University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers

PhD Thesis

- summary -

Scalable Software Architectures in Cloud Era

Author: Vasile M. Tovarnițchi

Doctoral committee:

Chairman	Prof. dr. ing. Mihnea Alexandru Moisescu	University Politehnica of Bucharest
Scientific Supervisor	Prof. Emerit dr. ing. Costică Nitu	University Politehnica of Bucharest
Reviewer	Prof. dr. ing. Nicolae Țăpuș	University Politehnica of Bucharest
Reviewer	Prof. dr. ec. Ion Smeureanu	Bucharest University of Economic Studies
Reviewer	Prof. dr. ing. George Culea	”Vasile Alecsandri” University of Bacău

Bucharest, Romania

2021

Contents

1 Introduction **1**
1.1 Thesis structure 1
2 Software Architecture **3**
2.1 Software Architecture Definition 3
2.2 Characteristics of a software architecture 4
2.3 Software architecture and the context abstractization 4
2.3.1 Architectural patterns 4
2.3.2 Architectural styles 5
2.3.3 Reference architectures 5
3 Scalable Infrastructures. Distributed Systems in the Era of Cloud Technologies **6**
3.1 Characteristics of distributed systems 6
3.2 Applications virtualization through Containerization. Functions virtualization: Serverless solutions 7
3.3 Cloud-Native Architectures 8
3.3.1 Green Computing 8
4 Software Systems Scalability **9**
4.1 Software Scalability 9
4.1.1 Vertical and horizontal scaling 9
4.2 Scalability in 3D format 10
4.3 Scalability and integration of the distributed components 10
4.4 Events as triggers of the interaction between components and changes indicators 12
4.4.1 Event-based architectures 12
4.4.2 Challenges raised by event-based architectural approaches 13
4.4.3 Event-based interoperability between distributed components. Integrity and consistency of data/information 13
5 Reactive Software Architectures **15**
5.1 The Reactive Manifesto 15
5.2 Actor Model 16
5.2.1 Actors versus Objects as paradigms in programming 16
5.2.2 The actor as fundamental unit of computing 16
5.2.3 Location transparency and interaction between actors 17
5.2.4 Resiliency 17
6 Intelligent Environmental Monitoring **18**
6.1 Digital Twin in Intelligent Systems 18
6.2 Scalable system for intelligent environmental monitoring 21
6.2.1 Requirements for an environmental monitoring system 22
7 Proposal of scalable software architecture for an intelligent environmental monitoring system **23**
7.1 Interaction and data flows in distributed software systems 23
7.2 Event Gateway 23
7.2.1 Functional roles 24
7.2.2 Advantages and weak points 24
7.3 Message queues. Functional roles 25
7.3.1 Advantages and weak points 25
7.4 Components responsible for message processing 25
7.4.1 Reactive and scalable module for message processing 25
7.4.2 Message processing 26
8 Environmental monitoring software system implementation details. Experimental aspects **28**
8.1 Events representation and ingestion. Interaction mediation 28
8.1.1 Configurations and service registry 28
8.1.2 Events ingestion, authorization and routing 28
8.1.3 Scaling the events ingestion functionalities 28
8.1.4 Interaction mediation 28
8.2 Events processing. Reactive distribution of tasks 29
8.2.1 Hierarchical structure of the processing node. Module scaling 29
8.3 Extending functionalities and system evolution 30
8.3.1 Details on functionalities externalization and extensions implementation 30
8.4 Implementation details summary 31
9 Conclusions. Contributions. Future directions **32**
9.1 Conclusions 32
9.2 Contributions 34
9.3 Future directions. Perspectives 36
9.4 Author's publications 37
Selected bibliography **37**

Chapter 1. Introduction

Humanity influences, but it is also deeply influenced by digital technologies, which are largely based on approaches related to distributed systems. Beginning of the 21st century is marked by a significant increase in the pace of innovation and development of digital technologies, but, at the same time, by the increasing demands of users of those technologies. These, among others, refer to higher and higher performances, even with tendencies to have "real-time" functionalities. In this regard, researchers and engineers have come up with new proposals for specialized concepts and techniques for data handling at various stages. The need of special approaches for event processing in the context of distributed systems comes from the need to process data as soon as they are acquired.

The software components of computer systems are no longer rigid, monolithic, operating independently, but are much more robust and flexible, able, if necessary, to exchange data with other software components, even remotely.

Improving and evolving trends in hardware components, the maturation of Cloud Technologies, as well as the evolution of algorithms and techniques for processing huge volumes of data, require the development of information system architectures in order to facilitate the optimal use and the efficient handling of data.

This work comes with a contribution in this regard, proposing several techniques, concepts and software architecture approaches that can be applied to the design and implementation of a software system involving the management of increased volumes of data, aiming at the following main principles: scalability, extensibility, resilience.

The elements developed in this thesis are materialized by proposing a reference software architecture given the context of an environmental monitoring system that has been confirmed by an experimental implementation.

1.1 Thesis structure

The thesis is structured into 9 chapters.
Chapter 1 – Introduction
Chapter 9 – Conclusions, contributions and future directions
Bibliography.

Chapter 2 includes the presentation of the software architecture field, as well as the presentation of the author's view regarding its role in development of a software system from the perspective of ensuring the system scalability and extensibility. The relevant elements of a software architecture are described, and definitions proposed by the author. It presents in an original way the relevant elements regarding the software architecture, as well as the significance of the activities related to this field over a software system throughout its entire lifespan. The details of a software system are highlighted on which the initial architectural decisions have a decisive effect on the quality attributes of the system, but especially those related to scalability and extensibility.

Chapter 3 is dedicated to distributed systems, an approach that is not lacking in the design of modern complex computer systems. It includes a presentation of the current approaches of accessing computing resources in the form of services, based on concepts and methods coagulated around Cloud Technologies. Modern ways of operating physical resources are presented through the use of dedicated software tools, but also advanced techniques that abstract the computing infrastructures necessary to implement the concrete functionalities of the software components.

The author's view is presented regarding the significance of the conceptual separation between software components - which, before implementation, require the development of an

appropriate architecture - and infrastructure components, so that the development and operation of software components is absolutely agnostic towards any infrastructure-specific detail.

It is shown the author's view on modern trends for the design of software systems related cloud-native architectures.

Chapter 4 includes the detailing of the various perspectives and modalities currently used in practice for the design and implementation of software components that provide a high scalability of a software system. The author also emphasizes that a modern computer system must be designed in such a way that it can be extended and modified as to ensure that it can evolve over time. It shows the methods and techniques that allow the implementation of computer systems based on software architectures considered evolutionary. From these perspectives, the need to develop a software architecture is highlighted, so that certain specific aspects of a computer system that are implemented according to appropriate architectural principles and approaches would decisively determine its main quality attributes.

The significance of abstracting specific details of interaction between the components of a software system that is reflected in its ability to be scaled and extended is emphasized. Event-based architectures are also being detailed, which form the basis for the implementation of modern IT systems with the following goals - to be scalable, extensible and resilient.

Chapter 5 is dedicated to detailing reactive software systems in connection with the use of the actor model. The specific approaches are highlighted in terms of the goals of the thesis that allow the implementation of a scalable, resilient and extensible IT system. *The Reactive Manifesto* is described as a current formalization of the principles according to which a modern reactive software system can be designed.

It is shown that the actor model is a suitable way to implement an extremely scalable, resilient and evolutionary software system.

Chapter 6 includes the detailing of smart systems concepts highlighting the significance of using such approaches in a monitoring system. Again, the emphasis is put on abstracting the context and focusing on the elements that are important in terms of the details of the monitored environment. As an evolved way of abstracting physical reality for an environmental monitoring system, it is proposed to be used *digital twins* approaches and the modern concept of *intelligent digital mesh*.

Given the possibilities and strengths of the actor model, it is proposed to model and implement *digital twins* using actors.

A conceptual software architecture for an environmental monitoring system is proposed and described.

Chapter 7 includes the detailed description of the reference architecture proposed by the author for a scalable environmental monitoring software system. It is a continuation of the previous chapter and includes a detailed description of the levels formulated from the perspective of satisfying the functionalities and requirements stipulated in defining the reference architecture and involving the use of techniques identified during current research.

Chapter 8 is dedicated to the experimental part and the presentation of the results of implementing the proposed software architecture. The chapter includes the presentation of some technical implementation details considered relevant from the perspective of the details developed and proposed within the thesis.

Chapter 2. Software Architecture

Today's computer systems have reached enormous complexities and dimensions, and the main challenges are no longer determined only by algorithms and data structures. Thus, when designing a system, especially involving the use of specialized distributed components, it is necessary to identify and assess effective solutions and approaches specific to the field of applicability related to the architecture of the future system and will determine its evolution and success. In particular, software systems are directly dependent on the software architecture developed for them in the early stages of the project. Lifespan, evolution, efficiency, maintenance, effort through which system changes can be made, etc., depend on the designed architecture.

2.1 Software Architecture Definition

The development of an architecture under different forms of representation represents the basic reference for the implementation of the functional requirements and the certainty of the maintenance in time of the construction in operational form. A complex computer system has a lot of elements, attributes, properties that must be taken into account in the design phase, so an architecture can prove to be successful in the end or vice versa - a total failure.

The emergence of this field coincides with the development of the works of David L. Parnas [PARNAS1972] who describes "modularization" as one of the methods to increase the flexibility of the system on the one hand and to streamline the implementation time on the other. Edsger W. Dijkstra proposed modeling a system by breaking it into segments that communicate with each other through messages [DIJK1968].

Architecture, by this definition, includes details about the *system*, its *organization* and its *fundamental elements* as a formalization (in the sense of abstraction) of specific goals and details. The system always relies on an (*embedded*) architecture, whether or not it has been defined a priori.

The software architecture emphasizes the significance of the structure of the components from the perspective of the interaction between them in terms of principles and rules that will govern the relationship between the components concerned, but also their evolution and, implicitly, the whole system over time [TOGAF]. This evolution involves both modifying the existing components as a result of optimization or of new requirements, and adding new components with distinct roles, which leads us to separate properties of software systems – *extensibility* and *evolution*.

A correct delimitation of the components within a system, as well as a clear and comprehensive definition of their input and output interfaces, streamlines the implementation and maintenance of the system, but also the activities related to extensibility, scalability and its further evolution.

The process of developing the software architecture involves identifying the relevant components and the links between them (**Fig. 2.1**), but the elements that do not affect the system in terms of structure, properties and behavior are avoided. This omission of “less relevant” details implies a certain level of abstraction, which allows us to simplify and manage the process of developing the architecture even for extremely complex systems.

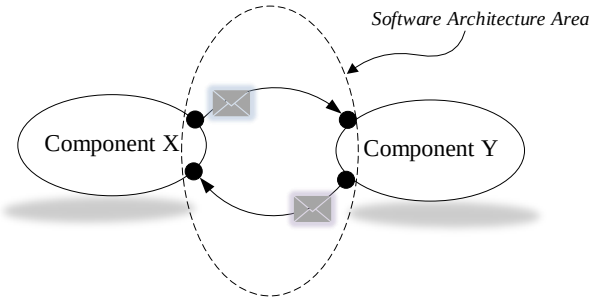

Fig. 2.1 - Software Architecture area

The software architecture represents the description in a certain form of a set of strategic decisions, related to the development of the detailed structure of the software system in terms of elements that abstract the active components of the system, as well as the interaction principles between them in order to outline the general image of a whole system or some relevant part of it.

2.2 Characteristics of a software architecture

A computer system consists of a collection of components that work collaboratively in order to achieve a well-established goal, but which cannot be accomplished by each component independently. The architecture focuses on the elements that facilitate the interaction between these components [BASS2003], while the particular, internal elements that do not influence the exterior of the component are not considered architectural elements.

A software system is considered to meet two types of requirements cumulatively:

1) ***functional*** – those that lead to the achievement of goals, namely what the system must do by implementing the functions with the system behavior. The result of implementing the functional requirements is contained in the system components.
2) ***non-functional*** – those that are not included in the final result but influence it. These requirements are also called *quality attributes* - such as scalability, reliability, availability, maintenance, performance, extensibility, etc.

Non-functional requirements can be grouped into two categories:

- *operational* – those that can be tracked while using the system through various metrics - performance, security etc.
- *evolutive* – scalability, elasticity, flexibility, extensibility etc.

2.3 Software architecture and the context abstractization

2.3.1 Architectural patterns

Software architecture also requires a "creative" aspect, as it is largely based on previous practical experience in designing and implementing software components and systems. Current experience in the field shows that after identifying distinct elements in a sufficiently common granularity within a system, it is very likely that someone has already encountered and found *solutions* to some similar *problems* in a similar *context*. Thus, formalizations on solving a type of problem, which involve performing steps in a certain order and which can be reproduced, have been called *patterns*.

Since in the opinion of the author of this thesis a pattern does not provide *solutions*, but *indications* to reach a certain solution, the definition below can be formulated.

A pattern is a set of reproducible steps, validated and documented in a consistent form, for a common problem in a similar context.

Regardless of the issues covered, a pattern must ensure that the non-functional requirements of the system architecture are met, in order to facilitate the maintenance and

evolution of complex systems. **For an optimal result, the congruence between goals, context and the chosen solution is very important.**

2.3.2 Architectural styles

Unlike system architecture, which is the image of the essential elements of the system and the connections between them, an architectural style represents the formalization of aspects of certain specific architectures [PERRY1992]. The architectural style determines the implementation of a certain set of components and the relationships allowed between those components, thus determining the system behavior. According to [KALE2019], "*an architectural style is based on the logical arrangement of software components*".

