DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

DEPARTMENT OF INDUSTRIAL ENGINEERING

FONDAZIONE BRUNO KESSLER

Doctorate Program in Industrial Innovation

EFA (Event Flow Architecture) principles illustrated through a software platform

Software architecture principles for IoT systems, implemented in a platform, addressing privacy, sharing, and fault tolerance

Andrea Eugenio Naimoli

Academic Advisor

Prof. Bruno Crispo

Università degli Studi di Trento

Industrial Tutor

Dr. Raja Marazzini

airpim srl

January, 2024

There will be an answer, let it be.

[Let it be, Beatles, 1970]

Abstract

The design and development of technology applications has to deal with many variables. Reference is obviously made to established hardware and software support, particularly with regard to the choice of appropriate operating systems, development model, environment and programming language. With the growth of networked and web-exposed systems, we are increasingly dealing with IoT (Internet-of-Things) systems: complex applications consisting of a network of often heterogeneous elements to be managed like an orchestra, using existing elements and creating new ones.

Among the many fields affected by this phenomenon, two in particular are considered here: industry and medical, key sectors of modern society. Given the inherently parallel nature of such networks and the fact that it is commonly necessary to manage them via the Web, the most prevalent de facto model employs an architecture relying on a paradigm based on data flows, representing the entire system as a kind of assembly line in which each entity acquires input data and returns an output in a perfectly asynchronous manner.

This thesis highlights some notable limitations of this approach and proposes an evolution that resolves some key issues. This has been done not only on a purely theoretical level, but with actual implementations currently operational and thus demonstrated in the field.

Rather than proposing an abstract formalisation of a new solution, the basic principles of a whole new architecture are presented here instead, going into more detail on some key features and with experimental and practical feedback implemented as a full blown software platform.

A first contribution is the definition of the principles of a new programming architecture, disseminated with some published articles and a speech in an international congress. A second contribution concerns a lightweight data synchronisation strategy, which is particularly useful for components that need to continue working during offline periods. A third contribution concerns a method of storing a symmetric encryption key combined with a peculiar retrieval and verification technique: this has resulted in an international patent, already registered. A fourth contribution concerns a new data classification model, which is particularly effective for processing information asynchronously. Issues related to possible integrations with artificial intelligence systems have also been addressed, for which a number of papers are being written, introduced by a presentation that has just been published.

Keywords

Architecture, Paradigm, IoT (Internet-of-Things), Privacy, Data, Flow, Events, Industry, Medicine

Acknowledgements

This work would not have been possible without the support of my whole family, wife and children in the first place, and of my parents too.
I am also grateful to my academic advisor who made it possible to undertake the journey to this point, to my industrial tutor who supported me during the research and to the company that made the collaboration with the university possible.
I thank the referees who supervised the work, including the referees and the final committee.
I also want to mention all the teachers I had from my early years of study to the most recent ones.
I also had great pleasure of working with my colleagues in the company, both in-house and external, and with one of the founding fathers of the technologies discussed here.
Thanks also to all the persons I met and who in some way contributed to the success of this work.

Contents

Abstract 5
Keywords 7
Acknowledgements 9
Contents 11
1. Introduction and context presentation 13
1.1. The context 13
1.1.1. The PhD course 13
1.1.2. The subject matter 14
1.1.3. Personal background and interest in development systems 14
1.1.4. Working and research environment 15
1.1.5. Main topic and focus 17
1.2. The Problem 20
1.3. The Solution 23
1.4. Innovative Aspects 25
1.4.1. Starsyncing 26
1.4.2. Superpassword 27
1.4.3. Fluid Data 28
1.5. Structure of the Thesis 31
2. State of the Art 33
2.1. Platform and paradigm 33
2.2. Framework and language 35
2.3. Comparing with innovation introduced 36
3. The Problems 39
3.1. What emerged 39
3.2. Key points addressed and resolved 39
3.2.1. Dynamic structure 40
3.2.2. Error handling 41
3.2.3. Privacy-by-design and Access Recovery 43
3.2.4. Data Manipulation and Synchronisations 44
4. Approach and Activity 45
4.1. Design and development 45
4.2. Workcases 45
4.2.1. Work Case “Energy”: Energy Supplier Company (gas, electricity) 46
4.2.2. Work Case “Cloud”: Global Ceramic Industry 47
4.2.3. Work Case “Sync”: International Natural Stone company 48
4.3. Principles, basic concepts and building blocks 49
4.3.1. EFA-EFP vs FBP/EDP 58

4.3.1.1. EFA-EFP vs FBP 58
4.3.1.1. EFA-EFP vs EDP 61
4.4. Components 62
4.5. Benchmarking references 64
5 Experimental results 67
5.1. Methodology 67
5.2. Critical features actually applied 67
5.2.1. Starsyncing 67
5.2.2. Superpassword 79
5.2.3. Fluid Data 92
6 Applied research: actual developed work 107
6.1. Actual implemented components currently running 107
6.2. System popularity, evaluation and personal contribute 109
7. Conclusion 113
8. Appendix 115
8.1. Python function signature type check 115
8.2. JavaScript catch-all handler 116
8.3. Node-RED flow sample 118
8.4. Basic classes to model tables for starsyncing 119
Bibliography 133

1. Introduction and context presentation

1.1. The context

1.1.1. The PhD course

This thesis is the final documentation of the industrial doctoral course “Doctorate Program in Industrial Innovation” co-founded by the University of Trento and the Fondazione Bruno Kessler. As stated by the official headline “companies are the key player” in the course as it is an interdisciplinary course in close contact with the corporate world. The author holds an “executive PhD” position, meaning that the activity is carried out primarily within the company in which he works. Unlike traditional doctoral studies, these kinds of courses focus on achieving high-impact applications in the industrial world. In fact, the research work has been accompanied by very intensive development, so much so that the survey results have been effectively implemented and are already in current use by several primary companies of national and international significance, with thousands of active users and millions of data records processed daily.

In addition to the fact that not only abstract research data, but a complete product currently actually used in the industrial field (called "Openbridge" [→∯6]) resulted from the work done, another peculiarity is the period of collaboration with the founding father of one of the basic theories discussed here [→∯1.1.5.1], mr. J. P. Morrison, who unfortunately passed away before seeing the conclusion of the work. Such collaboration was a privilege not to be taken for granted: like him, several influential personalities who gave birth to languages and paradigms (such as D. Ritchie for the C language or O. J. Dah and K. Nygaard for object-oriented programming) were born between the 1920s and 1940s, and unfortunately they have in many cases passed away before the current decade or are very old and often difficult to involve in personal studies.

Parallel to the activities carried out strictly within the company, developments in the medical field have been explored in depth with publications and speeches in the field of telemedicine, also touching on issues related to artificial intelligence [1]

As a final introductory note, it is worth noting that the entire substantial part of the pathway took place completely in the pandemic period (Covid-19 pandemic) between 2020 and 2023: this greatly conditioned the situation by incredibly limiting access to numerous resources and interactions with the dozens of companies involved in the project. Considerable additional effort was therefore required in order to formalise the model and arrive at drafting this document.

1.1.2. The subject matter

The partner company airpim srl [→∬1.1.4.2] proposed a study related to a software platform for effective management of customer information seen as a graph related to people, objects and data with multiple relationships between them: the topic therefore concerns the analysis of the highly varied needs of customers and the study of a cross-cutting solution by identifying a software model suitable for the purpose [→∬1.2].

Initially, considering the prototypes set up for the first customers involved, the implementation of a middleware relying on existing solutions was considered, but during the research it became clear that it would be advantageous to evaluate an update of the main paradigm adopted (FBP, created by ICT specialist mr. Morrison), and then finally decided to set up a completely new architecture, although strongly rooted in both the paradigm mentioned above and incorporating features of another paradigm flanked during the last period (EDP: Event Driven Paradigm). [→∬1.3]

1.1.3. Personal background and interest in development systems

Early programming was based on BASIC, Assembly and pure LM, and a few years after other languages such as Pascal, C, and AMOS came in, to develop utilities and games and to write articles on related topics. Later, new concepts have taken shape both through released utilities and videogames, and in the educational field. I then carried out in-depth studies on operating systems and

more generally on the development of programming environments, even collaborating on the design of some programming languages.

The recent PhD path stemmed from the initiative of the airpim company where I serve as CTO, which is very strong in research and innovation [see ∬1.1.4.2]: it can also be pointed out - as already noted - that the entire path had the honour of being favourably supported by mr. J. P. Morrison, a founding father in the 1970s of one of the pivotal principles discussed here (namely the “Flow-Based Programming” paradigm) [see ∬1.1.5.1], who unfortunately passed away in 2022.

1.1.4. Working and research environment

1.1.4.1. airpim

The high-tech firm airpim is based in Italy, having headquarters in Rovereto (near Trento) and satellite offices in other areas (including Verona, Bolzano and Milan), as well as a linked office in the United States of America. It is the main entity that promoted the present research, with the idea of building formal foundations for optimal software design in the solutions provided to customers, going beyond the current state of the art, based on common tools currently available, based on technologies from several years ago. It has been registered as the first innovative PMI in the whole Trentino Alto-Adige area. [3]

The present research produced not only experimental results, but solutions actually put into practice - and currently used by most of airpim’s customers - meeting the basic spirit of a doctorate classified as "industrial" . About twenty clients (and as many configurations) were involved, a significant partial list of which is given in paragraph 4.2.

1.1.4.2. SIT - Società Italiana di Telemedicina e Centro Studi Internazionale "Giancarmine Russo"

The Italian Society of Telemedicine has been active since 2007 aiming at a synergy between researchers and experts in digital medicine, thereby combining medicine and ICT. It has strong relationships with central authorities, corporations and universities and two international operative

liaisons for scientific and cultural projects, including the International Telemedicine Study Center and Research Unit “Giancarmine Russo”.
Collaborating with the latter as a technical advisor and researcher, I have pursued several publications on issues related to the present study [4, 5, 6, 7, 8, 9, 10] and promoted the popularisation of the architecture proposed here at the recent international congress [11]. The studies conducted in synergy with the Study Center have given a major boost to further investigations for possible cross-cutting applications and experimental testing in critical fields, such as medicine and paediatrics.

1.1.4.3. University of Trento

The University of Trento is a primary italian institution, with an extremely high level of quality: ranked first among the average universities in Italy according to the CENSIS institute. [2]
The teaching and research activities are of primary importance and are embedded in an area with numerous realities of different disciplines, particularly strong in the technological field (research, development, industry 4.0, industrial innovation), so much so that it is also known as Italian Silicon Valley.

1.1.4.4. Epityon

It is a technology innovation company that promotes research & development particularly supporting airpim which obviously has more commercial goals. About the present research, it has supported the designs of several software solutions by following numerous software filing practices and particularly followed the patent practice that arose during the PhD. [→∫5.2.2]

1.1.4.5. Elementica and more sideworks

All activity beyond productions strictly related to work orders are carried out by interfacing with public and private interlocutors through an additional entity called "Elementica": under this reference, secondary experiments and contacts in Italy and abroad have been carried out for possible future implementations.

1.1.5. Main topic and focus

The entire thesis addresses the issue of architectural choice for the design and implementation of software suitable for a very broad class of technology platforms, particularly characterised by being used in systems consisting of several interconnected units exchanging data with each other. Such systems typically fall into the IoT ("Internet-of-Things") world. [12]

The research is rooted in two paths that have met during these doctoral years: on the one hand, the study of telemedicine systems, in which the cooperation between various applications, diagnostic tools, detection sensors and other elements presents strong critical issues with regard to information processing and protection (think, for example, of privacy issues), and on the other hand, management platforms for medium and large industries (particularly for applications of the so-called "Industry 4.0", again with a great stress on data privacy [13]).

The first branch (studies on telemedicine systems) I have been pursuing for a few years, while the second (applications of the "Industry 4.0" world) concerns the airpim company's operational orders that have started using the best available solutions to integrate management environments, web applications, smartphone applications, peripheral devices (e.g. local sensors), and industrial machinery (including PLC machinery) for clients of national and international significance. The works in airpim were initially set up using the available state of the art and the main elements include: linux operating system, JavaScript and Python programming languages, Node-RED environment. The latter is a framework and environment suitable for IoT-based and IoT-like systems [14] and a kind of standard-de-facto in web applications adopted also by big players, such as Samsung [15] and others.

The main topic is therefore the definition of an operating model, an "architecture" more precisely, as an evolution of that derived from the above tools, fundamentally based on the FBP paradigm, later also accompanied by EDP [→ ∫1.1.5.1], to overcome the important issues that have emerged during its use.

1.1.5.1. Paradigm and languages

Computers as we know them today have a relatively recent history and actual milestones can be identified concentrated in the second half of the last century and the beginning of the current one. One of the most important steps is to be found in the possibility of programmable machines, with the separation of hardware and software becoming increasingly clear.
Going back to the 1950s, we find the ENIAC as one of the most important computing systems (without forgetting some notable earlier examples, such as the Z3 from a decade earlier and others), followed by various increasingly complex and efficient systems until the invention of the microprocessor, with the Intel 4004 being the first single-chip example in the 1970s.
With a vertiginous growth in power and complexity we then arrive at the most recent systems with considerable calculation and memory capacity, also thanks to network connections that allow the constitution of distributed and dynamic systems.

The programming of early systems basically consisted of operational sequences of basic actions, such as reading and writing information, comparing and branching (in the sense of making a choice). This is also the basic behaviour of microprogramming (i.e. inside microprocessors) and the philosophy of all early programming methods: it is a procedural, imperative paradigm [16]. Over the years, new paradigms have emerged and new programming languages built around them.
The concept of “programming paradigm” is located within a broader concept of “programming architecture”, which is the set of all elements involved in the implementation of software, including behaviours with the external context and how they interact with users. Surprisingly enough there is no universal definition for both concepts (one tip that emerges from this article is to inspire one), which will consider not the entire spectrum of architectures, paradigms and languages (since there is a plethora of them, including some esoteric ones), but only those of great interest to the scope of

operations in which we have moved, particularly that of the IoT world, large industries and telemedicine. In general, both "architecture" and "paradigm" will be discussed in this paper: the former concept is broader and includes not only the paradigm - which can be thought of as the logical model on how to organise and process information - but also details about the operating units, the type of hardware and software to be used, and more. Except where doubts may arise for which one of the two headings may be specified in a timely manner, it is possible for them to be used interchangeably when it is clear which detail is to be referred to.

