Computing (2023) 105:1745–1768
https://doi.org/10.1007/s00607-023-01165-x

REGULAR PAPER

Towards high-availability cyber-physical systems using a microservice architecture

Manel Mena[1] · Javier Criado[1] · Luis Iribarne[1] · Antonio Corral[1] · Richard Chbeir[2] · Yannis Manolopoulos[3]

Received: 14 June 2022 / Accepted: 9 February 2023 / Published online: 11 March 2023

Abstract

In the past few years the use of IoT devices has grown exponentially. When it comes to working with them, we find a series of problems that are not easy to solve. On the one hand, the simple fact of communicating with those devices can be problematic since they can use different types of technologies regarding that communication. On the other hand, these types of devices usually aim to perform their function using the lowest possible energy, meaning they have certain constraints in terms of performance. Trying to work with these devices in high-availability environments becomes difficult because of those restraints. This paper introduces digital dice, a virtual representation of IoT devices and cyber-physical systems based on microservices that uses the standard established by the W3C, the Web of Things, as the underlying framework to declare its possible interactions. The article puts forward the different strategies that Digital

Javier Criado, Luis Iribarne, Antonio Corral, Richard Chbeir and Yannis Manolopoulos have contributed equally to this work.

✉ Manel Mena
manel.mena@ual.es

Javier Criado
javi.criado@ual.es

Luis Iribarne
luis.iribarne@ual.es

Antonio Corral
acorral@ual.es

Richard Chbeir
richard.chbeir@univ-pau.fr

Yannis Manolopoulos
yannis.manolopoulos@ouc.ac.cy

1 University of Almería, Almería, Spain

2 University Pau & Pays Adour, E2S UPPA, LIUPPA, Anglet, France

3 Open University of Cyprus, Nicosia, Cyprus

Dice uses to mitigate the problems raised. Furthermore, we contrast the performance of Digital Dice with using the devices directly, demonstrating its advantages in a process that requires High Availability.

Keywords IoT · CPS · Microservices · WoT · High availability

1 Introduction

The increasing use of IoT devices [1] or Cyber-Physical Systems (CPS) [2] in all kinds of domains, such as smart cities, smart buildings and Industry 4.0, among others has caused the adoption of these devices to grow unchecked. In 2020, there were more than 20 billion devices, and there is no sign that it will stop anytime soon. One of the biggest problems when working with this large quantity of devices is that we find many different types of protocols when using them, which causes many headaches when working with heterogeneous ecosystems for these types of devices.

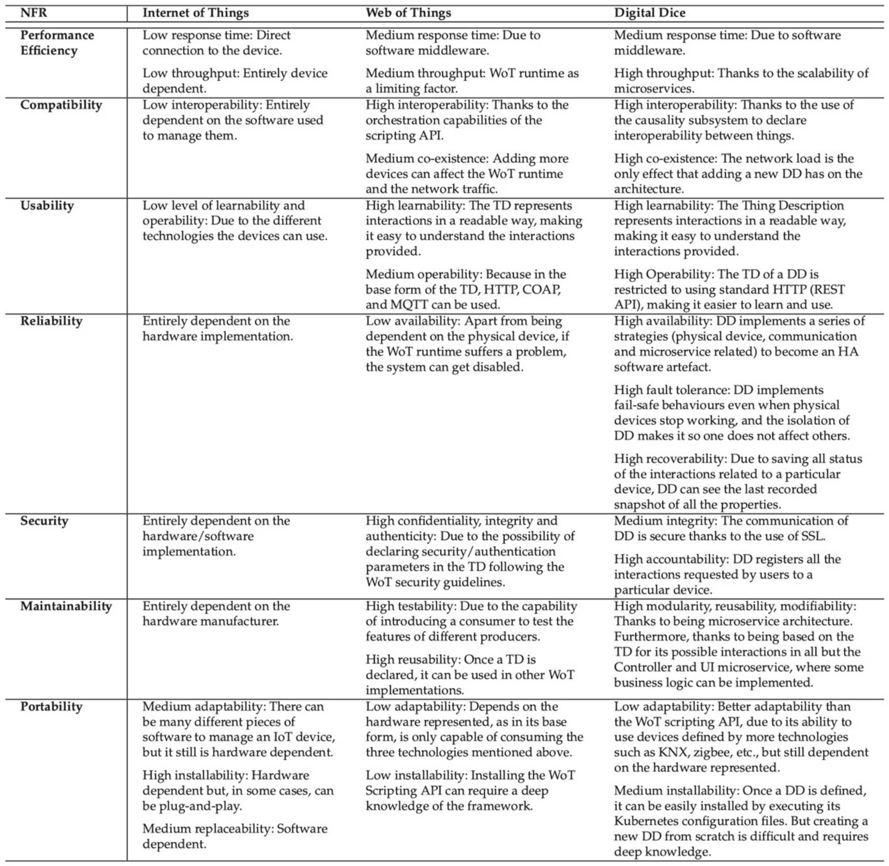
Furthermore, given the purpose of most IoT devices, where efficiency prevails, we find very noticeable performance restrictions when working with these devices. Moreover, the performance depends not only on the particular devices we are working with but, at times, the restrictions are the product of the technologies used, such as certain bus-based technologies like KNX or Modbus. These technologies have very marked restrictions on the number of clients who can be subscribed to a bus at a particular time.

Usability, performance, and availability are the three main problems we face in IoT-based architectures [3]. To solve them, we defined the concept of Digital Dice (DD) [4], the purpose of which is: (1) to create a "highly available" (HA) virtual representation of a Thing, hence the use of microservices [5], and (2) to help users and developers interact with the said Things using simple web protocols, abstracting the user from the particular underlying technologies of the Thing represented. To do so, a Digital Dice is represented by Thing Descriptions (TD) of the Web of Things (WoT).[1] The Web of Things is a set of standards and protocols developed by the W3C to build the IoT ecosystem in a flexible, scalable, and open way using web technologies as its application layer. It helps declare, discover and interact with Things in a standard way as well as monitor their state. The Thing Description is a definition language, similar to what OpenAPI is for RESTful services. Table 1 represents the efforts established in DD towards the attainment and improvement of the non-functional requirements (NFR) set out in the ISO/IEC 25010 related to the quality parameters of a software product, and compares them against an IoT device and the WoT reference implementation (WoT Scripting API).

As an architectural solution based on the WoT, Digital Dice deploys a set of microservices in a distributed containerized environment. This fact allows Digital Dice to become a highly scalable architecture, making this the first step towards attaining the HA requirement. Furthermore, the way that the microservices are deployed allows us to scale the architecture horizontally, increasing the number of instances of each

[1] W3C Web of Things (WoT) – https://www.w3.org/WoT/.

Table 1 NFR in IoT device, WoT reference implementation and Digital Dice



NFR	Internet of Things	Web of Things	Digital Dice
Performance Efficiency	Low response time: Direct connection to the device. Low throughput: Entirely device dependent.	Medium response time: Due to software middleware. Medium throughput: WoT runtime as a limiting factor.	Medium response time: Due to software middleware. High throughput: Thanks to the scalability of microservices.
Compatibility	Low interoperability: Entirely dependent on the software used to manage them.	High interoperability: Thanks to the orchestration capabilities of the scripting API. Medium co-existence: Adding more devices can affect the WoT runtime and the network traffic.	High interoperability: Thanks to the use of the causality subsystem to declare interoperability between things. High co-existence: The network load is the only effect that adding a new DD has on the architecture.
Usability	Low level of learnability and operability: Due to the different technologies the devices can use.	High learnability: The TD represents interactions in a readable way, making it easy to understand the interactions provided. Medium operability: Because in the base form of the TD, HTTP, COAP, and MQTT can be used.	High learnability: The Thing Description represents interactions in a readable way, making it easy to understand the interactions provided. High Operability: The TD of a DD is restricted to using standard HTTP (REST API), making it easier to learn and use.
Reliability	Entirely dependent on the hardware implementation.	Low availability: Apart from being dependent on the physical device, if the WoT runtime suffers a problem, the system can get disabled.	High availability: DD implements a series of strategies (physical device, communication and microservice related) to become an HA software artefact. High fault tolerance: DD implements fail-safe behaviours even when physical devices stop working, and the isolation of DD makes it so one does not affect others. High recoverability: Due to saving all status of the interactions related to a particular device, DD can see the last recorded snapshot of all the properties.
Security	Entirely dependent on the hardware/software implementation.	High confidentiality, integrity and authenticity: Due to the possibility of declaring security/authentication parameters in the TD following the WoT security guidelines.	Medium integrity: The communication of DD is secure thanks to the use of SSL. High accountability: DD registers all the interactions requested by users to a particular device.
Maintainability	Entirely dependent on the hardware manufacturer.	High testability: Due to the capability of introducing a consumer to test the features of different producers. High reusability: Once a TD is declared, it can be used in other WoT implementations.	High modularity, reusability, modifiability: Thanks to being microservice architecture. Furthermore, thanks to being based on the TD for its possible interactions in all but the Controller and UI microservice, where some business logic can be implemented.
Portability	Medium adaptability: There can be many different pieces of software to manage an IoT device, but it still is hardware dependent. High installability: Hardware dependent but, in some cases, can be plug-and-play. Medium replaceability: Software dependent.	Low adaptability: Depends on the hardware represented, as in its base form, is only capable of consuming the three technologies mentioned above. Low installability: Installing the WoT Scripting API can require a deep knowledge of the framework.	Low adaptability: Better adaptability than the WoT scripting API, due to its ability to use devices defined by more technologies such as KNX, zigbee, etc., but still dependent on the hardware represented. Medium installability: Once a DD is defined, it can be easily installed by executing its Kubernetes configuration files. But creating a new DD from scratch is difficult and requires deep knowledge.