Since an architectural style has a more restricted purpose, the general system architecture can include several styles. Architectural styles are formalizations that ignore the hardware part of the system, at the same time being imperceptible for the end user. They are used by those directly involved in the implementation of the components of the software system and, in principle, do not diminish the implementation effort, but at least ensure a conceptual coherence of the technical details of the software system.

2.3.3 Reference architectures

Modern software systems are also built by using dedicated software components. Given their complexity and diversity, the need has arisen to manage the integration and interoperability issues. Thus, formalized approaches have emerged that can be used as a reference. These include *reference architectures* that represent a model or architectural sketch for an entire system in a particular domain [ANGEL2012].

Reference architectures and concrete architectures evolve and are improved as a result of their application in practice. There is a continuous circular relationship between them over time [BASS2003] (**Fig. 2.2**).

In the system evolution process, the effect of the changes from the basic approaches propagates to the final approaches (**Fig. 2.3**).

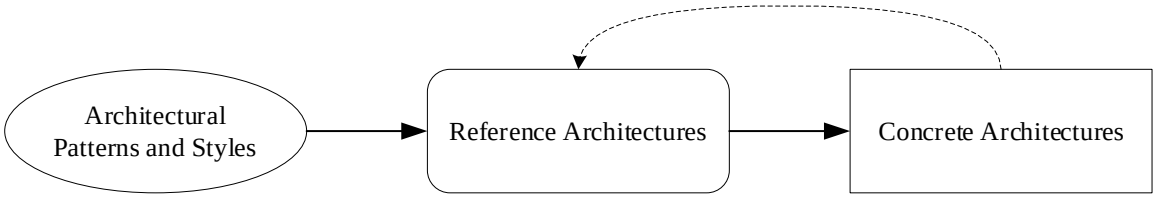

Fig. 2.2 - Circular relation between architectural approaches

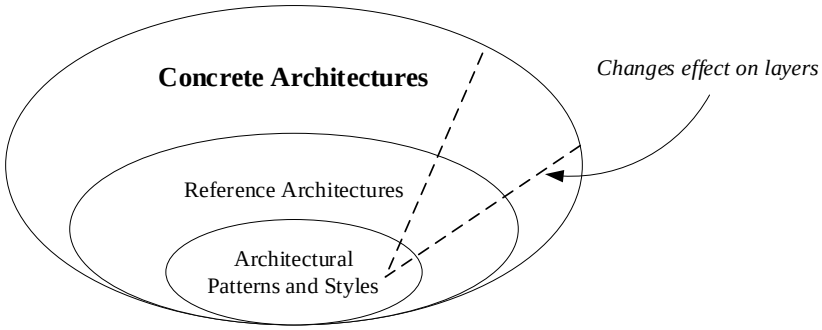

Fig. 2.3 - Architectural approaches and change effect

Chapter 3. Scalable Infrastructures. Distributed Systems in the Era of Cloud Technologies

The efficiency, performance, reliability and many other important characteristics of modern distributed systems are deeply dependent on software components that, on the one hand make possible the interaction between dispersed components, but on the other hand come with a high level of abstraction of computing resources (network and addresses/locations, protocols, threads etc.).

A distributed system represents a collection of autonomous but interconnected components that interact through specific methods and involve sharing resources collaboratively in order to achieve a well-defined goal.

The resources of a distributed system represent the purpose of the interaction between the components. These can be hardware, software or data. The role of the components is to allow the use of system resources in a coherent and transparent way, so that the end user does not perceive that those components are physically and conceptually divided and/or distributed.

3.1 Characteristics of distributed systems

The purpose of distributed systems is clear, namely to achieve a specific goal by using several interconnected components, but they also hide a number of aspects [COUL2001], [TAN2007] "invisible" to the end user, which must be addressed seriously by engineers from the initial phases of the project:

- *Heterogeneity*. Distributed systems consist of several elements, which differ according to the functions and roles they perform within the system.
- *Security*. The number of elements and vulnerability points in a distributed system is directly proportional to the number of distributed components of which the system is composed.
- *Reliability and Fault Tolerance.* The lack of a single possible point of failure represents a major advantage of distributed systems, but it increases the complexity of system deployment and maintenance, especially given the heterogeneity aspect of components.
- *Extensibility*. Another major feature of distributed systems is the ability to replace the existing components, to completely abandon some or vice versa - to add others with new features and roles. This feature underpins the concept of *Open Systems*, which implies that a system can be extended and modified in multiple ways [TOVARNITCHI2012]. Open systems contribute to the creation of more complex systems – systems of systems [TOVARNITCHI2012], facilitating the use of heterogeneous systems.
- *Scalability and Performance.* Performance is among the main features pursued and observable in the system operation, representing a significant quality attribute for all software applications [TOVARNITCHI2021].
- *Concurrency and Resource Sharing*. The concept of distributed system, by default, assumes the reuse of components or functionalities of components by several "beneficiaries", but this poses some challenges – it must be ensured that the elements concerned can be used *concurrently*.
- *Transparency*. Transparency refers to the "masking" of physical localization and specific properties of the components in relation to the "beneficiaries" (components, external systems, etc.).

  Other aspects of distributed systems where transparency can be applied are:
  - *Transparency of scalability* [COUL2001].
  - *Transparency of concurrency* [TAN2007].

3.2 Applications virtualization through Containerization. Functions virtualization: Serverless solutions

A particular modern form of virtualization is *containerization*, which is materialized through the use of so-called *containers*. The approach involves the virtualization of the operating system, which technically involves sharing the kernel of the operating system [SCHOLL2019].

Experience of current practices of applications development proves that final results are much better when programmers focus on writing the code, applying or developing algorithms or techniques to solve the problem in the best and most efficient way, without worrying about the operational aspects of the software components developed.

One of the current techniques for the fast delivery of such scalable infrastructure services is called *Serverless Computing*, where resources are fully managed by the service providers and mostly automatically. The basic idea is represented by the separation between the application itself, as a result of the development effort, and the operating part of the necessary infrastructure. Platform independence, implicit and non-intrusive scaling, with an emphasis on the optimal use of computing resources, are the elements that characterize this technique.

The serverless technique comes with an additional level of abstraction by separating the code execution (in the meaning of final result) from the location where it is actually executed. That model can be likened to a reactive architecture (discussed in **Chapter 5**) based on events (discussed in **Chapter 4**) [SCHOLL2019], where the function execution is triggered by a specific event via remote API call [SHAHRAD2019].

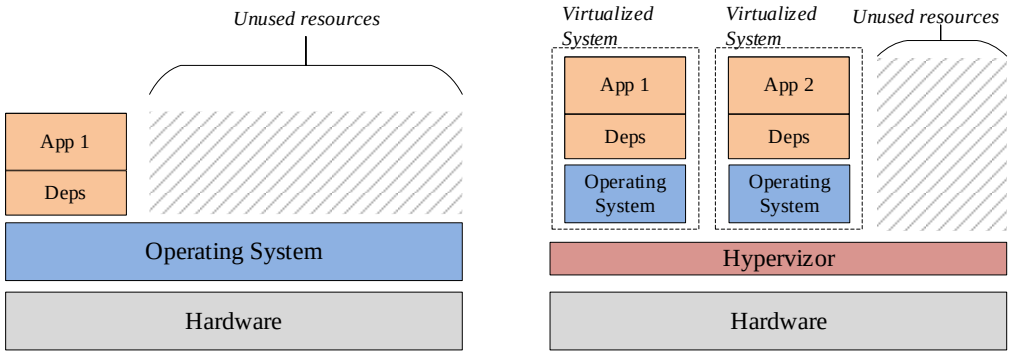

Fig. 3.1 – System virtualization: traditional system (left) versus virtualized system (right)

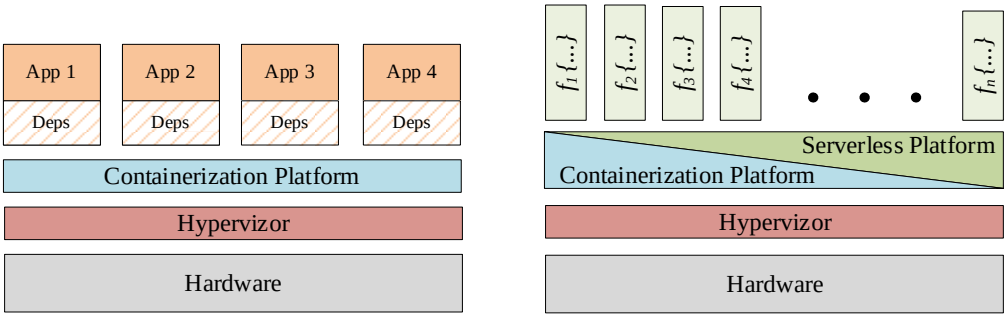

Fig. 3.2 – Application virtualization: containers (left) and serverless (right)

3.3 Cloud-Native Architectures

The design of information systems has been profoundly influenced by the increasing complexity of business requirements, which can be seen in the way they have been implemented.

Recently, in the context given by the huge diversity of digital technologies, a new concept was formalized - *Cloud-Native Applications* (maintained by *Cloud Native Computing Foundation*[1]), which involves a set of features - technical and architectural - that must meet a computer solution in order to provide an unitary operation mode, regardless of the platform on which they run. Such applications can be run identically in public, private, or hybrid cloud solutions.

The cloud-native approach is considered to be a technique that unifies the software architecture with the system architecture.

3.3.1 Green Computing

Classical methods of provisioning computing resources involve their a priori "reservation" and, most often, their continuous operation, even if they are not actually fully used. According to [BORAH2015] , the average annual use is 10%, which is extremely inefficient.

One issue that is being taken seriously today is the *Green Computing* approach - the efficient use of resources and therefore of the energy – the decrease in CO_2 and other greenhouse gases – which contributes to slowing the global warming. An extremely significant element, since it is estimated that the IT&C industry is responsible for 2% of total CO_2 emissions and for about 3% of global energy consumption [WEBB2008]. Recent studies show that by 2025, 4.5% of global energy consumption will go to data centers[2]. Being aware of the importance of the consequences of increasing energy consumption, major Cloud service providers (Google[3], Microsoft[4], Amazon[5] etc.) are moving towards the use of energy from renewable sources, undertaking to decrease the CO_2 footprint.

Therefore, the result of applying cloud-native techniques has an extremely significant impact both economically, technically, operationally and ecologically.

[1] https://www.cncf.io/ (accessed on March 2021)
[2] https://parkardigital.com/greener-computing-with-serverless-architecture/ (accessed on March 2021)
[3] https://www.google.com/about/datacenters/renewable/ (accessed on March 2021)
[4] https://blogs.microsoft.com/blog/2020/01/16/microsoft-will-be-carbon-negative-by-2030/ (accessed on March 2021)
[5] https://sustainability.aboutamazon.com/environment/sustainable-operations/renewable-energy (accessed on March 2021)

Chapter 4. Software Systems Scalability

A significant attribute and noticeable by the users of a computer system is the system performance. This can be improved in several ways, but one way is to scale the system – resources or processes.

4.1 Software Scalability

Scalability is an "ability" of the system that is an important attribute of computer systems, often mentioned and undertaken when it comes to their characterization and/or description, especially its significance has increased in the context of modern systems – determined by ever-increasing size and requirements. Scalability determines performance, but also has an important role in ensuring the reliability, quality of services and extensibility of systems.

A software system is considered to be scalable, if, following the increase in the demand for functionalities supported by the constituent components, it continues to work properly.

4.1.1 Vertical and horizontal scaling

Vertical scaling involves increasing the volume of computing resources involved (such as processors/cores, memory, storage space) per computing unit (either physical or virtual). This approach can also be called *monolithic* or having a *static* character.

Horizontal scaling involves altering the system structure in the meaning of adding or removing one or more new nodes (physical or virtual machines / computing units), an action that will be transparent for the end user, that's why we can call it *dynamic*.

Horizontal scaling is performed by using distributed resources, which reduces the degree of dependency towards possible errors that may occur during the execution of certain operations within the system.

Vertical scaling	Horizontal scaling
Mainly used as a support for monolithic software systems	Allows the use "on demand" of the necessary resources for distributed software components
High performance, consistent data	Acceptable performance, inconsistent data
Local communication (between processes)	Remote communication (network calls)
High probability of error in case of malfunctions (single point of failure)	Possibility to implement techniques in order to increase the resilience of the system when encountering failures
Limited scaling	Scaling (theoretically) unlimited

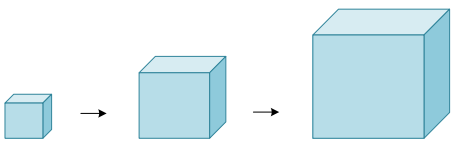

Fig. 4.1 - Vertical scalability

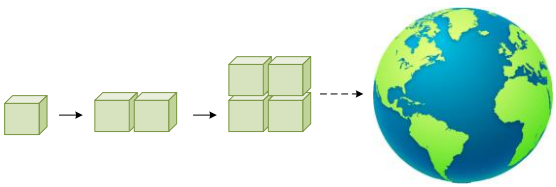

Fig. 4.2 - Horizontal scalability – (theoretically) unlimited

Scalability must be addressed in the design and development phases of the system architecture, because later changes of the level of system scalability, if possible – only with a need for additional resources (time, developers and other details with financial impact). It is recommended that in the analysis phase of the quality attributes of the future system, if scalability is identified as a relevant criteria, to identify in the system those resources that can later be resized and the system does not explicitly imply the use of that resources with a fixed size [BASS2013].

