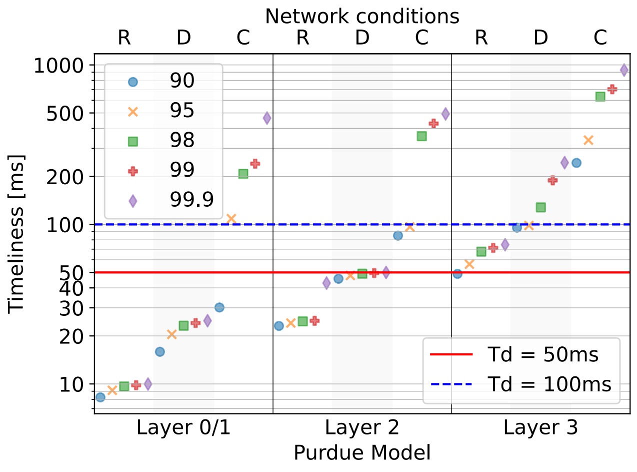
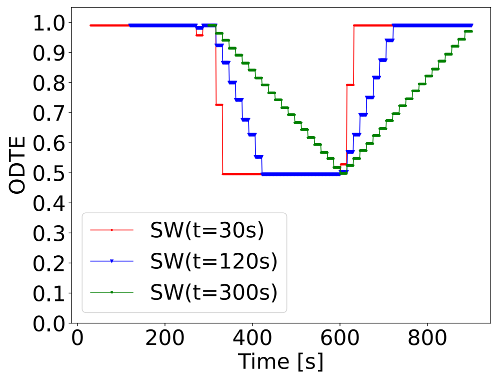


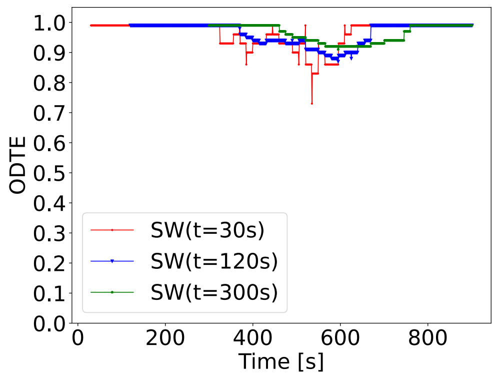
Figure 7.1: Purdue model.

minute window. Note that the left y-axis depicts the timeliness expressed in ms on a logarithmic scale (base 10). The bottom x-axis divides the figure into three vertical macro-sections, each representing a layer of the Purdue model. The top x-axis further divides those macro-sections based on plausible network conditions, i.e., regular, deteriorated, and critical, that might affect each layer of the Purdue model (see Table 7.1). For example, the yellow cross on the second column means that 95% of the status updates received by the DT had timeliness of at most 20 ms over the observed 5-minute window. The solid red horizontal line distinguishes the layers of the Purdue model that fit a DT with desired timeliness of 50 ms between those that do not. If the target is the 90th percentile, then layer 0/1 represents a suitable option under any network condition. Layer 2 is also a suitable option but only up to the deteriorated network conditions (the 90th percentile almost doubled the desired timeliness while critical network conditions occurred). The dashed blue horizontal line, instead, refers to a DT whose desired timeliness is 100 ms. If we still assume that the target is the 90th percentile, then both layers 0/1 and 2 are suitable deployment options under any network condition.

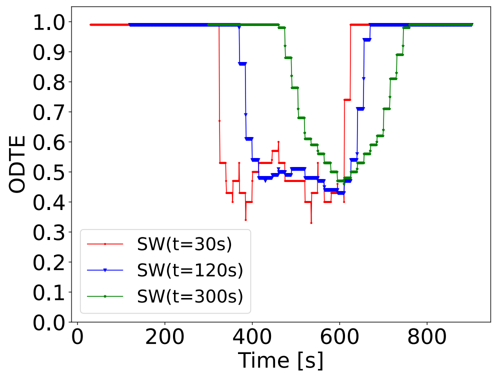
Figures 7.2a, 7.2b, 7.2c, and 7.2d show the responsiveness of the ODTE metric over a 15-minute time window concerning the experiments instantiating the (a) physical reconfiguration scenario, (b) digital reconfiguration scenario, (c) the anomaly detection scenario where the anomaly was an outburst of latency, and



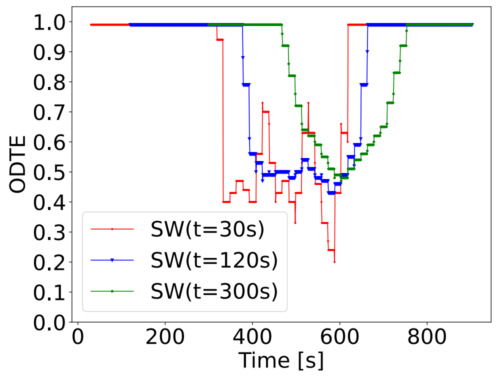
(a) Physical reconfiguration.



(b) Digital reconfiguration.



(c) Anomaly detection: Latency.



(d) Anomaly detection: Latency & Loss.

Figure 7.2: ODTE performance in different experimental scenarios.

(d) where two anomalies were in place simultaneously, i.e., latency and loss. Such figures depict three lines, i.e., red, blue, and green, each plotting the ODTE computed on a different sliding window (see the abbreviation SW in the legend), i.e., 30 s, 2 min, and 5 min, respectively. On the one hand, a shorter sliding window makes the ODTE metric more responsive (the red line reacts faster than the others to the scenario). On the other hand, a shorter sliding window also makes the ODTE metric more sensitive to noise. The wide fluctuations depicted in Figure 7.2d (see the red line) make evident the impact of a shorter sliding window on the ODTE metric. However, this does not mean that longer sliding windows are always better than shorter ones. The sliding window width choice should reflect

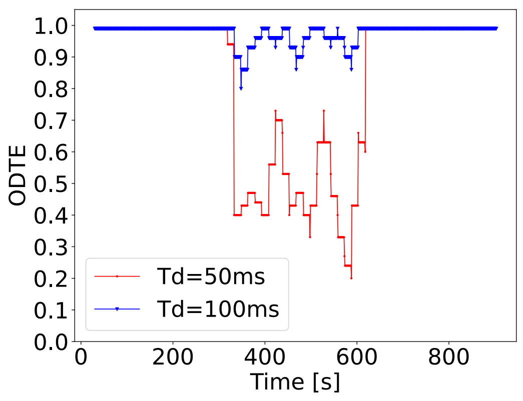


Figure 7.3: ODTE comparison between applications with different T_d.

the target DT sensitivity to short-lived variations of the entanglement over time. Finally, Figure 7.3 shows the ODTE (sliding window of 30 s) concerning two DTs whose desired timeliness was 50 ms (red line) and 100 ms (blue line), and both were performing under the same anomaly detection scenario with latency and loss as described above. Note that it is straightforward to understand if the DT is experiencing a "good" or "bad" entanglement. Also, the application-specific knowledge, i.e., the desired timeliness, is embedded within the ODTE metric. Finally, it is worth pointing out that the DT with 100 ms of desired timeliness was much less influenced (blue line) by the scenario than the one whose desired timeliness was 50 ms (red line).

Serverless Implementation

Figure 7.4 illustrates the deployments along the cloud-to-edge continuum we used to experimentally evaluate the serverless implementation, i.e., SDT, and compare it with the microservices one, i.e., MDT. Specifically, Figure 7.4-top and Figure 7.4-bottom depict the deployments used to assess the SDT and the MDT, respectively. They both involved two different applications: an edge application requiring strong cyber-physical entanglement and a third-party application designed to perform batch analytics with more relaxed entanglement requirements.

The first deployment (Figure 7.4-top) was used to evaluate the performance of the SDT to represent the PT, which was configured to transmit a status update

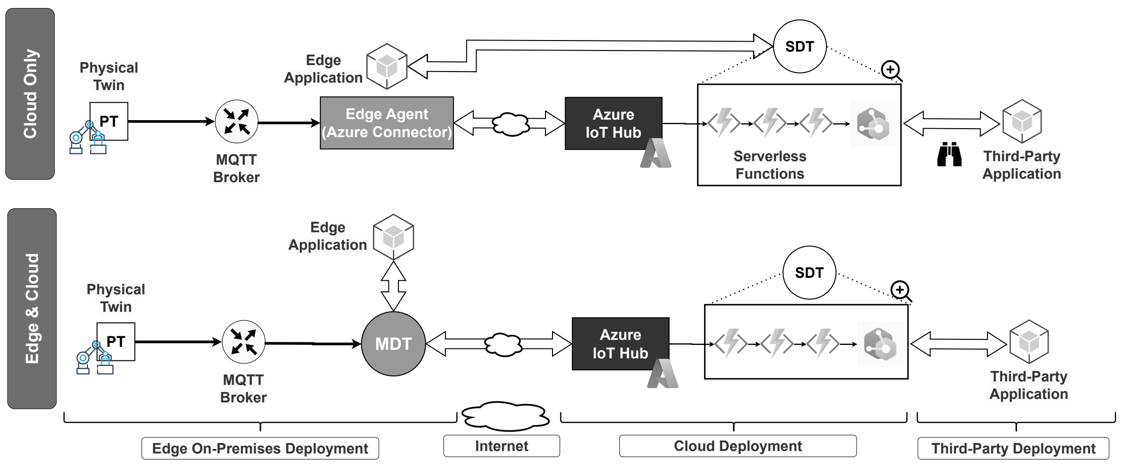
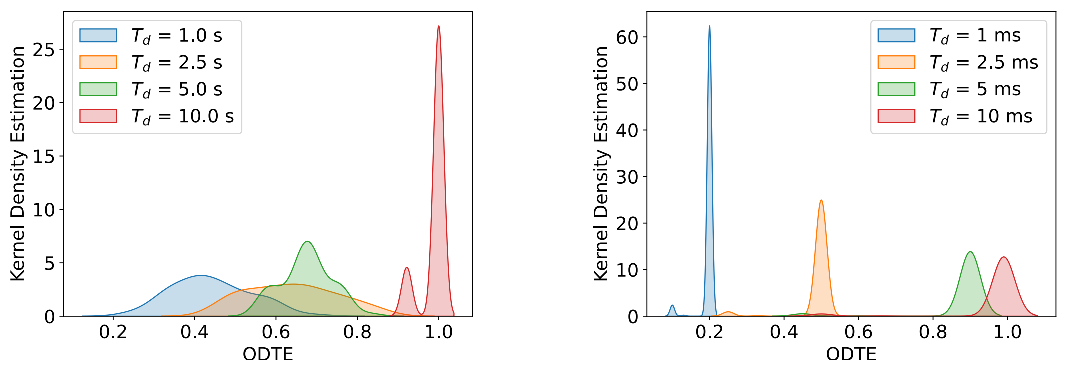


Figure 7.4: Cloud-only deployment (top) vs edge & cloud deployment (bottom).