microservice as needed. Digital Dice microservices are responsible for specific facets of the management of a thing, such as communication with the particular Thing, the control of events, the visual interface of the represented Thing, etc. Furthermore, the fact that each microservice has its own purpose facilitates maintaining them separately, improving the maintainability of the whole architecture.

Digital Dice takes its name from the concept of Digital Twin [6], as a result of both being virtual representations of physical devices or systems. However, whereas Digital Twin focuses on virtualizing devices to perform tests without influencing business processes, Digital Dice focuses on managing these devices. The word Dice arises from the different facets that are part of our microservices architecture, which we will deal with later.

In previous work, we conceptually defined the parts of Digital Dice [4]. After that, we saw how, through the so-called causality subsystem [7], Digital Dice handles interoperability between devices. Finally, we defined a way through a library called

WoTnectivity [8] to establish a common communication pattern to connect with IoT devices that use different technologies.

Extending our previous research, this paper focuses on the different strategies used for converting Digital Dice into a High Availability (or HA) alternative to more traditional Digital Twins. That aside, we compare the performance of different Digital Dice configurations with connecting directly to an IoT device. In addition, this paper describes how, through establishing the WoT as the definition layer, and also a well-defined API REST to define interactions, the use of Digital Dice can improve the operability of IoT devices. Lastly, to demonstrate the applicability of Digital Dice in a system where HA is a requisite, we show a virtual scenario, more specifically a garbage recollection system, aiming to improve energy efficiency in a smart city environment by optimizing the routes followed by the garbage trucks.

The facts mentioned above led us to pose the following research questions:

RQ1 *What does our proposal solve regarding communication with IoT devices?*

RQ2 *Are standard modeling languages expressive enough for the definition of our architecture? Is Digital Dice more expressive/operable than typical IoT devices?*

RQ3 *Can our solution improve the performance of requests sent to IoT devices?*

RQ4 *Is there any need for HA systems regarding IoT ecosystems?*

RQ5 *What do we propose to make our solution Highly Available?*

Digital Dice proposes a solution that establishes an architecture based on microservices, a REST API, and Server Sent-Events (SSE) for the management of IoT devices and cyber-physical systems applying the standards defined by the WoT. First, the article establishes a series of core concepts and fundamentals to understand how our system works, emphasizing the WoT, microservices architectures, and the concept of high availability (RQ1). Secondly, apart from using the language proposed by the WoT to define the interactions of the represented IoT devices, we include a definition language to generate different DD configurations. Accordingly, this solution considers the modeling of the system itself and the deployment configuration at the replication level, which is vital when talking about microservices (RQ2). Thirdly, we carry out a performance comparison of our solution versus a direct connection with devices and different DD configurations and check the difference in performance (RQ3). Furthermore, thanks to the example based on a smart city that we show in the article, one can see how there is a necessity for HA systems in IoT environments (RQ4). Finally, the article establishes the different strategies used to turn our solution into a high-availability solution (RQ5).

The rest of this paper is structured as follows. Sect. 2 describes the background information and the fundamentals required to understand Digital Dice. Sect. 3 offers an overview of the related works. Sect. 4 takes a closer look at the core concepts of DD. Sect. 5 shows the different strategies used by DD to become a HA system. Sect. 5.4 deals with the performance of DD in different configurations. Sect. 6 describes the example scenario. Sect. 7 goes over the answers to the proposed research questions. Finally, Sect. 8 explains the advantages of our approach and future work.

2 Fundamentals and background

IoT refers to any object, electronic device, object, animal, or person with which we can establish communication through a unique identifier as well as transfer data over the network via digital means [9]. The most critical problems we face when working with this type of things are defining its capabilities and establishing its digital representation, as there is no standard or homogeneous way to define it. The WoT proposes a language for defining the set of interactions of a Thing, the so-called Thing Description. Meanwhile, we propose a way to represent virtually different Things through a distributed system, Digital Dice. Combining the efforts of the ideas proposed by the WoT and our own, we established a solution for the management of IoT devices that offers the strength of microservice architectures and the standardizing efforts of the proposal of the WoT. In this section, we reviewed the fundamentals of Digital Dice, i.e, WoT, Microservice architectures and Highly Available Systems.

2.1 Web of Things

Web of Things (WoT) describes a set of standards to create flexible, scalable, and open Internet of Things (IoT) ecosystems using web technologies as the application layer. The concept of Digital Dice began as a Web of Things servient system, which is a software stack that implements some of the WoT building blocks. A servient can expose Things, as well as consume them. Digital Dice is what the WoT calls an intermediary servient software, a piece of software capable of doing both, consuming things and exposing them.

A Thing Description (TD) describes the metadata and interfaces of Things, which includes the set of properties, actions and events managed by that Thing. Moreover, the properties, actions, and events are what the Thing Description calls `InteractionAffordance`. The `Interactions` are simultaneously composed by one or more `Forms` or ways to access data.

All the `Interactions` of a Thing are instances of a `DataSchema`. This `DataSchema` defines the type of object that the interaction responds with (e.g., the status of a light has a Boolean as a data schema). Thanks to this, both ourselves and the system can know the type of data that will be returned when requesting an interaction. The `@type` property can be located in the Thing or in the various `InteractionAffordances`, and they essentially constitute an object with semantic tags (or types). Because it is a semantic value, the parameter can be used for a variety of purposes. For example, in the case of Digital Dice, it allows us to define specific characteristics of a thing custom-made for our Digital Dice, such as when an interaction has a user interface or when a device employs a protocol that the Web of Things does not fully support.

Thanks to using TD to represent the capabilities of a Thing, and the flexibility that the use of semantic values (`@type`) offers, DD can declare its capabilities in a user-readable way, that at the same time is rigid enough to be unambiguous to other computer systems.

2.2 Microservice architectures

Microservice-oriented architectures [10] are a type of Service-Oriented Architectures (SOAs) [11] as both promote the separation of the application into decoupled services. In microservice architectures, the microservices need to be fine-grained, meaning that they must fulfil very specific roles. Consequently, these microservices must communicate with each other using protocols as light as possible and agnostic to any technology, such as HTTP.

In general, we should try to make sure that when we design a microservice, it follows a principle of single responsibility, i.e., that "it does only one thing but does it well". Among the advantages that we find when developing microservice architectures are the modularity, scalability, and integration they provide. However, this type of architecture also comes with a series of problems such as higher latency due to the chain calls that can happen between microservices, the fact that it is often more challenging to test and develop these microservices, or that in this type of architectures we come across what is called the nightmare of microservices. When this so-called microservice nightmare appears, we find that we have hundreds of microservices up that sometimes we are not even sure what they do.

To help us cope with the problems involved with microservice architectures, the community has introduced some concepts or technologies, such as service meshes, Netflix OSS, or Kubernetes [12]. In the next section, we will see how some of these technologies are used to try not to fall victim to the problems of microservice architectures with our Digital Dice.

2.3 Highly available systems

One of the main metrics of a high availability system is its uptime. This parameter denotes the time that a system is performing to a normal degree for a certain amount of time. Architectures based on microservices are of special interest as they provide high availability to their constituent parts since those microservices tend to be pieces with a limited purpose that are easier to maintain. As such, these microservices can be replicated more easily than on a monolithic architecture. The three main principles of software system design that can help achieve that high availability are: removal of *single point of failure* that can make the system stop working, generation of reliable replicas (*crossover point*), and detection of failures that occur in the system in real-time and adapt to these possible errors.