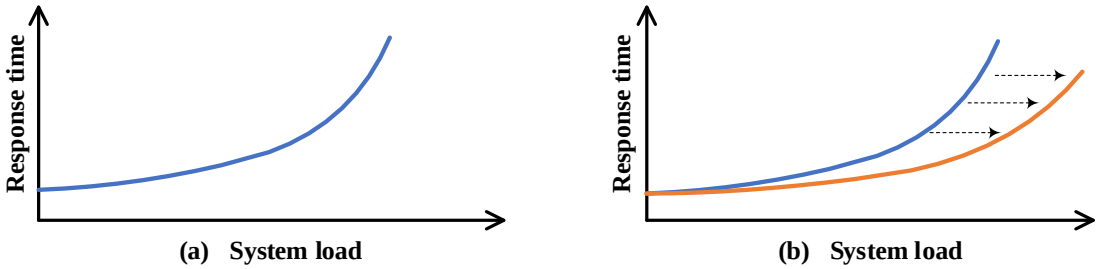

Fig. 4.3 - System load – response time relation

As the demand on the system increases, so does its response time - the time when requests are processed (**Fig. 4.3 / a**). This increase becomes steep at some point, at which point the system performance decreases significantly, and may lead to situations where some requests are processed late, with errors or not at all.

By contrast, in case of system scaling, its capacity increases in terms of its ability to cope with an increased number of demands (**Fig. 4.3 / b**), represented by moving the curve that reveals the demand / time-response relationship to the right and maintaining the system operation within acceptable parameters.

4.2 Scalability in 3D format

One of the limitations of a monolithic software system is its scalability. **The limitations of an information system are included in its architecture - the way how the constitutive components are designed and defined, as well as the integration between them.**

Abbott and Fisher popularized a three-dimensional scalability approach - x, y, z - each based on distinct scalability features. Formally, this model was called the *Scale Cube* [ABBOTT2009].

Following the approach proposed in the *Scalability Cube model*, we can say that scaling methods are also multiple, and the scalability of a computer system depends directly on its architecture (see **Chapter 2**). Each of the three scalability directions can be applied simultaneously, starting from the goals pursued and taking into account the advantages and disadvantages of each method. Thus, in case of scalability on the *z axis*, we can have multiple identical instances - scalable on the *x axis*, and the system in this case is scalable on the *(x, z)* direction. We can have instances of microservices (specialized) - scaling on the *y axis*, simultaneously with replicated instances of the same microservices, but dedicated for separate sets of data - scaling on the *z axis*, and the system in this case we consider to be scalable on the *(y, z)* direction. The point with the coordinates *(0, 0, 0)* is the representation of a system that is not scalable at all; the opposite point, on the other hand, represents the "infinite" scalability – we can say that the system is scalable on all axes.

4.3 Scalability and integration of the distributed components

The basic factors that differentiate the numerous architectures of computer systems, basically lie in the particularities of integration between components, and in case of an implemented computer system, the efficiency of integration between components largely determines its

essential quality attributes, whereas the operation of the components within the system depends on the effectiveness of "collaboration" between them. The significance of the integration aspect between the system components, but also the integration with external components, is given by the specialization on the components functionalities and, implicitly, the increase in their heterogeneity.

From architectural point of view, the integration between the components of a system is aimed at cooperative exchange of data between them in various forms and refer to the collaboration between those, represented by: (i) the connections between the components (in the sense of dependency), (ii) the interaction and (iii) restrictions (in the meaning of rules) on communication between those components.

Connections (i) between components can be of two types:

1. *Static*. We call the connections static, when the relationship between the components is explicitly defined in the system implementation. In such relationships, the dependency between components can only be changed by altering the application.
2. *Dynamic*. Dynamic we call the connections that can be changed at any time during execution. Such links are mainly used in distributed environment. Dynamic connections increase the flexibility of the system through independent operation and separate modification of components.

Interaction (ii) between components can be done in two ways:

1. *Synchronous*. It assumes that both components are active at the same time, and the initiating component of the interaction remains "pending" until it receives a response from the other component. This method also involves a temporal dependency between the components.
2. *Asynchronous*. The interaction between the components can be done separately in time. This mode allows the parallel execution of tasks, decoupling components in time, increasing the system performance and robustness.

Restrictions (iii) on communication between the components can be various and are primarily related to the specifics of the functional aspects of the system, but also to the connections and the interaction type between the components. These are reflected in the protocols and/or semantics through which the communication takes place.

Interoperability can be considered to have two aspects, respectively:

1. *Syntactic.* Details of the structure and format of the messages must be agreed by the components involved in the communication.
2. *Semantic.* It is important to correctly interpret the content and to understand the meaning of the message in a particular context.

The components or modules of a computer system have the role of embedding certain functionalities. This undertakes that the details of internal logic are kept hidden, so from this perspective, from the outside, the components are seen as black boxes and the functionalities are exposed to the outside by calling the *interfaces*.

The components of a distributed system can be considered as being **distributed in space**.

In case of asynchronous interaction, when the message processing can be done differently in time, we can say that we have components **distributed both in space and in time**.

The interface of a component represents the syntactic specification of inputs and outputs.

From technical point of view, the interfaces represent structural elements of the components, having the role of entry/exit gates with clear and exact specifications, which can be identified and located and facilitate the integration of components through collaborative exchange of data/messages according to the well-known and mutually agreed rules.

4.4 Events as triggers of the interaction between components and changes indicators

As an alternative to the explicit invocation [GARLAN1993], such as the *request-response* method, Garlan and Shaw mention another integration technique, which they call implicit invocation, based, on one hand, on "announcing" one or more events, and on the other hand the "announcement of interest" in the form of *subscriptions* to that type of events.

An event represents an abstraction of an accomplished fact that signals the quantifiable change in the state of an entity – either physical or digital – that occurs in a certain context.

Events can be regarded as ways of discreet representation of signals in terms of changes in a certain environment (in real, but also in digital world). Thus, the relevant details regarding the context of the changes are modeled as digital entities and transmitted in certain forms of representation as notifications to stakeholders.

4.4.1 Event-based architectures

Most characteristics of information systems, in order to meet the required needs of users, are determined by the quality attributes of the system, which depend directly on its architecture.

By implementing the communication logic around events (being essentially asynchronous), each component can perform its tasks in the best possible way, independently and without being blocked, and in case of errors, those are isolated locally.

Systems with an event-based architecture are composed of decoupled components, usually with rather narrow and well-defined purposes and which receive and process events asynchronously [RICHARDS2015].

The loose coupling that is provided by the event-based interaction approach comes with a number of advantages including an increased system scalability.

As the consumer's actions are reactions to certain events signaled by the manufacturer, they represent the basis of a reactive behavior (see **Chapter 5**). Reactive systems are considered to have event-based architectures [LAIGNER2020].

Event-based interaction is one of the essential methods for exchanging information in the context of Cloud technologies, ensuring that solutions are more agile and scalable.

Software architectures, where the specific actions of the components are triggered in response to an event or series of events, are called event-based software architectures.

Event-emitting components work independently from consumers, and don't even know their recipient. **Events facilitate the decoupling in time and space of the components involved in communication**, and their processing is not conditioned by interaction [DEBS2006]. The result will be scalable and extensible distributed systems, regardless of complexity.

4.4.2 Challenges raised by event-based architectural approaches

- ***Data consistency*.** Asynchronous event processing and loose coupling of components leads to low mutual control of the interaction, which can have negative consequences on the proper operation of the system from the perspective of the correct handling of events.
- ***Architectural complexity*.** Reliable processing of all events involves the use of several types of specialized components, each having a separate role throughout their life cycle.
- ***Interoperability*.** Given the heterogeneity of the components used in the architecture of such a system, it is necessary to take a series of approaches that, starting from the context, facilitate the cooperation between the components involved in the interaction in the easiest way.
- ***Non-determinism*.** Event-based systems are also considered reactive systems [LAIGNER2020], and since later are characterized as being inherently non-deterministic [KAISLER2005], we can say that event-based systems also come with a high level of non-determinism.
- ***Resources calibration*.** In the process of operating an event-based system, there are two unknown details, the solutions for which, in fact, also represent some of the significant advantages of the system architecture. The first refers to the fact that the volume or flow of the message perceived by the system cannot be determined a priori, in design phase. However, when designing the system architecture, taking into account these details that are difficult to estimate, it is necessary that the modules in the application that can be directly affected, to be designed by taking into account techniques such as ***elasticity*** and ***scalability***. The second unknown is given by the evolution of the system over time - ***evolutionary architectures***, determined by the emergence of new types of events that entail the need for new features in the system to be treated properly. This need is covered by a feature that we will call ***extensibility***, which involves the ***system evolution*** in terms of adding new features to the system or modifying the existing ones.

4.4.3 Event-based interoperability between distributed components. Integrity and consistency of data/information

An important detail to consider when designing the integration is the use of a reusable structure in terms of the semantics of the information involved in the interaction between the components. This structure makes possible the easy and flexible integration and collaboration between all the components within the system. Representing a form agreed by all parties involved in the communication, it can lead to a fluent collaboration, absolving us of the need

to manage multiple data structures. The use of a standardized form decreases the time for taking decisions and taking action, without the need to translate the signaling structures of the events issued by the producer. We can obtain a smaller volume of data transferred through the communication channel and an optimized use of storage space if they require persistence.

In an environmental monitoring system, the essential emphasis is focused on observing the dynamics of the components of which it is formed as a whole. Reformulated, the monitoring involves observing changes over time of certain attributes or properties of some components. The way of signaling the changes is abstracted in the form of issuing events as a way of presenting the relevant information about what happened. From this perspective, the event is the signaling of a change that took place in the past, of an accomplished fact, which must be accepted and treated as such by triggering a (re)action or ignored.

An event in itself, strictly as a carrier of information, whatever its significance, has no value, thus the details of the context in which it took place are important. The context is represented, first of all, by the *moment* and the *place* where the event was produced [CRISTEA2013]. These meta-data are essential in a monitoring system for placing the event in *time* and *space*.

Place of occurring (*space*) of the event is represented by the identifier of the event producer (which can be a reference, a unique name etc.).

Moment of time (*time*) in which the event happened is provided by the producer or, in the absence of value, by convention, is considered the time when it is received by the consumer.

A significant attribute of the event is its **unique identifier**. Considering that an event crosses several components until it is "consumed", it is essential to be able to identify it accurately and uniquely.

Another indispensable attribute for event distribution and processing is the **type of event**. Events of the same type have the same structure and semantics and usually signal the same type of changes.

In order to represent the details of the event, it has been proposed for use the ***CloudEvents***[6] specification. A unitary specification, such as CloudEvents, facilitates the interoperability between the distributed software components by providing a representation scheme with a common structure for consumers and event producers, disregarding the technology used for implementation. In the context of the large number of Cloud services offers and the huge diversity of dedicated software components and solutions, a common approach to event representation, routing and treatment allows the simplified integration of distributed software components, which is agnostic to the platform.

[6] https://cloudevents.io/ (accessed on March 2021)

Chapter 5. Reactive Software Architectures

The concept of ***reactive system*** was proposed as an approach [HAREL1985] that encompasses components with two distinct aspects – behavioral and implementation, and unlike classical (regular) systems, called transformational, which operate on input/output principle, ”*reactive systems are repeatedly prompted by the outside world and their role is to continuously respond to external inputs*”.

This initial definition, however, was not limited to a digital system. Later, Gérard Berry extended this concept in the context of software systems [BERRY1989]. According to him, three types of applications are differentiated:

1. *Transformational* – the input data is read and processed, and then the results are returned;
2. *Interactive* – they interact with users or other applications at their own pace;
3. *Reactive* – those that maintain a continuous interaction with its environment, but at the pace dictated by the environment and not by the application itself.

Reactive applications must be "able" (in the meaning of appropriate technical measures) to support a variable number of tasks due to the unpredictable volume of data involved. Real-time applications can generally be considered as reactive applications.

Since the vast majority of modern software systems involve their simultaneous use by multiple users, we are dealing with competing/simultaneous activities. Technically, these activities require a concurrent demand for system resources and depend directly on their availability. As an architectural approach, as mentioned above (see **Chapter 4**), it is advices to use techniques that facilitate the scalability of the system. Concurrency can be of two types:

1. *Local* – at the level of a single component;
2. *Distributed* – involves the interaction generated by processes running within two or more independent components.

5.1 The Reactive Manifesto

In order to satisfy (somehow) the current requirements in the design and implementation of modern software systems, a series of principles have been proposed according to which a ”*robust, resilient and flexible system*” [REACTMAN], can be built, all being systematized into a specification called *The Reactive Manifesto*.

The basic principles of this manifesto, confirmed, moreover, each separately as essential in any system, are:

1. *Responsiveness* – a series of measures must be taken to keep the system available so that its functionalities are fulfilled within a given time frame.
2. *Resiliency* – in case of some problems, the system must remain responsive without the user being aware of those problems.
3. *Elasticity* – in case of a variable flow of requests to the system, it may react by proportionally changing the volume of resources required to fulfill the tasks.
4. *Message-driven* – by applying the method of asynchronous communication through messages, the components ensure an increased autonomy from each other, but at the same time facilitate application of the principles of isolation and location transparency. Decoupling components and message-based communication increase the level of scalability, flexibility and robustness of the system, facilitating more granular control over its elements and interoperability between components.