A very relevant aspect that will be considered concerns multitasking programming since the execution of concurrent or even parallel processes is well suited to the realisation of IoT systems, due to the common presence of independent elements connected to each other. In this regard, a proposal for a new data model that takes full advantage of this possibility will be presented, freeing as much as possible from the need to set the structure of the various tasks a priori so that when tasks can be done in parallel, they are done so without the need to set it explicitly or to define synchronisation points.
In order to achieve this effect, a new data classification method (data type) is proposed to transparently exploit the possibility of applying functions on data even if it has not yet been fully processed [→∫5.2.3].

1.1.5.2. IoT and Data-Flowing

Under the term IoT many systems can be included: usually we refer to networks of devices and sensors connected through higher structures, but it is also possible to think of a more complex formalisation, for example, distinguishing a series of "layers" ranging from the more "physical" level (hardware: the sensors as objects in the real world for example) to the more "logical" level (the software application that governs the entire structure). [25]
What is of interest here concerns only the upper levels of such layering, however, not from the point of view of purpose, but from the point of view of conceptual model.

In a nutshell, the idea is to provide the principles of a model that can represent an IoT system and then lend itself to practical implementation using some kind of consensus development tool.

Without reinventing the wheel, we started with readily available tools and-as mentioned in part above-the solutions initially sketched out and then dissected and evolved here were based on the Node-RED tool and then the FBP paradigm: the idea is that typically in an IoT system there are peripheral devices (usually interfaced with the outside world through one or more sensors for acquiring data from the real world) that perform input data processing and then coordinate with other devices or integrators at higher levels. It becomes natural to think of a paradigm that models such devices by seeing them as operational units that react to input data with continuity over time: this concept is exactly the basis of the FBP paradigm, and so a tool such as Node-RED allows it to be applied simply and effectively. Since it is then based in turn on the NodeJS platform and the JavaScript language, an enormous amount of material and resources are available. These are by far among the most popular and widely used technologies in the world, particularly in networking and web. [26]

All in all a Node-RED application is basically an endless flow of data: when a node gets some input, it processes it and sends an output away, as a kind of factory chain.

1.2. The Problem

There are many details that can be taken into consideration, but for the purpose of this study we will focus on only a few relevant aspects.

Work orders of airpim can all be framed as IoT systems (and in fact, this is how they were managed, agreed upon, implemented, and currently used): typically a general administrative panel installed on a server connects to several remote subsystems-often connected to local sensors-and allows for the detection and activation of various interactions.

Using the tools described in the previous paragraphs, including Node-RED, basically two main issues emerged at first, that could not be solved by effective alternative choices [17]:

- *Allow a dynamic structure: in FBP the network of nodes is the program itself, so once defined is typically static and needs a new deployment to be modified even slightly*
- *Embed a well-defined error handling system: the autonomy of the nodes is both a blessing and a curse when dealing with accidents or glitches as a simple failure of a node can be. This*

is much clearer in Node-RED where the only effective method to handle this case is the "catch" node, an obvious outlier in that it has no standard input channel at all, even if it receives data.

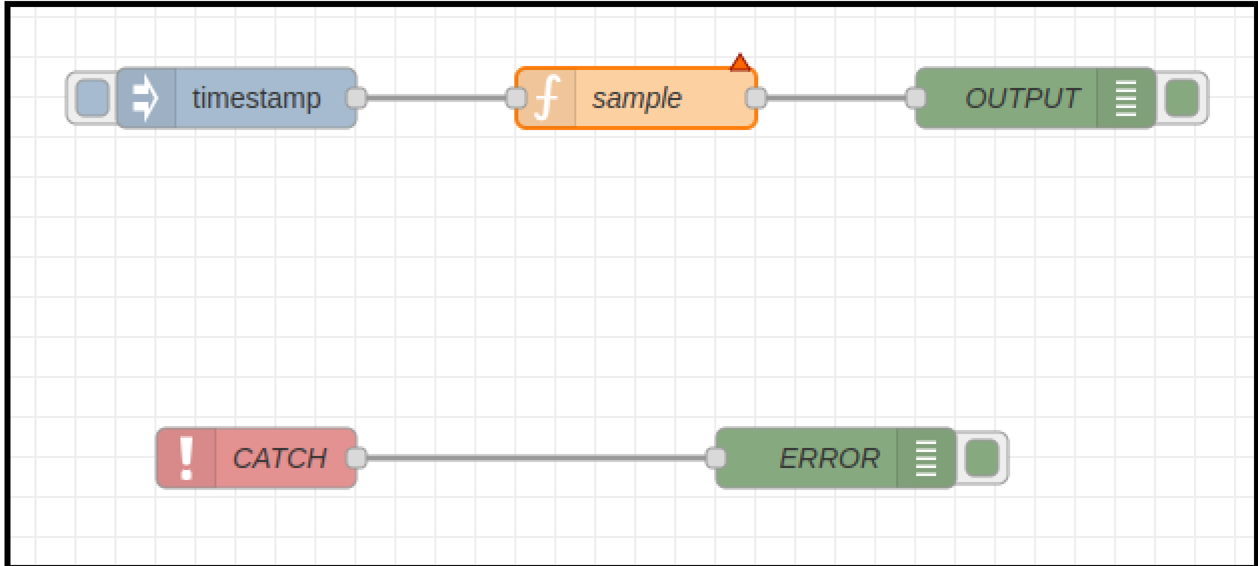
These are both points of enormous criticality: the very nature of an IoT system is subject to constant evolution (just think of the issues related to physical sensors that can be prone to failure and need to be replaced), and in addition, the absence of fine error management can cause malfunctions or even a general shutdown for even the slightest problem in the smallest of components.

The question about errors refers to deep failures (including syntax errors in the code to be executed) and not to simple "logical" errors, which obviously can be handled by dedicated procedures. Incredibly, for this aspect there is no proper tool of any kind: in fact, the official Node-RED documentation itself reports:

"*[These kinds of errors] can occur when a Node [i.e. a single logic unit] fails to properly handle an internal error. They cause the entire Node-RED runtime to shutdown as that is the only safe thing to do.*"

This is obviously a condition conceptually unacceptable and absolutely must be overcome in order to arrive at a viable supply for a work order.

Here follows a very simple Node-RED program that shows part of the problem (see also ∫8.2 for the XML code):



sample Node-RED program with a CATCH component

The “function” node contains few issues: Node-RED does its best to detect and report problems (the icon with the triangle indicates possible problems), but especially in asynchronous systems there can be "uncaughtException errors" that are totally unmanageable.

In particular, only two situations are possible: either the problem is intercepted by the "catch" node (if any) or it is not handled.

Meanwhile, in both cases the operational flow is completely diverted so that all nodes following the error node will be in an inconsistent condition, in addition any "side-effects" would be applied anyway (so it would be appropriate to have a dedicated flow to handle a possible rollback). Furthermore, even in the case where a "catch" node is present, it should be noted that this only triggers a new flow! This is therefore a non-solution since such a flow is nothing more than a branch of the whole application that could in turn generate errors.

Subsequently, other issues related to the data processed by the various systems also emerged, and in particular two of them were studied, addressed, and eventually solved:

- confidentiality and protection of information:

    - there are, of course, many methodologies for managing data security, but a privacy-by-design approach that makes such management transparent is of paramount importance
    - the chance to protect even with encryption methods some data is important, but having the guarantee that it can be accessed even in extreme situations, especially with a secure system to have the access key even through a disaster recovery procedure
- ability to manage data as flexibly as possible, concentrating efforts on processing them, not on managing and synchronising communications: unfortunately, almost all programming systems require explicit management of communications and specific attention to asynchronous ones, i.e., all those typical of an IoT system.

In chapter 3 we provide a brief discussion of each of the points listed above.

1.3. The Solution

A new paradigm is defined within a broader concept of an event-flow-based architecture.
The basic idea is that the components involved are seen as an interconnected network of elements that react to inputs by providing an output (as in the FBP paradigm) but also having a broader view that can respond to more complex actions. Such actions (including a possible crash of a single component) are classified as "events" and triggered over time. All "inputs" and "outputs" are included in this new categorization and seen themselves as events. In summary, the entire system is a cloud of components immersed in a stream (over time) of events to which they react.

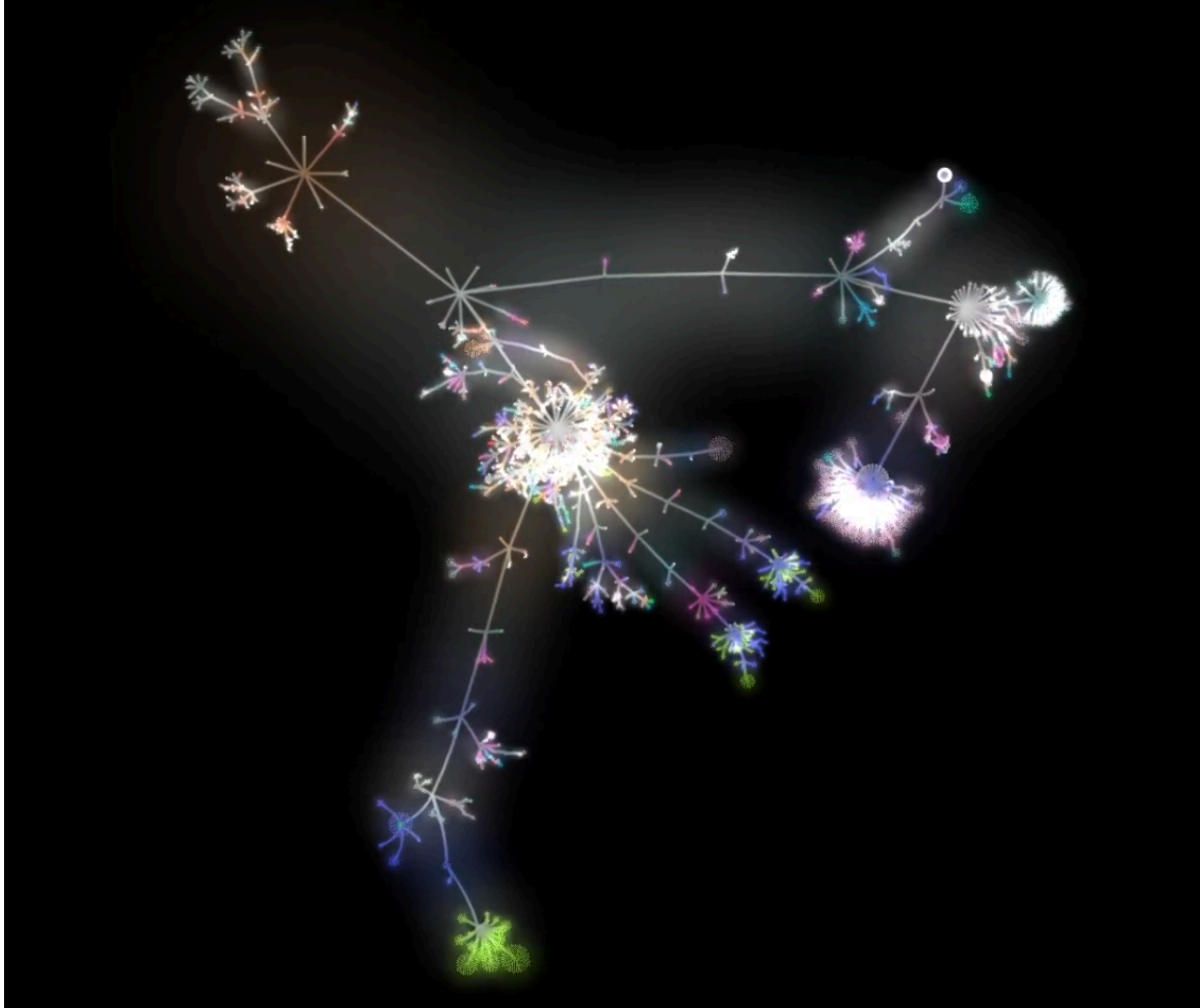
Thus, the architecture is based on: components (containing and executing the operational logic that processes information), relationships (the interconnections), events (events that happen over time, including receiving input, producing output, and crashing a component), and data (the information processed, transmitted, and shared).

The following is the basic nomenclature defined for this work.

- **System**: the entire network of components involved, including their relations. It must be viewed from the conceptual point of view, and is typically implemented as a set of intercommunicating software applications installed on multiple devices networked together and typically also exposed on the web
- **Unit**: a single component of the system. It can be a complex subsystem or a simple element: in many cases it corresponds to a specific hardware device (such as a server, for a component that processes data or acts as a bridge between other elements, or a sensor or wearable device)
- **Channel**: it is a relationship between units in which information can move: it normally has no physical correspondence (but the transmission medium), but is only a software abstraction
- **Attributes** and **State**: *attributes* are named characteristic quality of an element that can take on different values (some are typical of architecture, others are custom-definable), while the *state* is an ordered set of the attributes (of any element: the whole system, a single unit, or a single channel)
- **Event**: any change in the state of an individual unit, channel, data, or the entire system
- **Data**: informations processed by the system through its units

The entire system is based on interactions between units, whose activity is stimulated by the succession of events of various kinds over time: hence the basic concept of "event flow" (Event Flow) from which the paradigm (EFP: Event Flow Paradigm) and architecture (EFA: Event Flow Architecture) are named.

All of the general principles and specific solutions designed have been effectively implemented in the latest versions of a custom software platform named “openbridge” [→6] (fully engineered within the airpim company) currently active and used daily by hundreds of users and managing millions of records in various work environments. Few excerpts are reported in the appendix, while the whole platform is made up of hundreds of thousand of lines and has a complex custom toolchain (developed ad-hoc) and it is composed by personal snippets, proprietary code (of the airpim company) and some open source add-ons.



graphical representation of the "openbridge" project managed with the GIT versioning system

1.4. Innovative Aspects

By addressing the various issues that emerged in the course of the research, some particular solutions were developed that are broad in scope and thus applicable in different contexts as well. Among the various innovations developed - it should be emphasised that these are in all cases not only experimental activities, but concretely implemented and to date actively used on a daily basis in various supplies - are to be highlighted in particular:

- An incremental synchronisation method for efficient alienation of information between different components code-named “starsyncing” [→∫ 1.4.1, 5.2.1]
- An encrypted information management and retrieval system that resulted in a software patent code-named "superpassword" [→∫ 1.4.2, 5.2.2]
- A new methodology of general organisation and classification of software data code-named “Fluid Data” [→∫ 1.4.3, 5.2.3]

1.4.1. Starsyncing

Dealing with systems composed of multiple units often presents the problem of having to synchronise information: in IoT systems it is very common to have peripheral units (often of low computing power), often integrated with sensors, that must interact with a main unit. Not infrequently, such peripheral units must also be able to work during more or less prolonged phases of "offline" (i.e., unavailability of communication to other units): a very recurrent configuration therefore consists of systems or subsystems headed by a main unit that coordinates numerous peripheral units. Such a configuration can be represented as a kind of "solar system" with one main node connected with many secondary nodes.