Microservice architectures almost intrinsically adhere to these principles. In most cases, these architectures use the replication of microservices as a redundancy technique to eliminate the *single point of failure*, and the inclusion of operating patterns such as the *circuit breaker* to define the behavior in real-time when these failures occur in *crossover points*.

3 Related work

Digital Dice aims to achieve interoperability and integration of IoT devices. Previous research developed by Guinard [13] proposes an architecture and offers a series of good practices to achieve this objective. The `Webthings.io` [14] approach uses WoT as the definition language to describe the possible interactions of an IoT device. However, `Webthings.io` focuses on the use of IoT devices, while Digital Dice seeks the performance improvement and virtualization of those devices, not just their use.

Another approach that uses the WoT as a definition language is WoT Store [15]. This solution allows the discovery and management of all Things available in the environments in a seamless way. However, it does not really explore any approach about the devices mentioned in those environments. This is where Digital Dice thrives, as both approaches are perfectly compatible. WoT Store can be used to discover Digital Dice. Related to the WoT, a lot of different solutions to the different layers of it have been proposed [16]. However, none of them focuses on the performance improvement of IoT devices.

Microservices are an interesting way of improving performance when handling IoT devices. In [17], the authors carry out an implementation based on Jolie and Java to manage platforms that operate with a multitude of concurrent applications in Smart Buildings. However, this solution has the handicap of being based on a particular domain and not a much broader solution in terms of managing IoT devices and cyber-physical systems in general.

The approach in [18] provides a generic method for using microservices in a non-domain-specific way. It uses microservices to isolate features of IoT-centric systems like security, events and devices, among others. This method, however, does not take full advantage of microservices since those microservices are complex, making system replication and maintenance more challenging. The Digital Dice architecture, on the other hand, uses less sophisticated microservices to improve the capabilities of these systems.

The IoT-A project [19] is an Architectural Reference Model (ARM) that develops a shared knowledge of the IoT domain and offers a common technological basis and set of rules for IoT system developers to derive a concrete IoT system architecture. This ARM is used to create communication across distinct groups of IoT devices in both the WoT and Digital Dice designs.

At the industrial level, there are several developments of different Digital Twins [20]. However, they focus on the most classic definitions of the concept, where the simulation of devices prevails, forgetting the facet of Digital Twins as possible virtual abstractions of physical devices to be used as middleware for the management of these devices. Our Digital Dice describes the necessary communication between devices for the full integration of those with their physical counterparts. Digital Dice embraces the concept of virtual entity and the ability to establish Physical-to-Virtual, Virtual-to-

Table 2 Features comparison of CPS and IoT management solutions

	[14]	[17]	[18]	[23]	[24]	[25]	[26]	[27]	[28]	DD
Interoperable	●	●	●	●	●	●	●	●	●	●
Fault tolerance									●	●
Virtualization										●
Management	●	●	●	●	●	●	●	●	●	●
Scalable		●	●			●			●	●
Highly available									●	●
Standard	WoT			WoT	WoT	SDN	WoT	WoT		WoT
Thing-2-thing						●				●
Domain	G	SB	G	SB	E	G	G	SB	G	G
Implementation	●									●

Physical and Virtual-to-Virtual communication between different artefacts (devices and software abstractions). The work undertaken in [21], thoroughly explains these different definitions and terms that surround the concept of Digital Twin.

Another work of interest in the industrial domain is [22], which uses a microservice approach to develop a framework for manufacturing assembly lines. One of the particularities of their approach is that workers and other artefacts involved in the assembly process are transformed into cyber-physical entities. This work just like Digital Dice uses an MDE approach to design the orchestration and choreography of their approach. Contrary to the general solution of Digital Dice, this solution is focused on solving a particular problem in the mass customization of assembly lines.

Table 2 summarizes the main features of different alternatives to Digital Dice. For clarification, the table uses the following abbreviation in the Domain row: G for General, SB for Smart Buildings, and E for Environmental. As we can see, there are several works that use WoT as the underlying definition language. However, none of them focuses on the performance and scalability of the solution, which is the main goal of Digital Dice and HA systems. In addition, virtualization is an important feature that we only tend to see in the industrial domain and Digital Dice provides this feature.

The management of CPS in a HA environment is a complex task. As far as we know, DD is the first approach that uses WoT as the definition language and aims to become a highly available solution for managing CPS. In addition, another feature of DD is its capability to establish Thing-to-Thing communication. Finally, we provide an actual implementation of our approach, that has not been carried out in any of the works mentioned in the table, except [14].

4 Digital dice: core concepts

This section summarizes Digital Dice and describes its different parts, challenges and strengths in an IoT scenario. Digital Dice was created with the idea of providing virtual representations of *things* capable of managing IoT devices and trying to extend their functionality in the best possible way. To achieve our goal, we decided to adopt WoT as a common language for the definition of Digital Dice. Using a standard supported by the W3C ensured the compatibility of Digital Dice with systems and platforms that use the WoT as a definition language, like `WebThings.io` [14], a platform for monitoring and managing devices over the web, recently liberated by the Mozilla foundation.

As an intermediary WoT servient, Digital Dice can consume and expose *things*. Moreover, it adopts microservices as one of its building blocks. The use of microservices adds complexity to our infrastructure but also offers the possibility of adding replicas to the services that need it. As we mentioned previously, one of the main problems that we have when working directly with physical devices is that they tend to have constrained resources in some cases. The use of microservices, as well as other strategies that are shown in the next subsection, helps to alleviate this problem.

Figure. 1 represents the metamodel for the definition of Digital Dice, in which we can see the microservices that can be part of it, each of which has different `Environments` where the necessary environment and configuration variables are declared. Furthermore, for internal and external communication, each microservice has a series of `Resources` defined by `href` and its `DataSchemas` input and output. The green classes denote those already established in the Thing Description of the WoT. `DeployConfig` sets the parameters required for the scaling and target CPU usage of microservices. The microservices that DD will use are generally defined by the type of interactions the device can use, as seen in some of our previous work [4]. Digital Dice can make use of the following microservices:

– `Controller` (C). This microservice is responsible for communicating with clients or other software components. Furthermore, it orchestrates the requests to the rest of the microservices.
– `Reflection` (R). This microservice is responsible for recording the interactions that occur on the IoT device onto a database, defined by the `DBConfig`, as well as executing the interactions requested by the user on said device. This microservice establishes a permanent connection with the IoT device it represents.
– `DataHandler` (DH). This microservice is responsible for communication with the underlying database. It records all the interactions requested for a particular device on the database and recovers the data requested by users. This microservice acts as an intermediary and decides when a request can be resolved with the database without establishing a direct connection with the IoT device.
– `EventHandler` (EH). This microservice handles the events generated by the IoT device. It is also responsible for listening to events that affect some interactions of the represented device.
– `User Interface` (UI). This microservice is responsible for offering user interfaces when necessary. These interfaces can be at the interaction level, thing level, or both. Furthermore, the interfaces must be declared as a `link` in the TD that represents the particular DD.
– `Virtualizer` (V). This microservice is a substitute for the physical IoT device, simulating its operation if required. It is the closest thing to the general concept of Digital Twin within Digital Dice.

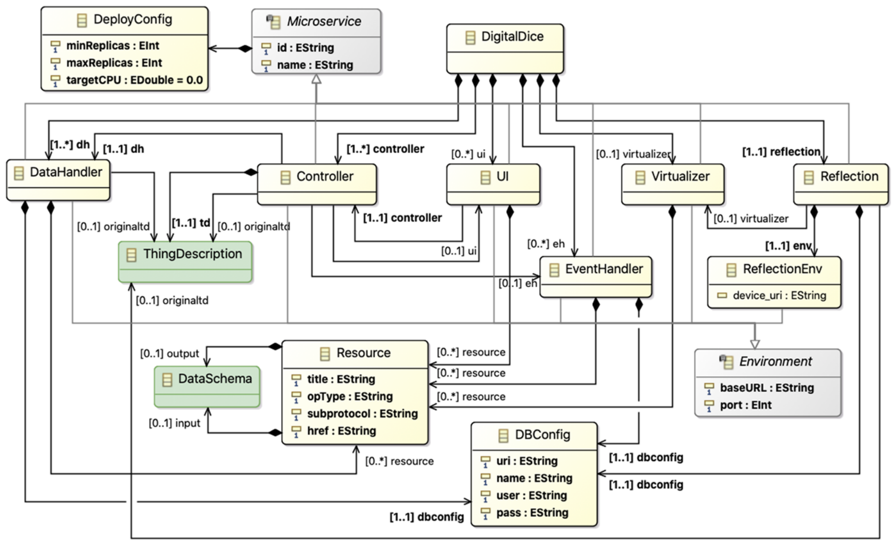


Fig. 1 Digital Dice metamodel

As seen in Fig. 1, microservices C, R and DH are mandatory in every Digital Dice, while EH, UI and V are optional, depending on what is being represented. Finally, DD offers a Thing Description to interact with the represented device through the Controller. The `td` parameter allows DD to be compatible with other WoT systems.