A system that meets the above criteria is called ***Reactive System***.

Through a controlled latency a system can be considered *receptive*. The architectural aspects through which the receptivity of a system can be ensured are *resiliency* and *elasticity*.

These two characteristics are interdependent, but both are determined by the interaction type. Thus, an asynchronous interaction, message-driven, is the basis of a reactive system.

Reactive systems can be considered as the implementation of a particular architectural style, which involves dividing the system into several components, but which work collaboratively and coherently as a whole, are receptive to environment stimuli and elastic to their variable volume, as well as resilient in case of possible errors.

Reactive architectures refer to the reactive systems design process.

”Reactive Principles” represent a particular set of technical, primarily architectural, approaches that facilitate the construction of coherent systems in heterogeneous distributed environments. The architecture of a reactive software system facilitates the implementation of a complex system, composed of a large number of components with distinct functionalities, but which work collaboratively as a whole and which react in a coordinated way to stimuli from the external environment. A reactive system remains functional in situations of rapid increase of ingested data volumes, the necessary computing resources being ensured by specific scaling techniques.

5.2 Actor Model

The *actor model* has been formalized in the form of a methodology that can be used as a theoretical and practical tool to address complex issues of concurrency in computing systems [HEWITT1973]. An *actor* represents the universal computing primitive unit with which the actor model operates.

For software systems that require functionalities involving concurrent or parallel activities, or interactions in distributed, scalable environment, the actor model provides the theoretical support needed to abstract these aspects, which significantly simplifies not only their implementation, but also their maintenance and evolution.

5.2.1 Actors versus Objects as paradigms in programming

In Object Oriented Programming (OOP) the fundamental units of computing are modeled as objects. In the approach proposed by Carl Hewitt, when we design a system or, at a more granular level, some modules/components according to the Actor Model principles, “anything represents an actor”. Objects interact through the explicit call of the method that exposes the necessary functionality, can expose properties (internal variables) to be read or modified from the outside. Actors, in contrast to objects, interact exclusively through the exchange of messages, they do not directly expose their internal state to be read or modified from the outside, but only through messages and as a result of decisions taken internally based on the current behavior of the actor, which, however, may change [AGHA1985].

5.2.2 The actor as fundamental unit of computing

As a computing unit, an actor consists of the following elements:

- *Processing* – since it has certain tasks to fulfill;
- *State* – must be able to memorize certain details, which are used in processing;
- *Communication* – necessary from the perspective of interaction with other computing units.

In particular, an actor consists of mailbox, status, behavior, supervision strategy and references to child actors.

The state of an actor at a given time is represented by the values of the internal variables at that time, which are related to the internal logic and determine the **behavior of the actor** or, in other words, how the next message will be processed. There is no competition within an actor, which ensures that his state can only be altered by that actor, which guarantees a **consistent behavior**.

5.2.3 Location transparency and interaction between actors

Physically, the components can be located locally or remotely, the interaction being made strictly by involving specific network elements. Logically, depending on certain domain-specific criteria, components can be considered local, even if they are physically remote, or remote, even if they are physically local (next to). Location transparency refers to the ability to communicate with a component in the same way, regardless of its location.

The interaction between actors is achieved exclusively through the asynchronous transfer of messages, and this is achieved only if the address of the recipient is known. Unique identifiers are used for actor's localization, generally valid and interpretable within the system or cluster. The physical location is not revealed or relevant, which provides a number of advantages already mentioned in distributed systems. Thus, the communication between actors is done by using the same semantics, regardless of their location.

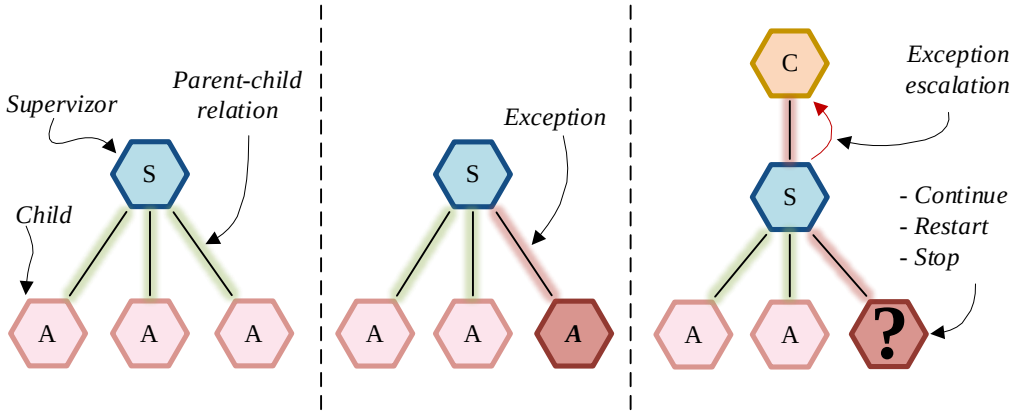

Fig. 5.1 - Actors hierarchies. Supervision and interaction

5.2.4 Resiliency

If the system faces certain situations that generate errors that it cannot deal with, then it becomes unstable, having negative effects on a whole series of important attributes, such as performance, level of quality of service, etc. In order to prevent such situations, certain measures are used to provide the resiliency and a high level of fault tolerance.

Resiliency has an approach that primarily involves receptivity to errors. In essence, a resilient system, in the case of an issue with a component, tries to "resuscitate" it, and if this is not possible, instantiates a new component that is able to take over its tasks.

Resiliency is part of the properties of the system related to its architecture, thus the necessary measures must be explicitly taken in the architecture development phase.

Chapter 6. Intelligent Environmental Monitoring

Worrying state of the environment, through its accelerated degradation and the consequent effects, directly influences our health [KUM2013] (both physical and mental), climate, ecosystems and the interaction between them.

The environment is presented as an extremely complex system, with subsystems with very sophisticated interactions, with a multitude of indicators and variable parameters that, in their turn, are determined by a lot of factors. Monitoring solutions must allow the interpretation, in various forms, and the exact understanding of the causes of the events and phenomena monitored, which is the key element in taking decisions for possible actions to correct, prevent or remove certain unwanted events and/or phenomena.

6.1 Digital Twin in Intelligent Systems

The advancement and maturation of digital technologies in the last decade have created favorable conditions for the design and implementation of information systems that involve interaction with objects and beings from the real world. These systems are called *Smart*. „*The most essential feature of smart environments is the ability to control devices remotely, even automatically.*" [COOK2005]

Environmental monitoring is characterized by the use of techniques, methods and technologies specific to the concerned environment, but all are based on the identification of phenomena in time and space. By integrating digital technologies in such processes, it results into the creation of so-called cyber-physical systems [DUMITRACHE2017]. Thus, accurate measurements can be performed in near real time, at a high level of automation, as well as by applying artificial intelligence techniques [CURRY2020].

The characteristic aspects that make a system smart (or even intelligent) and autonomous, inevitably increase its complexity, but, as shown above, in order to efficiently manage these details in all phases of system development - design, implementation, operationalization - the solution is the appropriate modularization of functionalities, an activity that must be taken into account starting with the design of the system architecture. One of the essential approaches in this regard is the *Digital Twin*, where the key element consists in data and specific techniques/methods of handling it.

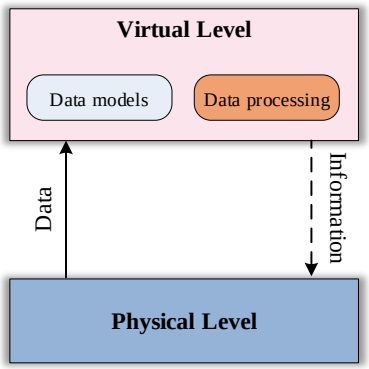

Fig. 6.1 - Digital twin model

As a general-valid approach, the digital twin involves the interconnection of the virtual world with the physical one through the digital representation of an object in the real world [GRIEVES2014]. From the perspective of environmental monitoring, the digital twin represents the image of a physical device, as an element of abstraction of the main characteristics that form the scope of observation or manipulation. That representation is used to perceive, interpret and operate the state of the modeled device.

Th. Borangiu et al. in the paper [BORANGIU2019] provides a concise and comprehensive description of the concept of digital twin: "*The holistic view of the real capabilities, status and features of an entity (equipment, process, product) including its digital representation, execution context, history of behavior, time evolution and status can be encapsulated in a digital twin – defined as an extended virtual model of a physical asset, process, product, that is persistent even if its physical counterpart is not always on line/connected; this extended digital model can be shared in a cloud database with other entities.*"

Digital twins are used to abstract four-dimensional physical reality that differ in purpose or granularity:

1) objects, components, basic devices – the individual digital representation of an entity in a system;
2) complex devices or components – these can be seen as an aggregate structure, being composed of several elements of type (1), but as a whole one from the perspective of their operation;
3) processes – which are intended for the digital representation of the behavior of modeled entities and can be seen as a representation of their dynamics in time and space, but also being dependent on context;
4) environments or systems and systems of systems – the most complex, but also complete way of digital representation of an environment, through which its holistic image is built.

The digital twin is the digital representation of an entity/object or of a group of entities/objects from the real world by representing accurately and updated in real time, in both directions, the values of the main attributes and characteristics pursued in the monitoring process.

Digital twins are the interface between the real world and the digital world, providing a continuous connection, possibly in real time, between these two worlds, due to which the border between them becomes even less noticeable.

In the proposed system architecture, the roles of the digital twin are:

- to be the "image" of the physical device in the modeled system by the exact reflection of its status, represented by the last known values of the pursued attributes;
- to represent a single reference/access point in the context of accessing the values of the attributes both in terms of the current status (last values) and historical values;
- to represent a single point of contact if you want to interact with that device;
- to ensure the interaction between the physical device "from the field" with the digital part of the system;
- to react to specific events produced by the physical device;
- to complete the functionalities of the real devices by taking over and executing some tasks within the internally implemented functions or by the intermediate transfer/offload [TAPUS2013] to other specialized components either from inside the system or from outside;
- actioning of real devices in response to events received from them or the retransmission of commands to them as a result of decisions taken or events launched by other components involved in certain specific data flows.

Being a digital entity, the digital twin can be managed extremely easily like any other type of digital entity. Thus, it can be created, recreated and destroyed, persisted, moved or copied to another remote system. Unlike real entities, their digital replicas can be accessed concurrently by other components, and in addition access control policies or other specific methods can be applied to them in order to manage them in a controlled and secure way.

The digital replica of any device can be used to take over and perform some tasks and to expand its capabilities in terms of data processing and integration with other components. This approach is an important advantage from the device point of view, as its energy consumption can be considerably reduced [TAPUS2013], [TAPUS2019]. At the same time,

tasks are performed on servers where processing power is not an issue and in a controlled way by the logic implemented in the digital twin. From the perspective of supporting digital services and resources in this regard, Cloud technologies provide the necessary solutions – allow the operationalization of scalable, resilient, high-performance, extensible, robust, flexible solutions.

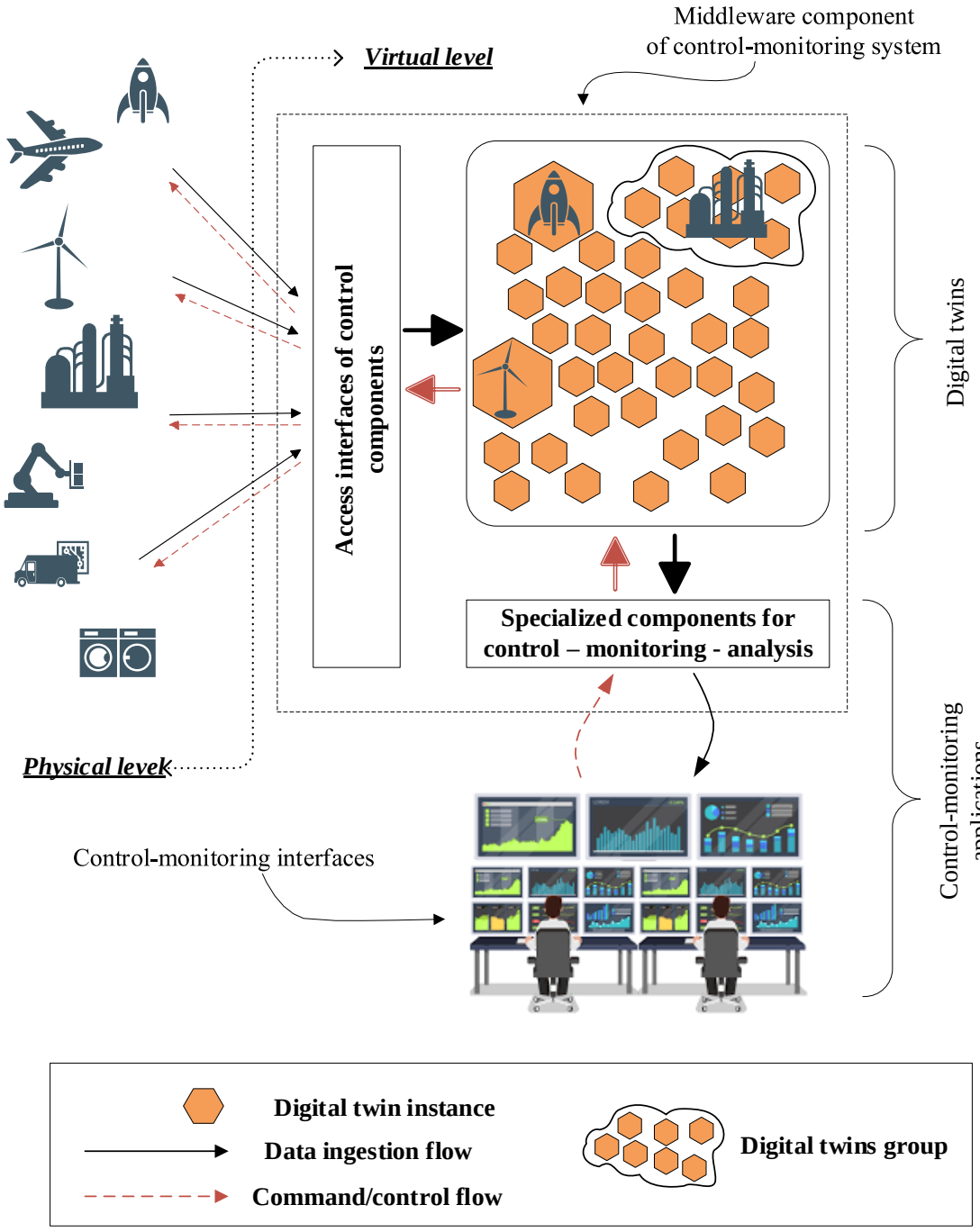

Fig. 6.2 - Conceptual architecture of a monitoring system which implies using of digital twins

6.2 Scalable system for intelligent environmental monitoring

The architecture of a system represents the view of the system architect on the environment but depending on a certain context. In order to be implemented, however, it must include the essential elements from the perspective of the specific issues of the given context, in a coherent and unitary way.