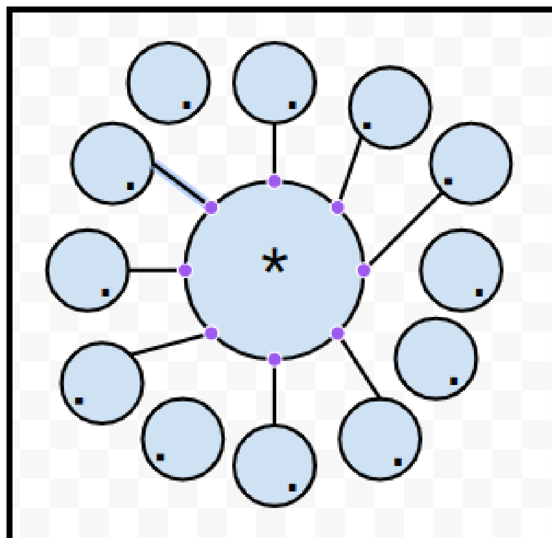


diagram of a bunch of peripheral units linked to a main one

To effectively manage a synchronisation of an entire database in such a configuration, the most common "distributed database" technologies are ill-suited by taking precisely the above characteristics into account: in particular, secondary units can hardly support a full database engine installation, and in addition, it is often necessary to integrate data from external sources for which it is necessary to be able to properly mark records in order to distinguish them (a "key" may not necessarily be present) and to know their status (whether updated since the previous synchronisation or not for example).
A particular system has been designed and implemented that is very well suited to such situations, code-named "starsyncing" [→∫5.2.1]: it is currently fully operational in the "openbridge" platform [→6] and is being used in various instances for clients of international scope.

1.4.2. Superpassword

A critical aspect of data management in an IoT system is data protection: the presence of distributed peripheral units in the "real world" exposes data to significant risks. Small sensors or wearable computing units can be easily accessed by unauthorised parties, so some form of data encryption is often necessary. Studies exist about several applicable solutions [29] but one particular aspect remains essentially unresolved: in order to properly protect the data, it is necessary that even direct access (to the physical medium) does not allow the information to be interpreted, so a strong form of encryption, just on a key, may be a solution. However, the problem of being able to have a method of retrieving such a key can arise, and this can happen in various situations such as:

- the person who knows the key is not available (e.g., in health care or national security, this could be a designated official who is no longer able or refuses to do so)
- the person who knows the key loses or otherwise forgets it
- any "brute force" solutions are not applicable because they are too costly and perhaps to be applied on systems with a limited number of attempts

The last point raises another very sensitive issue: the desirable retrieval system should also be such as to ensure that the result obtained (the retrieved key) is actually correct, and this is without using it.

It may be noted that the situation described above is common in a number of areas: for example, in the medical field, it is admittedly true that there is confidential health information that is accessible only to patients and referring physicians, but in case, for example, there is an individual physician involved in that context for some information and perhaps he or she becomes unavailable (trivially because he or she unfortunately passes away or even more simply because he or she is unreachable in an emergency), there is usually a form of hierarchy whereby some other party can take over. Another related case occurs in various political contexts where certain apex roles (e.g., a president of a nation) have access to confidential information, however, there are usually control groups (assemblies of senior representatives) that, subject to an appropriate majority, can replace the principal person. This also may apply in another critical field: the military context.

The idea of having a system for retrieving a subject's own key, the result of which is verifiable before any use, and which can be triggered through a kind of "majority" within a group of other subjects, has generated an analytical method that has been named "superpassword." [→∬5.2.2]
This method has been generalised and formalised into an already granted international patent, managed by more than 20 professional proxies.
The process of setting up and obtaining a patent, and in particular a software patent, is very long and difficult and has taken up quite some time during the course of the present research, but patents are also one of the main goals of an industrial PhD such as this one, so it was decided to proceed.

1.4.3. Fluid Data

Whatever programming paradigm and language you use you are dealing with two basic elements: code and data. The code enables application logic by managing information through the use of formal references, typically static syntactic elements (e.g., numeric values or alphanumeric strings, to represent hard-coded values) or mnemonic elements (variables, which can also semantically represent constants, but through mechanisms related to the execution engine: these are basically references to data in some kind of memory).

Data is processed by applying more or less complex functions, in which the above references become the arguments.

Let us take a simple example in the C language:

```
#include <stdio.h>
void main() {
	int x=12, y=2, z;
	z = (x+y)/2-7;
	printf("%d\n", z);
}
```

In this case there are 6 references: variables “x”, “y” and “z”, and constants “2”, “7” and “12” .
The first line inside “main” declares the three variables, setting also some values.
The next line applies some functions: assignment, grouping, sum, division and subtraction. It is done using infix operators, that is a convenient way to apply mathematical functions, while the same could be achieved with a chain call - providing the corresponding functions are actually defined - with something like:

```
assignment(z, subtraction(division(grouping(sum(x, y)), 2), 7))
```

A feature common to essentially all common programming languages and environments is the fact that typically, to apply a function that uses a datum, it is necessary to know the value of the datum. In the example above, in order to execute the assignment function (in the snippet represented by the infixed operator "=") all the data involved must first be correctly computed.

There are actually some nuances to this, as in the following example in Python:

```
def T():
    print("T")
    return True
# #enddef T

def F():
```

```
    print("F")
    return False
# #enddef F

print("  I: ", F() or  T(), "\n")
print(" II: ", T() or  F(), "\n")
print("III: ", T() and F(), "\n")
print(" IV: ", F() and T(), "\n")
```

The execution of which produces the following result:

```
F
T
  I:  True

T
 II:  True

T
F
III:  False

F
 IV:  False
```

The two functions “T” and “F” simply return the values True and False respectively:
The final four lines call these functions as arguments to the boolean functions "or" and "and" changing the order of the operands from time to time.
In cases "I" and "III" we can see that both functions are actually executed (this is evidenced by the fact that each of them as the first action within it emits an output, just to get feedback that it is executed), while in cases "II" and "IV" only the first operand is used, while the second is ignored.

This technique is known as "short-circuiting" and is commonly applied to Boolean functions and operators by exploiting certain peculiarities: in particular, in case II we exploit the fact that the

"logical or" operator takes on the Boolean value "true" if even one of its operands is "true," just as in case IV we exploit the analogous fact for the "logical and" operator in which, however, it is verified that if even one of the operands is "false," the overall result is necessarily "false."

This technique avoids potentially unnecessary processing, thus providing increased execution speed and efficiency. It should be noted that if the operands involved cause side effects, these occur only for those actually computed, so it is up to the code designer to consider such effects. Of course, it is always possible to call all operands before applying the link function, so that side effects can be activated if desired.

Another aspect that should not be underestimated is the fact that in cases similar to the above, the execution engine could compute the various operands in competition (in parallel, if possible), possibly interrupting the processes once the overall computation is defined (e.g., in the case of a chain of operands connected with a "logical or," one could process the various operands in parallel, interrupting the ongoing ones as soon as even one is evaluated as "true"). For this latter variant, there is a need to take into account that any side effects would be executed until the eventual interruption of processing (so if an operand can potentially generate multiple side effects, only a portion could be completed).

In this research, a new data classification model was developed that takes full advantage of some of the above features, generalising the possibility of using data as operands even when their value is undefined or only partially calculated. [→∬5.2.3.]

1.5. Structure of the Thesis

The thesis is organised with an introduction in Chapter 1 that presents the general background and all topics addressed, underlying some innovative aspects of the research. Chapter 2 describes the current state-of-the-art of programming paradigms and tools for IoT systems. Chapter 3 lists the main problems encountered and addressed by adopting currently available solutions. Chapter 4 describes

the approach taken to overcome the stated problems and also presents some real-life work cases in which the results of this research were then applied in practice. The experimental results and peculiar solutions that have been studied and implemented in practice, are described in chapter 5, while chapter 6 recaps the current status of developed framework based on proposed architecture. Finally, conclusions can be found in Chapter 7. Chapter 8 is an appendix with some technical insights and significant parts of developed code.

2. State of the Art

2.1. Platform and paradigm

The most widely used platform at present for managing systems such as those discussed here is Node-RED, based on the NodeJS framework, entirely founded on the JavaScript language [26, 30, 31, 32], built around the FBP paradigm.
The choice of a JavaScript tool is particularly suitable for distributed systems on web-exposed networks, since it is the standard programming language for browsers and can be exploited both client-side and server-side.

The official documentation has the following annotation:

Built on Node.js
The light-weight runtime is built on Node.js, taking full advantage of its event-driven, non-blocking model. This makes it ideal to run at the edge of the network on low-cost hardware such as the Raspberry Pi as well as in the cloud.
With over 225,000 modules in Node's package repository, it is easy to extend the range of palette nodes to add new capabilities.

[from official Node-RED website]

Node-RED has a huge amount of configurable "ready-made" modules, and it is very easy to create new ones. Customization then is very easy to initiate, since logical flows for processing information can be created quickly.

It directly inherits major advantages and disadvantages of the underlying engine and language used as a base, namely NodeJS and JavaScript.
We note among the advantages: modularity, total transversality (JavaScript engines are available for all popular systems, both for local and networked execution, particularly via the Web and for use via a browser), debugging (being an interpreted language, step-by-step execution is possible).
Disadvantages include: strong interdependence between modules (dependency chains often become inconsistent), proliferation of libraries and support resources (even small projects may require the

import of hundreds of files perhaps for the use of very few sub-functions), low tolerance for certain types of errors [see ∫ 3.2], high difficulty in handling inter-process communication, synchronisation and concurrency.

The platform has a huge amount of "half-ready" modules, and for the basic language (JavaScript) there are ready-made functions and libraries in large quantities. Support tools (e.g. for debugging) are also widely available: moreover considering that the main mode of use of the finished application is typically a web browser, it is very convenient to handle interactive debugging (since all popular browsers natively support the language).

Since it is based on the FBP paradigm, it offers its advantages, but it suffers from its limitations, and in particular from the inability to handle dynamic changes (any change in the "Node-RED program" requires a new deployment and thus an interruption and restart) and to be able to effectively intercept errors. These issues are highlighted in section 3.

It is also critical to point out that the FBP paradigm should be run in a fully concurrent environment (allowing for good parallelism in task execution) whereas the totality of JavaScript execution engines in popular browsers are essentially single-task.
It is still possible to handle asynchronous and concurrent activities, however with extreme care in the development of the logic, which must be concerned with handling possible conflicts, locking, and crashes.

Another critical point is related to the fact that the referenced paradigm is not embedded in a broader context of an "architecture": this implies that a general method of setting up an application is defined, but it is necessary to handle "manually" divine details albeit recurring. This last aspect e.g. implies that there is no already established system to manage data security or concurrency of activities, elements that-as better illustrated below-are of high relevance in our case.

While many approaches are possible, most IoT systems are set around data management, so FBP - as the only paradigm or as the main one [36, 37] - is an absolutely prevalent choice, especially with interactions across the Web where the reference language is the worldwide standard for all browsers.

2.2. Framework and language

Since the reference tools (particularly the Node-RED tool, but more generally the underlying JavaScript language) generate a "web" type application, the entire system is set up with a development environment that is also entirely web-based. The main engine through which the application is built is completely visual, except possibly defining additional snippets, which, however, tend to be small blocks of code (e.g. to be used within nodes, normally with a simple "copy&paste").

Basically, the fruition of the development environment is through the use of a Web browser. The advantages are those related to the confidence with it (e.g., the ability to use the widely established built-in debugging tools), but there are some important limitations:

- the possible different development and usage experience depending on the browser used
- the fundamentally non-multitasking mode of JavaScript's runtime engines

A final important limitation that we would like to emphasise further is the fact that the development environment and the execution environment are basically integrated: in fact, when you "publish" a Node-RED application you actually have to enable the entire environment, thus also exposing all the editing tools.

having the JavaScript language as the basis of the environment, there are numerous supported frameworks and lots of libraries available [38] with different advantages and disadvantages: using snippets or libraries to handle recurring cases (e.g., for information synchronization or for encoding through hashing algorithms not natively implemented in the language engine) can obviously expose one to risks of various kinds, not the least of which is the fact that one does not have direct control over the logical activities, except to analyze the codes used punctually (if open source, clearly).

2.3. Comparing with innovation introduced

The next chapter describes in more detail the main practical problems encountered, addressed and then solved. This section proposes a very simple schematic comparison of some substantive elements between the state of the art and the proposed new solutions (for simplicity just referred to as FBP and EFA, respectively)..

As in part already mentioned, the currently most widely used reference model rests on a paradigm that was conceived many years ago and developed in synergy with its JavaScript-based software implementation, thus in some ways also suffering from the limitations already discussed and in any case not delving into a shared standard.
A small parallel can be drawn, to make the idea better, with the "Logo" programming language, which, while laying some foundations for a formal evolution of an innovative paradigm (mainly procedural-functional, but with the peculiar feature of graphical representation by means of its "turtle"), has experienced independent evolutions related to the different implementations for the many platforms on which it was available.[1] It is a bit like how in JavaScript an infinite loop such as `while (true) {...}` is forbidden because it cannot be handled directly in web browsers, which are its most popular environments by far.

A first argument concerns the interconnected components of the system: in FBP the various processes are basically rigidly interrelated, whereas in EFA the approach is totally flexible, effectively separating the structural design part from the data. The original approach in FBP of having single communication packets often ill-fits real systems where digital data are handled more flexibly. As detailed in the specific section [∬3.2.1], EFA implementations can avoid having to rebuild the connection tree structure so that system operations do not have to be interrupted.

A second element of fundamental importance is related to the total independence of the various processes in FBP with the underlying engine that manages the data streams: this can expose implementation difficulties about error handling, which in fact happens commonly [∬3.2.2].