The language represented by the metamodel, allows us to automatically generate DD code simpler than that established in [4] since the process is more direct than starting from a Thing Description, limiting the casuistry and the number of rules that are applied for its generation.

Fig. 2 represents the architecture of Digital Dice with all the different artefacts. We can see how the parts interact with each other and how Digital Dice offers an API for communication with users or other devices. Digital Dice can have different configuration states, as well as different declared microservices. In this case, DD#1 has an EH, and the DH microservice is replicated. Meanwhile, DD#2, has a UI microservice, and the controller has a replica.

Apart from the microservices established in the metamodel for each DD, the architecture requires a series of common artefacts for their correct operation. First, Digital Dice needs a database. For that, we use MongoDB due to its capability of working with all types of unstructured data and the possibility of making replicas, which allows simpler scalability than other types of databases.

It is important to note that DD establishes its own schema when saving the data generated by the IoT device or the requests made by the user. Creating a general schema is necessary since the data required or provided by different IoT devices can be diametrically different. In the best case, these devices will offer data in the form of readable JSON or XML-type schemas, although this is not always the case, since we can also find textual or binary data. Table 3 shows the parameters used in this schema.

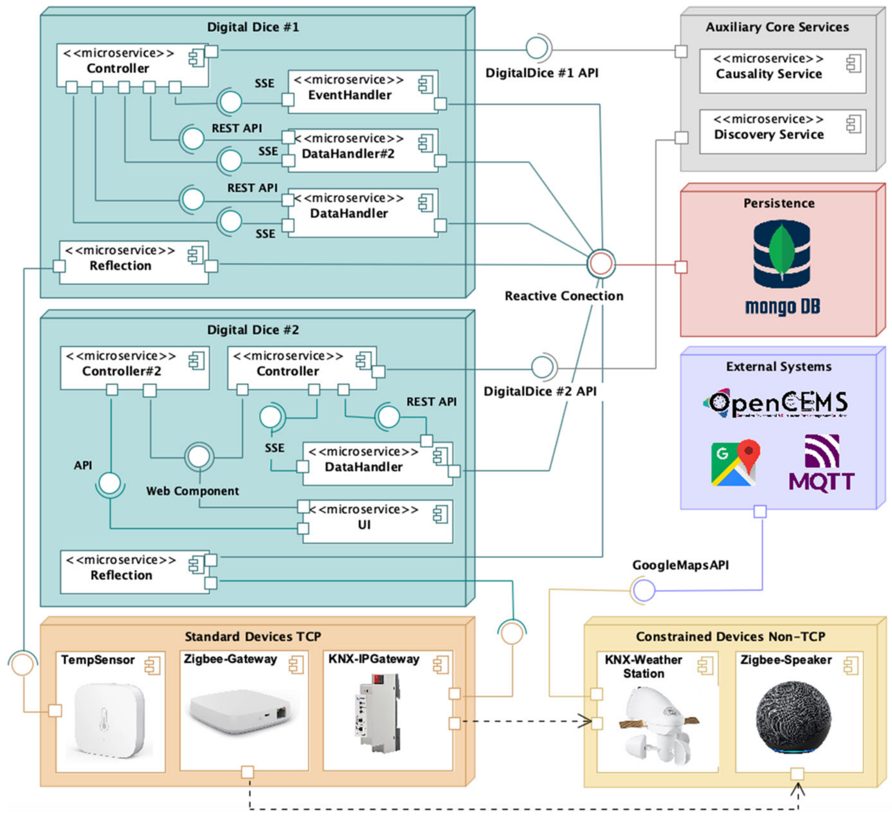

Digital Dice #1
<<microservice>> Controller
SSE
<<microservice>> EventHandler
REST API
<<microservice>> DataHandler#2
SSE
REST API
<<microservice>> DataHandler
SSE
<<microservice>> Reflection
DigitalDice #1 API
Auxiliary Core Services
<<microservice>> Causality Service
<<microservice>> Discovery Service
Persistence
mongo DB
Reactive Conection
Digital Dice #2
<<microservice>> Controller#2
<<microservice>> Controller
REST API
SSE
<<microservice>> DataHandler
API
Web Component
<<microservice>> UI
<<microservice>> Reflection
DigitalDice #2 API
External Systems
OpenEMS
MQTT
GoogleMapsAPI
Standard Devices TCP
TempSensor
Zigbee-Gateway
KNX-IPGateway
Constrained Devices Non-TCP
KNX-Weather Station
Zigbee-Speaker

Fig. 2 Digital Dice Architecture

Table 3 Thing interaction schema (*) action.*, property.* or event.*

Field	Description	Assignment	Type
id	Unique number assigned to a Thing Interaction.	required	ObjectID
device	Parameter used to identify the related device. This parameter has to be equal to the ID of the Thing Description that represents the DD.	required	String
interaction	Name and type of the interaction that is going to be saved in the database.	required	String (*)
data	Raw data generated or needed for the interaction. This is the only visible data for the end user the rest entities are for internal use only.	optional	Object
createdAt	Timestamp for the creation of the Thing interaction.	required	Date
updatedAt	Timestamp for the possible modification of the Thing Interaction.	optional	Date

As a result, we can establish a set of standard metadata for all the requests performed on the system, making it easier to carry out simple data analysis afterwards.

The second piece necessary for the proper operation of DD is the causality subsystem [7]. It allows the establishment of a mechanism to define interactions between different devices. This relationship is formed by two entities, *causes* and *effects*. Causality relationships help us to define how different devices interact without human intervention, all through a common definition schema.

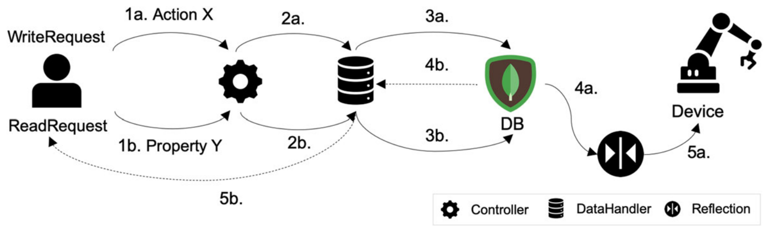


Fig. 3 Flow of write and read request in a Digital Dice

Finally, our architecture has a discovery service to be able to consult the different Thing Descriptions that are available on the network. This service appears in the recent modifications included in the WoT recommendations, which call for a discovery subsystem to establish public domain implementations. We created a discovery service, which can be found in the WoT-Lab[2] of our research group, in which the users can interact with the different physical and virtual devices that make up the experimentation environment.

In some cases devices or the different parts of DD may require other services, such as OpenData platforms (e.g., OpenCEMS [29]), external APIs (e.g., Google Directions API), or auxiliary services (e.g., Mosquitto MQTT server).

Due to the requirements of working with IoT devices, such as the necessity of being able to work with real-time data, some of the DD microservices, like the DataHandler, do not follow the traditional stateless pattern of RESTful APIs. This microservice, in particular, needs to be stateful because besides offering a REST API to its users, it offers communication through Server-Sent Events (SSE) so that users can access the data required in real-time.

We have now seen all the pieces that make up the architecture of DD. However, it is interesting to see how different types of requests are orchestrated in this architecture to understand its operation better. To do so, in Fig. 3 we depict two different types of requests. On the one hand, the invocation of an action `X`, and, on the other, the request of a property `Y` to a Digital Dice. In the first request, the user asks the controller of the DD that represents the device to invoke the action `X`, while the controller requests the action from DH. DH then registers the aforementioned request on the database, which will be seen by the Reflection of the device that executes the action `X` on the device. In the second request, the user asks the controller for the value of the property `Y`, before the controller redirects the request to the DH, which asks the database about this property. Finally, the database retrieves the last value of this property and returns it to the DH which will then send the response to the user. The choice of these two operations is not accidental. The first is a `write` operation, while the second is a `read` operation.

It is essential to realize that only `write` operations require communication with the device. In contrast, `read` operations use the database as a kind of cache for the data requested by the IoT device. When we are talking about TD, we see that the possible interactions of an IoT device according to this schema are actions, properties, and

[2] ACG WoT-Lab - https://acg.ual.es/wot-lab.

events. The invocation of an action will always be a `write` operation while listening to an event is always a `read` operation, and in the case of properties, these can be both `read` and `write` operations.