(a) (b)

Figure 7.5: SDT (a) vs. MDT (b): ODTE for different T_d values.

every five seconds. Figure 7.5a shows the kernel density estimation of the ODTE for several values of T_d. The ODTE mean was 0.44 ± 0.10, 0.64 ± 0.11, 0.68 ± 0.06, and 0.99 ± 0.03 for T_d of 1 s, 2.5 s, 5 s, and 10 s, respectively. These results confirm a positive correlation between T_d and the ODTE and indicate that the SDT was successfully entangled only when T_d was set to 10 s, which is a relatively high value. However, it is worth remarking that the performance within the cloud may fluctuate significantly and may be influenced by a wide variety of factors, such as the type of subscription, the quality of resources, and the resource quotas, among others.

The second deployment (Figure 7.4-bottom) addresses the limited performance observed in the cloud-only deployment. In this case, there was also an MDT running on edge on-premises, thus closer to the PT. Figure 7.5b shows the kernel density estimation of the ODTE confirming the positive correlation observed for the SDT. However, due to proximity to the PT and the simplicity of the implementation compared to the SDT, T_d could be set an order of magnitude lower than in the previous case.

This experiment shows how different implementations of the same DT can coexist along the cloud-to-edge continuum. Indeed, both the SDT and the MDT implement (in different ways) the architecture described in Figure 6.4. From the perspective of the applications, it makes no difference whether to interact with the SDT or the MDT. The choice between the two can be based only on performance or economic factors. Generally, cloud-based implementations provide lower performance (i.e., worse ODTE measures) in exchange of more cost-effective and scalable deployments compared to edge implementations.

7.3.2 Entanglement-Aware Digital Twin Ecosystem: Overhead

Figure 7.6 describes the phases of the experiments to assess the entanglement-aware DT ecosystem overhead: i) initial deployment (left); ii) context variation (centre); and iii) deployment adaptation (right). Each phase is described and detailed in the following paragraphs.

Initial Deployment The initial deployment phase describes a typical industrial setting in a steady state. As illustrated in Figure 7.6, we distinguish three logical layers: physical, digital twin, and application. The physical level is on the shop floor and comprises three IIoT devices that publish their status information on a MQTT message broker, i.e., the so-called IIoT Devices Broker. Such a broker may be defined as a physical broker to refer to its responsibility to exchange data with physical devices. In contrast, the digital level spreads from the control room to the shop floor. Here, three elementary DTs consume the information published on the physical broker to clone the IIoT devices (i.e., PT) into software

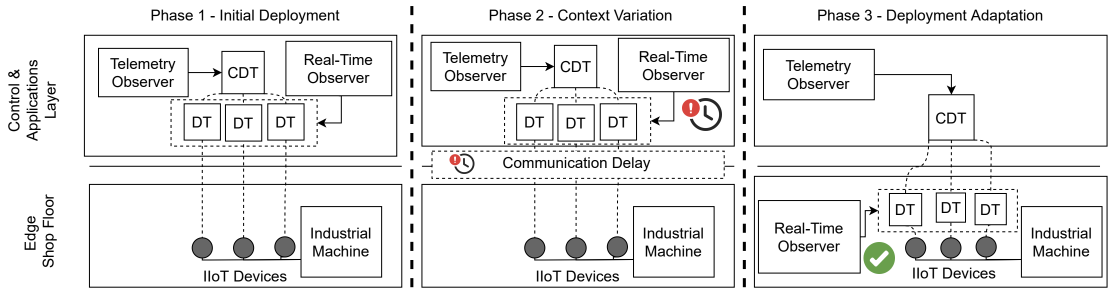

Phase 1 - Initial Deployment
Phase 2 - Context Variation
Phase 3 - Deployment Adaptation
Control & Applications Layer
Edge Shop Floor
Telemetry Observer
CDT
Real-Time Observer
DT
DT
DT
Industrial Machine
IIoT Devices
Communication Delay

Figure 7.6: Phases of the experiments for overhead assessment.

counterparts. Each PT handles three sub-resources (energy consumption, battery level, and temperature) and publishes on independent topics with a configurable message rate (ranging from 10 ms to 100 ms) and an average payload size for each sensor information of 100 bytes.

Each DT is responsible to digitalize a target IIoT device managing incoming packets to: i) process and adapt received payloads to the standard Sensor Measurement Lists (SenML) [200] data format; ii) evaluate and maintain the internal status; and iii) handle possible incoming commands and re-configuration requests sent by applications. Such elementary DTs publish their status variation (always using SenML) to a dedicated MQTT message broker, i.e., the so called Digital Twins Broker. Such a broker may be defined as a digital broker to identify its responsibility to handle only packets from DTs and applications.

According to the design patterns presented in Section 5.2, each DT was structured as a pod where the core engine container is put side by side with a *sidecar* to support communication proxy functionalities and an *ambassador* to handle a uniform and fine-grained metric collection. The Composed Digital Twin (CDT) is responsible for periodically aggregating information and statuses from other active DTs and exposing the new computed representation to applications and services interested in having an aggregate representation. The CDT directly observes the variations of connected DTs, reading data from the digital broker and publishing its new status on the same broker but on a different topic.

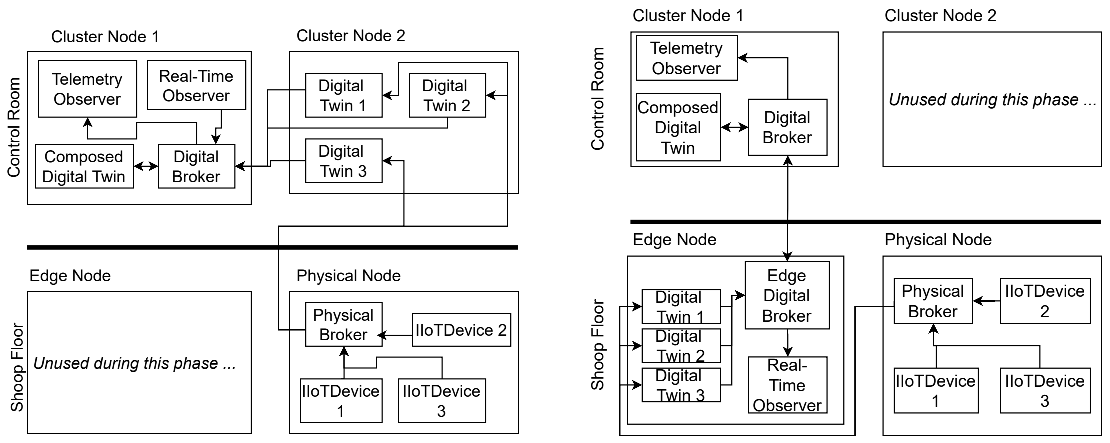
The application level is about industrial applications, i.e., those applications built on top (and by means) of the abstractions provided by the digital twin level. A telemetry observer and a real-time telemetry observer are the industrial applications part of this illustrative scenario. It is worth noting that such observers differ in the entanglement they demand. Specifically, the real-time telemetry observer demands that the observed DTs are strongly entangled with their PT. This means that the information dispatched among PTs and DTs must flow upward and downward as close to real-time as possible.

Context Variation The context variation phase is the result of a drop in network performance that slows the information circulation between DTs executed in the control room and IIoT devices active on the shop floor. As a result, DTs can no longer guarantee a strong entanglement with their PT. Note that the acceptable misalignment between DT and PT states is strongly associated with the nature of applications and services, and different tolerance levels may coexist in the same deployment. In this setup, only the real-time observer is affected by a misalignment between PTs and DTs, while the telemetry observer is used only for reporting purposes.

Thanks to the possibility of effectively monitoring every aspect of the involved entities (e.g., through ambassadors in each DT adapting and collecting metrics), we may have multiple decision points able to detect the context variation and react to it by adapting the deployment to restore the target working conditions. In the conducted experiments, this responsibility was delegated to the control room, since it has global awareness of the deployment and thus can determine how and when it should take management actions.

Deployment Adaptation The context variation phase triggers the deployment adaptation phase, which ends in a steady state. The objective of the deployment adaptation phase is to properly react to meet target conditions. In the context of these experiments, the objective is to dynamically re-configure DTs and applications to meet the demanded level of entanglement through a migration of target components directly on edge nodes in the shop floor.

This adaptation requires to: i) deploy a new MQTT digital broker at the edge and configure it to work in bridged mode with the other one already running in the control room to automatically synchronize target topics; ii) migrate DTs and the real-time telemetry observer on the edge to be physically close to the IIoT devices;



(a) Initial deployment. (b) Adaptive deployment.

Figure 7.7: Traffic flows at the digital and application level.

and iii) re-configure DTs and the observer to work in the new environment with the correct brokers.

The performance results detail i) the overhead introduced by the technology stack used to enable an entanglement-aware DT ecosystem and ii) how the whole system behaves while adapting to triggering events. The discussion about the performance results revolves around the steady and transitory states the ecosyste goes through while performing the phases described. In particular, we measured both resource (i.e., CPU and memory) and network consumption. We extracted the performance results from Prometheus with a per-pod granularity.

Steady State The steady state (i.e., a stable configuration over a period) occurs twice: throughout the initial deployment phase (up to the context variation phase) and in the adaptive deployment phase once the new configuration occurs. Figure 7.7 details the pods for the digital and application levels during the steady states of the initial (left side) and adaptive (right side) deployment.

Table 7.2 shows the average resource consumption in a steady state of the following components: Kubernetes, Istio, monitoring addons, and DTs. Kubernetes consumed the majority of resources overall. Specifically, it took 265.75 milliCPU, 2.02 GB (memory), 450.23 KB (traffic in), and 516.06 KB (traffic out). Although the highest in this comparison, the resources allocated for Kubernetes are still

Table 7.2: Averaged performance metrics collected at steady state

Metric	**Unit**	**Component**	**AVG**	**STD**
CPU	milliCPU	Kubernetes	265.75	8.05
CPU	milliCPU	Istio	12.00	0.78
CPU	milliCPU	Monitoring Addons	4.51	0.78
CPU	milliCPU	Digital Twins	43.31	4.38
Memory	GB	Kubernetes	2.02	0.01
Memory	GB	Istio	0.13	0.01
Memory	GB	Monitoring Addons	0.23	0.001
Memory	GB	Digital Twins	0.48	0.003
Traffic In	KB	Kubernetes	450.23	18.01
Traffic In	KB	Istio	2.62	0.44
Traffic In	KB	Monitoring Addons	70.67	7.40
Traffic In	KB	Digital Twins	4.74	1.56
Traffic Out	KB	Kubernetes	516.06	19.31
Traffic Out	KB	Istio	27.47	4.52
Traffic Out	KB	Monitoring Addons	7.02	0.82
Traffic Out	KB	Digital Twins	15.26	0.75

minimal. This makes Kubernetes suitable for typical devices within industrial environments. In addition, note that there are also Kubernetes distributions designed explicitly for resource-constrained scenarios. Such distributions may represent a reasonable option for those environments that cannot afford the overhead introduced by vanilla Kubernetes or need to support specific use cases (e.g., semi-autonomous edge nodes).