---

[1] “Logo” is an educational programming language created in 1967 heavily based on “LISP” with a particular mixture of logical aspects and interaction: its visual cursor named “turtle” (named after an actual turtle-shaped robot) can be handled with basic instructions. Another intriguing feature is that by creating custom procedures, these are then available as they were built-in, so that not knowing the specific implementation it can be difficult to understand which instructions are built-in and which are custom.

Wanting to somehow identify a somewhat disruptive feature, one can reason that in FBP individual processes interact with the external world basically only with data communication channels (in/out), while in EFA the interaction can also take place through state indicators, including in this case the possibility of a "crash" of the main logic.

Using again a simple parallelism at the practical and implementation level, it is like having a code that cascades a series of externally implemented functions which, however, are subject to possible execution errors: in FBP such external functions would produce an undefined situation in the event of an error, whereas in EFA one can imagine them being enclosed by a simple control code (of the try/catch or try/except type common to many programming environments, as well as a possible timeout system) that allows a valid response to be generated at all times (possibly with values signalling the presence or absence of errors, clearly).
This is a crucial aspect that reverses the idea of having an exception handling system as a basic tool (typical, for example, of the Python language) in favour of more fine-tuned and controlled case management (as, for example, in the Go language).

In ∫3 some specific problems are described individually, while a comparison between the various paradigms considered (EFA-EFP, FBP, EDP) are given in ∫4.3.1

3. The Problems

3.1. What emerged

Many complications arose during the use of Node-RED and other related software (libraries, modules, and others) that were then addressed from time to time by proactive methods until we were able to create a classification to frame them into a few recurring cases, so that we could then develop a deeper logic that completely overcomes them in the new EFA architecture. The next section sets out that classification.

3.2. Key points addressed and resolved

Node-RED is based on a visual editor through which you create FBP-like programs by manipulating various components and linking them in graph form: you can install additional components to have them available and configure parameters directly through a browser. Some settings require custom code (JavaScript, of course) that can be entered directly from the development window. There is in particular a basic element called "function" that allows you to insert a completely custom node containing any desired logic, and it is widely used because it particularly facilitates the ability to manipulate data to pass it correctly along the flow by adapting its format.
Development is basically done through a browser directly on the execution server, and once the changes are complete, a "publish" of the work is done: this really just means to stop any execution that may be in progress (which is based on the previous state), to consider the new version of the system being built, and to restart the various operational streams.

It is critical to emphasise that the Node-RED tool is fundamentally a "monolithic" system so there is no clear distinction between the development environment and the execution environment: in general, installation of the system implies that the editor is always present in the final systems.

An initial security issue emerges from what has been reported, in that since there is no provision for some sort of "compilation" and the editor is in fact always present on the servers, it is necessary to protect access to development functions. In fact, the official Node-RED documentation itself reports:

By default, the Node-RED editor is not secured - anyone who can access its IP address can access the editor and deploy changes.

[from official Node-RED website]

3.2.1. Dynamic structure

As described above, the publication of a new or updated project results in an interruption of the flow and a subsequent restart.

Even changing a single parameter, a minor element (e.g., correcting a small typo in some label text) or perhaps slightly more complex but still simple actions such as deleting or adding related components in an assembly (e.g., adding a "sensor" node to a monitoring unit that controls perhaps dozens of them), as well as, of course, any complex action, requires an interruption of the service.

This is completely unacceptable in some critical situations where there can be no interruption.

Of course, the cases of components that need to be replaced (perhaps for scheduled and periodic maintenance) are also relevant.

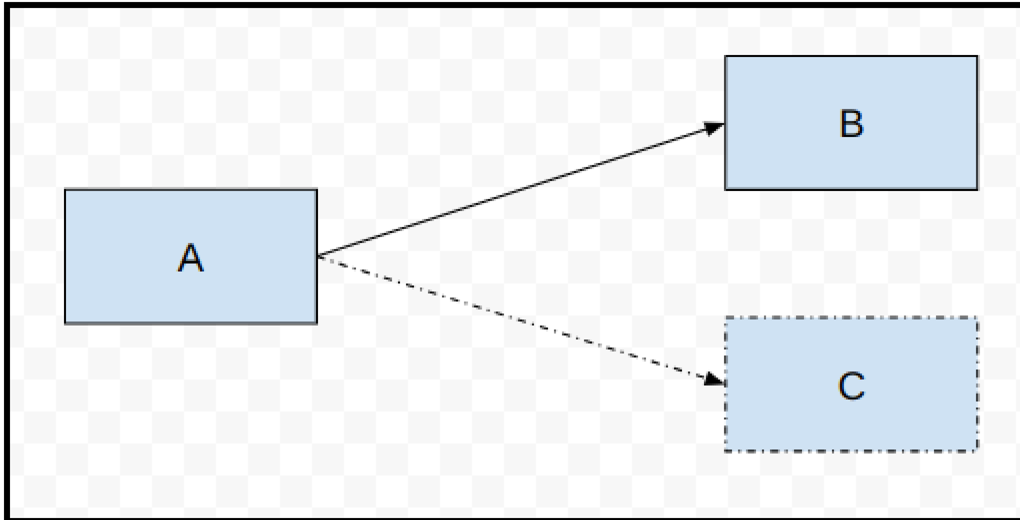
It must be said that the problem of being able to have a dynamic representation that does not necessarily require a "reboot" of the system relates more to Node-RED than to the FBP paradigm itself, which, however, somewhat favours it because of the characteristics of the various components.

In FBP, in fact, each node is avowedly seen as a "black-box" and therefore has no associated external "parameters": it basically reacts to an input by generating an output. Theoretically, one could update a node even with the general application active, but one could have an inconsistent state if the node is processing data.

In EFA this problem is solved by decoupling the processing part from the structural part, and this is basically the same evolution as with the first programmable calculators, separating the basic structure from the code: the individual nodes, the EFA units, are still black-boxes as far as the processing part

(the internal logic) is concerned, but they have a standard interface so that they can interact with the outside world.

Here is a simple case:



node A linked to node B and willing to link to node C at a certain point

Having node/unit "A" linked to another "B" at first and with the need to have an additional one "C" at some time. In FBP node "A" must be built from scratch with two output ports: in Node-RED if a second port is added a new deploy is required! In EFA the units don't have a preset number of channels: seen from another point of view, it can be said that channels are available inside a unit, but are manageable also from outside [→∫4.3]

3.2.2. Error handling

Error management is perhaps the most critical of the elements discussed here. Once again, the first signs come directly from the official Node-RED documentation:

Node-RED provides two ways for a node to report an error. It can either just write a message to the log or it can notify the runtime of the error and cause a flow to be triggered.

If the error is only written to the log, you will see the message in the Debug sidebar and log output, but you will not be able to create a flow to handle it. These are uncatchable errors.
If it does notify the runtime properly, then it is a catchable error that can be used to trigger an error-handling flow.
There is a third sort of error that can cause the Node-RED runtime to shutdown. These uncaughtException errors cannot be handled in the flow and are caused by bugs in nodes.

[from Node-RED official documentation on website, re-edited]

It seems clear that the first two types are not errors from a programming point of view, but logical anomalies related to the context of the data. Only the last case is relevant and it is clearly indicated that it is neither handled nor manageable at all.

What is rigidly stated in the last sentence of the above quotation is then too reductive: it is in fact spoken of as the only actual cause of "bugs." This is true in most cases, but there are also other more complex situations that can occur including possible typos or syntax errors (note that all code is executed with the same execution engine so compatibility issues or deprecated or invalid forms are not verifiable by the developer of the individual node who cannot know in what environment it will be executed) or inclusion of external logic over which one does not have direct control.

It is in some ways a transposition of the system of "catchable exceptions" found in many programming languages (try/except constructs in Python and try/catch in JavaScript for example) with two special features:

- there are no uncatchable exceptions: even a possible syntax error is catchable;
- the "bubbling" typical of exception handling (whereby any unhandled exception is passed to the next context of the execution call) does not generate a "crash" at the main level. This means that even in the event that any exception is not explicitly handled by the application logic, it cannot "crash." Taking an example with a common Python script and standard handleable exceptions, it is as if the entire code were enclosed within a try...except construct.

3.2.3. Privacy-by-design and Access Recovery

Privacy-by-design.

Any application consists of a logic part-typically implemented through code in a given language-that processes information and a data part (the information that is processed). The choice of programming language, paradigm and platform is not indifferent to the exposure of the data itself, which needs to be represented in a specific format in order to be processed. For example, in the case of an object-oriented programming paradigm, encapsulation of actions through the use of specific methods avoids undesirable alterations since it creates, in essence, a centralization of control.

In certain cases it is necessary to have high levels of data protection, so it is necessary to somehow encrypt the information (encryption decryption actions) and use communications to secure partners: in general, data safeguarding can have very complex aspects with variables related to the type of access (read, write, modify) and other parameters. A formalisation can also be defined to represent these complexities [18], but in general adherence to the identified principles remains entirely up to the developer of the application.

In general, "Privacy" is used to refer to all aspects of data access security (in all forms and for all types of actions) and "Privacy-by-design" to indicate an overall approach in which this aspect is a constituent part of the overall design. As with any other type of approach, there cannot be a generalist technical solution, but there can certainly be features that facilitate such an approach (this also applies to the earlier example of method isolation in object-oriented programming: the functionality is there, but it is up to the designer to use it and not to circumvent it).

It was noted that this almost always has no specific facilitation, so it was thought to include some advanced possibilities at the paradigm level.

In the EFA architecture, it is envisioned that each individual unit can generate a unique key used to encode memorised data at runtime: the same key is required for accessing the information. In this way not even having physical access to the data can be intercepted, while interaction with the running process remains the only possible channel.

Access Recovery

Another relevant question that arose concerns disaster recovery: what happens, for example, if an EFA unit as described above (with data therefore encrypted) for some reason has to be replaced (e.g.,

because it fails)? In this regard, the idea is that each unit can generate an encryption key, but also that this key is managed through the "superpassword" system [→∫5.2.2] with the designation of a control group of other units for eventual restoration.

3.2.4. Data Manipulation and Synchronisations

When you need to coordinate multiple asynchronous process units that process data you need to manually implement some pieces of code. For example, in JavaScript if you are calling a remote data ("fetching") you have to create a special handling function that will be invoked at the end of the call, thus generating a branch in the logic: you can possibly draft a more "procedural" code using the "async/await" constructs but in this case you force a kind of "locking" dependent on the execution time of the call.

In EFA, the synchronisation of information can be done either explicitly - for which, however, a particularly effective technique has been designed and developed for IoT, particularly useful for progressive synchronisation of multiple clients to a server and operation even in offline mode [→5.2.1] - or implicitly, taking advantage of a new data classification model and lazy processing of information [→5.2.3]

4. Approach and Activity

4.1. Design and development

The research started by intervening directly in jobs already set up for airpim customers: since one of the key tools was Node-RED, we found ourselves having to intervene with customizations and modifications of it, so the initial reference model was FBP, from which many underlying concepts are derived. Others derived from other models were joined by building new ones as well, as better described below.

All of the cases below are active airpim customers and use the platform I designed and developed based on EFA [→∬ 4.3] already set up with the architecture described in this thesis on a daily basis.

The primary impetus has been to make supporting tools to make the new features that have been studied as transparent as possible, but the ultimate goal remains to make a complete development environment (with dedicated tools and possibly even a new programming language) that generates a distributable application.

4.2. Workcases

The study was carried out, as anticipated, working on real and currently all operational cases. Listed below are significant examples also used as references in the following paragraphs. For matters of confidentiality and industrial secrecy, only descriptive and useful information is given in this paper.

4.2.1. Work Case “Energy”: Energy Supplier Company (gas, electricity)

Leading operator in northern Italy established as a merger of other operators active in the area for many years. Supplies reach more than 250000 end customers.

The job was to build a programmable management panel with real-time monitoring of 8 sections:

- *Lighting*: real-time monitoring of a lighting network with counting of devices by type and detection of those on, off, and in error
- *Water*: real-time monitoring of downspouts, catch basins and irrigation networks, including moisture level verification
- *Environment*: real-time access to general data such as temperature, humidity, atmospheric pressure, and precipitation levels
- *Solar Energy*: updated statistics on KWh production with solar source
- *Parking*: survey of parking stalls at operational units
- *Access*: transit detection of people and vehicles at operational units
- *Charging*: monitoring electric charging stations
- *Waste*: monitoring are waste recovery

For some sections, it was also necessary to implement local sensing (parking lots and access in particular), while others required interfacing with existing subsystems. [Figure 4.2.]

One of the most critical points concerns the management of tanks in the "Water" section because access to physical information is very limited (you basically have only one level sensor for each tank), and some local and some remote logic needed to be implemented. Basically based on the water level, some actions have to be triggered locally (even in case of network problems to the outside) and some remote (whose logic is, however, managed peripherally).

The new "Fluid Data" model [see ∫ 5.2.3] has been exploited for this section.