Automated monitoring processes require a digital infrastructure with an increased focus on aspects of efficient connectivity and interoperability between constitutive components. Thus, the architecture proposed for the monitoring system contains a set of specialized components that pass messages between each other, according to rules determined either by external factors (configurations, specific details present in the message structure, etc.) or by the primary purpose of the component, (for example persistence).

In current approach, a message can have one the following purposes:

- *State tracking* – the message is used to retrieve relevant information and to keep it as a "current value" or "last known value";
- *Transient* – the message may be distributed to a component external to the system for some particular purpose – the message leaves the system and is transmitted to another one;
- *Persistency* – the message is captured in order to be saved (usually in the database) for future operations.

The message can represent a command or to signal the occurrence of an event, which will trigger the execution of certain operations related to the functionalities of that component. The basic method of interaction between components is the asynchronous transfer of messages. However, depending on the specifics of the functionality, the message may also go through some steps that run synchronously.

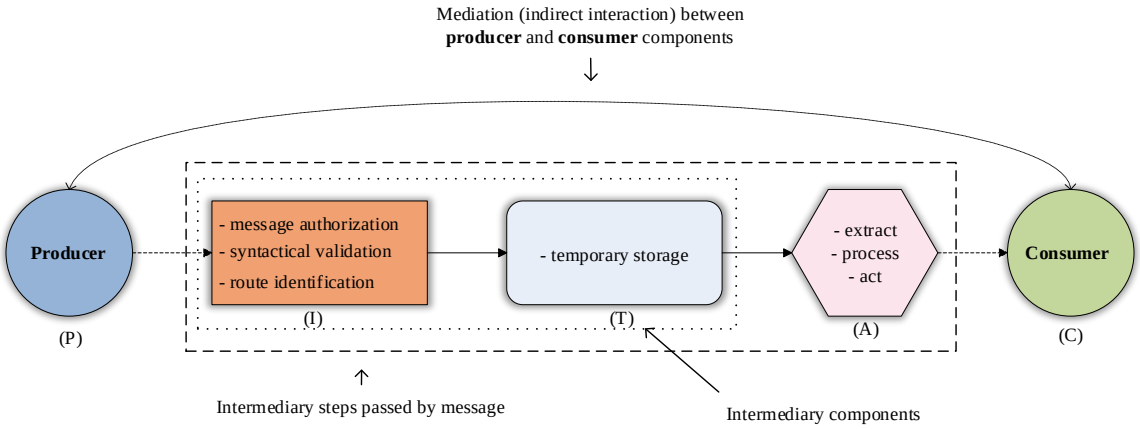

Fig. 6.3 – Modules of a monitoring system

From the perspective of the proposed architecture, we have the following general types of components:

- *Message producers* (P) – the message can be generated either by an external component to the system or by an internal one;
- *Message ingestion* (I) – those are responsible for such actions as component authorization (if applicable), syntactic validation of the message, redirecting to a temporary location;
- *Temporary storage* (T) – components optimized for temporary and orderly message storage. A significant goal of these components, among others, is to decongest/depressurize the flow of messages;

- *Active components* (A) – represents the "brain" of the system or the "smart/intelligent" part of it. Within these components, message processing or the supervision/management of message processing takes place if this aspect is delegated to specialized component;
- *Message consumer* (C) – they can be represented both by internal and external components. In case of external components, the messages can also act as instructions/commands for the actioning/triggering purpose.

6.2.1 Requirements for an environmental monitoring system

Below are listed the characteristics, set out by the author of this thesis and in [TOVARNITCHI2016], which must satisfy an intelligent distributed environmental monitoring system. The characteristics were also detailed in the works [TOVARNITCHI2017], [TOVARNITCHI2019A], [TOVARNITCHI2019B], [TOVARNITCHI2021].

Essential characteristics:

- provide flexible and efficient solutions for interconnecting distributed measuring devices (sensors) or actuators to central entities (servers);
- use efficient and reliable communication techniques, but standardized, between heterogeneous components, adapted to certain environments and needs;
- provide support for the storage, processing and analysis of a large volume of data, through the use of efficient algorithms and dedicated software components;
- is able to operate with a certain level of autonomy. The system can be set up so that in some situations it behaves according to certain rules, with exact successions of steps, depending on certain situations, without the need for intervention by the human operator;
- is able to operate at least reactively. Depending on the data that reaches the system, but also on other conditions, it can call other systems or can act remote devices;
- is able to integrate with other systems. It must expose software interfaces (APIs) through which it can be integrated with other systems according to strict and well-defined rules in order to benefit from the functionalities of other systems or to expose its own functionalities and data (see SaaS or PaaS paradigm - depending on the context). Integration methods are currently a very important issue, given that nowadays closed systems are less preferred compared to those that allow extensibility (open systems) [TOVARNITCHI2012];
- provide the possibility to use the services of data analysis systems in order to be able to harness the data and to provide support for the decisions taken by the system either automatically or not;
- provide visual interfaces through which a human operator can intervene upon the system or can monitor the situations that occur in the system in real time or after the occurrence in the system of some events;
- can cope with the variable needs of computing resources - scalability. The number of distributed components that produce or require data may increase or decrease rapidly [TOVARNITCHI2021].

Chapter 7. Proposal of scalable software architecture for an intelligent environmental monitoring system

In the software design stage, the basic concern involves the mandatory completion of the steps related to the identification of software components and the delimitation of their responsibilities. These steps decisively determine the deduction and formalization of the interactions between the identified components, as well as the way how the respective connections are to be handled so as to achieve the maximum coherence between components and with direct effect on achieving the ultimate goal for which the system is designed and implemented. Thus, depending on the responsibilities of each identified component in the system, appropriate patterns and principles will be selected that will govern the relationships between those components in an optimal way and will form the basis for a flexible, evolving, robust system.

Depending on the **origin**, **destination/purpose** and **direction** (sometimes) relative to the system - input or output - messages can be grouped into two types: *events* and *actions*.

An **event** is an accomplished fact, which refers to the past, upon which we cannot intervene, and we must treat it as such. Events can be related to a fact both outside and inside the system. They can act as a stimulus or trigger.

An **action** is an intention, an activity that is to take place in the future. It has an imperative role and can be considered a command - consequences of internal decisions taken as a result of the events received.

The base entities (in the meaning of components) with which a monitoring system operates can fall into one of the following two categories:

- *producers*
- *consumers*

7.1 Interaction and data flows in distributed software systems

The components of a system can communicate directly with each other or through mediators – – intermediate components, with delimited roles and dedicated to aspects that are related to the semantics and syntax of communication, but also to the particularities related to: communication dynamics, restrictions imposed by the system architecture, infrastructure features, etc.

This system uses two types of mediating components:

1) Event Gateway;
2) Message Queues.

7.2 Event Gateway

Exposing a series of particular functionalities on one hand, and on the other hand incapsulating and hiding specific implementation details, is a common and necessary approach in the context of distributed system architectures.

The implemented Event Gateway component has the following features:

- operates on the basis of *HTTP*/*HTTPS* protocols or others compatible with them (e.g. *CoAP*);
- the supported interaction is based on the specifications of the REST approach [FIELD2000];
- the interaction is of request-response type and stateless;

- it is based on open standards (REST, JSON, etc.), which makes it accessible to be used and integrated easily and without restrictions;
- has the role of interface of the system with the external environment and represents the entry point of the messages into the system.

7.2.1 Functional roles

The designed system interaction with the external environment is characterized by an extremely high dynamic. Specifically, the dynamics is determined both by the variation of the number of third-party entities that interact with the system, but also by the variable rate of the volume of information involved in communicating with those entities. From this perspective, there is a need for a flexible, consistent, but primarily scalable component.

I. Flexible and robust mediator

Event Gateway is a loosely coupled component responsible for mediating the system interaction with the external environment and acts as an entry point of messages into the system, where the authorization of third party components takes place and where the messages are approved and routed to internal components for further processing. The component is agnostic to the technologies on which operate the external components that generate incoming messages.

II. Router de nivel 7 (eng. *Layer 7 Router*)

The API Gateway component plays a significant role in handling the incoming messages. Depending on the business goals or on certain functional or technical aspects, ingested messages are to be redirected to internal distributed components for further processing.

III. Reverse Proxy

Exposure of the internal details of the system to the outside environment must be reduced to a minimum. Components responsible for message processing can be implemented in any technology, there is no restriction in this regard.

IV. Message translator

To simplify and generalize the use of the component, the ingested messages have a standard syntactic structure. Instead, after ingestion, they may undergo some adjustments or transformations, in order to be syntactically prepared for the subsequent internal components to which the messages are to be transmitted during the flow.

In the scientific literature, this approach is outlined in the form of a special pattern – *Anti-Corruption Layer*, formulated for the first time by Eric Evans in his well-known work in the field of software design - "*Domain-Driven Design*"or *DDD* [EVANS2003].

7.2.2 Advantages and weak points

The main advantages of using this software component are the following:

- *Financial*. Hardware gateways are significantly more expensive.
- *Flexibility*. They can be scaled and installed inside the system by intermediating the communication between internal components of the system (east-west traffic). They are loosely coupled and have a low dependency on external components, which allows them to be easily altered or completely replaced. It increases system security by authorizing third-party components and incapsulating the functionalities exposed by internal services;

- *Facilitates the architecture evolution*. Being robust and relatively easy to use, they can be reconfigured and adapted to new situations, approaches, techniques.

Although they bring a big plus in distributed architectures by decoupling in space the components involved in the interaction, they do not decouple them in time, which means that all the components involved must be available during the interaction.

7.3 Message queues. Functional roles

The components involved in asynchronous communication operate with messages also in a collaborative way. In order to allow the message to be processed at a later time, without the producer being waiting for the processing to complete, the message needs to be stored for a period of time by an intermediate component. The technical approaches in this case are multiple, from some general ones, such as files or databases, to more specialized ones – message queues.

7.3.1 Advantages and weak points

Among the advantages brought by message queues are:

- provides the location transparency;
- facilitates asynchronous generation/publication and consumption of messages;
- asynchronous notification of consumer;
- any message producing or consuming components can be removed or added at any time, which brings a significant flexibility to the system and significantly increases its ability to be scaled [BATES1998] and extended;
- brings a plus in abstracting the system functionalities;
- facilitates the separation of concerns.

The advantages brought by using message queues also come with some sensitive points:

- location transparency leads to increased latency in message flow;
- partially, fault tolerance must be addressed by the involved components;
- there is a risk that message will be processed multiple times or will be lost;
- communication is one-way;
- message order is not guaranteed from producing to processing.

7.4 Components responsible for message processing

Once with the advancement of computing systems based on multi-core processor architectures, for design and implementation of software systems it has become indispensable to apply approaches that intensely exploit principles focused on concurrent and parallel processing.

7.4.1 Reactive and scalable module for message processing

The basic ways to ensure the scalability, extensibility and resilience is the proper modularization of the system and loose coupling through asynchronous interaction based on message exchange.

Actors and actor systems in such situations are appropriate from the perspective of the following characteristics:

- the default interaction is of asynchronous type;
- the message, within this implementation represents an immutable data structure, through which an event or an intention is abstracted. It requires processing and

possibly a certain reaction, but depending on the context, which is determined by the current state of the actor, resulting from previous actions and decisions;

- state encapsulation and adjustable behavior, depending on the context;
- due to the internal functioning details of an actor, there will be no concurrency within it, which gives us the guarantee that its internal state will be consistent and can be modified in a controlled and coherent way;
- messages are processed in the same order in which they were received;
- provides support for the implementation of techniques for supervising the execution of child actors, which increases the system resiliency;
- by implementing appropriate supervision policies, subordinate actors and hierarchies (which include child actors) can be "shut downed" thus completely freeing up the resources used (first of all memory) - scaling down;
- although technically there is a difference in the performance of a local or remote call over the network, in terms of the approach used in the programming model supported by the actor model there is no difference, as all actors, regardless of their physical location are considered as being part of the same system;
- bring an increased level of flexibility to the system by applying the principles known as microservice-based architectures – encapsulation of status and behavior, specialization on limited functionalities – scalability;
- the system can be extended by adding new hierarchies of actors, which allows the extension of the functionalities of the entire system - horizontal scaling;
- due to the asynchronous interaction mode and location transparency, these components can be easily scaled both vertically (using multiple processors/cores) and horizontally (components running on separate physical/virtual machines can be added to the system through the network) allowing the implementation of extremely elastic, extensible/evolutionary systems;
- supports several ways of distributing and routing messages adapted to various situations that may occur in the context of distributed components.