An important outcome of the above performance results is that the MDT implementation is extremely frugal. It is worth noting that the item "Digital Twins" in Table 7.2 regards the DTs themselves and also sidecars and ambassadors (deployed as containers within the same pod). In particular, the DTs altogether consumed 43.31 milliCPU, 0.48 GB (memory), 4.74 KB (traffic in), and 15.26 KB (traffic out). The overhead introduced by sidecars and ambassadors is negligible. This notable result fosters the use of design patterns whose benefits go far beyond their costs.

Transitory State A transitory state occurs while moving from one configuration to another, and its analysis allows us to quantify the overhead of a given transition.

Table 7.3: Average execution time for involved migration steps

Id	Action	Entity	Location	Exec. [ms]
1	CREATE	Edge Digital Broker	on Edge	7.34
2	CREATE	Digital Twin 1	on Edge	8.19
3	CREATE	Digital Twin 2	on Edge	8.13
4	CREATE	Digital Twin 3	on Edge	8.28
5	CREATE	Real-Time Observer	on Edge	7.36
6	DELETE	Digital Twin 1	from Control Room	3.34
7	DELETE	Digital Twin 2	from Control Room	3.23
8	DELETE	Digital Twin 3	from Control Room	3.20
9	DELETE	Real-Time Observer	from Control Room	2.40

Table 7.4: Average execution time for involved rollback steps

Id	Action	Entity	Location	Exec. [ms]
1	CREATE	Digital Twin 1	on Control Room	8.19
2	CREATE	Digital Twin 2	on Control Room	8.18
3	CREATE	Digital Twin 3	on Control Room	8.32
4	CREATE	Real-Time Observer	on Control Room	7.31
5	DELETE	Digital Twin 1	from Edge	3.42
6	DELETE	Digital Twin 2	from Edge	3.22
7	DELETE	Digital Twin 3	from Edge	3.28
8	DELETE	Real-Time Observer	from Edge	2.39
9	DELETE	Edge Digital Broker	from Edge	2.46

Typically, a transition is triggered by a context variation, which forces the system to move towards a new configuration. The transition is from the initial deployment to the adaptive deployment. Such a transition occurs because the configuration in the initial deployment phase no longer meets the entanglement demanded by the real-time telemetry observer. The proposed ecosystem can deal with context variation, moving to a new configuration, i.e., the adaptive deployment.

For completeness, we experimentally assessed both the transition from initial deployment (Figure 7.7(a)) to adaptive deployment (Figure 7.7(b)) and vice versa to roll back to the initial configuration of the deployment. Table 7.3 itemizes the steps to move from the initial deployment to the adaptive deployment and Table 7.4 presents the opposite. It is worth mentioning that the container images are pre-pulled on the nodes. Therefore, a CREATE step does not require downloading the related container image from a repository. Also, we executed the steps sequentially,

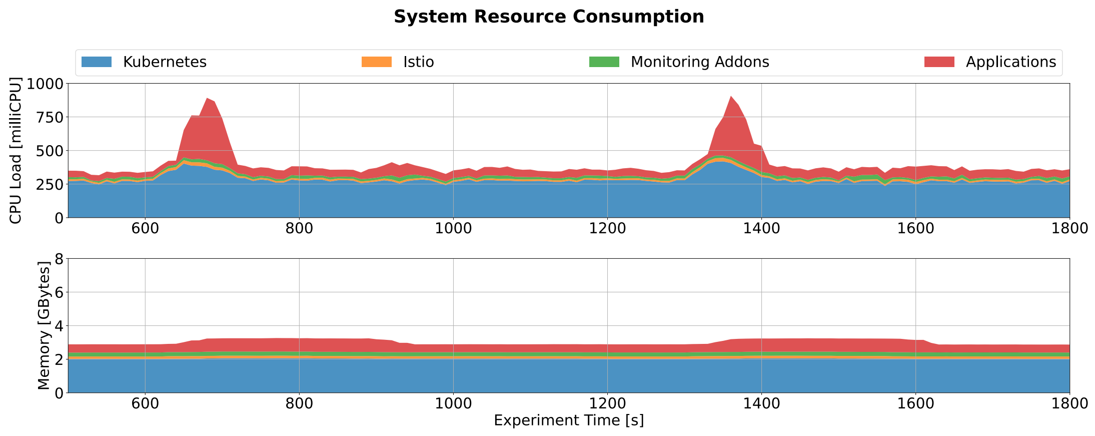


Figure 7.8: CPU and memory consumption overhead (migration and rollback).

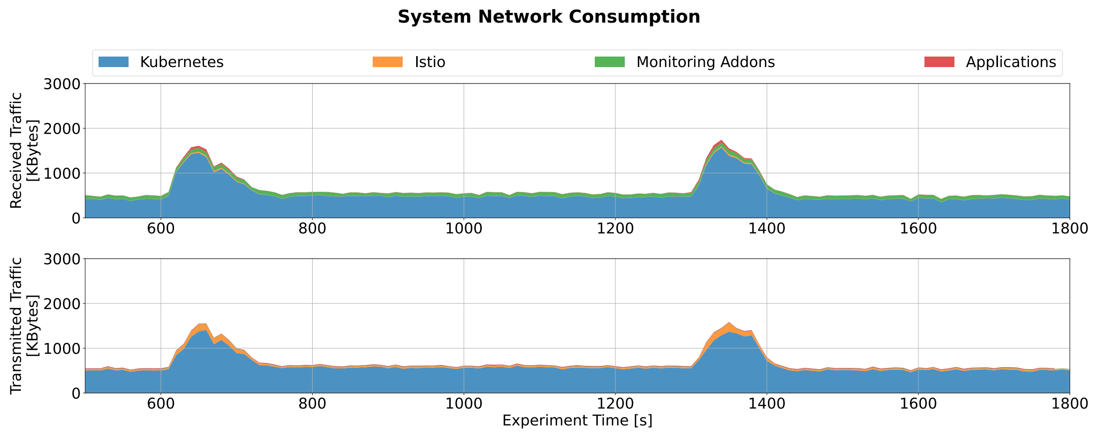
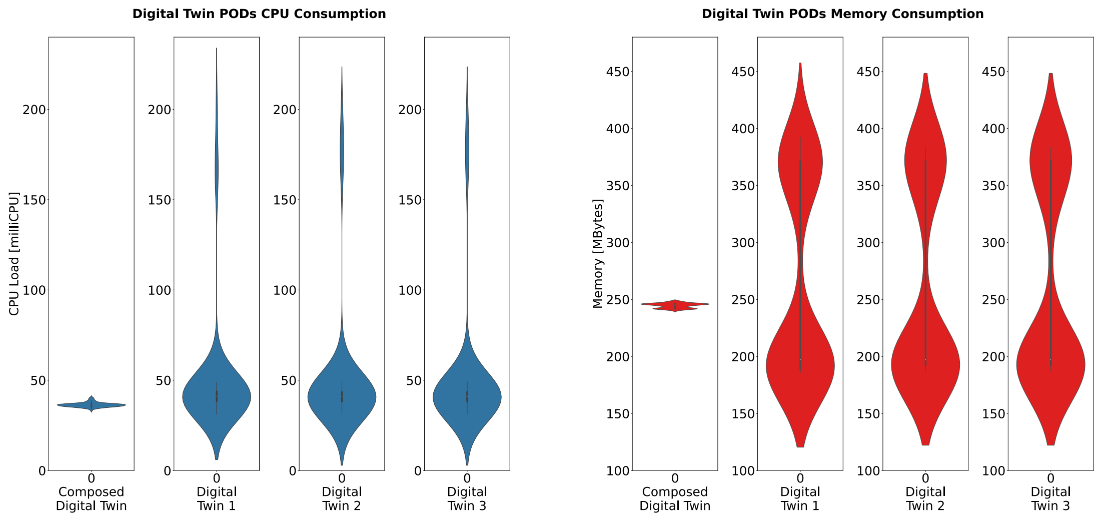


Figure 7.9: Received and transmitted KB (migration and rollback).

which means that the total time of a given transition is the sum of every single step. Note that some steps may be executed concurrently to speed up the transition. As a result (without parallel step execution), on average the overall migration time was around 55 s and the rollback procedure required about 50 s. Such time intervals include the reported action steps, the execution time required by the container to start internal modules, connect to brokers, and start processing incoming data.

Figures 7.8 and 7.9 depict the resource consumption during the above-mentioned transitory states. The first peak regards the transition from the initial deployment to the adaptive deployment, whereas the second one is the rollback. The peak went over a steady-state resource consumption of around 500 milliCPU, 360 MB



(a) CPU Consumption [milliCPU]. (b) Memory Consumption [MB]

Figure 7.10: DT CPU and memory overhead (migration and rollback).

(memory), 1600 KB (traffic in), and 1800 KB (traffic out). Graphs in Figure 7.10 report instead the distribution of CPU and memory consumption of the DT pods considering both the migration and the rollback procedures. As expected, reported values confirm the trends illustrated in the previous timeline analysis, i.e., the overall limited resource consumption of DT pods and their small variation during the transitions. The CDT kept the same value distribution since it was not directly involved in the migration procedures while DTs increased CPU and memory load only during the transitory period. In both analysis, it is important to stress that the memory occupied by removed containers was released by the virtualization system only after a specific time period (around 5 minutes). For this reason, in Figure 7.8 the occupied memory does not decrease immediately after the transition peak and in Figure 7.10 there are two main density areas associated to the memory (which also takes into account the allocation of removed pods).