Reducing the number of requests executed by the IoT device is very important, especially when talking about industrial environments. In these environments, many devices or cyber-physical systems use bus-type protocols, such as Modbus or Profibus, which have more than obvious limitations in the number of agents carrying out operations.

5 Digital dice as a highly available solution

There are two main reasons behind the creation of DD. First, to homogenize the ecosystem of IoT devices. Generally, these types of ecosystems tend to be very heterogeneous due to the myriad of different devices that tend to coexist in different deployments. Secondly, to improve the performance of the devices, as they tend to have limited resources. In previous sections, we have explained how DD establishes a common query language for different IoT devices based on the use of the definition language established by the Thing Description of the WoT. This section will look at the different strategies we have used to provide DD with the concept of high availability (HA) and see how these strategies help improve the system's overall performance.

To better understand these strategies, we have divided them into three different types: (a) Physical device strategies related to communication with a particular physical device or cyber-physical system; (b) Microservices replication strategies related to how the different microservices generate more replicas or stop some of them; and (c) Communication strategies related to the inter-communication between microservices and the communication between the said microservices and the end-user.

5.1 Physical device strategies

`Reflection` is the only point of connection with the physical device and prevents the device from collapsing; it is the primary decision regarding device communication strategies. As we have previously said, many IoT devices have notable limitations in terms of performance since they are often governed by micro-controllers of the ESP32 type or similar, which are designed above all to prioritize energy saving. Furthermore, in some IoT environments, such as the bus-based KNX or Modbus installations, we find limitations in the number of connections we can establish. When many users want to request data simultaneously, the bus can collapse and then all the requests are denied.

The use of `Reflection`, as well as using the database as an intermediate system, lead to the reading operations not having to reach the physical device, which already avoids many requests to the said devices. In addition, having a single point of connection with the device ensures that the writing operations are on a first-come, first-served basis. The `Reflection` operation is based on subscribing to the *reading* interactions of the device. If the device does not allow the subscription to interact, `Reflection` is responsible for making long-polling requests to the device every `X` seconds, `X` being a configurable time. Furthermore, `Reflection` listens to the *writing* operations related to the controlled device. When the microservice detects that a given operation requests changes on the device, it will execute it. The choice of MongoDB means that the `Reflection` does not need to be continuously requesting data from the database. MongoDB allows us to subscribe to particular queries through a reactive pattern, making the database responsible for notifying the subscriber when new data meets the parameters established in the query.

5.2 Microservices replication strategies

As for microservices replication strategies, we must first understand that the microservices which are part of DD are based on docker-type containers. Thanks to the use of containers, we can isolate each of the microservice executions and work environments from the deployment machines. Using containers offers the great advantage of being able to deploy our solution in any machine, both in bare-metal machines or in cloud infrastructure.

When deploying Docker containers, we can use different clients, like the original docker client (docker CLI), docker-compose, Rancher and Kubernetes, among others, each one having its advantages and disadvantages. However, due to its direct compatibility with different cloud services and the possible installation of small infrastructures based on them in our machines, we decided to deploy them using Kubernetes.

Cross-platform support is not the only reason we have decided to use Kubernetes. Unlike some of its alternatives, this system has a series of pre-installed services that save us from having to install new artefacts necessary for Digital Dice to work, as well as a series of advantages:

(a) It offers the automatic generation of subnets to isolate the microservices for each of our DD.
(b) It allows communication between the different containers by using the container's name as an internal DNS, thus avoiding the need to install a discovery service since Kubernetes has already implemented it.
(c) Moreover, thanks to how widespread this client is, we are able to work with a series of tools that allow us to track the performance and other metrics for each container, like Prometheus, which can be used to design a dashboard using Grafana, to facilitate the tracking of these metrics.

Kubernetes also allows us to replicate containers automatically, unlike other clients. To do this, we establish a series of predefined parameters: the number of CPU requested, the CPU limit, the minimum and the maximum number of replicas that each microservice can have, and the target CPU utilization for each container of a particular microservice. Then, when the Kubernetes system realizes that any container is receiving a CPU load close to the established limit, it automatically launches another replica of that microservice. The opposite takes place in the event of it realizing that there is a container not being sufficiently used, meaning it will be responsible for stopping it. It should be noted that Kubernetes, by default, not only uses the level of CPU utilization for the replication of microservices but can also use the memory level.

However, given our experimentation with handling microservices of Digital Dice, it is better to use the CPU instead of the memory. When we use containers with Kubernetes, in front of every one of them, there is a *service* commonly called entryPoint, which is responsible for distributing requests to each of the replicas from the corresponding microservice through a predefined strategy. In the scenario example that we describe in the next section, a deployment configuration file for Digital Dice can be seen[3]. Each of the microservices (deployments) are paired with a configuration of type *service*. Moreover, each deployment has another configuration of type *HorizontalPodAutoscaler* for handling the replicas of a particular microservice.

5.3 Communication strategies

When discussing communication strategies in DD, we have to divide them into two different viewpoints, namely, how DD communicates with the end-user and what happens with the communication between the microservices.

As we have already mentioned, DD uses the standard proposed by the WoT. This standard establishes a definition file, the Thing Description, which encapsulates all the interactions DD can manage. The Thing Description aims to be to things or virtual things (Digital Twins - Digital Dice) what Open API is to a service. One of the Digital Dice premises is to facilitate its use by any web or mobile developer, so from the inception of DD, we decided that end-users should be able to interact with DD through an API. However, the use of an API in the field of IoT brings with it a series of limitations when we talk about data in real-time, so in the end, our DD uses what we could call a hybrid connection pattern. Firstly, with a REST API, that by definition is Stateless, which means that the query is made and answered at the moment. Secondly, the use of Server-Sent Events (SSE), a stateful protocol, means that the user is subscribed to a method until the connection is closed. In DD, we can subscribe to any read interaction through SSE, meaning that we can subscribe to a device's property through an HTTP request and then receive the changes of the property in real-time. All without the end-user having to learn a technology outside their domain. The use of SSE presented us with a problem that, in the beginning, we had not taken into account. With HTTP/1 connections, a service can only keep five connections open with each user. HTTP/2 and HTTP/3 solve this problem by removing that limitation. In our Digital Dice, we have chosen to establish the connection with the user through the HTTP/2 protocol since HTTP/3, although it is faster than HTTP/2 as it works over UDP, is not supported by most browsers. Furthermore, using HTTP/2 requires that every DD path is secured by an SSL certificate due to the need for this by the HTTP/2 protocol.

The use of HTTP/2 in communications with the user means that internal communications in each subnet of DD must also be secured internally, so we also use the HTTP/2 protocol. As we discussed earlier, when using a microservice architecture, we find that turning replicas on and off can cause what is commonly called the nightmare of microservices. In this type of architecture, it can happen that requests are being made to containers which are no longer available, some of the containers are

[3] Garbage Containers Kubernetes file - https://cutt.ly/YIgtLAW.

Table 4 Horizontal-pod-autoscaler configuration parameters

Service	ReqCPU	MaxCPU	Min replicas	Max replicas	Average utilization
Controller	300 m	600 m	1	5 || 7	50%
Data handler	300 m	600 m	1	5 || 7	50%
Reflection	200 m	1000 m	1	1	Non-Applicable

continuously causing errors, or a communication overload is caused in the subnets without us noticing.

Digital Dice uses a service mesh, more specifically `envoy` and `istio`, to solve these problems. A service mesh is used to control how microservices share data. Unlike other systems that can also manage communication, the service mesh is a layer integrated into the application, which is visible and allows us to record whether the interaction between different parts of an application is good or bad. It facilitates the optimization of communication and prevents downtime as the application grows. Apart from monitoring, it also applies a series of recommended patterns in microservice architectures. One of them is the *circuit breaker*, which avoids communication with a microservice method if it continuously generates problems, forcing a default response. Furthermore, the use of `envoy` and `istio` allows load balancing between the different replicas of a microservice automatically and without the need to implement it in the microservice itself.

5.4 Performance evaluation

Performance evaluation needs to establish a baseline for the characteristics of the infrastructure where DD are deployed. For this test, even though Digital Dice can be deployed on bare metal, we decided to deploy them on a cloud service, in this case, Google Cloud. Of course, making this choice comes with a penalty in response times. However, we can experiment with more fine-grained control and fewer limitations when allocating resources for each microservice.

Table 4 shows the different parameters required by the different microservices used to measure the performance of Digital Dice. Those parameters are the minimum number of replicas, the maximum number of replicas, the requested CPU level measured in milicores (1000 m = 1 core), the maximum CPU level and the target CPU utilization requested by the system for each microservice container. Apart from the parameters of each microservice, we need to define the number of replicas used in our MongoDB deployment. We use a replica set with two machines equipped with 2 vCPU and 2 GB of RAM each in this experiment. We decided to use two different configurations (config#1 and config#2) to see the difference in performance. As can be seen, the only difference in the second configuration (config#2) is that we added two more possible replicas to the maximum number of replicas allowed.