The balanced assignment of threads to actors that have in queue messages waiting to be retrieved for processing ensures the continuous and simultaneous use of all processor cores assigned to the actor system. **The continuous demand for cores and the uninterrupted use of threads, without keeping them pending or blocked, is "*the most efficient concurrent processing system we can hope for*."** [VERNON2015].

7.4.2 Message processing

The approach chosen for digital representation of real world elements is related to the use of digital twins.

Actors representing the digital image of physical devices are responsible directly by coordinating actual processing of events (messages).

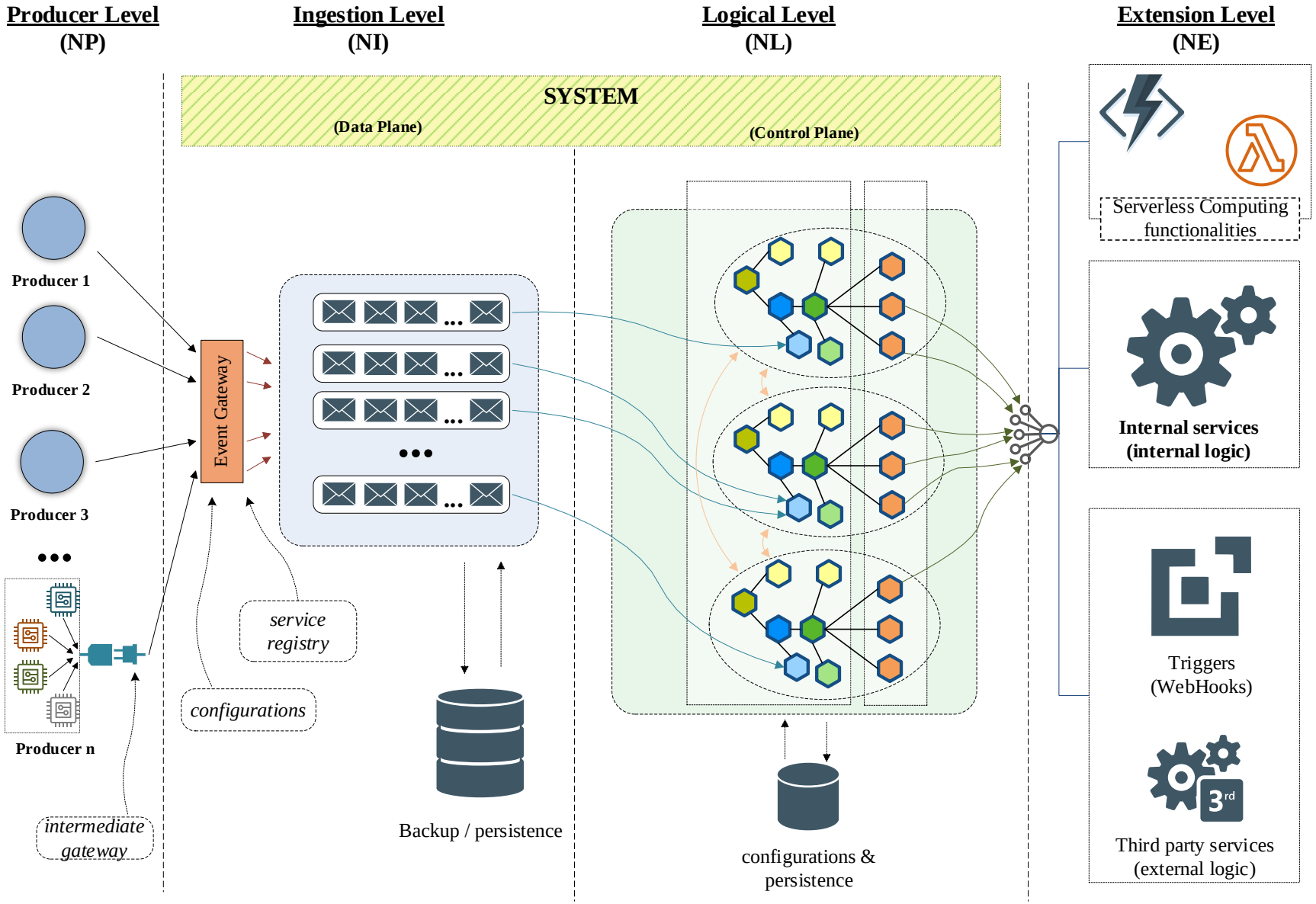

Fig. 7.1 – System Architecture

Chapter 8. Environmental monitoring software system implementation details. Experimental aspects

Choosing the right technologies for implementing the computer components identified in the architecture design phase of a complex software system are strategic decisions that will largely determine its quality parameters.

8.1 Events representation and ingestion. Interaction mediation

To describe the events and the context of their producing, as previously specified, the *CloudEvents* specification is proposed to be used. The relevant details, as specified, are represented in *JSON* [7] format.

8.1.1 Configurations and service registry

The component behavior is given by the specific configurations, which also determine the routing of the events. Configurations include dynamic aspects of the system, which can be changed at any time, separately from the application, even during execution.

8.1.2 Events ingestion, authorization and routing

From this perspective, the respective component imposes a set of access policies and rules for the message-producing components.

By using the EventGateway component, the entire complexity of internal components is abstracted, and the external components can benefit from their functionalities in a standardized manner, punctually adapted to the necessary technical requirements.

The EventGateway component enhances the agility of the system architecture, as it can be reconfigured, adapted, or replaced without affecting the rest of the components inside the system. In addition, due to its simple implementation, it can be easily adapted in order to correspond to new policies and functioning rules.

8.1.3 Scaling the events ingestion functionalities

The foundation of a reliable functioning is determined by the scalability of certain component. Scalability is ensured from two perspectives:

- ***conceptual*** – consists in using approaches that provide the premises of scalability. The most important role here is the isolation by functionally decoupling the EventGateway from internal components, but also from external ones.
- ***technical*** – implementing the approaches described in the previous chapters (see **Chapter 4**) which include software or hardware solutions.

8.1.4 Interaction mediation

In the current implementation, the interaction specificity is established depending on the following two criteria:

- ***interaction dynamics***, which requires that, in the situation where it is needed, one topic can be scaled independently from the others, so it must meet certain requirements in terms of performance;

[7] https://tools.ietf.org/html/rfc8259 (accessed on March 2021)

- ***cohesion*** between producers and consumers, which means that one topic will provide support for communication between those components that have a close collaborative relationship and a well-defined common goal.

8.2 Events processing. Reactive distribution of tasks

The criteria on which the decision to choose this model was based on:

- high-level programming model that allows the execution of multi-threaded tasks and the abstraction of the complexity of the details specific to the implementation of concurrent multi-threaded execution functionalities;
- abstracts the interaction through the network and the distributed execution of tasks;
- allows the creation of independent modules, but which can be organized in clusters for the distributed, collaborative and organized execution of tasks and the **creation of resilient, extensible, evolutionary systems that can be scaled both vertically and horizontally**, depending on the needs;
- provides the possibility of safely saving the current state on persistent environment, which increases the probability that the received messages will be processed, but also the possibility to "recreate" the actor, maybe even in another physical location, and to continue the initial flow as if that interruption had not taken place;
- facilitates the implementation of scalable components and complies with the principles of the *Reactive Manifesto*, which allows the implementation of applications that meet the requirements of cloud-native architectures.

8.2.1 Hierarchical structure of the processing node. Module scaling

In **Fig. 8.1** is represented the proposed conceptual structure of the hierarchy of actors of which a processing node is composed. The actors in the proposed hierarchy can be grouped into four general categories:

I) Coordination (AC, ACCh, ACG);
II) Configuration (ACfg, ADCfg);
III) Management (AMCh, AMG);
IV) Device Virtualization – represented by the acronym AD (Device Actor).

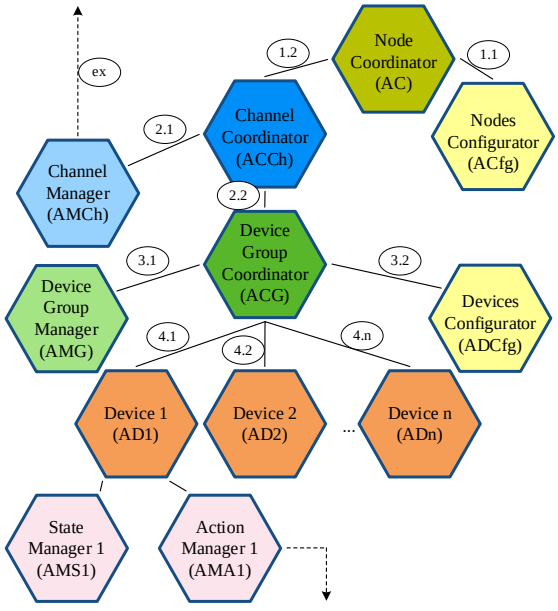

Fig. 8.1 – Conceptual Architecture of Processing Node

Loosely coupled stand-alone modules, which interact based on asynchronous message exchange, and location transparency, are the essential scalability approaches that were taken into account when implementing the environmental monitoring system detailed in this thesis.

Location transparency implies a common form of representation that abstracts the interaction between components, regardless of whether the message sent is to be processed locally or remotely. In addition, ensuring the location transparency allows new components to be added dynamically.

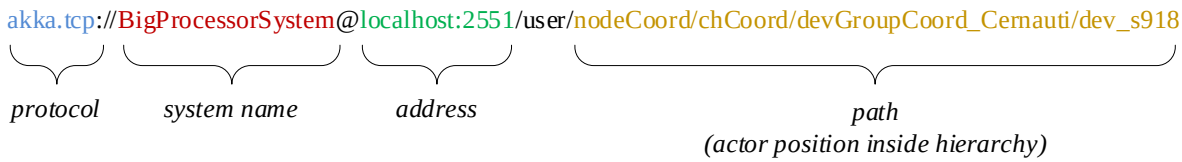

Fig. 8.2 – Actor location representation example

The asynchronous message transmission method functionally decouples the components of a software system, which allows the manufacturer and the consumer to be **scaled vertically** separately from each other.

The inclusion of several actor systems in a cluster facilitates **horizontal scaling** and **high availability** of the resulting system. Clustering, by adding and removing nodes during system operation, allows it to be expanded or contracted depending on the load.

We consider clustering as one of the methods to provide a dynamic elasticity and the system evolution.

The nodes that are part of a cluster have as basic goal the sharing of functionalities in order to achieve the results. The addition of new nodes allows the extension of system functionalities.

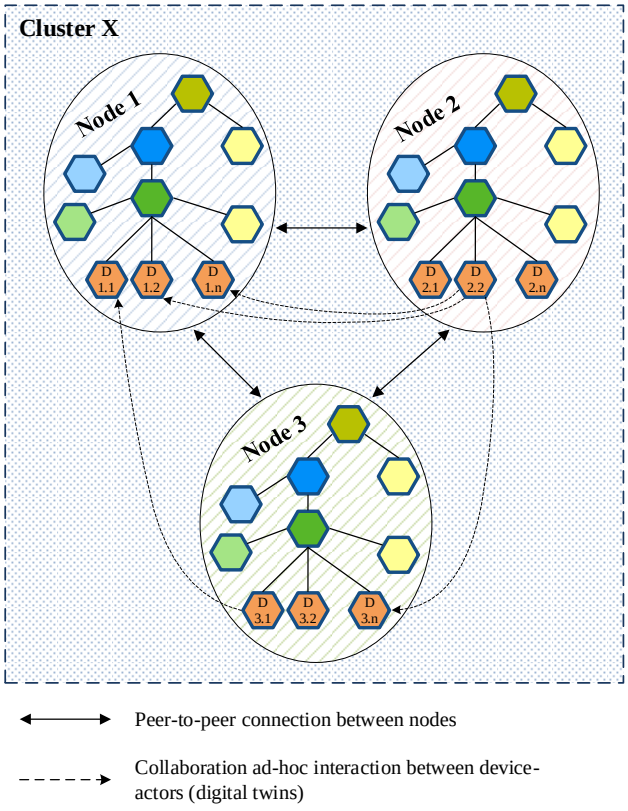

Fig. 8.3 - Cluster of nodes and collaboration between device-actors

8.3 Extending functionalities and system evolution

As stipulated in the papers of the author of this thesis [TOVARNITCHI2017], [TOVARNITCHI2019A], a modern computer system must be designed so that it can be extended and evolve with the requirements of the beneficiaries.

These features come in tandem with the possibility of the system being designed as an open [TOVARNITCHI2012] and scalable [TOVARNITCHI2021] one, which provides the advantage that it can be included in other systems by complementing the functionalities of the latter and creating systems of systems.

8.3.1 Details on functionalities externalization and extensions implementation

The system described in the thesis was designed so that its modules can be replaced with minimal effort and cost. At the same time, the existing functionalities can be completed with new ones. This feature has been implemented as a way to extend the functionalities of virtual

devices at the level of actors responsible for the digital image of the physical devices. From this perspective, these actors act as mediators regarding the extension of the functionalities of physical devices, a feature mentioned as one of the advantages of applying the concepts of digital twins [TAPUS2013], [BORANGIU2019].