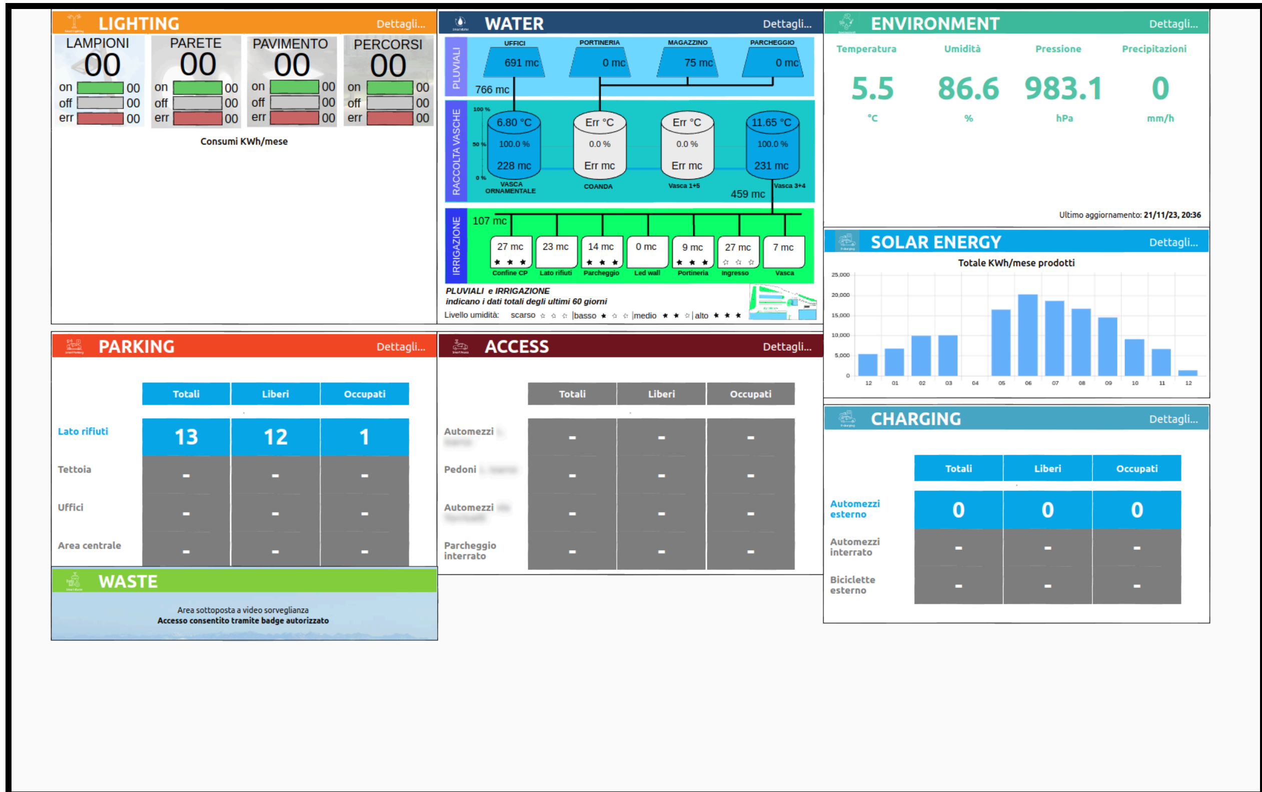


Figure 4.2.

Work Case "Energy": interactive interface with an overview on all subsystems and sensors (actual running instance of openbridge [∫6])

4.2.2. Work Case "Cloud": Global Ceramic Industry

Leading industrial player in the global ceramic industry with over 1 billion of sales per year, more than 3000 employees and more than 70 millions mq of production capacity. It is a corporation covering 12 companies each with its specific products and collections.

The work mainly concerns the management of data coming in from the group's companies, which are distributed all over the world: this data-as well as being used as a confidential archive-generates reports and triggers other types of automation, such as automated alerts. The information has different levels of confidentiality, but basically four can be identified:

- externally accessible data (with sharing web links and access passwords)

- data accessible by all operators (several hundred)
- data accessible only by a few local administrators
- data accessible only by a general administrator

For the highest level of confidentiality, a full encryption system has been agreed upon, whereby the access key functions as a decryption key, and without them the data cannot be accessed even with physical access to the storage servers. Moreover, in the event of an attempted breach the data, according to a particular policy (typically a maximum number of attempts), may eventually be destroyed. Each company in the group has a superadministrator - a figure designated by a kind of council of people appointed for this purpose who also intervenes to act in his place in case of temporary unavailability by indicating a substitute - who is the only one with such a key. When a turnover of such superadministrators is necessary (due to end of term or other reason) the key is obviously passed to the successor (who then updates it).

The “Superpassword” (∫ 5.2.2) solution has been applied for a branch of this application.

4.2.3. Work Case “Sync”: International Natural Stone company

One of the most successful international natural stone companies in the world with a huge purchasing and sales network in Australia, Scandinavia, Korea, the USA, Africa, and Brazil.

The B2B approach requires sellers to be able to browse the complete catalogue of available products constantly updated from various locations around the world, through a dedicated application that must also be able to operate completely offline.

The operational request to airpim consists of various supplies, but in particular the availability of a mobile application (implemented as a “Progressive Web App”, a web-based application) that can synchronise data with the general database (reachable on a custom server) even in a partial and progressive manner, which can be used in fully offline mode and with automatic data realignment when switching to online mode.

Consultation of the catalogue is done in front of the end customers so it must be possible to view the large amount of photographic material in particular: in the first stage quickly though not at the

highest quality (e.g., while scrolling through the catalogue), in the second stage at high quality (possibly by preloading the data ad-hoc). Product information (many thousands of fact sheets and dozens of images per document) is typically updated a couple of times a day in a scattered manner.

This job was the most complex test bed for the required synchronisation strategy, so the "Starsyncing" method (∫ 5.2.1) was appropriately applied-refined to the best of our ability

4.3. Principles, basic concepts and building blocks

It is necessary to provide a description of the constitutive elements of the proposed paradigm, listed in ∫1.3. It is worth noting that the overall idea is to have a single development tool without having to integrate multiple environments and languages, with a tierless approach [35]

First, the **system** as a whole is in principle a mesh-up of several subsystems which may also be characterised by different architectures. Obviously, in order to be able to interact, certain minimal requirements must be met, but these are fundamentally always present in all software solutions and certainly in all the cases discussed in this study. Two simple basic requirements have been identified:

- the ability to activate a subsystem and verify its effects (this is normally done by passing an input to the subsystem and retrieving its output)
- having knowledge of the general state of the subsystem, at least as a condition of a state of the type "ready"/"running"/"unavailable"

It should be obvious that these are not at all restrictive requirements: basically it simply must be possible to reach the subsystems and use them

Individual subsystems are referred to as "units" (a single one is a “**unit**”) in the jargon proposed here: the level of granularity depends on the specific needs of the solution being adopted and the freedom of access to those components.

The following are some real cases of units encountered during this study:

- Work Case “Energy” (see ∫4.2.1): each of the 8 monitored sections constitutes one unit, an additional one was then further implemented to realise the data management and collection server. However, some sections are actually EFA subsystems, such as that of the “Water” where in particular, a subunit was made for the tanks that controls an additional 4 sub-subunits (one per tank)
- Work Case “Cloud” (see ∫4.2.2): there are 14 instances of a cloud system with a couple of thousands of users and some millions of files managed everyday. A branch with some advanced features is in progress to enable refined data management so that the data are encrypted and accesible through a specific key that is, however, easily retrievable through a combined action of multiple responsible.
- Work Case “Sync” (see ∫4.2.3): the main system is a kind of huge catalogue for thousands of records, each with dozens of images. Server-side there is complex management that integrates with other management and ERP software, while end operators use a specially developed app that uses the star synchronisation technique (see ∫5.2.1) so that the process is as streamlined as possible.

Each and every unit interacts with others through one or more channels (“**channel**” in the singular): communication typically takes place through established protocols: in the totality of the cases examined, systems are always exposed on the Web although almost always through proprietary, high-security subnets (VPNs). All cases examined use API interfaces except at the lowest levels for sensor access, which is wired directly via hardware to special devices that map physical information (analog and digital) to a local file-system.
It is of critical importance that any implementation can be realised effectively with simple "API" calls between components (this includes the ability to easily build "PWA" applications, which are very common in the IoT environment) [19]

A channel is identified by a tuple of 7 elements:

- the source (one or more units, possibly can be empty)
- the destination (one or more units, possibly can be empty)
- the data in transit (if any)
- the time instant of input of the data (i.e., when the source produces it: possibly null if there is no data)

- the time instant of output of the data (i.e., when the destination receives it: possibly null if there is no data)
- a temporal context of definition
- a temporal instant of "embedding" (possibly null)

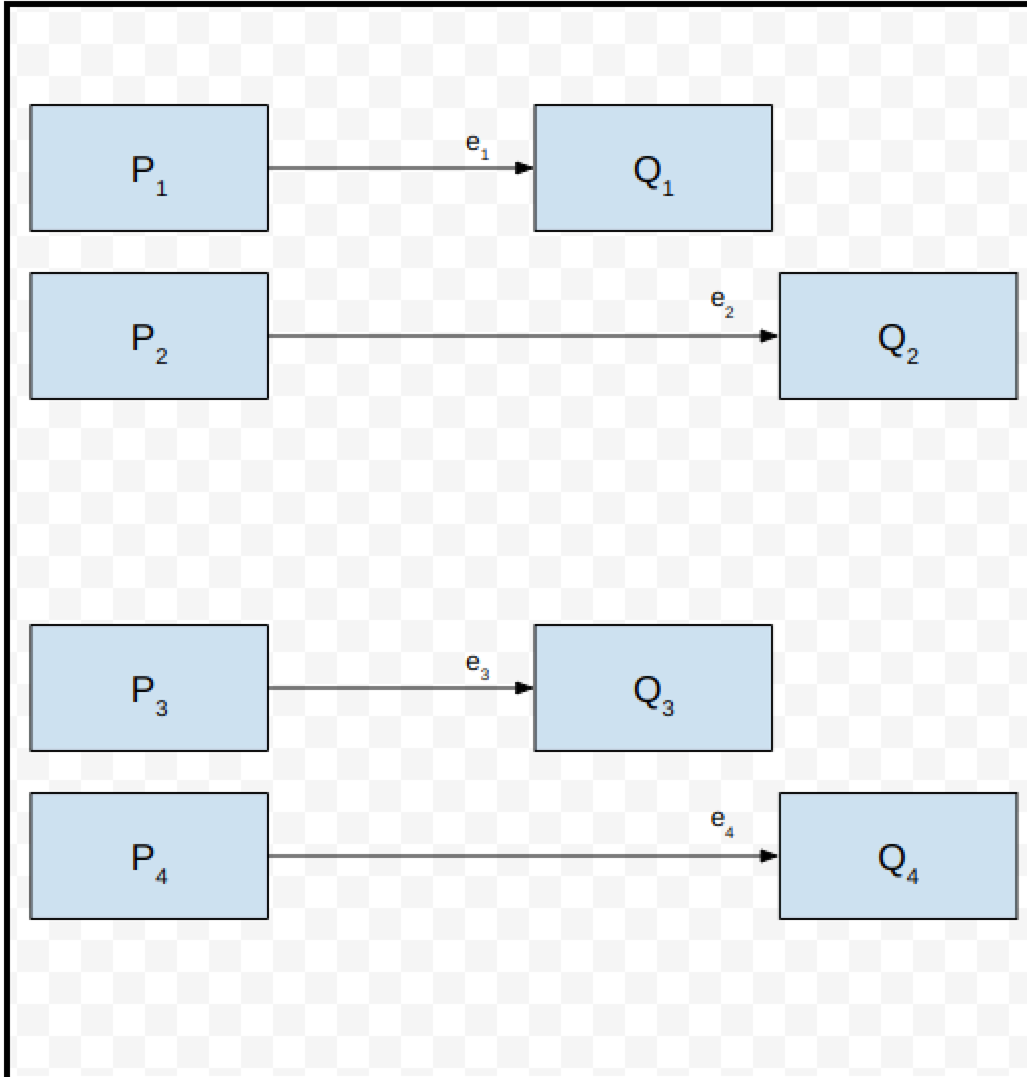
The first five elements are source, destination, data and time instants related to data (of input and output): they are simply the linked units and the information travelling from source to destination (and the time marker when data is produced and delivered).

The last two elements, on the other hand, refer to the fact that each channel is embedded in a "time frame" on the basis of which an ordering between channels can be defined as described more fully below.

In the following diagrams:

- each process is represented by a rectangle with the reference identifier inside (e.g., P_1).
- a connection arrow, oriented, between two rectangles with an identifier annotation above it (e.g., e_1) represents an event (e.g., transmission of a data item) that is triggered in the direction indicated (e.g., a data item travelling from P_1 to P_2).
- a line intersecting multiple connection arrows is used to represent a temporal dependence between connections by annotating the event identifiers in order (e.g., if events e_1 and e_2 are to be met only in that order, there will be a line intersecting the corresponding arrows with the annotation "e_1-e_2")
- it is also possible to indicate cascading sorting by drawing additional intersecting lines as in the last example below

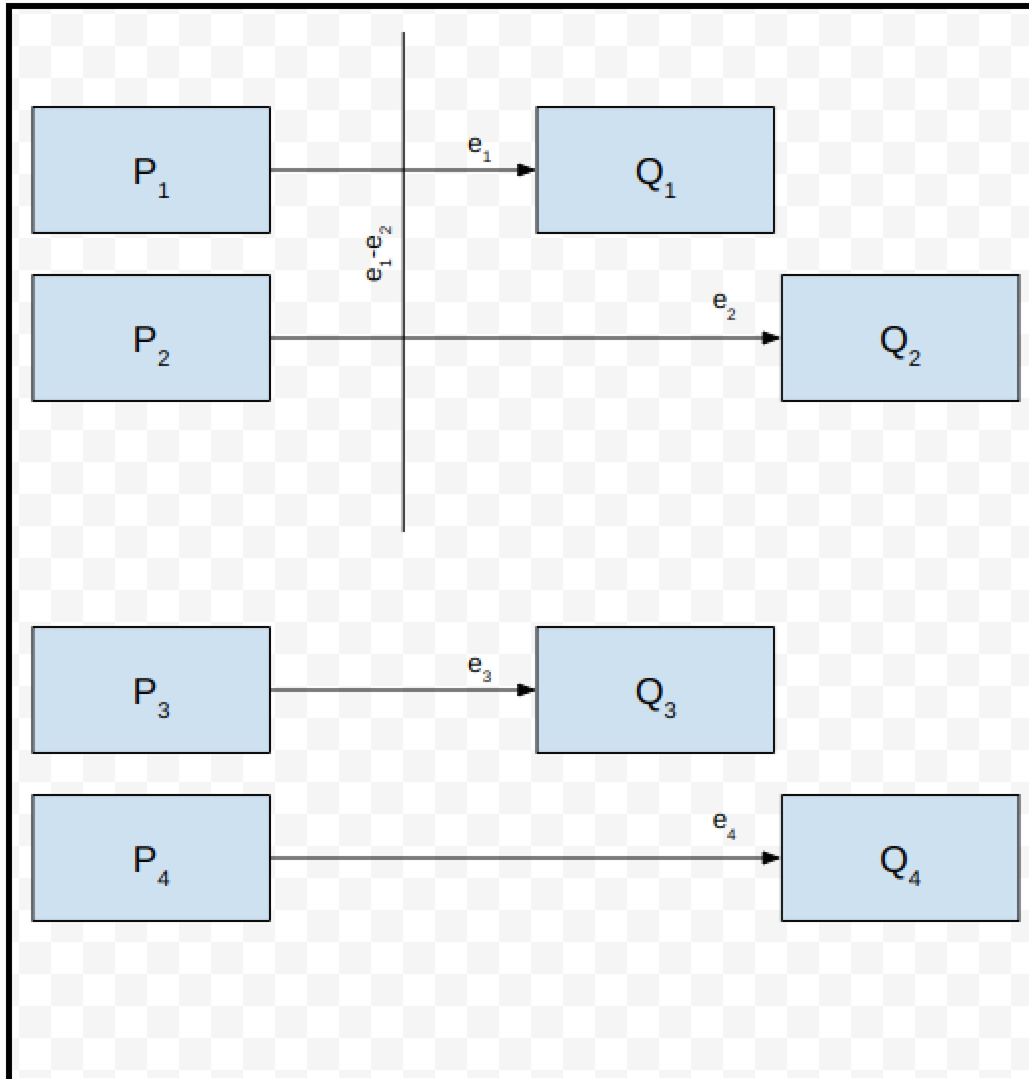
Consider four processes P_1, P_2, P_3 and P_4 that produce data for Q_1, Q_2, Q_3 and Q_4, respectively.