7.3.3 Entanglement-Aware Digital Twin Orchestration

We designed an experiment comprising three phases to demonstrate the effectiveness of the entanglement-aware middleware in maintaining the desired quality of entanglement over time. This experiment was carried out in a cloud-to-edge con-

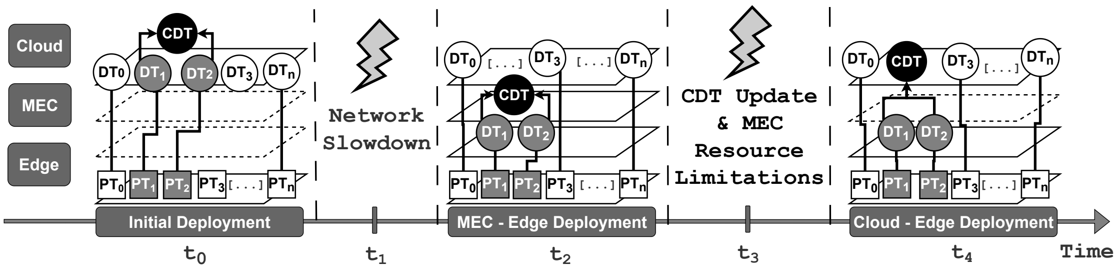


Figure 7.11: Cyber-physical application deployment over time.

tinuum scenario. The network slowdown phase introduced a bottleneck between the physical broker (where PTs publish their status updates) and the two DTs running in the cloud. We used Chaos Mesh to set 750 ms of latency, an equivalent jitter, and a correlation of 25% between consecutive packets to simulate a real-world event. The CDT reconfiguration phase simulated a re-configuration of the CDT replacing the internal model with an alternative one requiring more computational resources (we forced the DT to calculate the first 100000 prime numbers while performing state transitions). Lastly, the baseline phase did not introduce any effect to undermine the quality of entanglement. The experiment, lasting 25 minutes overall, consisted of the previous phases executed sequentially, with the baseline phase occurring before and after any of the other phases.

Figures 7.11, 7.12, and 7.13 show the evolution and performance of the cyber-physical application over time. More specifically, Figure 7.11 shows the deployment location, Figure 7.12 shows the ODTE measure, the DT life cycle state, and CPU consumption, and Figure 7.13 shows the amount of network traffic generated within the cloud-to-edge continuum (i.e., edge on-premises, MEC, and cloud).

All DTs were initially deployed in the cloud as specified in the application configuration. The first 5 minutes represent the baseline phase. As Figure 7.12 shows, both ODTE and life cycle state values remained stable at 1.0 and 5 (meaning Entangled), respectively (see Figure 6.3 for more details about the life cycle).

The second five minutes represent the network slowdown phase. In this case, the ODTE measure fell well below the 0.9 thresholds for each deployed DT, thus making the DTs switch life cycle state to 4 (Disentangled). This, in turn, triggered the middleware, which enforced a different deployment. Specifically, the middleware migrated the CDT to the MEC and the two DTs to the edge, succeeding in

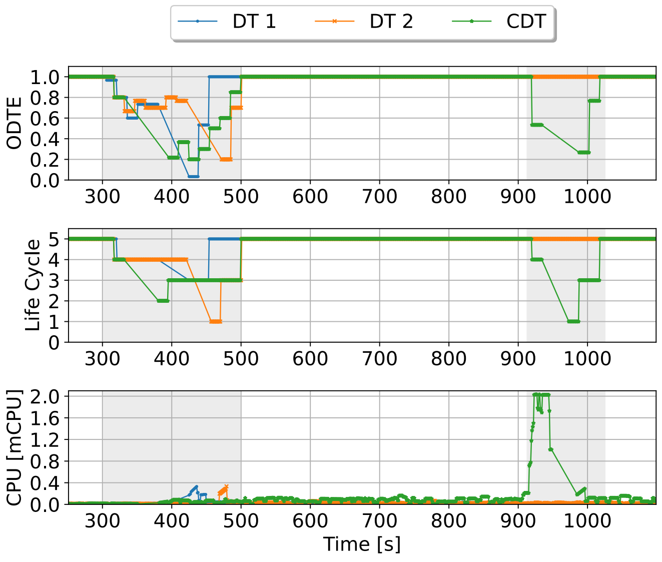


Figure 7.12: ODTE values, life cycle states, and CPU consumption.

making them Entangled again. Note that this would not be possible without the successful implementation of cloud-to-edge mobility, entanglement awareness, life cycle, and declarative application description (see Chapter 3). In fact, the DTs accurately quantified the quality of entanglement over time. Because the network slowdown phase caused a violation of the cyber-physical application requirements (i.e., ODTE fell below the 0.9 thresholds), a state transition occurred—from Entangled to Disentangled. This was promptly recognized by the middleware, which migrated the DTs along the cloud-to-edge continuum, physically closer to their PTs. The container migration required minimal networking resources as shown in Figure 7.13.

The third five minutes represent the baseline phase again. Since no effects were injected, the quality of entanglement remained stable at 1.0 for all DTs.

The fourth five minute represent the CDT reconfiguration phase in which we simulated an update of the CDT model to a version more CPU-hungry, thus forcing the migration of the CDT from the MEC to the cloud, which is usually richer in resources. As shown in Figure 7.12, the CDT reconfiguration phase caused a peak in CPU consumption, saturating the resources available for the CDT, which, in turn, impacted the time needed for updating the DT state, thus causing a drop in

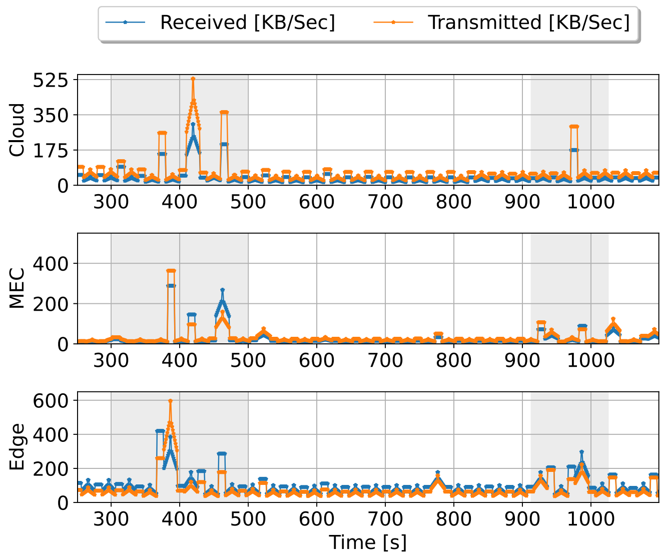


Figure 7.13: Received and transmitted network traffic along the continuum.

the ODTE measure. As soon as the DT life cycle state moved to 4 (Disentangled, see the second gray area in Figure 7.12, the container was migrated back to the cloud where the CDT could find enough resources to run a CPU-intensive model. In this case, having a variable load resilience ecosystem (see Chapter 3) revealed to be fundamental.

Finally, another baseline phase occurred. As Figure 7.12 shows, the entanglement remained unchanged, leading to no changes in the deployment strategies until the end of the experiment.

Lastly, Figure 7.14 elaborates on Entanglement Recovery Time—a measure of the time required by the middleware to restore the desired cyber-physical entanglement—over ten experiment runs. It is worth noting that the Entanglement Recovery Time depends on several configuration factors, such as how frequently Kubernetes monitors the cluster, how frequently Prometheus scrapes metrics, how frequently the Orchestrator queries Prometheus to get the current DT states, etc. Depending on the use case, these factors may be fine-tuned to make the middleware (and the overall system in general) more, or less, responsive. The Entanglement Recovery Time is noticeably longer for the CDT due to its complex structure comprising two underlying DTs. In fact, when one of the two DTs gets Disentangled,

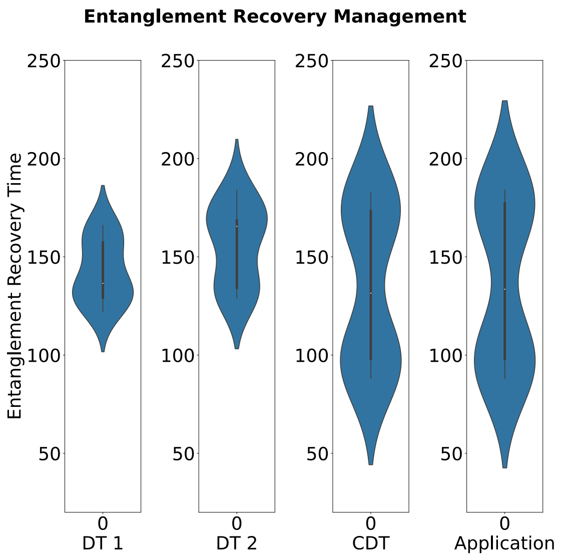


Figure 7.14: Entanglement recovery time.

the CDT follows, and the CDT becomes Entangled again if and only if both the underlying DTs gets Entangled.

7.4 Chapter Summary

This chapter discussed the implementation of the proposed entanglement-aware DT ecosystem and demonstrated its effectiveness through experimentation.

Section 7.1 provided insights on the realized microservices and serverless implementations of the entanglement-aware DT ecosystem. In the microservices setup, the entanglement-aware DTs were implemented using the WLDT library. The middleware was developed using open source technologies such as Etcd, Kubernetes, and Prometheus. Its primary objective is to maintain the quality of entanglement within the cyber-physical application constraints. The serverless implementation, on the other hand, relied on the serverless framework provided by Microsoft Azure. In this context, DTs were Function Apps.

Section 7.2 elaborated on the testbed used for experimentation. It was automatically configured in a reproducible manner using Ansible. The project is publicly available on GitHub [199].

Section 7.3 discussed the experiments and their results. First, Section 7.3.1 not only demonstrated that ODTE is effective in measuring the quality of entanglement in IIoT scenarios of practical interest, but also the advantages and drawbacks of the entanglement-aware DTs implemented as microservices and serverless functions. Then, Section 7.3.2 quantified the overhead of the proposed entanglement-aware DT ecosystem, detailing the impact of the entire technology stack. Lastly, Section 7.3.3 demonstrated the middleware's orchestration capabilities along the cloud-to-edge continuum in enforcing the desired quality of entanglement of a cyber-physical application in spite of failures.

Chapter 8

Conclusion and Future Work

This work proposed a vision of DTs that revolves around the concept of entanglement. According to this vision, a DT is a virtual entity entangled with an object, whether tangible or intangible, of which the DT provides a (augmented) representation in the virtual space, thus decoupling the object from the observer in space and time.

Given the limitations of existing platforms and metrics, this work engineered, implemented, and experimentally evaluated an entanglement-aware DT ecosystem. This involved:

- A metric to measure the entanglement. In this regard, the original contribution was ODTE—a concise yet expressive metric to measure entanglement. The effectiveness of the ODTE metric was evaluated experimentally in scenarios of practical interest.

- Entanglement-aware DTs. This work originally discussed how to apply software design patterns for building DTs as microservices and a methodology for building DTs with serverless functions. The proposed architecture for entanglement-aware DTs is event-driven in nature, making it suitable regardless of the specific implementation strategy, be it microservices or serverless. The MDT implementation was extremely frugal in terms of resource consumption, with negligible overhead introduced by sidecars and ambassadors. This notable result encourages the use of design patterns whose benefits far outweigh their costs. On the other hand, the SDT implementation provided lower performance (i.e., worse ODTE measures) in exchange for more cost-effective and scalable deployments.