The physical device used to measure the performance of Digital Dice is a simple light since we can quickly deploy this device with different technologies, more specifically a Philips light controlled by the Zigbee protocol, a KNX light and light controlled by a web server mounted on an ESP32 microcontroller. This choice allows us to carry out a performance comparison by abstracting from the technologies that have been used. At the same time, this device allows us to work with the three types of operations we deem as "basic" in a thing, namely read operations, write operations and subscriptions.

In Listing 1, we can see the Thing Description of the Digital Dice that governs these devices, which is the same in all three cases, since although the internal implementation is different in the case of Reflection (microservice that establishes the connection with the physical device) for each of them, they all have the same operations, and the Thing Description is a black box for the user when operating with our DD.

```
{
"@context": "https://www.w3.org/2019/wot/td/v1", "id": "acg:lab:light", "title": "ACG Lab
      Light",
"properties": {
  "status": {
    "type": "object",
    "properties": {"value": { "type": "boolean" }},
    "forms": [{
      "href": "https://example.com/acg:lab:light/property/status/",
      "contentType": "application/json"}, {
      "href": "https://example.com/acg:lab:light/property/status/sse",
      "subprotocol": "sse", "response": {"contentType": "text/event-stream" }}] }},
"actions": {
  "switch": {
    "input": {
    "type": "object",
    "properties": {"value": { "type": "boolean"}}},
    "required": [ "value" ],
    "forms": [{
      "href": "https://example.com/acg:lab:light/action/switch/",
      "contentType": "application/json" }]} }
}
```

Listing 1 Thing Description of Digital Dice Light.

In Fig. 4, we measure the performance of Digital Dice for the two different configurations (config#1 and config#2) stated in Table 4, by checking the level of expected responses, along with errors that appear as we increase the number of requests to the system. In this figure, we mean by *errors* those requests where the response is part of the 400 Series Client Error Status Codes, like the responses 404 Not found or 408 Request Timeout. To perform the measurement, we used two different platforms for the synthetic generation of requests on our system, Locust and JMeter.

In the upper-left chart of Fig. 4a, we see that errors appear from a relatively low number of requests per second in the different physical devices we used as a baseline. Interestingly, in bus-based connections, like KNX, we find that after eight simultaneous reading operations, time-outs begin to surface since the data buffer of this system collapses.

Conversely, in the upper-right chart of Fig. 4b, we can see how our Digital Dice performs as far as reading operations are concerned. As we increase the number of requests, our system responds much better than any baseline devices shown in the upper-left chart because we do not need to establish direct communication with the physical devices. However, in the chart, we can see how some errors appear for approximately every 500 users making requests to our Digital Dice. The errors arise when a new replica is being booted, which causes requests to be directed to that new container before it finishes booting. At this point, the number of errors appears to stabilize again. It is important to see that, although we have previously defined that our system can work with five replicas (config#1), in the graph, the fifth spike

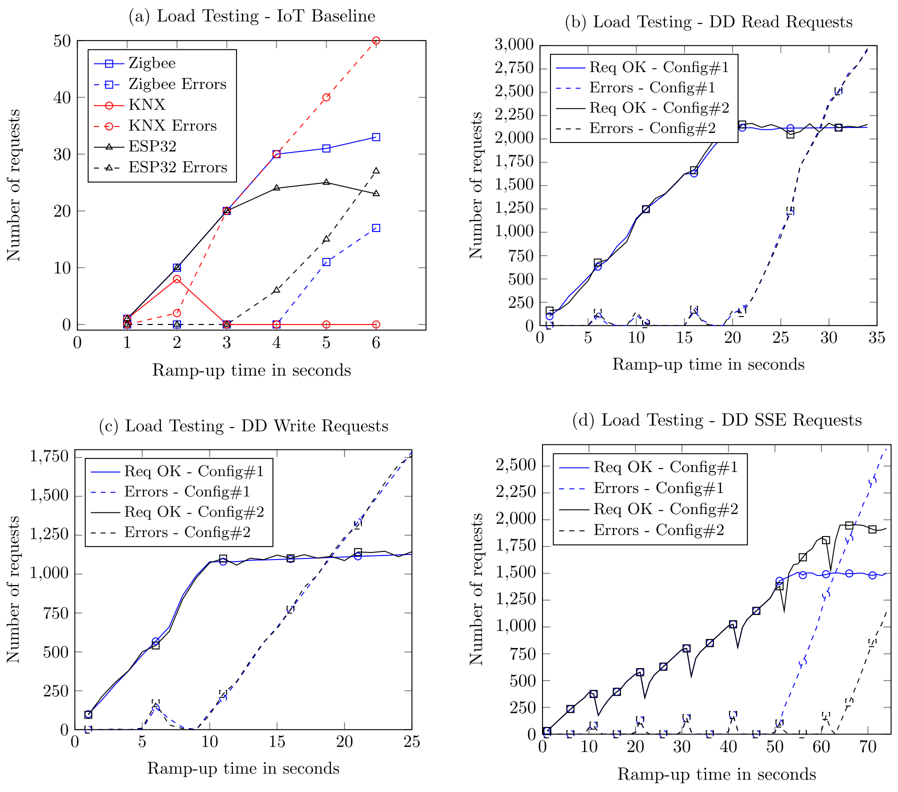


Fig. 4 Performance charts: Example IoT devices vs Digital Dice

in the number of errors does not appear. However, from a certain point, the system reaches the limit for the number of requests it can handle. This point is reached when we get to the maximum amount of requests that can be handled by the database simultaneously. At that time, the database becomes the limiting factor. After increasing the database resources, we observed that the number of requests increased as expected. Furthermore, suppose we increase the number of replicas of the microservices and the number of replicas of the database in our deployment to more than five. In that case, we reach another limiting factor, the bandwidth from where we launch the synthetic requests. We suspect that if we used a network of nodes to launch the requests in a parallelized way, our system would continue to scale the requests as expected, but due to the cost it would entail, we have not yet been able to confirm this hypothesis. The same happens with config#2, as DD is not the limiting factor.

The lower-left chart of Fig. 4c represents the write requests that our system can handle. Again, the behavior is similar to `read` operations, but collapse occurs much earlier in the two configurations. The collapse happens because Reflection establishes a queue with the write requests that arrive at the database, requests that will be sent sequentially to the physical device. This behavior occurs when the number of requests that reach the database exceeds the number of requests that our physical device can accept, and the queue in Reflection gets filled up. As such, we run into the limitation of writing operations onto the physical device.

Finally, the result displayed in the lower-right chart of Fig. 4d allows us to see the behavior of server-sent events (SSE). This type of request is quite challenging to test since practically none of the load testing tools can work with these. To test SSE, we must generate custom test scripts using a Java Sampler. Once launched, we see that the behavior differs from the usual read operations. First, fewer requests can be processed at the same time. In addition, we can observe that the error spikes, even though they appear to be in sync with the replication of a service, are more pronounced and seem to affect the number of requests processed by the system. Furthermore, using five as the maximum replication factor, the system does not find any limitations in the database, unlike with the read operations. In config#2, where the replication factor for the microservices is set at seven, we can see how we start to find the same limiting factors again as with the read operations.

The replication becomes erratic with SSE using the CPU limit as a replication metric because the containers are almost idle until a message is triggered. We are studying how to establish other custom replication metrics in future work, such as the number of requests per second our system receives. With this metric, microservices will anticipate the need for replication, making our system even more resilient.

Augmenting the number of replicas or giving more resources to the database certainly improves the performance of the different operations in Digital Dice. However, due to the cost of the operation using more replicas, we decided that the configuration used was good enough to see the behavior of Digital Dice for all the different types of operations.