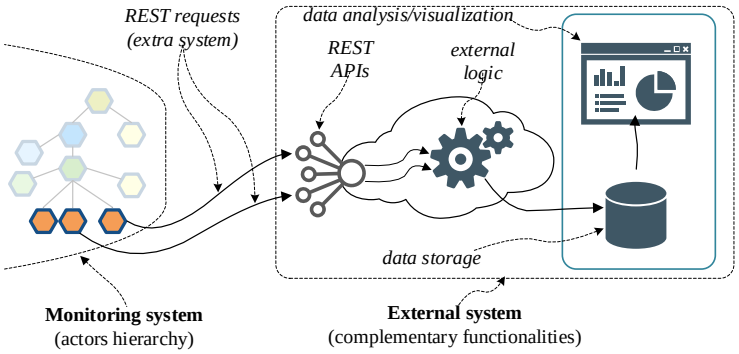

Fig. 8.4 – Functionalities externalization

The environmental monitoring application presented in this thesis involves working with events that are identifiable in *time* and *space*. Labeling events with information related to time and place of occurrence, comes with a significant advantage – allows correlating of multiple events and determining the context for real-time monitoring over a large area, but also for further processing for analysis, forecasting or model creation.

8.4 Implementation details summary

It is noted that the detailed implementation elements are deeply embedded in the system architecture at all levels, and any changes to it would come with a significant effort, and, therefore, with a significant cost. This confirms once again the significance of developing an optimal system architecture, together with a thorough and comprehensive analysis of the initial requirements.

The asynchronous execution of tasks determines the optimized use of the involved computing resources, allowing the completion of several tasks within a certain period of time. The controlled expansion of the system by adding additional nodes organized in clusters, along with the asynchronous execution of tasks, brings a major operational advantage: it allows the involvement of the required volume of computing resources at a certain time and optimized use of available computing resources.

The implementation of the environmental monitoring system, which was carried out in accordance with the architecture proposed and presented in the previous paragraphs, aimed:

- implementation of a distributed system, through the use of modern technologies and techniques;
- practical application of techniques proposed and detailed in the previous chapters, allowing the system to be scaled;
- using the principles according to which the system should operate reactively;
- the system functionalities can be extended and/or modified;
- experimenting on the use of actors for the digital representation of physical devices by using the concepts of digital twins;
- development of a hybrid Cloud architecture;
- the system should be cloud-agnostic, confirmed by the use of technologies that are not specific to a certain Cloud service provider;
- the use of provided Cloud services by distinct providers, confirming the possibility of implementing large distributed systems, with the possibility of extending or replacing the functionalities.

Chapter 9. Conclusions. Contributions. Future directions

9.1 Conclusions

The design of complex information systems is a fundamental field and decisively determines the implementation and operation of the system throughout its entire life cycle

A software system is designed to meet a specific purpose and is based on a set of well-established functional requirements. To abstract the complexity of the requirements related to the desired functionalities, it is used to formalize them in various forms, resulting in a *software architecture*.

Identifying and grouping functionalities from the same field (conceptually connected) into common components or modules and conceptual segregation between modules belonging to complementary domains covered by system functionalities facilitates the management of the complexity of modern software systems determined by the size and versatility of requirements.

The scalability and extensibility of a software system is ensured from the architecture designing stage, by specifying the interactions between the components and not their implementation details. Evolutionary architectures offer the possibility to modify and update the system over time

The main factors that differentiate the various architectures of computer systems rely primarily in the particularities of integration between components. In case of an implemented computer system, the efficiency of the integration between the components largely determines its essential quality attributes, since the functioning of the components within the system depends on the effectiveness of the “collaboration” between them. In particular, distributed software systems consist of specific components that can also run on computing systems with completely different properties and architectures.

The software architecture comes to provide effective technical solutions in order to meet non-functional requirements, such as: scalability, extensibility, resilience, availability, maintainability. The abstraction of the system components, used by the software architecture, facilitates the simplified understanding of highly complex systems.

At the same time, the integration of third-party software systems and tools, which come with delimited functionalities in order to complete the necessary ones within the designed system, is one of the main concerns of software system architects.

A successful computer system is based on an optimal architecture. However, the development of software architecture also contains a less accurate aspect, which cannot be quantified - it is based to some extent on experience and intuition.

The identification and delimitation of the components of a software system, dispersed either physically or logically, as well as the efficient interoperability between them, require the use of various specific approaches, adapted to the context

From technical, conceptual, economic point of view there is no reason to design a complex computer system as a single monolithic component. This is the main reason for developing the system architecture as consisting of several distinct components, which are delimited by grouping similar functionalities.

Modern infrastructure solutions, due to the advancement of virtualization techniques, provide possibilities (somehow) "unlimited" in terms of volumes of computing resources that can be provided. The services implemented on their basis, allow the implementation of information systems not only of "unlimited" dimensions, but also with a series of essential properties such as: scalable, resilient, flexible, extensible, dynamic.

Another significant aspect that software architects must take into account is the abstraction of infrastructure elements. This detail allows the implementation of infrastructure-agnostic solutions, which provides the basis for the implementation of so-called *cloud-native architectures*. These architectures are portable, scalable and extensible.

Modern software architectures should not be designed as rigid ones, with possibilities and functionalities limited strictly to the initial requirements. According to the current experience in the field, architectures must be *evolutive*. This requires the early development of the software architecture, so that its components can be modified over time or even replaced. Moreover, given the current trends and dynamics of technology development, along with the diversification and sophistication of end-user requirements of information systems, modern software architectures must be designed so that the resulting systems are *extensible* in order to add new functionalities or are included in even larger systems (systems of systems).

The demands on a modern computer system are variable and spontaneous

Information systems, as one of the basic requirements, must operate in such a way as to provide a certain level of quality of service. This is not possible without two fundamental details: a proper sizing of computing resources and the ability to use them optimally. They form the practical basis for *scalability techniques*.

It is noted that there is a trend where the design and operation of software components becomes completely independent from the infrastructure – through the use of *containerization technologies* and *serverless technologies*.

The ability to make efficient use of the available computing resources is extremely significant as to ensure the proper scalability of the system.

From the software architecture point of view, this is achieved through some essential basic approaches:

- abstraction of the future system functionalities;
- delimitation of functionalities by grouping them into distinct components;
- establishing the way of interoperability between components by designing the interfaces with the appropriate semantics and syntax depending on the context, as well as the interaction mode.

Proper modularization and abstraction of functionality details are one of the essential solutions for managing the complexity of an IT system. Abstractions do not have to depend on abstracted details, and details do not have to be implemented depending on abstractions. The interaction between the components must be performed according to these abstractions. Components that interact based on these abstractions are considered to be *loosely coupled*.

Loose coupling between components, asynchronous and message-based communication promise to be the main architectural approaches for a scalable, high-performance, resilient, extensible and scalable software system.

The interaction between the components of a modern distributed system is continuous and spontaneous, characterized by a changing dynamic

The operating environment of a distributed computer system consists, among others, of users and third-party components that are not under its direct control, and their intervention in the sense of interaction can be spontaneous and variable. Modern information systems must maintain a continuous link with their environment, and this requires that they be implemented in such a way that they are able to respond appropriately to requests. Such systems are called *reactive systems*, and one of the approaches according to which reactive systems can be built is the *actor model*.

Although the implementation of systems based on the actor model requires a different way to see the problem, this model allows the implementation of reactive applications, and the resulting systems to be scalable, extensible, resilient, resilient, elastic and responsive.

The physical and digital worlds are becoming more and more integrated, operating as a unitary whole, and the demarcation line is becoming more and more diffuse. The convergence between the two worlds - physical (real) and digital (virtual) becomes inevitable, and the latter, which (theoretically) has no limits, comes to extend the tangible reality

The use of modern approaches of tangible reality abstraction such as *digital twins* provides the possibility of digital modeling of physical objects and processes from the real world, which allows their management and monitoring at an extremely high level of granularity and accuracy, even in real time.

Today's technologies and techniques connect people, devices, processes and services in a collaborative environment in the form of *intelligent digital mesh*, approaches that have the potential to significantly increase the competitiveness of modern enterprises and help them come up with services and products with an increased added value.

Such techniques open new opportunities for research and development of automated solutions in all areas of our lives and work, especially where it is necessary to collect, analyze and take quick and correct decisions based on data collected from multiple distinct, scattered physical entities, but which act in an organized manner and are in some form of physical or logical dependency.

The combined use of these approaches allows the implementation of monitoring and management systems of physical systems in an extremely accurate and prompt manner.

9.2 Contributions

The contributions made in this thesis are predominantly applicative, the main direction being to pursue a practical and applicable finality of the research results. Results materialization represents the proposal of a reference software architecture for an intelligent environmental monitoring system, confirmed at the end through a practical implementation of a detailed environmental monitoring system. The proposed architecture was developed in compliance with the primary goals set at the beginning of the research - *scalability*, *extensibility*, *resilience* of software components of modern computer systems. The fields of applicability of the results of this research are multiple, where, with relatively few adjustments, the proposed architecture can be extended and applied successfully. The target domains include, first of all, *smart environments*.

A considerable effort in this research has been made in order to identify the details, elements, approaches, techniques and tools actively used today, which allow the implementation of a modern monitoring software system so as to ensure, depending on the context, its maximum possible scalability. A secondary direction that was taken into account in this research, was to identify practical solutions to ensure high indicators on the following quality attributes of a software system: performance, extensibility, resilience.

In addition to presenting the current ways of approaching scalability from various perspectives, the author's view was presented on the specific details of software architecture development so that the future system is both scalable, extensible and resilient. The significance of abstracting the specific details of interaction between the components of a software system that is reflected in its ability to be scaled and extended was emphasized. *Event-based architectures* have also been detailed, which form the basis for the implementation of modern IT systems with the goals pursued - to be scalable, extensible and resilient.

A special contribution made in this thesis is to deepen and expand the concept of a *system for intelligent environmental monitoring*. In summary, the following key proposals have been made in that direction:

- it was proposed to use digital twins approaches in the digital modeling of a physical environment for the purpose of intelligent monitoring;
- it was proposed to use in an original way the actors from the actor model for modeling and implementing digital twins as a basis for a monitoring system that is scalable, extensible, and also resilient;
- a set of requirements has been proposed that an intelligent environmental monitoring system must meet according to the proposed architecture;
- it was proposed for use a series of software architecture components detailing the modules grouped by functionalities, for an environmental monitoring system. Emphasis was placed on the architectural elements and principles identified in the research in order to ensure the scalability, resilience and extensibility of the designed system;
- it was proposed to use a standardized format for event representation, as a basis for providing the interoperability between heterogeneous software components and systems.

The essential contribution of the present thesis consists in the proposal of a reference software architecture for intelligent monitoring of the environment, which has been validated through a conceptual implementation of a software system.

The proposed software architecture for an environmental monitoring system is original from several perspectives. First, for its development, it was proposed to use modern techniques and concepts that were substantiated and justified within an extensive research. The proposed details have been applied to the design of the modules of the future system that meet the identified requirements and consider the specifics of each required functionality. The particular way how the system modularization was proposed provides the basis of an *evolutionary architectures*.

Secondly, given the context of the intensive evolution of digital technologies where everything "moves to the Cloud", the originality of the proposed architecture is given by the fact that it was designed to be approached as a *cloud-native architecture*. This aspect is essential in providing the competitiveness of a modern software solution, since it conditions the maximum possible decoupling of all software components implemented by the infrastructure components. Additionally, cloud-native applications are evolutionary, scalable, resilient and extensible. Not to be neglected is the efficiency of such architectures both technically and financially.

A significant effort was made for the practical validation of the developed architecture, materialized by implementing a scalable and extensible software system for the intelligent monitoring of the environment. It was described the specific implementation aspects that represent the implementation of the architectural details proposed and designed in the development phase of the architecture underlying the respective system. Also were presented the technical details considered relevant from the perspective of the goals of this thesis that were taken into account when implementing the system.

The implementation was approached from the perspective of two distinct, but complementary aspects. The first aspect is determined by the specifics of implementing the system modules so that they are scalable. Depending on the identified particularities of the functionalities within each module, various appropriate technologies has been used, the decision being argued for each module. A standardized event representation format has been proposed for use, providing the interoperability between heterogeneous components and software systems, both internal and external.

The second significant aspect regarding the development of the system was determined by the implementation details of the processing module. The theoretical basis is the actor model, but the implementation is original, from the perspective of several specific details. This

module is scalable horizontally and vertically and allows the system to be expanded at the cluster level. Additionally, the components of this module are extensible and resilient. It includes the implementations of the actors that represent the physical devices which manage the interaction with the external environment (based on the digital twins approach), but also the tasks offloading to third party components.

Another significant advantage of the solution is the system extensibility. Extensibility is managed at the processing node level and can be applied in two ways:

- adding native functionalities, by connecting in cluster the additional nodes with the necessary new functionalities;
- offloading functionalities, by connecting the external components (to the system) by using generic interaction methods, agnostic to the used technology.

9.3 Future directions. Perspectives

The concepts treated and proposed in this thesis have an increased potential for practical implementation in the development of complex information systems with applicability in most economic fields, including those related to our personal life.

As a practical example, in this thesis was proposed a conceptual software architecture for an environmental monitoring system. Our environment is an extremely complex "organism" and is influenced by a lot of actors and factors, which are, without a doubt, in a relationship of interdependency. This field is a generous one in challenges and in various details that determine it to be a very complex one.