- Entanglement-aware middleware. The resource overhead for the technology stack required by the entanglement-aware middleware was fairly limited. The middleware orchestration capabilities were demonstrated experimentally in a cloud-to-edge continuum scenario. The implemented entanglement-aware middleware successfully enforced the desired quality of entanglement of a cyber-physical application, even in the presence of failures.

Future work may extend this research in several directions. An in-depth assessment of the scalability and performance limits of the proposed ecosystem would be crucial. This assessment could unravel the challenges arising from the exponential growth in the number of DTs, exploring optimization strategies and potential bottlenecks in the system performance. In this regard, further investigation into serverless solutions at the edge on-premises could provide insights into their suitability to accommodate demanding entanglement requirements while still benefiting from the serverless approach. This exploration aims to leverage the advantages of serverless computing in a localized setting, offering a nuanced perspective on the integration of edge computing and serverless solutions within the entanglement-aware DT ecosystem.

Another important avenue for future research would involve a comprehensive evaluation of the ODTE metric while measuring the entanglement within ensembles or hierarchies of DTs. This exploration could provide a better understanding of how entanglement dynamics propagate in complex systems. Not only would this further validate the scalability of the ODTE metric from individual DTs to a web of DTs, but also its sensitivity in dealing with the propagation of subtle entanglement changes across the hierarchy.

Lastly, an exploration of the concept of trustworthiness in the context of DTs emerges as a crucial area for future investigation. This would require understanding how users perceive and establish trust in the information provided and consumed by DTs, especially in mission-critical cyber-physical applications. In this regard, a potential approach towards trustworthiness could be the following. On one hand, DTs should shoulder the responsibility of verifying the trustworthiness of the environment where they are deployed. On the other hand, the orchestration middleware should ensure the verification of the deployed DTs for trustworthiness. Therefore, trustworthiness would be conceived as an ongoing process where the involved parties continuously verify each other.

In summary, future research directions would encompass a multifaceted exploration of scalability, performance, entanglement, and trustworthiness within the realm of DTs.

References

[1] F. Maggi and M. Pogliani, "Attacks on smart manufacturing systems," Trend Micro Research: Shibuya, Japan, Tech. Rep., 2017.

[2] T. J. Williams, "The purdue enterprise reference architecture," *Computers in industry*, vol. 24, no. 2-3, pp. 141–158, 1994.

[3] M. Hankel and B. Rexroth, "The reference architectural model industrie 4.0 (rami 4.0)," *Zvei*, vol. 2, no. 2, pp. 4–9, 2015.

[4] S.-W. Lin, B. Miller, J. Durand, *et al.*, "Industrial internet reference architecture," *Industrial Internet Consortium (IIC), Tech. Rep*, 2015.

[5] A. Corradi, L. Foschini, C. Giannelli, *et al.*, "Smart appliances and rami 4.0: Management and servitization of ice cream machines," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1007–1016, 2019. DOI: 10.1109/TII.2018.2867643.

[6] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2019. DOI: 10.1109/TII.2018.2873186.

[7] B. R. Barricelli, E. Casiraghi, and D. Fogli, "A survey on digital twin: Definitions, characteristics, applications, and design implications," *IEEE Access*, vol. 7, pp. 167 653–167 671, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2953499.

[8] A. Ricci, A. Croatti, and S. Montagna, "Pervasive and connected digital twins—a vision for digital health," *IEEE Internet Computing*, vol. 26, no. 5, pp. 26–32, 2022. DOI: 10.1109/MIC.2021.3052039.

[9] P. Bellavista, C. Giannelli, M. Mamei, M. Mendula, and M. Picone, "Digital twin oriented architecture for secure and qos aware intelligent communications in industrial environments," *Pervasive and Mobile Computing*, vol. 85, p. 101 646, 2022, ISSN: 1574-1192. DOI: `10.1016/j.pmcj.2022.101646`.

[10] S. Malakuti and S. Grüner, "Architectural aspects of digital twins in iiot systems," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ser. ECSA '18, Madrid, Spain: Association for Computing Machinery, 2018, ISBN: 9781450364836. DOI: `10.1145/3241403.3241417`.

[11] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018, 16th IFAC Symp. on Information Control Problems in Manufacturing 2018, ISSN: 2405-8963. DOI: `10.1016/j.ifacol.2018.08.474`.

[12] R. Wagner, B. Schleich, B. Haefner, A. Kuhnle, S. Wartzack, and G. Lanza, "Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products," *Procedia CIRP*, vol. 84, pp. 88–93, 2019, 29th CIRP Design Conference 2019, 08-10 May 2019, Póvoa de Varzim, Portgal, ISSN: 2212-8271. DOI: `10.1016/j.procir.2019.04.219`.

[13] K. Josifovska, E. Yigitbas, and G. Engels, "Reference framework for digital twins within cyber-physical systems," in *2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, 2019, pp. 25–31. DOI: `10.1109/SEsCPS.2019.00012`.

[14] K. M. Alam and A. El Saddik, "C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE Access*, vol. 5, pp. 2050–2062, 2017. DOI: `10.1109/ACCESS.2017.2657006`.

[15] E. Tantik and R. Anderl, "Potentials of the asset administration shell of industrie 4.0 for service-oriented business models," *Procedia CIRP*, vol. 64, pp. 363–368, 2017, 9th CIRP IPSS Conference: Circular Perspectives on PSS, ISSN: 2212-8271. DOI: `10.1016/j.procir.2017.03.009`.

[16] Y. Fang, C. Peng, P. Lou, Z. Zhou, J. Hu, and J. Yan, "Digital-twin-based job shop scheduling toward smart manufacturing," *IEEE transactions on industrial informatics*, vol. 15, no. 12, pp. 6425–6435, 2019.

[17] Z. Zhou, Z. Jia, H. Liao, *et al.*, "Secure and latency-aware digital twin assisted resource scheduling for 5g edge computing-empowered distribution grids," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 7, pp. 4933–4943, 2022. DOI: 10.1109/TII.2021.3137349.

[18] C. Hu, W. Fan, E. Zeng, *et al.*, "Digital twin-assisted real-time traffic data prediction method for 5g-enabled internet of vehicles," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 4, pp. 2811–2819, 2022. DOI: 10.1109/TII.2021.3083596.

[19] A. Castellani, S. Schmitt, and S. Squartini, "Real-world anomaly detection by using digital twin systems and weakly supervised learning," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 7, pp. 4733–4742, 2021. DOI: 10.1109/TII.2020.3019788.

[20] Z. Lv, J. Guo, and H. Lv, "Safety poka yoke in zero-defect manufacturing based on digital twins," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2022. DOI: 10.1109/TII.2021.3139897.

[21] H. V. Dang, M. Tatipamula, and H. X. Nguyen, "Cloud-based digital twinning for structural health monitoring using deep learning," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 6, pp. 3820–3830, 2022. DOI: 10.1109/TII.2021.3115119.

[22] "IEC 62443: Industrial network and system security," International Electrotechnical Commission, Tech. Rep.

[23] N. Suri, E. Benvegnù, M. Tortonesi, C. Stefanelli, J. Kovach, and J. Hanna, "Communications middleware for tactical environments: Observations, experiences, and lessons learned," *IEEE Communications Magazine*, vol. 47, no. 10, pp. 56–63, 2009.

[24] M. R. Brannsten, F. T. Johnsen, T. H. Bloebaum, and K. Lund, "Toward federated mission networking in the tactical domain," *IEEE Communications Magazine*, vol. 53, no. 10, pp. 52–58, 2015. DOI: `10.1109/MCOM.2015.7295463`.

[25] J. Nobre, D. Rosario, C. Both, E. Cerqueira, and M. Gerla, "Toward software-defined battlefield networking," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 152–157, 2016. DOI: `10.1109/MCOM.2016.7588285`.

[26] K. Poularakis, G. Iosifidis, and L. Tassiulas, "Sdn-enabled tactical ad hoc networks: Extending programmable control to the edge," *IEEE Communications Magazine*, vol. 56, no. 7, pp. 132–138, 2018.

[27] Q. Chen, H. Wang, and N. Liu, "Integrating networking, storage, and computing for resilient battlefield networks," *IEEE Communications Magazine*, vol. 57, no. 8, pp. 56–63, 2019. DOI: `10.1109/MCOM.2019.1900186`.

[28] I. Zacarias, L. P. Gaspary, A. Kohl, R. Q. A. Fernandes, J. M. Stocchero, and E. P. de Freitas, "Combining software-defined and delay-tolerant approaches in last-mile tactical edge networking," *IEEE Communications Magazine*, vol. 55, no. 10, pp. 22–29, 2017. DOI: `10.1109/MCOM.2017.1700239`.

[29] *Nato selects thales to supply its first defence cloud for the armed forces*, `https://www.businesswire.com/news/home/20210125005006/en/`, [Online; accessed 2022-02-02], Jan. 2021.

[30] A. F. Mendi, T. Erol, and D. Doğan, "Digital twin in the military field," *IEEE Internet Computing*, vol. 26, no. 5, pp. 33–40, 2022. DOI: `10.1109/MIC.2021.3055153`.

[31] E. Glaessgen and D. Stargel, "The digital twin paradigm for future nasa and us air force vehicles," in *53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA*, 2012, p. 1818.

[32] D. Lehner, J. Pfeiffer, E.-F. Tinsel, *et al.*, "Digital twin platforms: Requirements, capabilities, and future prospects," *IEEE Software*, vol. 39, no. 2, pp. 53–61, 2021.

[33] Y. Lu, X. Huang, K. Zhang, S. Maharjan, and Y. Zhang, "Communication-efficient federated learning and permissioned blockchain for digital twin edge networks," *IEEE Internet of Things Journal*, vol. 8, no. 4, pp. 2276–2288, 2020.

[34] D. Loghin, L. Ramapantulu, and Y. M. Teo, "Towards analyzing the performance of hybrid edge-cloud processing," in *2019 IEEE International Conference on Edge Computing (EDGE)*, IEEE, 2019, pp. 87–94.

[35] K. M. Alam and A. El Saddik, "C2ps: A digital twin architecture reference model for the cloud-based cyber-physical systems," *IEEE access*, vol. 5, pp. 2050–2062, 2017.

[36] M. Picone, S. Mariani, M. Mamei, F. Zambonelli, and M. Berlier, "Wip: Preliminary evaluation of digital twins on mec software architecture," in *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2021, pp. 256–259. DOI: `10.1109/WoWMoM51794.2021.00047`.