4 producers + 4 consumers processes, all independent

With basic modelling, the 4 events (generation given by a P_i process and delivery to the corresponding Q_i) are completely independent.

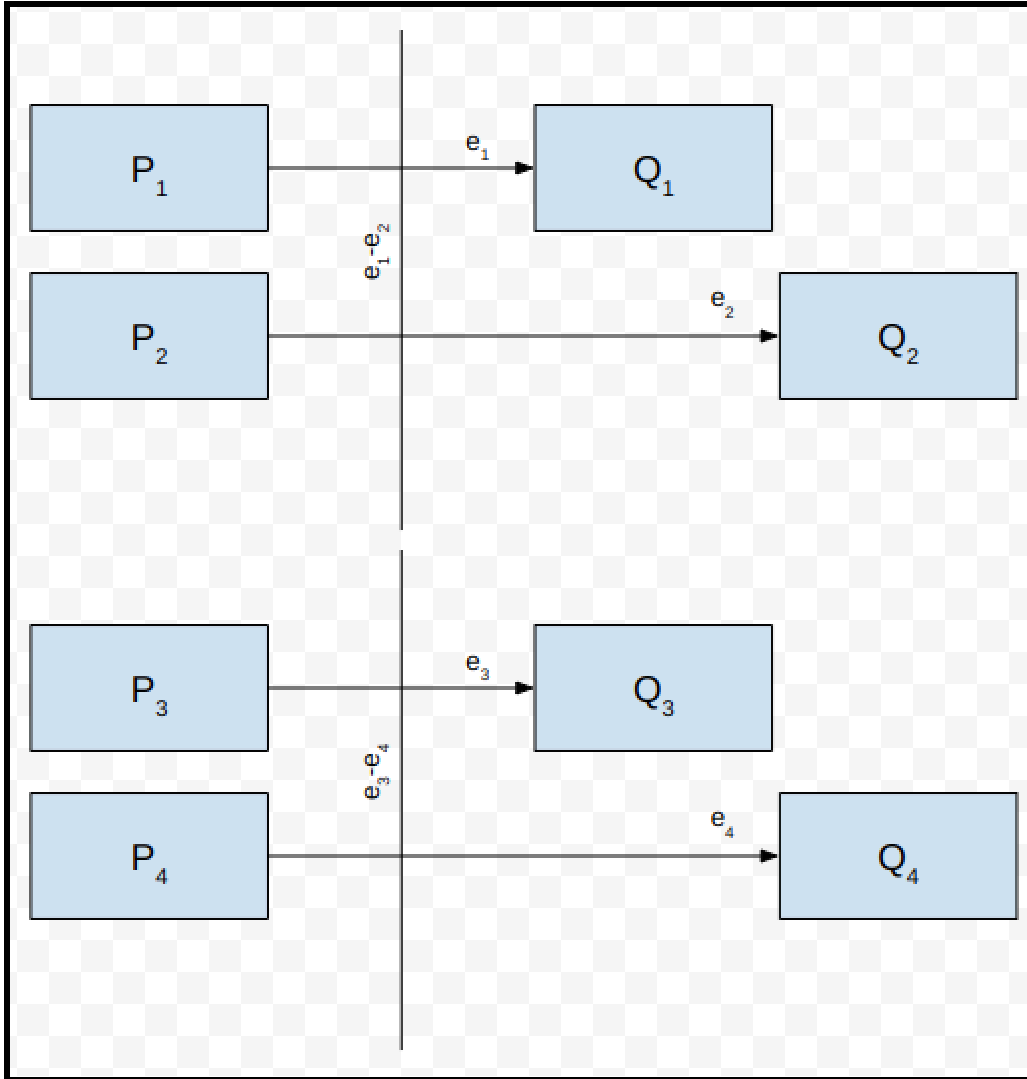
However, in EFP it is possible to organise events on timelines and coordinate them with each other. For example, one can connect e_1 and e_2 with a connector that makes explicit an ordering between them with e_1 having to precede e_2 (connector "e_1-e_2"):



4 producers + 4 consumers processes, having e_2 depending on e_1

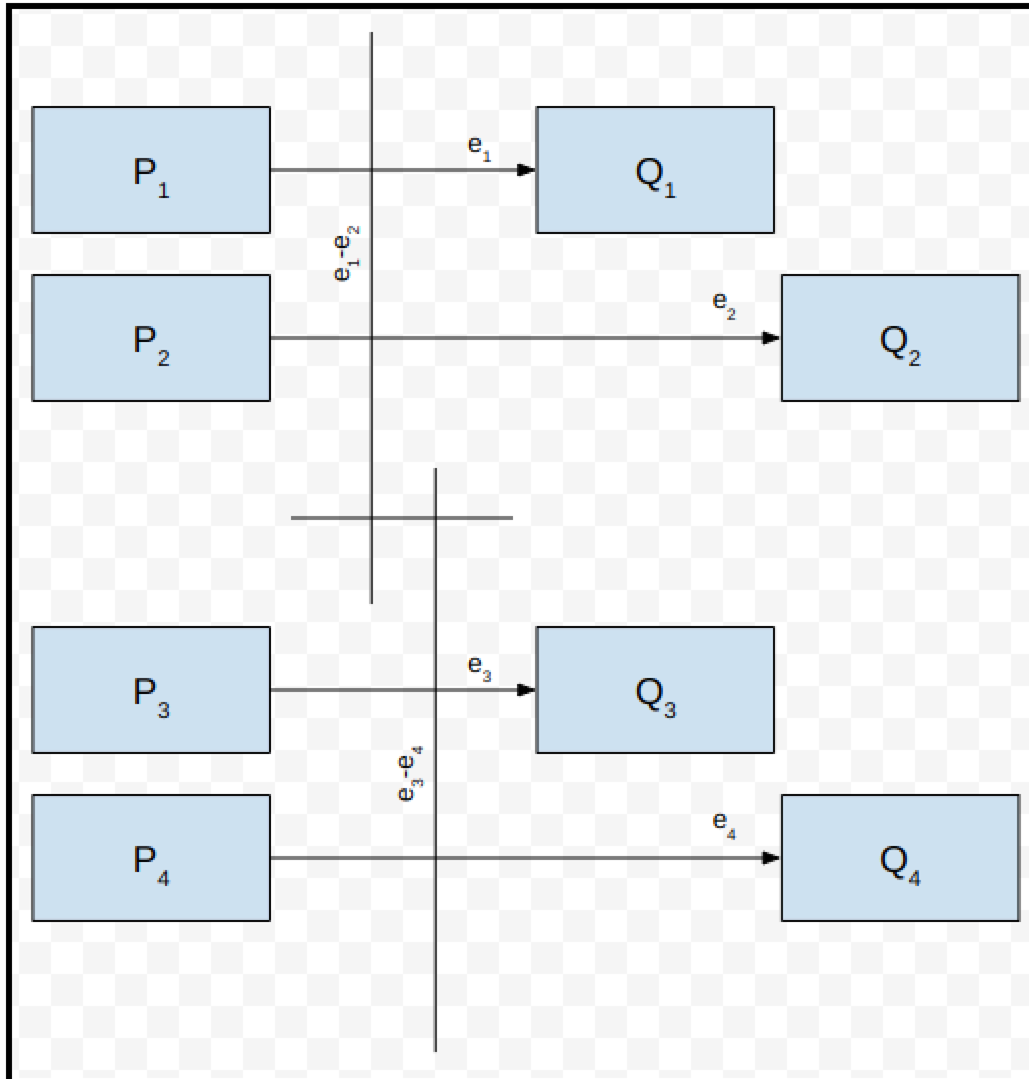
In this way we want to mean that if e_2 occurs before e_1 then the system should NOT deliver the data to Q_2, but wait until e1 also occurs and then proceed.

A similar condition could be imposed for events e3 and e4, but without interfering with the former:



4 producers + 4 consumers processes, having e_2 and e_4 depending respectively on e_1 and e_3

You could even link the two connectors together and create an additional dependency. Graphically, one could simply use a spatial ordering to indicate that e_1 and e_2 must occur first and only then e_3 and e_4 (again with the respective partial ordering already indicated):



4 producers + 4 consumers processes, having e_2 and e_4 depending respectively on e_1 and e_3 but also e_3-e_4 depending on e_1-e_2

Representation

The previous representations are convenient for small patterns, but for more complex systems it is worthwhile to take advantage of a more common graph-based representation (this is also with a view to possibly using facilitation through the various algorithms available for such structures). A natural representation of an EFA application is obtained, then, using a directed graph with some advanced decorations.

We assert a one-to-one correspondence between an EFA system and a graph as recalled below and further specified. A graph that meets the characteristics listed below is called an "EFA graph".

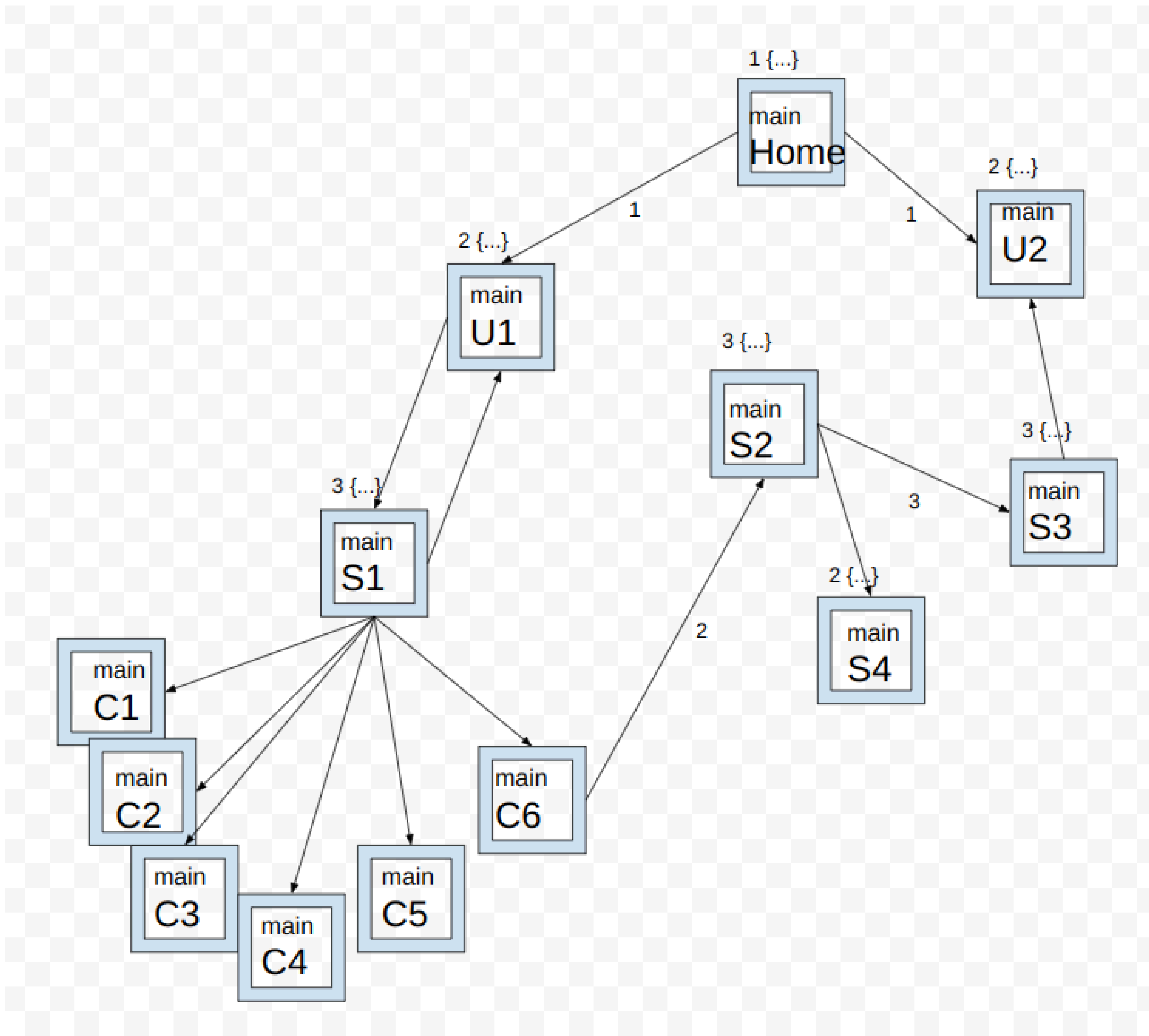
We consider a weighted, labelled, annotated, directed (possibly disconnected) graph G, that is an ordered pair (V, A), so that: G = (V, A), where:

- V is the set of vertices
- A is a set of ordered pairs of vertices, so called “arcs”

All elements (both vertices and arcs) may have (optionally) a label, annotation and weight to define the **attributes** (and so the **state** [→∫1.3]):

- labels and annotations are structured information (can be easily represented with JSON objects or similar structures, or in abbreviated form).
    - Labels have at least two keys with string values: "kind" to indicate the type of an element and "name" to indicate its name. For example, a vertex might have a label {"kind": "main", "name": "Home"}. The abbreviated form can be used if there are no confusions or other keys, and is obtained by writing the value of the "kind" field small with the value of the "name" field below it.
    - Annotations have two keys: "system" and "config". The latter contains arbitrary, application-dependent key-value entries, while the former contains at least the sub-entry "status", whose value can vary depending on the application, but with at least an indication of the progress of the internal process (e.g., with values such as "waiting," "running," "error"). **This value is typically updated not by the individual element it refers to, but by an outer element or by the main engine**..
- weights are signed numerical values.