The main direction for future development consists in improving the proposed architecture and create a *Holistic Environmental Monitoring System*, which would explicitly take into account the diversity of indicators, factors and actors in diverse, heterogeneous environments, but which, apparently or not, are in any relationship, even indirect. From this perspective, it is considered appropriate to expand the system with artificial intelligence capabilities, given the maturity and evolution of the concepts and the entire theoretical basis related to this field.

For effective implementation of details of artificial intelligence, as a second direction, it is necessary to adjust the interaction capabilities of entities that simulate real-world objects - digital twins, which are implemented as actors according to the actor model. First of all, this would involve their design and implementation so that they operate with a high level of autonomy, on the basis of decisions taken. Because this would require dynamic interactions between virtual entities, it is necessary to implement their interaction so that they are able to initiate and participate in complex flows and coordinated activities.

Human-machine interfaces are indispensable in an advanced environmental monitoring system. Their continuous evolution is observed, together with the fields of related digital technologies. Thus, another perspective of the necessary improvement of the system proposed in this thesis, imposed by the current trends, consists in including advanced human-machine interfaces based on the augmented reality. In the context of the diversity and complexity of the types of data with which a monitoring system can currently operate, this type of interfaces has an increased potential to provide an extremely useful tool in terms of simplifying the observation and real-time control of monitored situations.

9.4 Author's publications

1. **Vasile M. Tovarnițchi**, "Algorithms for data processing in environmental analysis", CSCS18: The 18th International Conference on Control Systems and Computer Science, Bucharest, 2011
2. **Vasile M. Tovarnițchi**, "Open Source Hardware as a Framework", 12th International Multidisciplinary Scientific GeoConference - SGEM2012, Bulgaria, 2012, pp. 193-197, **WOS:000348533800026, ISI Indexed**
3. **Vasile M. Tovarnițchi**, "Arhitecturi software și tehnologii cloud în monitorizarea distribuită a mediului", partea VII (3 capitole) din: Nitu C., Dobrescu A. S., Oprea A., Tovarnițchi M. V. , Popescu S., Iurescu L., "Sisteme Inteligente în Ecologie. Surse regenerabile de energie. Aplicații", Matrix Rom, București, 2016, pp. 437-530, ISBN: 978-606-25-0241-6
4. **Vasile M. Tovarnițchi**, "Cloud-based Architectures for Environment Monitoring", CSCS21: The 21th International Conference on Control Systems and Computer Science, Bucharest, 2017, **WOS:000449004400102, ISI Indexed**
5. Vladimir F. Krapivin, Ferdenant A. Mkrtchyan, Vladimir Yu. Soldatov, **Vasile M. Tovarnițchi**, "An Expert Systems for the Aquatic Systems Investigation", CSCS21: The 21th International Conference on Control Systems and Computer Science, Bucharest, 2017, **WOS:000449004400101, ISI Indexed**
6. **Vasile M. Tovarnițchi**, "Designing Distributed Scalable and Extensible System using Reactive Architectures", CSCS22: The 22th International Conference on Control Systems and Computer Science, Bucharest, 2019, **WOS:000491270300081, ISI Indexed**
7. **Vasile M. Tovarnițchi**, "Intelligent Wireless Sensor Network Applications in Cloud Era", CSCS22: The 22th International Conference on Control Systems and Computer Science, Bucharest, 2019, **WOS:000491270300082, ISI Indexed**
8. **Vasile M. Tovarnițchi**, "Scalable Approaches for Environmental Monitoring Solutions", CSCS23: The 23th International Conference on Control Systems and Computer Science, Bucharest, 2021, (accepted, to be published)**, ISI Indexed**

Bibliography

[ABBOTT2009] Martin L. Abbott, Michael T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise 1st Edition, Addison-Wesley Professional (2009)

[AGHA1985] Agha, G.A., ACTORS: A Model of Concurrent Computation in Distributed Systems, Doctoral thesis, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab (1985)

[ANGEL2012] Angelov S., Grefen P., Greefhorst D.; A framework for analysis and design of software reference architectures. Information & Software Technology. 54. 417-431 (2012); 10.1016/j.infsof.2011.11.009

[BASS2003] Bass L., Clements P., Kazman R., Software Architecture In Practice, Addison-Wesley Professional, 2rd Edition (2003)

[BASS2013] Bass L., Clements P., Kazman R., Software Architecture In Practice, Addison-Wesley Professional, 3rd Edition (2013)

[BATES1998] Bates John, Bacon Jean, Moody Ken, Spiteri Mark, Using Events for the Scalable Federation of Heterogeneous Components. Proceedings of 8th ACM SIGOPS European Workshop Sintra, Portugal, (September 1998) doi: 10.1145/319195.319205

[BERRY1989] Berry Gérard., "Real Time Programming: Special Purpose or General Purpose Languages." IFIP Congress (1989)

[BORAH2015] Borah Amlan, Mucharary Deboraj, Singh Sandeep, Borah Janmoni. "Power Saving Strategies in Green Cloud Computing Systems.", International Journal of Grid Distribution Computing. 8. 299-306, (2015) doi: 10.14257/ijgdc.2015.8.1.28.

[BORANGIU2019] Borangiu T., Oltean E., Răileanu S., Anton F., Anton S., Iacob I. (2020) Embedded Digital Twin for ARTI-Type Control of Semi-continuous Production Processes. In: Borangiu T., Trentesaux D., Leitão P., Giret Boggino A., Botti V. (eds) Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future. SOHOMA 2019. Studies in Computational Intelligence, vol 853. Springer, Cham

[COOK2005] Cook Diane, Kumar Das Sajal, "Smart Environments: Technology, Protocols and Applications" (1st Edition), Wiley-Interscience, 2005

[COUL2001] - Coulouris G., Dollimore J., Kindberg T., Distributed Systems: Concepts and Design (3rd edition), Addison-Wesley (2001)

[CRISTEA2013] Cristea V., Dobre C., Pop F., "Context-Aware Environments for the Internet of Things", In: Bessis N., Xhafa F., Varvarigou D., Hill R., Li M. (eds) Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence. Studies in Computational Intelligence, vol 460. Springer, Berlin, Heidelberg (2013) doi: 10.1007/978-3-642-34952-2_2

[CURRY2020] Curry E., Real-time Linked Dataspaces: A Data Platform for Intelligent Systems Within Internet of Things-Based Smart Environments. In: Real-time Linked Dataspaces. Springer, Cham, (2020), doi: 10.1007/978-3-030-29665-0_1

[DEBS2006] Introduction. In: Distributed Event-Based Systems. Springer, Berlin, Heidelberg (2006), doi: 10.1007/3-540-32653-7_1

[DIJK1968] Dijkstra E. W., The structure of the "THE" multiprogramming system. Communications of the ACM 11, 5 (May 1968), pp. 341–346, doi: 10.1145/800001.811672

[DUMITRACHE2017] Dumitrache I., Caramihai S. I., Sacala I. S. and Moisescu M. A., "A Cyber Physical Systems Approach for Agricultural Enterprise and Sustainable Agriculture," 2017 21st International Conference on Control Systems and Computer Science (CSCS), Bucharest, 2017, pp. 477-484, doi: 10.1109/CSCS.2017.74.

[EVANS2003] Evans Eric, "Domain-Driven Design: Tackling Complexity in the Heart of Software", 1st Edition, Addison-Wesley Professional, (2003)

[FIELD2000] Roy T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Doctoral Dissertation, 2000

[GARLAN1993] Garlan D., Shaw M., ”An Introduction to Software Architecture”, Carnegie Mellon University, Pittsburgh, PA, USA (1994)

[GRIEVES2014] Grieves Michael, "Digital twin: manufacturing excellence through virtual factory replication." White paper 1 (2014), pp. 1-7

[HAREL1985] Harel D., Pnueli A. (1985) On the Development of Reactive Systems. In: Apt K.R. (eds) Logics and Models of Concurrent Systems. NATO ASI Series (Series F: Computer and Systems Sciences), vol 13. Springer, Berlin, Heidelberg, pp 477-498

[HEWITT1973] Hewitt Carl, Bishop Peter, Steiger Richard, A universal modular ACTOR formalism for artificial intelligence. In Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI’73). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973) pp. 235–245.

[KAISLER2005] Kaisler S. H., "Software Paradigms", 1st Edition, Wiley-Interscience, (2005)

[KALE2019] Kale V., Parallel Computing Architectures and APIs: IoT Big Data Stream Processing (1st Edition), Chapman and Hall/CRC (2019)

[KUM2013] A. Kumar, H. Kim and G. P. Hancke, "Environmental Monitoring Systems: A Review," in IEEE Sensors Journal, vol. 13, no. 4, pp. 1329-1339, (April 2013) doi: 10.1109/JSEN.2012.2233469

[LAIGNER2020] Laigner Rodrigo, Kalinowski Marcos, Diniz Pedro, Barros Leonardo, Cassino Carlos, Lemos Melissa, Arruda Darlan, Lifschitz Sergio, Zhou Yongluan, "From a Monolithic Big Data System to a Microservices Event-Driven Architecture" (2020)

[PARNAS1972] Parnas D. L., On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12 (December 1972), pp. 1053–1058. doi: 10.1145/361598.361623

[PERRY1992] Perry D. E., Wolf A. L., Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes Vol. 17, No. 4 (October 1992), 40–52. doi: 10.1145/141874.141884

[REACTMAN] https://www.reactivemanifesto.org/ (accessed on 13.02.2020)

[RICHARDS2015] Richards Mark, "Software Architecture Patterns", O'Reilly Media (2015)

[SCHOLL2019] Scholl Boris, Swanson Trent, Jausovec Peter, "Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications", 1st Edition, O'Reilly Media (2019)

[SHAHRAD2019] Shahrad Mohammad, Balkind Jonathan, Wentzlaff David. "Architectural Implications of Function-as-a-Service Computing.", In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 1063–1075, (2019) doi: 10.1145/3352460.3358296

[TAIBI2020] Taibi Davide, Nabil El Ioini, Claus Pahl, and Jan Raphael Schmid Niederkofler, "Serverless Cloud Computing (Function-as-a-Service) Patterns: A Multivocal Literature Review." In Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER'20) (2020)

[TAN2007] - Tanenbaum S. Andrew , Van Steen M., "Distributed Systems - Principles and Paradigms", 2nd Edition, Pearson Prentice Hall (2007)

[TAPUS2013] Olteanu A., Tapus N., Iosup A., "Extending the Capabilities of Mobile Devices for Online Social Applications through Cloud Offloading," 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Delft, 2013, pp. 160-163, doi: 10.1109/CCGrid.2013.52

[TAPUS2019] Mohammed M.A., Kamil A.A., Hasan R.A., Tapus N., "An Effective Context Sensitive Offloading System for Mobile Cloud Environments using Support Value-based Classification", Scalable Computing: Practice and Experience, 20, (2019) pp. 687-698

[TOGAF] TOGAF® Standard, Version 9.2, a standard of The Open Group, https://pubs.opengroup.org/architecture/togaf9-doc/arch/ (accessed on May 2020)

[TOVARNITCHI2011] Tovarnițchi Vasile, "Algorithms for data processing in environmental analysis", CSCS18: The 18th International Conference on Control Systems and Computer Science, Bucharest, 2011

[TOVARNITCHI2012] Tovarnițchi Vasile, "Open Source Hardware as a Framework", 12th International Multidisciplinary Scientific GeoConference - SGEM2012, Bulgaria, 2012, pp. 193-197

[TOVARNITCHI2016] Tovarnițchi Vasile, "Arhitecturi software și tehnologii cloud în monitorizarea distribuită a mediului", partea VII (3 capitole) din: Nitu C., Dobrescu A. S., Oprea A., Tovarnițchi M. V. , Popescu S., Iurescu L., "Sisteme Inteligente în Ecologie. Surse regenerabile de energie. Aplicații", Matrix Rom, București, 2016, pp. 437-530, ISBN: 978-606-25-0241-6

[TOVARNITCHI2017] Tovarnițchi Vasile, "Cloud-based Architectures for Environment Monitoring", CSCS21: The 21th International Conference on Control Systems and Computer Science, Bucharest, 2017

[TOVARNITCHI2017A] Vladimir F. Krapivin, Ferdenant A. Mkrtchyan, Vladimir Yu. Soldatov, Vasile M. Tovarnițchi, "An Expert Systems for the Aquatic Systems Investigation", CSCS21: The 21th International Conference on Control Systems and Computer Science, Bucharest, 2017

[TOVARNITCHI2019A] Tovarnițchi Vasile, "Designing Distributed Scalable and Extensible System using Reactive Architectures", CSCS22: The 22th International Conference on Control Systems and Computer Science, Bucharest, 2019

[TOVARNITCHI2019B] Tovarnițchi Vasile, "Intelligent Wireless Sensor Network Applications in Cloud Era", CSCS22: The 22th International Conference on Control Systems and Computer Science, Bucharest, 2019

[TOVARNITCHI2021] Tovarnițchi M. Vasile, "Scalable Approaches for Environmental Monitoring Solutions", CSCS23: The 23th International Conference on Control Systems and Computer Science, Bucharest, 2021, (accepted, to be published), ISI Indexed

[VERNON2015] Vernon Vaughn, "Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka" (1st Edition), Addison-Wesley Professional (2015)

[WEBB2008] Webb Molly, "SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI)." Creative Commons (2008)