[37] R. Al-Sehrawy and B. Kumar, "Digital twins in architecture, engineering, construction and operations. a brief review and analysis," in *International Conference on Computing in Civil and Building Engineering*, Springer, 2020.

[38] B. Tekinerdogan and C. Verdouw, "Systems architecture design pattern catalog for developing digital twins," *Sensors*, vol. 20, no. 18, p. 5103, 2020.

[39] M.-H. Hung, Y.-C. Lin, H.-C. Hsiao, *et al.*, "A novel implementation framework of digital twins for intelligent manufacturing based on container technology and cloud manufacturing services," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 3, pp. 1614–1630, 2022.

[40] M. Picone, M. Mamei, and F. Zambonelli, "A flexible and modular architecture for edge digital twin: Implementation and evaluation," *ACM Trans. Internet Things*, vol. 4, no. 1, Feb. 2023, ISSN: 2691-1914. DOI: `10.1145/3573206`. [Online]. Available: `https://doi.org/10.1145/3573206`.

[41] X. Tao, K. Ota, M. Dong, H. Qi, and K. Li, "Performance guaranteed computation offloading for mobile-edge cloud computing," *IEEE Wireless Communications Letters*, vol. 6, no. 6, pp. 774–777, 2017.

[42] T. Liu, L. Tang, W. Wang, Q. Chen, and X. Zeng, "Digital-twin-assisted task offloading based on edge collaboration in the digital twin edge network," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1427–1444, 2021.

[43] T. Do-Duy, D. Van Huynh, O. A. Dobre, B. Canberk, and T. Q. Duong, "Digital twin-aided intelligent offloading with edge selection in mobile edge computing," *IEEE Wireless Communications Letters*, vol. 11, no. 4, pp. 806–810, 2022.

[44] R. Minerva, G. M. Lee, and N. Crespi, "Digital twin in the iot context: A survey on technical features, scenarios, and architectural models," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1785–1824, 2020. DOI: 10.1109/JPROC.2020.2998530.

[45] A. Hyre, G. Harris, J. Osho, M. Pantelidakis, K. Mykoniatis, and J. Liu, "Digital twins: Representation, replication, reality, and relational (4rs)," *Manufacturing Letters*, vol. 31, pp. 20–23, 2022.

[46] P. Almasan, M. Ferriol-Galmés, J. Paillisse, *et al.*, "Network digital twin: Context, enabling technologies, and opportunities," *IEEE Communications Magazine*, vol. 60, no. 11, pp. 22–27, 2022.

[47] P. Bellavista, M. Fogli, C. Giannelli, and C. Stefanelli, "Application-aware network traffic management in mec-integrated industrial environments," *Future Internet*, vol. 15, no. 2, 2023, ISSN: 1999-5903. DOI: 10.3390/fi15020042.

[48] M. Vaezi, K. Noroozi, T. D. Todd, *et al.*, "Digital twins from a networking perspective," *IEEE Internet of Things Journal*, vol. 9, no. 23, pp. 23 525–23 544, 2022.

[49] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Requirements and design patterns for adaptive, autonomous, and context-aware digital twins in industry 4.0 digital factories," *Computers in Industry*, vol. 149, p. 103 918, 2023.

[50] A. Ricci, A. Croatti, S. Mariani, S. Montagna, and M. Picone, "Web of digital twins," *ACM Trans. Internet Technol.*, vol. 22, no. 4, Nov. 2022, ISSN: 1533-5399. DOI: 10.1145/3507909. [Online]. Available: https://doi.org/10.1145/3507909.

[51] H. Zhang, Q. Qi, W. Ji, and F. Tao, "An update method for digital twin multi-dimension models," *Robotics and Computer-Integrated Manufacturing*, vol. 80, p. 102 481, 2023.

[52] L. Hui, M. Wang, L. Zhang, L. Lu, and Y. Cui, "Digital twin for networking: A data-driven performance modeling perspective," *IEEE Network*, 2022.

[53] J. Pfeiffer, D. Lehner, A. Wortmann, and M. Wimmer, "Towards a product line architecture for digital twins," in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, IEEE, 2023, pp. 187–190.

[54] A. Kanak, N. Ugur, and S. Ergun, "A visionary model on blockchain-based accountability for secure and collaborative digital twin environments," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, IEEE, 2019, pp. 3512–3517.

[55] D. Lee, S. H. Lee, N. Masoud, M. Krishnan, and V. C. Li, "Integrated digital twin and blockchain framework to support accountable information sharing in construction projects," *Automation in construction*, vol. 127, p. 103 688, 2021.

[56] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[57] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[58] N. McKeown, T. Anderson, H. Balakrishnan, *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[59] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[60] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-defined networking (sdn): Layers and architecture terminology," *RFC 7426*, 2015.

[61] *Arista 7150 series*, `https://www.arista.com/en/products/7150-series`, [Online; accessed 20-July-2021].

[62] *Centec v350 series*, `https://www.centecnetworks.com/solution/9`, [Online; accessed 20-July-2021].

[63] *Open vswitch*, `https://github.com/openvswitch/ovs`, [Online; accessed 24-March-2021].

[64] E. L. Fernandes, E. Rojas, J. Alvarez-Horcajo, *et al.*, "The road to bofuss: The basic openflow userspace software switch," *Journal of Network and Computer Applications*, p. 102 685, 2020.

[65] O. N. Foundation, "Openflow switch specification, version 1.5.0," 2014.

[66] J. Halpern, J. H. Salim, *et al.*, "Forwarding and control element separation (forces) forwarding element model," *RFC 5812*, 2010.

[67] M. Bjorklund *et al.*, "Yang-a data modeling language for the network configuration protocol (netconf)," *RFC 6020*, 2010.

[68] R. Presuhn, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Management information base (mib) for the simple network management protocol (snmp)," *RFC 3418*, 2002.

[69] A. Doria, J. H. Salim, R. Haas, *et al.*, "Forwarding and control element separation (forces) protocol specification.," *RFC 5810*, 2010.

[70] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 127–132, ISBN: 9781450321785. DOI: `10.1145/2491185.2491190`.

[71] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014, ISSN: 0146-4833. DOI: `10.1145/2602204.2602211`.

[72] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, and N. Weidenbacher, "Opflex control protocol, 2016," IETF draft, work in progress, Tech. Rep.

[73] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Network configuration protocol (netconf)," *RFC 6241*, 2011.

[74] D. Harrington, R. Presuhn, and B. Wijnen, "An architecture for describing simple network management protocol (snmp) management frameworks," *RFC 3411*, 2002.

[75] B. Pfaff and B. Davie, "The open vswitch database management protocol," *RFC 7047*, 2013.

[76] Z. Latif, K. Sharif, F. Li, M. M. Karim, S. Biswas, and Y. Wang, "A comprehensive survey of interface protocols for software defined networks," *Journal of Network and Computer Applications*, vol. 156, 2020.

[77] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "Sdni: A message exchange protocol for software defined networks (sdns) across multiple domains," Tech. Rep., 2012.

[78] *Apache zookeeper*, `https://github.com/apache/zookeeper`, [Online; accessed 19-July-2021].

[79] *Advanced message queuing protocol*, `https://www.amqp.org`, [Online; accessed 19-July-2021].

[80] J. Stribling, Y. Sovran, I. Zhang, *et al.*, "Flexible, wide-area storage for distributed systems with wheelfs," *NSDI'09*, pp. 43–58, 2009.

[81] Q. Vohra and E. Chen, "Bgp support for four-octet autonomous system (as) number space," *RFC 6793*, 2012.

[82] J. P. Vasseur, J. L. Le Roux, *et al.*, "Path computation element (pce) communication protocol (pcep)," *RFC 5440*, 2009.

[83] F. Bannour, S. Souihi, and A. Mellouk, "Distributed sdn control: Survey, taxonomy, and challenges," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[84] N. Foster, R. Harrison, M. J. Freedman, *et al.*, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[85] A. Voellmy and P. Hudak, *Nettle: Taking the sting out of programming network routers* (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)). 2011, vol. 6539 LNCS, pp. 235–249.

[86] A. Voellmy, H. Kim, and N. Feamster, "Procera: A language for high-level reactive network control," in *HotSDN'12 - Proceedings of the 1st ACM International Workshop on Hot Topics in Software Defined Networks*, 2012, pp. 43–48.

[87] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software-defined networks," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*, 2013, pp. 1–13.

[88] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013. DOI: 10.1109/MCOM.2013.6461198.

[89] N. Gude, T. Koponen, J. Pettit, *et al.*, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008, ISSN: 0146-4833. DOI: 10.1145/1384609.1384625.

[90] Z. Cai, A. L. Cox, F. Dinu, T. Ng, and J. Zheng, "The preliminary design and implementation of the maestro network control platform," Tech. Rep., 2008.

[91] Z. Cai, A. L. Cox, and T. Ng, "Maestro: A system for scalable openflow control," Tech. Rep., 2010.

[92] D. Erickson, “The beacon openflow controller,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 13–18, ISBN: 9781450321785. DOI: `10.1145/2491185.2491189`.

[93] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, “On controller performance in software-defined networks,” in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE 2012*, 2012.

[94] *Floodlight sdn openflow controller*, `https://github.com/floodlight/floodlight`, [Online; accessed 04-March-2021].

[95] *The pox network software platform*, `https://github.com/noxrepo/pox`, [Online; accessed 04-March-2021].

[96] *Ryu component-based software defined networking framework*, `https://github.com/faucetsdn/ryu`, [Online; accessed 09-March-2021].

[97] S. Li, D. Hu, W. Fang, *et al.*, “Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability,” *IEEE Network*, vol. 31, no. 2, pp. 58–66, 2017. DOI: `10.1109/MNET.2017.1600030NM`.

[98] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey, and G. Wang, “Meridian: An sdn platform for cloud network services,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 120–127, 2013. DOI: `10.1109/MCOM.2013.6461196`.

[99] S. Shin, Y. Song, T. Lee, *et al.*, “Rosemary: A robust, secure, and high-performance network operating system,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’14, Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 78–89, ISBN: 9781450329576. DOI: `10.1145/2660267.2660353`.

[100] E. A. Brewer, “Towards robust distributed systems,” ser. PODC ’00, Portland, Oregon, USA: Association for Computing Machinery, 2000.

[101] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: `10.1145/564585.564601`.

[102] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "Cap for networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 91–96. DOI: 10.1145/2491185.2491186.

[103] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 473–478, Sep. 2012, ISSN: 0146-4833. DOI: 10.1145/2377677.2377767.

[104] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *2010 Internet Network Management Workshop / Workshop on Research on Enterprise Networking, INM/WREN 2010*, 2010.