Using a polygonal shape to draw a vertex (a circle or a rectangle) and lines to draw arcs, labels are written inside the shape for vertices and above the line for arcs, while annotations are written respectively outside and below. Weights are written near annotations.



schematic diagram of an EFA tree

An EFA application may be represented by an EFA graph G, where:

- G is the application itself
- V is the set of units
- A is the set of channels

Here it is possible to see an initial evolution from the FBP setting: like the latter, it is possible to have a visual representation of the application via a graph, but in addition to it, each operational node (black box in FBP, unit in EFA) now has additional information managed outside its logic. This goes beyond the FBP's narrow concept of "black-boxes," while maintaining the idea of their autonomy: in

particular - as you should have already guessed - the "system.status" entry of an annotation is used to handle any serious errors without causing the entire system to crash.

To represent any temporal channel constraints it is not necessary to use explicit graphical structures (as in the examples above) as it is also possible simply to indicate this information with specific annotations: in this regard it can be particularly convenient to use "weights" to indicate the time instant of reference for example (if, for example, an event has no "temporal dependencies" one can give 0 weight, while for those with dependencies it is sufficient to give a weight greater than those on which it depends).
The main reference structures are derived from those of FBP [39] to which, however, the refinements discussed in Section 3.2 must be applied: in one of the research sessions in which I interacted with mr. J. P. Morrison, he said that he would actually like to apply some of these features in a later evolution of the paradigm (he spoke of a hypothetical "FBP 2"), but his reference company did not push in that direction as it was not within the core business.

4.3.1. EFA-EFP vs FBP/EDP

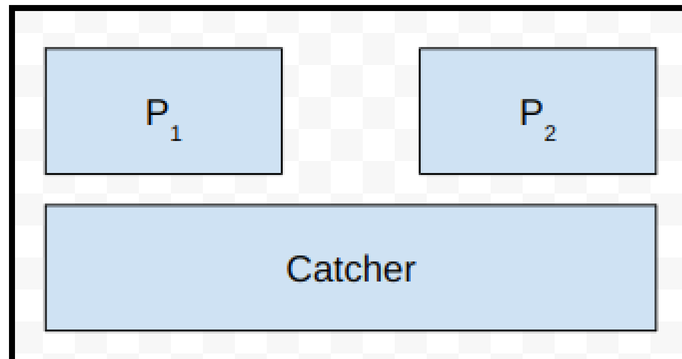
How does this new architecture and its paradigm compare with those that partly stimulated its development? In particular, it is useful to see some key differences between the new EFP paradigm and the FBP and EDP paradigms.

4.3.1.1. EFA-EFP vs FBP

In FBP each process unit is avowedly viewed as a "black-box": the only interaction actually possible is to provide an input and wait for an output [→3.2.1]. In EFP this is only partially true: in particular, it is possible to intercept some secondary effects, not the least of which is a possible "crash."

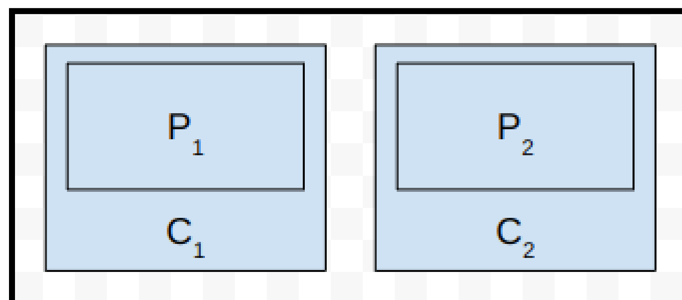
Let us take a simple example with two nodes (for simplicity, no input/output connections are shown) that can generate secondary effects (they can crash or timeout for example).

In FBP a representation can be as follows:



where the two nodes P1 and P2 operate independently of each other and there is a third "Catcher" node that tries to intercept special cases (e.g., some types of errors: note that NOT all cases are manageable [→∫3.2.2.])

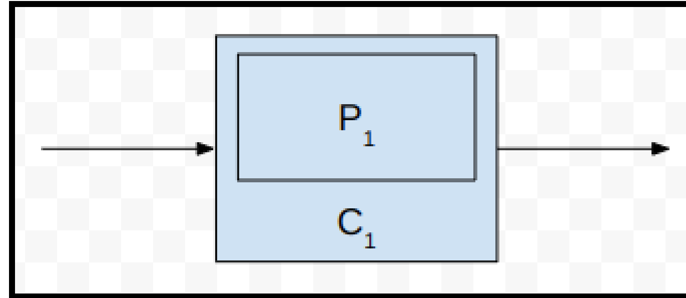
The same application may be designed in EFP as follows:



This is possible because this paradigm explicitly provides for the possibility of having nested operational units, and in addition, exploiting the features described in ∫3.2.2 yields two additional notable advantages:

- any kind of side-effect is managed (including possible crashes!)
- the two nodes/units are managed independently of each other

In general, it is always possible to immerse one operating unit inside another control unit:



In this way the basic unit is in charge of implementing the deep logic of data manipulation, while the external unit can make a control action by, for example, always providing consistent output, which could reasonably always be composed of two elements: the actual data and an acknowledgement of the calculation (a kind of error flag). This strategy is similar to that adopted by some recent widely used programming languages. For example, in GO (golang) there are essentially no "exceptions," unlike many other widely used languages (particularly Python, which instead bases much of its logical error handling on such methodology): in short, each function should output a value consisting of the actual data and an "error" object (possibly "null" in case of success).

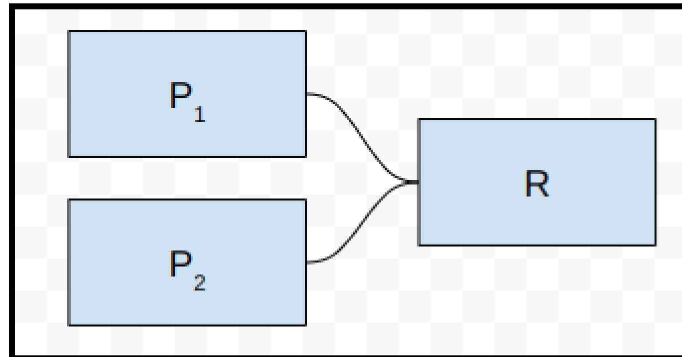
The fact that operational units can interact with the outside world not only through the flow of data in/out has led to considering such phenomena (input = data arrival and output = data production) as "events" and including them in a broader range of other phenomena, so that everything can be managed as a reaction to some kind of event (in this sense, the arrival of a piece of data is not viewed so differently from the possible request for a change of state from "active" to "deactivated", for example: this is part of the idea of being able to dynamically manage the structure of an application [→∫3.2.1]).

A further difference between FBP and EFP is that the latter fits into a broader concept of architecture that natively provides for the idea of encoding data through a dedicated mechanism facilitating privacy-by-design [→∫3.2.3] by otherwise having to specially develop an algorithm in the application that deals with it.

4.3.1.1. EFA-EFP vs EDP

In EDP, events are basically changes in information that can be "observed" by an operating node, which sets up logic (commonly referred to as "handlers") to be triggered appropriately.

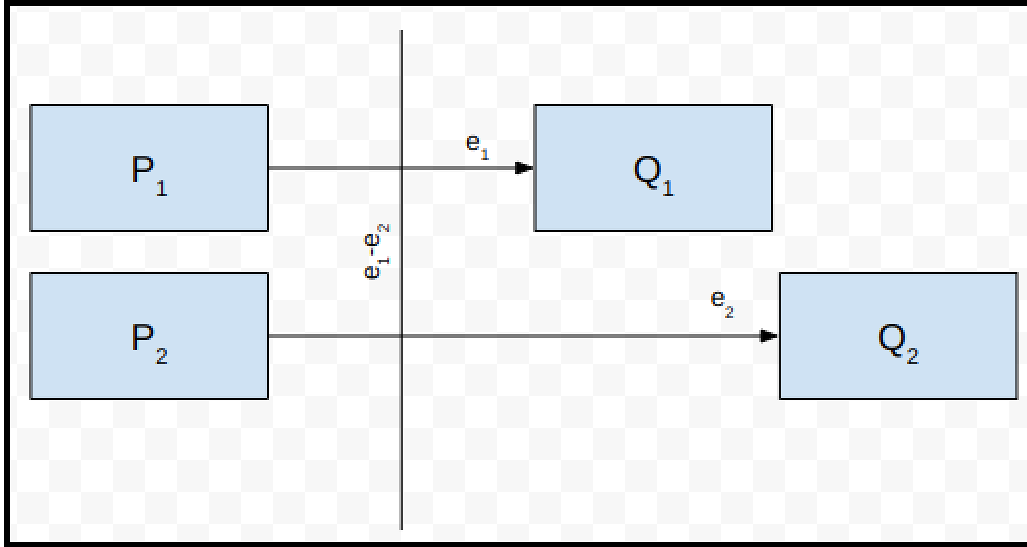
Consider, for example, an operating unit "R" that reacts to the arrival of a piece of data that can be generated by "P_1", "P_2" or both (with different effects)



In an EDP system, the R operating unit could define a handler to execute upon receipt of a data item, at which point it would have to verify the origin of the data item and react accordingly. In this case the event is only one ("data arrival") and the only object affected is R itself.

In an EFP system there is one substantial difference: R always reacts to the receipt of data, but there are now three distinct events, depending on whether the data is produced by P_1, P_2, or both. This last situation is the most significant to describe: if R were to perform some computation only upon receiving data from both of the other processes, this condition would have to be manually implemented internally, and this could be particularly difficult (what would happen, for example, if first a piece of data arrived from only one of the other processes and then later a piece of data from both? R would have to maintain a state of the cases and behave accordingly).

Another key distinguishing feature is the fact that each event can be associated with a timeline so that a kind of "hierarchy" among events can be defined.



In the example depicted, event e_1 occurs when P_1 generates information that triggers the execution of Q_1, so it occurs similarly for e_2 involving P_2 and Q_2: the e1-e2 connector indicates, however, that the two events are not independent, but oriented in time, and this is graphically evidenced by the position of Q2 not aligned with Q1.

The e_1-e_2 connector indicates, however, that the two events are not independent, but oriented in time, and this is graphically evidenced by the position of Q_2 not aligned with Q_1. If we consider P_1 and P_2 to be active, in case e_1 occurs first then Q_1 proceeds and then later when e_2 occurs Q_2 would also proceed, but if e_2 were to occur first then the system would also wait for e_1 to occur and only at that point Q_1 and Q_2 would be triggered.

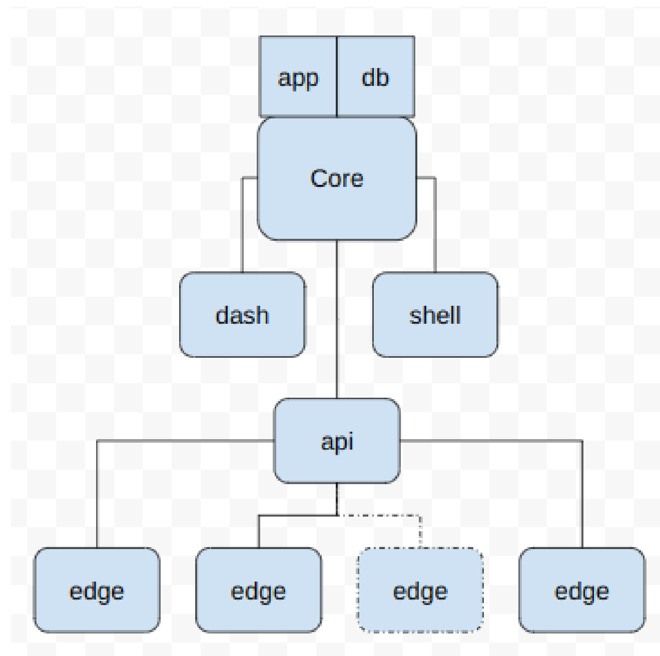
It is then possible to build a "hierarchy" of events without delegating complex synchronisation management to individual units. Fundamentally, this is where the concept of "event flow" comes from since these events can be pigeonholed within one or more timelines.

4.4. Components

The platform implementing the architecture proposed here uses various hardware and software elements, with a high degree of flexibility.

Notwithstanding the fact that through appropriate dedicated operating units it is possible to interface with any third-party system, as a reference structure we adopt:

- a main "Core" module that serves as overall coordination: this in turn is composed of an "application" module (which contains the general logic) and a "database" module (for fine data management): of course, distributed configurations can be created if necessary. The underlying technology used is container technology (specifically the current version uses Docker and Swarm) with a Linux operating system installed.
- a "dash" module that implements the typical functionality of a management interface, with a graphical interface that can be accessed through Web technologies
- a "shell" module that implements a "terminal-like" management interface
- an "api" module that enables interactions through programming interfaces (usually REST-type)
- an "edge" module that is executable even on small systems with low resources and that interfaces with the core through the "api" module: this relies on: this is used to implement peripheral nodes, such as when sensor objects are to be implemented in an IoT system. Among the most commonly used physical media (present in all the work cases presented) are Raspberry and IONO devices.



structure of the openbridge platform

4.5. Benchmarking references

To evaluate operational choices, several benchmarks were made using different parameters as described below.

Time unit. An "operational time unit" (OTU) of 8.76 hours was defined as the base unit: 1 OTU = 8h 45' 36". This time unit represents a work shift in a day (24 hours) with 3 shifts (24/3 = 8 hours) with an additional margin due to overlapping of the band that ensures absolute continuity. The margin was estimated from a series of surveys over the year and then approximated slightly to normalise the final value.

Semester	Number of supports	Total hours	Average hours
2022-I	60	364.80	6.08
2022-II	68	355.98	5.24
2023-I	80	342.80	4.29
2023-II	96	298.56	3.11

Customer care monitored in 2022 and 2023

Several support interventions were monitored where there is a need to consider that the time required is obviously not only related to analysis and modification of software code, but also to interactions with the customer and detection of issues in a timely manner.
From the data collected, there is an estimate of about 4.68 hours per intervention (that is about 0,5 OTU).
It can already be seen that the average intervention time has decreased over time and is significantly lower in the last period analysed: this is indeed a direct consequence of the fact that more and more services have been converted to the new architecture.