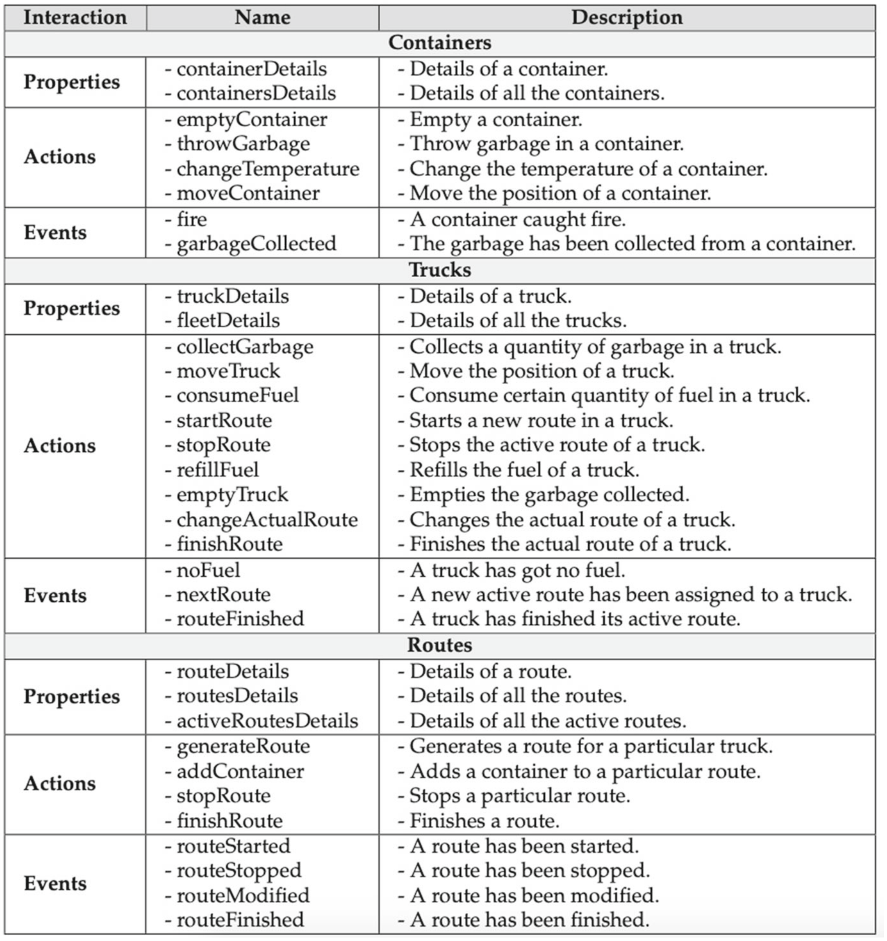
6 Smart city example scenario

In this scenario, we intend to demonstrate the viability of Digital Dice in a smart city environment. To do so, we decided to develop a virtualized system based on a process very important for citizens, the garbage collection process. We chose this scenario because it aligns with our DD's strengths. In this process, the actual citizen is a stakeholder, so developing a high-availability alternative is of great interest to citizens as it provides them with real-time information about the status of the containers and the routes followed by the garbage collection trucks. Moreover, since we will work with a completely virtual system, we present DD as a proper implementation of a DT.

The scenario comprises three Things to control: containers, trucks, and routes. The possible interactions available in each of the Things are defined in Table 5. It is worth noting that each of the Thing Descriptions in our scenario does not represent an individual entity. This means that, for example, in the containers, the Thing Description does not just represent a single container; it represents a complete set. One of the significant advantages of WoT is its flexibility when defining the concept of Thing. Conceptually, the Thing exposed in the example could be considered an aggregated Thing in WoT terms. The public code repository is online[4] and provides more details

[4] Garbage collection repo. - https://github.com/acgtic211/garbage-iiot-dd-public.

Table 5 Garbage collection scenario - Interaction Affordances



Interaction	Name	Description
	Containers	
Properties	- containerDetails	- Details of a container.
	- containersDetails	- Details of all the containers.
Actions	- emptyContainer	- Empty a container.
	- throwGarbage	- Throw garbage in a container.
	- changeTemperature	- Change the temperature of a container.
	- moveContainer	- Move the position of a container.
Events	- fire	- A container caught fire.
	- garbageCollected	- The garbage has been collected from a container.
	Trucks	
Properties	- truckDetails	- Details of a truck.
	- fleetDetails	- Details of all the trucks.
Actions	- collectGarbage	- Collects a quantity of garbage in a truck.
	- moveTruck	- Move the position of a truck.
	- consumeFuel	- Consume certain quantity of fuel in a truck.
	- startRoute	- Starts a new route in a truck.
	- stopRoute	- Stops the active route of a truck.
	- refillFuel	- Refills the fuel of a truck.
	- emptyTruck	- Empties the garbage collected.
	- changeActualRoute	- Changes the actual route of a truck.
	- finishRoute	- Finishes the actual route of a truck.
Events	- noFuel	- A truck has got no fuel.
	- nextRoute	- A new active route has been assigned to a truck.
	- routeFinished	- A truck has finished its active route.
	Routes	
Properties	- routeDetails	- Details of a route.
	- routesDetails	- Details of all the routes.
	- activeRoutesDetails	- Details of all the active routes.
Actions	- generateRoute	- Generates a route for a particular truck.
	- addContainer	- Adds a container to a particular route.
	- stopRoute	- Stops a particular route.
	- finishRoute	- Finishes a route.
Events	- routeStarted	- A route has been started.
	- routeStopped	- A route has been stopped.
	- routeModified	- A route has been modified.
	- routeFinished	- A route has been finished.

about the different interactions as well as the input and output parameters required. This repository contains all the Thing Descriptions of the scenario, the source code of the different microservices, and the configuration files of both Docker and Kubernetes to replicate the scenario when necessary.

This scenario seeks to maximize the length of the different routes in real-time as the garbage containers fill. To execute this behavior, we need to declare a series of causality relationships represented in Fig. 5, whose formal definitions are found in the Event Handlers of each of the Things that intervene in the scenario. These relationships lead to the execution of a series of actions or modifications of a certain property in their Things. For example, we can see in Fig. 5 how when a container exceeds 90% of its capacity (cause), the rule tries to add it to the route through the `addContainer` action (effect), which will internally look for the optimal route to add the container,

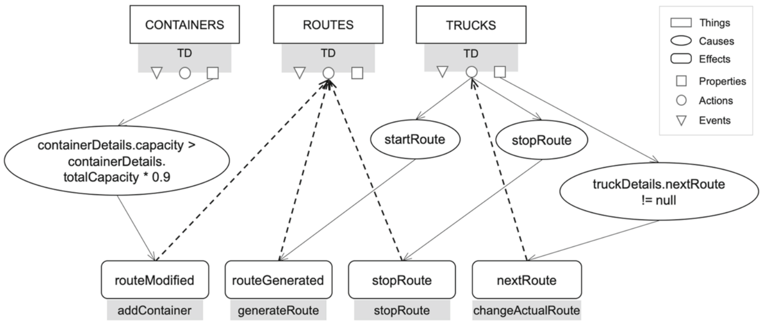


Fig. 5 Garbage collection scenario-causalities

prioritizing active routes that neither exceed the fuel capacity, nor the load capacity of the particular truck that is working on it.

In this scenario, the virtualization services are responsible for imitating the behavior of containers and trucks. The latter's movements are established from the generated route, which is defined by the containers to be collected, whose optimal route is calculated thanks to Google Directions API. Unless the user indicates how much to fill a particular container manually, the containers can execute a virtualization behavior, which fills each container every x seconds with a semi-random weight depending on the container we are dealing with.

In the ACG WoT-Lab you can find an interface for the management and monitoring of this scenario, specifically in this case, the interface to follow the data from truck 2. Furthermore, WoT-Lab shows a scenario related to smart homes and gives access to Thing Descriptions that represent physical devices, virtual devices, and Digital Dice. These Thing Descriptions are available through a public discovery service to experiment with this technology.

7 Discussion

If we go over the challenges and the research questions discussed in Sect. 1 , we can now see how our proposed solution meets all the expectations.

In summary, Digital Dice is a virtual abstraction of IoT devices that aim to improve the interoperability, integration, and management of these types of devices in multiple domains. The idea arises from creating Digital Twins based on microservices, but providing them with the capability of virtualizing devices and managing them by establishing a common pattern of communication that is completely agnostic with the underlying technologies in the device (RQ1). We also describe how Digital Dice uses the standard established by the W3C, the Web of Things, to declare the functionality provided by each virtual device abstraction. Furthermore, thanks to the Thing Description of the WoT, Digital Dice is compatible with the platforms that use this standard (RQ2).

The article presents the pieces that can form part of a Digital Dice, both microservices and external artefacts. Furthermore, this work introduces a language represented by the DD metamodel, which helps define the different microservices and configuration parameters of a specific deployment (RQ2).

In addition, we have seen how Digital Dice establishes behaviors and communication between different devices, forming a series of rules called causality relationships. These rules facilitate the possible automation of processes, avoiding the need for human intervention.

On top of that, we have described the different strategies that Digital Dice uses to improve its performance, focusing on strategies regarding communication with physical devices, microservice replication strategies and communication strategies between them and the user (RQ5). Moreover, we have verified empirically and experimentally how the application of these strategies improves performance, making it a high-availability alternative for managing devices with performance restrictions (RQ3).