[105] T. Koponen, M. Casado, N. Gude, *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, 2010, pp. 351–364.

[106] P. Berde, M. Gerola, J. Hart, *et al.*, "Onos: Towards an open, distributed sdn os," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 1–6, ISBN: 9781450329897. DOI: 10.1145/2620728.2620744.

[107] *The opendaylight project*, `https://github.com/opendaylight`, [Online; accessed 10-March-2021].

[108] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12, Helsinki, Finland: Association for Computing Machinery, 2012, pp. 19–24, ISBN: 9781450314770. DOI: 10.1145/2342441.2342446.

[109] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *Proceedings - International Conference on Network Protocols, ICNP*, 2014, pp. 569–576.

[110] S. Jain, A. Kumar, S. Mandal, *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 3–14, ISBN: 9781450320566. DOI: 10.1145/2486001.2486019.

[111] K.-K. Yap, M. Motiwala, J. Rahe, *et al.*, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," ser. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 432–445, ISBN: 9781450346535. DOI: 10.1145/3098822.3098854.

[112] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014.

[113] *In-network computing*, `https://www.sigarch.org/in-network-computing-draft/`, [Online; accessed 2023-10-11].

[114] D. R. K. Ports and J. Nelson, "When should the network be the computer?," ser. HotOS '19, Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 209–215, ISBN: 9781450367271. DOI: 10.1145/3317550.3321439.

[115] R. Friedman and D. Sainz, "An architecture for sdn based sensor networks," in *ACM International Conference Proceeding Series*, 2017.

[116] A. S. Yuan, H. .-. Fang, and Q. Wu, "Openflow based hybrid routing in wireless sensor networks," in *IEEE ISSNIP 2014 - 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Conference Proceedings*, 2014.

[117] B. Trevizan De Oliveira, L. Batista Gabriel, and C. Borges Margi, "Tinysdn: Enabling multiple controllers for software-defined wireless sensor networks," *IEEE Latin America Transactions*, vol. 13, no. 11, pp. 3690–3696, 2015.

[118] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, "Sdn-wise: Design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks," in *Proceedings - IEEE INFOCOM*, vol. 26, 2015, pp. 513–521.

[119] H. I. Kobo, A. M. Abu-Mahfouz, and G. P. Hancke, “Fragmentation-based distributed control system for software-defined wireless sensor networks,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 901–910, 2019.

[120] H. Huang, P. Li, S. Guo, and W. Zhuang, “Software-defined wireless mesh networks: Architecture and traffic orchestration,” *IEEE Network*, vol. 29, no. 4, pp. 24–30, 2015.

[121] A. Detti, C. Pisa, S. Salsano, and N. Blefari-Melazzi, “Wireless mesh software defined networks (wmsdn),” in *International Conference on Wireless and Mobile Computing, Networking and Communications*, 2013, pp. 89–95.

[122] S. Babu, P. V. Mithun, and B. S. Manoj, “A novel framework for resource discovery and self-configuration in software defined wireless mesh networks,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 132–146, 2020.

[123] H. Elzain and Y. Wu, “Software defined wireless mesh network flat distribution control plane,” *Future Internet*, vol. 11, no. 8, 2019.

[124] P. Bellavista, A. Dolci, C. Giannelli, and D. D. Padalino Montenero, “Sdn-based traffic management middleware for spontaneous wmns,” *Journal of Network and Systems Management*, vol. 28, no. 4, pp. 1575–1609, 2020.

[125] X. Chen, T. Wu, G. Sun, and H. Yu, “Software-defined manet swarm for mobile monitoring in hydropower plants,” *IEEE Access*, vol. 7, pp. 152 243–152 257, 2019.

[126] K. Poularakis, Q. Qin, E. M. Nahum, M. Rio, and L. Tassiulas, “Flexible sdn control in tactical ad hoc networks,” *Ad Hoc Networks*, vol. 85, pp. 71–80, 2019.

[127] K. Poularakis, Q. Qin, K. M. Marcus, K. S. Chan, K. K. Leung, and L. Tassiulas, “Hybrid sdn control in mobile ad hoc networks,” in *Proceedings - 2019 IEEE International Conference on Smart Computing, SMARTCOMP 2019*, 2019, pp. 110–114.

[128] M. Abolhasan, J. Lipman, W. Ni, and B. Hagelstein, “Software-defined wireless networking: Centralized, distributed, or hybrid?” *IEEE Network*, vol. 29, no. 4, pp. 32–38, 2015.

[129] I. Ku, Y. Lu, M. Gerla, R. L. Gomes, F. Ongaro, and E. Cerqueira, “Towards software-defined vanet: Architecture and services,” in *2014 13th Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET 2014*, 2014, pp. 103–110.

[130] K. L. K. Sudheera, M. Ma, G. G. M. N. Ali, and P. H. J. Chong, “Delay efficient software defined networking based architecture for vehicular networks,” in *2016 IEEE International Conference on Communication Systems, ICCS 2016*, 2017.

[131] Z. He, J. Cao, and X. Liu, “Sdvn: Enabling rapid network innovation for heterogeneous vehicular communication,” *IEEE Network*, vol. 30, no. 4, pp. 10–15, 2016.

[132] J. Liu, J. Wan, B. Zeng, Q. Wang, H. Song, and M. Qiu, “A scalable and quick-response software defined vehicular network assisted by mobile edge computing,” *IEEE Communications Magazine*, vol. 55, no. 7, pp. 94–100, 2017.

[133] S. Correia, A. Boukerche, and R. I. Meneguette, “An architecture for hierarchical software-defined vehicular networks,” *IEEE Communications Magazine*, vol. 55, no. 7, pp. 80–86, 2017.

[134] G. Secinti, P. B. Darian, B. Canberk, and K. R. Chowdhury, “Sdns in the sky: Robust end-to-end connectivity for aerial vehicular networks,” *IEEE Communications Magazine*, vol. 56, no. 1, pp. 16–21, 2018.

[135] F. Xiong, A. Li, H. Wang, and L. Tang, “An sdn-mqtt based communication system for battlefield uav swarms,” *IEEE Communications Magazine*, vol. 57, no. 8, pp. 41–47, 2019.

[136] N. Hu, Z. Tian, Y. Sun, *et al.*, “Building agile and resilient uav networks based on sdn and blockchain,” *IEEE Network*, vol. 35, no. 1, pp. 57–63, 2021.

[137] W. Qi, Q. Song, X. Kong, and L. Guo, "A traffic-differentiated routing algorithm in flying ad hoc sensor networks with sdn cluster controllers," *Journal of the Franklin Institute*, vol. 356, no. 2, pp. 766–790, 2019.

[138] K. Chen, S. Zhao, N. Lv, W. Gao, X. Wang, and X. Zou, "Segment routing based traffic scheduling for the software-defined airborne backbone network," *IEEE Access*, vol. 7, pp. 106 162–106 178, 2019.

[139] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[140] A. S. Tanenbaum and H. Bos, *Modern operating systems.* Pearson, 2015.

[141] D. C. Van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Towards a flexible, lightweight virtualization alternative," in *Proceedings of the 7th ACM International Systems and Storage Conference, SYSTOR 2014*, 2014, pp. 165–174.

[142] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[143] *Linux programmer's manual - lxc(7)*, `http://man7.org/linux/man-pages/man7/lxc.7.html`, [Online; accessed 29-September-2023].

[144] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, 2015, pp. 386–393.

[145] *Arch linux - chroot*, `https://wiki.archlinux.org/index.php/Chroot`, [Online; accessed 29-September-2023].

[146] *Linux programmer's manual - chroot(2)*, `http://man7.org/linux/man-pages/man2/chroot.2.html`, [Online; accessed 29-September-2023].

[147] *Linux programmer's manual - namespaces(7)*, `http://man7.org/linux/man-pages/man7/namespaces.7.html`, [Online; accessed 29-September-2023].

[148] *Linux programmer's manual - network_namespaces(7)*, http://man7.org/linux/man-pages/man7/network_namespaces.7.html, [Online; accessed 29-September-2023].

[149] *Arch linux - cgroups*, https://wiki.archlinux.org/index.php/Cgroups, [Online; accessed 29-September-2023].

[150] *Control groups - the linux kernel documentation*, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, [Online; accessed 29-September-2023].

[151] *Linux programmer's manual - cgroups(7)*, http://man7.org/linux/man-pages/man7/cgroups.7.html, [Online; accessed 29-September-2023].

[152] *Kubernetes*, https://github.com/kubernetes/kubernetes, [Online; accessed 29-September-2023].

[153] *Kubernetes - objects*, https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, [Online; accessed 29-September-2023].

[154] *Kubernetes - pods*, https://kubernetes.io/docs/concepts/workloads/pods/, [Online; accessed 29-September-2023].

[155] *Kubernetes - glossary*, https://kubernetes.io/docs/reference/glossary/?all=true, [Online; accessed 29-September-2023].

[156] *Kubernetes - labels and selectors*, https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/, [Online; accessed 29-September-2023].

[157] *Kubernetes - components*, https://kubernetes.io/docs/concepts/overview/components/, [Online; accessed 29-September-2023].

[158] *Etcd*, https://github.com/etcd-io/etcd, [Online; accessed 29-September-2023].

[159] *Kubernetes - cluster networking*, https://kubernetes.io/docs/concepts/cluster-administration/networking/, [Online; accessed 29-September-2023].

[160] M. Fogli, T. Kudla, B. Musters, *et al.*, "Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks," in *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, 2021, pp. 1–7. DOI: `10.1109/ICMCIS52405.2021.9486396`.

[161] M. Fogli, G. Pingen, T. Kudla, S. Webb, N. Suri, and H. Bastiaansen, "Towards a cots-enabled federated cloud architecture for adaptive c2 in coalition tactical operations: A performance analysis of kubernetes," in *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 225–230. DOI: `10.1109/MILCOM52596.2021.9653050`.

[162] T. Kudla, M. Fogli, S. Webb, G. Pingen, N. Suri, and H. Bastiaansen, "Quantifying the performance of cloud-oriented container orchestrators on emulated tactical networks," *IEEE Communications Magazine*, vol. 60, no. 5, pp. 74–80, 2022. DOI: `10.1109/MCOM.003.00975`.

[163] *Kubernetes (k8s): Production-grade container scheduling and management*, `https://github.com/kubernetes/kubernetes`, [Online; accessed 2022-02-02].

[164] *Lightweight kubernetes (k3s)*, `https://github.com/k3s-io/k3s`, [Online; accessed 2022-02-02].

[165] *Kubernetes native edge computing framework (kubeedge)*, `https://github.com/kubeedge/kubeedge`, [Online; accessed 2022-02-02].