Finally, we have established an example scenario based on the garbage collection process to see how Digital Dice could work without significant problems in a smart city environment. From this scenario, we have provided a public repository with the source code and the Thing Descriptions and deployed a web interface to enable experimentation (RQ4).

Through the article, we have seen how Digital Dice can be used as a virtual abstraction of CPS, particularly IoT devices, and how the proposed architecture helps us to achieve high availability and scalability. This brings about a series of challenges yet to overcome, such as the fact that the current version of Digital Dice has some potential improvements related to the load balancing offered by the service mesh the moment that the number of replicas of a microservice is increased or decreased, especially when talking about SSE requests and in the case of write requests, as we saw in Sect. 5.4. Furthermore, even though Digital Dice can establish and use typical authorization mechanisms, such as OAuth2, and uses secure communication protocols, such as HTTPS, it is still necessary to implement a security layer capable of protecting the data exchanged between the different microservices. As of right now, for the data exchanged, we rely on the security mechanisms offered by the service mesh, but we are aware that this is not enough.

8 Conclusion

Digital Dice provides a new way to manage CPS by providing an architectural approach that allows the creation of virtual abstractions of IoT devices. This approach is based on creating microservices that represent the different functionalities of the devices, which can be deployed in a distributed way. The main advantage of this approach is that it allows the creation of high-availability and scalable systems, which can be used to manage IoT devices with performance restrictions. However, we are aware that there are still some challenges to overcome, as discussed in Sect. 7. Still, we believe this approach is a good starting point to create a new way to manage CPS that does not require proprietary solutions and that can be used in a wide range of domains and applications, especially in those that require high availability and scalability.

We have a series of open issues when discussing future work related to Digital Dice. Regarding the replication behavior of microservices, we plan to establish new custom metrics to anticipate peak requests and improve the performance as far as SSE requests are concerned. These performance issues can be seen in the Sect. 5.4. To solve them, we will try to apply smoothing strategies when microservices are replicated, as well as in the load balancing phase applied by our service mesh. Moreover, another aspect we want to improve about our Digital Dice is the security of our system, going further than the suggestions offered in the security guidelines of the WoT. To do so, we want to implement request filtering in our service mesh to mitigate the possible malicious code injection in the communication between our microservices. Another open line in our architectural proposal is to make Digital Dice compatible with some open data platforms, such as Open CEMS [29], to serve as data sources. Finally, we want to create an application to ease the generation of Digital Dice models using the proposed language.

Acknowledgements R&D Projects UrbanITA (PY20_00809) and HERMES (PID2021-124124OB-I00). Manel Mena: Spanish Government FPU17/02010.

Funding Funding for open access publishing: Universidad de Almería/CBUA.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

1. Laghari A, Wu K, Laghari R et al (2021) A review and state of art of internet of things (IoT). Arch Comput Methods Eng. 1–19. https://doi.org/10.1007/s11831-021-09622-6
2. Pivoto DG, de Almeida LF, da Rosa RR et al (2021) Cyber-physical systems architectures for industrial internet of things applications in Industry 4.0: a literature review. J Manuf Syst 58:176–192. https://doi.org/10.1016/j.jmsy.2020.11.017
3. Botta A, De Donato W, Persico V, Pescapé A (2016) Integration of Cloud computing and Internet of Things: a survey. Future Gener Comput Syst 56:684–700. https://doi.org/10.1016/j.future.2015.09.021
4. Mena M, Criado J, Iribarne L, Corral A (2021) Assembling the web of things and microservices for the management of cyber-physical systems. J Univers Comput Sci 27(7):734–754. https://doi.org/10.3897/jucs.70325
5. Bucchiarone A, Dragoni N, Dustdar S et al (2020) Microservices. Springer Cham. https://doi.org/10.1007/978-3-030-31646-4
6. Tao F, Zhang H, Liu A, Nee AY (2018) Digital twin in industry: state-of-the-art. IEEE Trans Industr Inform 15(4):2405–2415. https://doi.org/10.1109/TII.2018.2873186

7. Mena M, Criado J, Iribarne L, Corral A (2021) Defining interactions of WoT servients with causality relations. In: Proc. MEDES'2021, pp 112-119. https://doi.org/10.1145/3444757.3485102
8. Mena M, Criado J, Iribarne L, Corral A (2020) WoTnectivity: a communication pattern for different web of things connection protocols. In: Proc. COMPSAC'2020, pp 1059-1064. https://doi.org/10.1109/COMPSAC48688.2020.0-133
9. Whitmore A, Agarwal A, Da Xu L (2015) The internet of things-a survey of topics and trends. Inf Syst Front 17:261–274. https://doi.org/10.1007/s10796-014-9489-2
10. Pautasso C, Zimmermann O, Amundsen M et al (2017) Microservices in practice, part 1: reality check and service design. IEEE Softw 34(1):91–98. https://doi.org/10.1109/MS.2017.24
11. Papazoglou MP, van den Heuvel WJ (2007) Service oriented architectures: approaches, technologies and research issues. VLDB J 16(3):389–415. https://doi.org/10.1007/s00778-007-0044-3
12. Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables Devops: migration to a cloud-native architecture. IEEE Softw 33(3):42–52. https://doi.org/10.1109/MS.2016.64
13. Guinard DD, Trifa VM (2016) Building the web of things with examples in node. js and raspberry Pi. manning publications
14. WebThings. https://webthings.io. Acc.: 2022-12-29
15. Sciullo L, Gigli L, Trotta A, Felice MD (2020) WoT store: managing resources and applications on the web of things. Internet Things 9:100164. https://doi.org/10.1016/j.iot.2020.100164
16. Sciullo L, Gigli L, Montori F, Trotta A, Felice M (2022) A survey on the web of things. IEEE Access 10:47570–47596. https://doi.org/10.1109/ACCESS.2022.3171575
17. Khanda K, Salikhov D, Gusmanov K et al (2017) Microservice-based iot for smart buildings. In: Proc. AINAW'2017, pp 302-308. https://doi.org/10.1109/WAINA.2017.77
18. Sun L, Li Y, Memon RA (2017) An open IoT framework based on microservices architecture. China Commun 14(2):154–162. https://doi.org/10.1109/CC.2017.7868163
19. Krčo S, Pokrić B, Carrez F (2014) Designing IoT architecture(s): A European perspective. In: Proc. WF-IoT, pp 79-84. https://doi.org/10.1109/WF-IoT.2014.6803124
20. Uhlemann THJ, Lehmann C, Steinhilper R (2017) The digital twin: realizing the cyber-physical production system for industry 4.0. Procedia CIRP 61:335–340. https://doi.org/10.1016/j.procir.2016.11.152
21. Jones D, Snider C, Nassehi A et al (2020) Characterising the digital twin: a systematic literature review. CIRP J Manuf Sci Technol 29:36–52. https://doi.org/10.1016/j.cirpj.2020.02.002
22. Thramboulidis K, Vachtsevanou DC, Kontou I (2019) CPuS-IoT: a cyber-physical microservice and IoT-based framework for manufacturing assembly systems. Annu Rev Control 47:237–248. https://doi.org/10.1016/j.arcontrol.2019.03.005
23. Ibaseta D, García A, Álvarez M et al (2021) Monitoring and control of energy consumption in buildings using WoT: a novel approach for smart retrofit. Sustain Cities Soc 65:102637. https://doi.org/10.1016/j.scs.2020.102637
24. Teriús-Padrón J, Simeoni E et al (2019) Autonomus air quality management system based on web of things standard architecture. In: Proc. UIC-ATC'2019, pp 184-189. https://doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00074
25. Jararweh Y, Al-Ayyoub M, Darabseh A et al (2015) SDIoT: a software defined based internet of things framework. J Ambient Intell Humaniz Comput 6:453–461. https://doi.org/10.1007/s12652-015-0290-y
26. Benomar Z, Longo F, Merlino G, Puliafito A (2020) A stack 4 things-based web of things architecture. In: Proc. Cybermatics'2020, pp 113-120. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics50389.2020.00036
27. Silva B, Khan M, Lee K et al (2020) Restful web of things for ubiquitous smart home energy management. In: Proc. ICNC'2020, pp 176-180. https://doi.org/10.1109/ICNC47757.2020.9049774
28. Yang H, Kim Y (2019) Design and implementation of high-availability architecture for IoT-cloud services. Sensors 19:3276. https://doi.org/10.3390/s19153276
29. Mansour E, Chbeir R, Arnould P, Allani S, Salameh K (2021) Data management in connected environments. Computing 103:1121–1142. https://doi.org/10.1007/s00607-020-00884-9