[166] "Resilience engineering: Learning to embrace failure," 11, vol. 55, New York, NY, USA: Association for Computing Machinery, Nov. 2012, pp. 40–47. DOI: `10.1145/2366316.2366331`. [Online]. Available: `https://doi.oxrg/10.1145/2366316.2366331`.

[167] A. Basiri, N. Behnam, R. de Rooij, *et al.*, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016. DOI: `10.1109/MS.2016.60`.

[168] A. Basiri, L. Hochstein, N. Jones, and H. Tucker, "Automating chaos experiments in production," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 31–40. DOI: `10.1109/ICSE-SEIP.2019.00012`.

[169] M. Fogli, C. Giannelli, F. Poltronieri, C. Stefanelli, and M. Tortonesi, “Chaos engineering for resilience assessment of digital twins,” *IEEE Transactions on Industrial Informatics*, pp. 1–9, 2023. DOI: 10.1109/TII.2023.3264101.

[170] F. Siqueira and J. G. Davis, “Service computing for industry 4.0: State of the art, challenges, and research opportunities,” *ACM Comput. Surv.*, vol. 54, no. 9, Oct. 2021, ISSN: 0360-0300. DOI: 10.1145/3478680.

[171] Z. Wang, R. Gupta, K. Han, *et al.*, “Mobility digital twin: Concept, architecture, case study, and future challenges,” *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 17 452–17 467, 2022. DOI: 10.1109/JIOT.2022.3156028.

[172] L. Li, S. Aslam, A. Wileman, and S. Perinpanayagam, “Digital twin in aerospace industry: A gentle introduction,” *IEEE Access*, vol. 10, pp. 9543–9562, 2022. DOI: 10.1109/ACCESS.2021.3136458.

[173] V. Damjanovic-Behrendt and W. Behrendt, “An open source approach to the design and implementation of digital twins for smart manufacturing,” *International Journal of Computer Integrated Manufacturing*, vol. 32, pp. 366–384, 4-5 2019. DOI: 10.1080/0951192X.2019.1599436.

[174] M. Azarmipour, H. Elfaham, C. Gries, T. Kleinert, and U. Epple, “A service-based architecture for the interaction of control and mes systems in industry 4.0 environment,” in *IEEE International Conference on Industrial Informatics (INDIN)*, 2020, pp. 217–222, ISBN: 9781728149646. DOI: 10.1109/INDIN45582.2020.9442083.

[175] H. Howard and R. Mortier, “Paxos vs raft: Have we reached consensus on distributed consensus?” In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–9.

[176] G. Adzic and R. Chatley, “Serverless computing: Economic and architectural impact,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 884–889.

[177] J. Wen, Z. Chen, Y. Liu, *et al.*, "An empirical study on challenges of application development in serverless computing," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 416–428.

[178] V. Zambrano, J. Mueller-Roemer, M. Sandberg, *et al.*, "Industrial digitalization in the industry 4.0 era: Classification, reuse and authoring of digital models on digital twin platforms," *Array*, vol. 14, p. 100 176, 2022.

[179] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 445–451.

[180] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing data processing jobs with lithops," *IEEE Transactions on Cloud Computing*, 2021.

[181] R. Chard, Y. Babuji, Z. Li, *et al.*, "Funcx: A federated function serving fabric for science," in *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.

[182] P. Patros, J. Spillner, A. V. Papadopoulos, B. Varghese, O. Rana, and S. Dustdar, "Toward sustainable serverless computing," *IEEE Internet Computing*, vol. 25, no. 6, pp. 42–50, 2021.

[183] A. Golchin and R. West, "Jumpstart: Fast critical service resumption for a partitioning hypervisor in embedded systems," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2022, pp. 55–67.

[184] K. Fizza, A. Banerjee, K. Mitra, *et al.*, "Qoe in iot: A vision, survey and future directions," *Discover Internet of Things*, vol. 1, no. 1, pp. 1–14, 2021.

[185] V. M. Sachidananda, A. Khelil, and N. Suri, "Quality of information in wireless sensor networks," in *MIT International Conference on Information Quality*, 2010.

[186] D.-H. Shin, "Conceptualizing and measuring quality of experience of the internet of things: Exploring how quality is perceived by users," *Information & Management*, vol. 54, no. 8, pp. 998–1011, 2017.

[187] Y. Ikeda, S. Kouno, A. Shiozu, and K. Noritake, “A framework of scalable qoe modeling for application explosion in the internet of things,” in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, IEEE, 2016, pp. 425–429.

[188] M. Suryanegara, D. A. Prasetyo, F. Andriyanto, and N. Hayati, “A 5-step framework for measuring the quality of experience (qoe) of internet of things (iot) services,” *IEEE Access*, vol. 7, pp. 175 779–175 792, 2019.

[189] W. Robitza, A. Ahmad, P. A. Kara, *et al.*, “Challenges of future multimedia qoe monitoring for internet service providers,” *Multimedia Tools and Applications*, vol. 76, no. 21, pp. 22 243–22 266, 2017.

[190] L. Li, M. Rong, and G. Zhang, “An internet of things qoe evaluation method based on multiple linear regression analysis,” in *2015 10th International Conference on Computer Science & Education (ICCSE)*, IEEE, 2015, pp. 925–928.

[191] I. de la Torre Diez, S. G. Alonso, E. M. Cruz, and M. A. Franco, “Measuring qoe of a teleconsultation app in mental health using a pentagram model,” *Journal of medical systems*, vol. 43, no. 7, pp. 1–5, 2019.

[192] M. Picone, M. Mamei, and F. Zambonelli, “Wldt: A general purpose library to build iot digital twins,” *SoftwareX*, vol. 13, 2021.

[193] *Istio*, `https://github.com/istio/istio`, [Online; accessed 2023-10-11].

[194] *Prometheus*, `https://github.com/prometheus/prometheus`, [Online; accessed 2023-10-11].

[195] *Chaos mesh*, `https://github.com/chaos-mesh/chaos-mesh`, [Online; accessed 2023-10-11].

[196] *Ansible*, `https://github.com/ansible/ansible`, [Online; accessed 2023-10-11].

[197] *Cri-o*, `https://github.com/cri-o/cri-o`, [Online; accessed 2023-10-11].

[198] *Flannel*, `https://github.com/flannel-io/flannel/`, [Online; accessed 2023-10-11].

[199] *Kubernetes-based cluster maker*, `https://github.com/fglmtt/kubemake/`, [Online; accessed 2023-10-11].

[200] C. Jennings, Z. Shelby, J. Arkko, A. Keränen, and C. Bormann, "Sensor Measurement Lists (SenML)," Request for Comments, no. 8428, Aug. 2018.

Author's Publication List

[1] H. Bastiaansen, J. v. d. Geest, C. v. d. Broek, *et al.*, "Federated control of distributed multi-partner cloud resources for adaptive c2 in disadvantaged networks," *IEEE Communications Magazine*, vol. 58, no. 8, pp. 21–27, 2020. DOI: 10.1109/MCOM.001.2000246.

[2] M. Fogli, T. Kudla, B. Musters, *et al.*, "Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks," in *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, 2021, pp. 1–7. DOI: 10.1109/ICMCIS52405.2021.9486396.

[3] P. Bellavista, M. Fogli, L. Foschini, C. Giannelli, L. Patera, and C. Stefanelli, "Qos-enabled semantic routing for industry 4.0 based on sdn and mom integration," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–6. DOI: 10.1109/HPSR52026.2021.9481869.

[4] M. Fogli, G. Pingen, T. Kudla, S. Webb, N. Suri, and H. Bastiaansen, "Towards a cots-enabled federated cloud architecture for adaptive c2 in coalition tactical operations: A performance analysis of kubernetes," in *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 225–230. DOI: 10.1109/MILCOM52596.2021.9653050.

[5] M. Fogli, C. Giannelli, and C. Stefanelli, "Software-defined networking in wireless ad hoc scenarios: Objectives and control architectures," *Journal of Network and Computer Applications*, vol. 203, p. 103 387, 2022, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2022.103387.

[6] M. Fogli, C. Giannelli, and C. Stefanelli, "Edge-powered in-network processing for content-based message management in software-defined industrial networks," in *ICC 2022 - IEEE International Conference on Communications*, 2022, pp. 1438–1443. DOI: `10.1109/ICC45855.2022.9838863`.

[7] T. Kudla, M. Fogli, S. Webb, G. Pingen, N. Suri, and H. Bastiaansen, "Quantifying the performance of cloud-oriented container orchestrators on emulated tactical networks," *IEEE Communications Magazine*, vol. 60, no. 5, pp. 74–80, 2022. DOI: `10.1109/MCOM.003.00975`.

[8] M. Fogli, C. Giannelli, and C. Stefanelli, "Joint orchestration of content-based message management and traffic flow steering in industrial backbones," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022, pp. 325–330. DOI: `10.1109/WoWMoM54355.2022.00067`.

[9] P. Bellavista, M. Fogli, L. Foschini, C. Giannelli, L. Patera, and C. Stefanelli, "A framework for qos- enabled semantic routing in industrial networks: Overall architecture and primary protocols," in *2022 IEEE Future Networks World Forum (FNWF)*, 2022, pp. 58–63. DOI: `10.1109/FNWF55208.2022.00019`.

[10] P. Bellavista, M. Fogli, C. Giannelli, and C. Stefanelli, "Application-aware network traffic management in mec-integrated industrial environments," *Future Internet*, vol. 15, no. 2, 2023, ISSN: 1999-5903. DOI: `10.3390/fi15020042`.

[11] M. Fogli, C. Giannelli, F. Poltronieri, C. Stefanelli, and M. Tortonesi, "Chaos engineering for resilience assessment of digital twins," *IEEE Transactions on Industrial Informatics*, pp. 1–9, 2023. DOI: `10.1109/TII.2023.3264101`.

[12] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, "Requirements and design patterns for adaptive, autonomous, and context-aware digital twins in industry 4.0 digital factories," *Computers in Industry*, vol. 149, p. 103 918, 2023, ISSN: 0166-3615. DOI: `10.1016/j.compind.2023.103918`.

[13] P. Bellavista, N. Bicocchi, M. Fogli, C. Giannelli, M. Mamei, and M. Picone, “Measuring digital twin entanglement in industrial internet of things,” in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 5897–5903. DOI: 10.1109/ICC45041.2023.10278787.

[14] T. Albers, M. Fogli, E. Harmsma, E. Lazovik, and H. Bastiaansen, “A taxonomy for workload deployment orchestration in the edge-cloud continuum,” in *Service-Oriented and Cloud Computing*, G. A. Papadopoulos, F. Rademacher, and J. Soldani, Eds., Cham: Springer Nature Switzerland, 2023, pp. 239–250. DOI: 10.1007/978-3-031-46235-1_16.