Next, it was observed that a work year is commonly represented as 365 days (leap years have 366 days, but the additional day can be "absorbed" with appropriate approximations or calculated separately: this situation did NOT occur during the research because the years 2022 and 2023 were analysed) and you have:

1 year = 365 days = 365*24 hours = 8760 hours = 8760 h

1 OTU = 8h 45' 36" = 8h + 45’(60’/60’) + 36” = 8h + (45’/60’)h + 36” = 8h + 0,75h + 36”(3600”/3600”) = 8h + 0,75h + (36”/3600”)h = 8h + 0,75h + 0,01h = 8,76h

from which we get:
1 OTU = 8,76h = (8760/1000)h = 8760h/1000 = 1 year / 1000 (and 1 year = 1000 OTU)

so 1 OTU is equivalent to one thousandth of a year as a time value.

Salary per year. Since the entire work was done in-house the actual costs for personnel were available: however, additional estimates were made to also calculate external costs (e.g., temporary staff or outside consultancies) and borne by the end client (of which we did not have direct access). This value was then compared with eurostat data and in particular with the “NAMA_10_FTE__custom_4232263” dataset, which reports an average annual wage in the Europe area of 33627€/year. It was found that 1 or 2 human resources were engaged in the various interventions in most cases, thus with an arithmetic mean of 1.5. The weighted annual cost is thus found to be equal to the annual salary multiplied by this average: 33627€/year * 1.5 = 50440.5€/year. Considering that much of the activity was carried out in Italy, which has a slightly lower average wage than the European average, this value was reasonably rounded to the figure of €50000/year.

Taking into account a work commitment of about 42 hours per week yields 2184 annual hours so a full year is covered by 8760/2814=4.01 salaries which we can round up to 4. The personnel cost per time unit thus results in:
€50000*4/year = €50000*4/1000OTU = €200000/1000OTU = €200/OTU

5 Experimental results

5.1. Methodology

All the basic principles were applied in a stepwise manner to active services to airpim customers, modifying the platform with special functions, libraries, and configurations. In most cases, situations implemented with the old architecture could be compared with the new one (sort of A/B testing) since many conditions are recurring.
Three distinctive features that have been studied and implemented are described below, and some comparison reports - which obviously include other secondary features not described here - about the goodness of the new solution follow.

it is important to emphasise two aspects that strongly characterised the present research:

- the conduct of the research took place substantially entirely during the period of the "covid pandemic" obviously influenced the work considerably, also because of the impossibility of being able to carry out some activities in the desired way, with very high limitations in interactions with the outside world and in the use of some operational tools;
- a great number of results are represented by software codes designed, implemented, tested and even put into production (see ∬6) consisting of millions of lines of code that are obviously not reported here: part then went to support the already mentioned patent and part was filed in the Italian public software registry

5.2. Critical features actually applied

5.2.1. Starsyncing

In managing data in an IoT system, issues related to information synchronisation are obviously to be kept in mind: it is common to need to keep several components aligned.

In managing data in an IoT system, issues related to information synchronisation are obviously to be kept in mind: it is common to need to keep several components aligned.
From the point of view of "distributed synchronisation" in general, there are different approaches depending on the needs (e.g., whether it is more relevant to speed of execution, at the cost of having data that is not quite up to date, or accuracy, at the expense of the time required for alignment). [20]
Several studies can be found that look at the efficiency of different approaches to such problems, but they almost always refer to methodologies that primarily view the problem as managing a distributed database with some specialization. [21]
It is worth noting that there are robust generalist solutions in this regard, such as MariaDB Galera Cluster for SQL engines or many so-called NoSQL databases. A key feature of all these systems is also a major limitation for the types of problems analysed in this study: the components cooperating in distributed management are essentially equivalent in power. Taking the case of MariaDB Galera Cluster, the individual nodes are "servers" set up in a particular way that can be primary/master or secondary/slave (depending on the desired configuration) as desired.

Why is the above a very strong limitation in the IoT cases studied here? There are two main reasons:

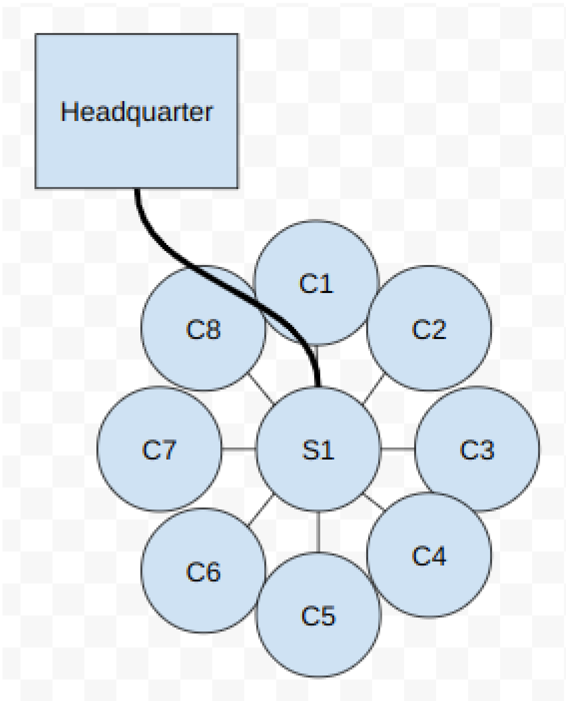
- topology: communication structures are often trees with a few core nodes handling large amounts of data (e.g., control units) and many peripheral nodes exchanging information with core ones
- morphology: individual components are often unbalanced in terms of power, with many peripheral devices of low power and capacity

It was found that the recurring topology for all the jobs analysed consists of a main node - typically a server - and several secondary nodes - typically lightweight clients - that must synchronise information to/from the former.
The organisation of data can still take advantage of a traditional shared database model, however, alongside a new one, that in traditional database jargon can be seen (and used) for efficient synchronisation between a group of clients and a server.
The model designed and implemented to efficiently handle this situation is described below, having found that no available solution was a practical fit (obviously the idea of managing each component as a cluster node for a distributed database is impractical, especially since often the only usable mode of interaction is through interfacing with software APIs (see ∫4.3)

As a real case, in Work Case “Sync” [∫4.2.3] there is a management environment (with various servers and machines attached) installed at the customer's site that exposes a server on a dedicated network accessed by numerous clients.

The following representation shows the main environment, the server ("S1") and some clients (there are numerous, but 8 are shown in the representation, from "C1" to "C8").

The "S1" server makes internal synchronizations with various other systems, while individual clients need to synchronise in a streamlined way with the "S1" server.



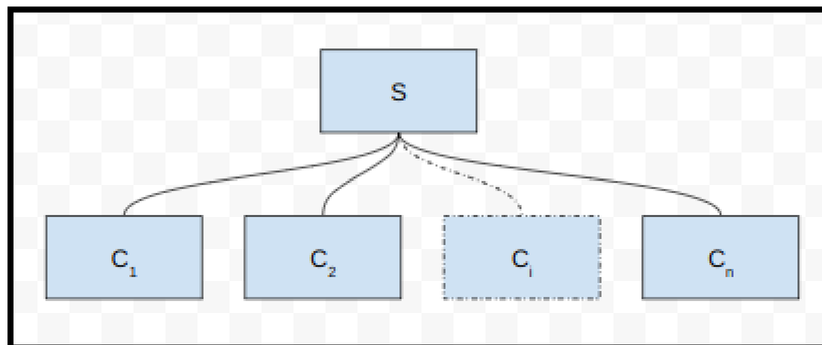
bunch of clients linked to a server, itself linked to an headquarter

In the remainder of this section, references - if not otherwise stated - will be made to relational databases, tables and records only for convenience of description, keeping in mind that all procedures are applicable to any type of database by remapping the concepts appropriately.

5.2.1.1. Topology and Morphology

Let “S” be a main node, a component where the reference database resides, and “C_i“ the secondary nodes that need to synchronise information.

Let "T_j" then be the tables of the various categories of information (at the implementation level they can correspond, for example, exactly to "tables" for relational databases and "collections" for non relational databases) and "$R_{j,k}$" be the items to be synchronised (at the implementation level they can correspond, for example, to "records" for relational databases or "documents" for non relational ones).
Node "S" - without loss of generality - can be a monolithic server or a single node of a complex distributed cluster. The nature of "S" is of no particular relevance: it is sufficient to take into account that it makes data access available.
All nodes "C_i" must synchronise information with "S" efficiently: the goal is that they can synchronise the entire database or a selected part of it using a simple protocol that is easily adaptable even for devices with low computing power and through APIs (see ∫4.3)



clients linked to a server needed to synchronise data

5.2.1.2.Protocol

From the given assumptions, the following can be written:

$S = \{ T_j \mid j \in [1, n_s] \}$ (where n_s is the number of tables)
$\forall s \in S \,.\, s = \{ R_{j,k} \mid k \in [1, n_j] \}$ (where n_j is the number of records for table j)

Each "C_i" node must be able to synchronise information locally: the chosen approach involves an initialization phase in which a copy of the data is made and subsequent "alignment" phases based on a differential update with actions of:

- creation: for records that are generated locally

- insertion: for new records present on the server but not locally
- deletion: for records present locally but not (anymore) on the server
- update: for records that are also already locally present on the server but are changed

5.2.1.2.1. Global Sequential Counter, Data Complexity Level, Record Hash Identifier

Some very simple elements are used to facilitate synchronizations: a global sequential counter, a data complexity indicator and a record hash identifier.

The Global Sequential Counter.

The *Global Sequential Counter* (*GSC* for short) is simply an ever-growing counter: each time any record is updated its value is associated with it and it must be incremented anytime before the next update (usually it happens almost simultaneously).
It is not necessary that the counter produces consecutive values, only that it represents unique values, increasing over time. Nor is it essential that the update occurs only when a record is updated, as long as this condition remains true: a more convenient way is to use a simple time counter (possibly with high precision) whose increment is therefore dependent on the passage of time (and not on an update action). Finally, it is not further strictly necessary that the value that is associated with the update of a record be referable to a single record in the entire database, since it can also be associated with several of them.

When a record (of any table) is updated, the current counter value is associated with it.
At the time of creation you may not associate that value, although for convenience it is preferable to do so, or to assign a conventional value such as 0 (zero), in order to facilitate later procedures.

We denote by GSC(t) the function that returns the current value of the counter at time instant *t*: if a clock is used, its value is simply the current "time" (timestamp).
We then denote by $G(R_{j,k})$ the value of the counter associated with the record $R_{j,k}$.

The Data Complexity Level

In order to better manage synchronisation, it was thought appropriate to be able to differentiate the level of complexity of information because it is possible, for example, that some information may require different degrees of alignment.

The solution proposed here allows any field of all records in a table to be assigned a level of "complexity," so that during synchronisation it is possible to decide whether or not to include that detail. These levels are definable at the lowest level of individual records, more precisely for each characteristic item: for example, referring to a relational database, one can assign such a level to a single field in a table (so each record can have different levels for each column).

Using a positive integer as a reference for the level, one can assign the minimum level (of complexity it is understood) equal to 1 to all fields in all tables and larger values for more complex fields (e.g., larger size).

We call $D_r(R_{j,k}, l)$ the function that, given record $R_{j,k}$ and complexity level l, returns a representation of $R_{j,k}$ that includes all fields of complexity equal to or less than l.

$D_r(R_{j,k}, l)$ = *representation of* $R_{j,k}$ *with all fields of complexity equal to or less than l*

Given a table T_j for a single j, and a complexity level l we further define a function D_t as follows:

$D_t(T_j, l) = \{ D_r(R_{j,k}, l) \mid R_{j,k} \in T_j \}$ (so the representation of a table "T" for a given a level "l", is the list of the representations of all its records for stated level)

We also define a more general data representation function D* ("D star") that given a reference to a table, a complexity level, and a counter value "c" returns representations of the data as follows.

$D^*(T_j, l, c) = \{ r \in D_t(T_j, l) \mid G(r)>c \}$

It means that given a table (argument T_j), a level of complexity (argument l) and a counter value (argument c) we get all the records from D_t (defined above) whose associated counter is greater than the one given as an argument.

Recalling that the counter is a value according to which it is possible to have an overall sorting of records from the least recently updated to the most recently updated, the function actually returns all records updated subsequent to a given reference "instant" (the value "c" passed as the argument).

In Work Case “Sync” (∫ 4.2.3) two levels of complexity were needed: a "high" level for fields containing medium to large resources ("blobs" for images) and a "low" level for all other fields (Booleans, numeric, strings, date-hours). The principle is that it is desirable to be able to synchronise by including or not including large images.
In the actual implementation of the provision currently in use, 9 levels of complexity coded with the numbers 1 to 9 were preset, but of which only two are currently used, based on the type of field: level 9 is assigned to "blob" type fields (large binaries), level 1 to all other fields (for all tables).

The Record Hash Identifier

Almost all of the work considered uses relational-type databases where each table typically has key fields: if not present, it is trivial to add an auto-incremental column to be used for that purpose. In non-relational databases, a key-value type association is commonly exploited to represent information. In all cases the information unit is typically a "record."

- properly so called in the case of relationals, corresponding to a row in a table, identifiable by a key (possibly with the caveat noted above)
- a row of a collection associated with the reference key, in the case of non-relationals

For convenience-since it is always possible to fall back therefore to this model conceptually-we speak generally of a "record" of a "table", identified by a "key".
Taking into consideration a single table (properly for a relational database, or a generic "collection" in the case of a non-relational database, then), we will have a list of fields (columns: a fixed number in the case of relational databases and a potentially variable number in the case of non-relational database), of which one or more have a "key" function (as given above).

A record “R” can thus be represented as a set of 'field-value' pairs that can be partitioned into two subsets: one partition “K” contains the fields constituting the reference 'key' (as indicated above), while the other “M” contains all the other fields.

$R = K \cup M = \{ (k_i, v_i) \mid i \in [0, \#K) \} \cup \{ (m_j, v_j) \mid j \in [0, \#M) \}$
(where k_i are the key-fields, m_j other fields, v_i and v_j the values)

Each individual record is associated with an identifier obtained by applying an appropriate (it is sufficient to estimate the desired length: any modern algorithm that generates keys of at least 128bit is normally sufficient for any application) hashing algorithm to the subset of values v_i:

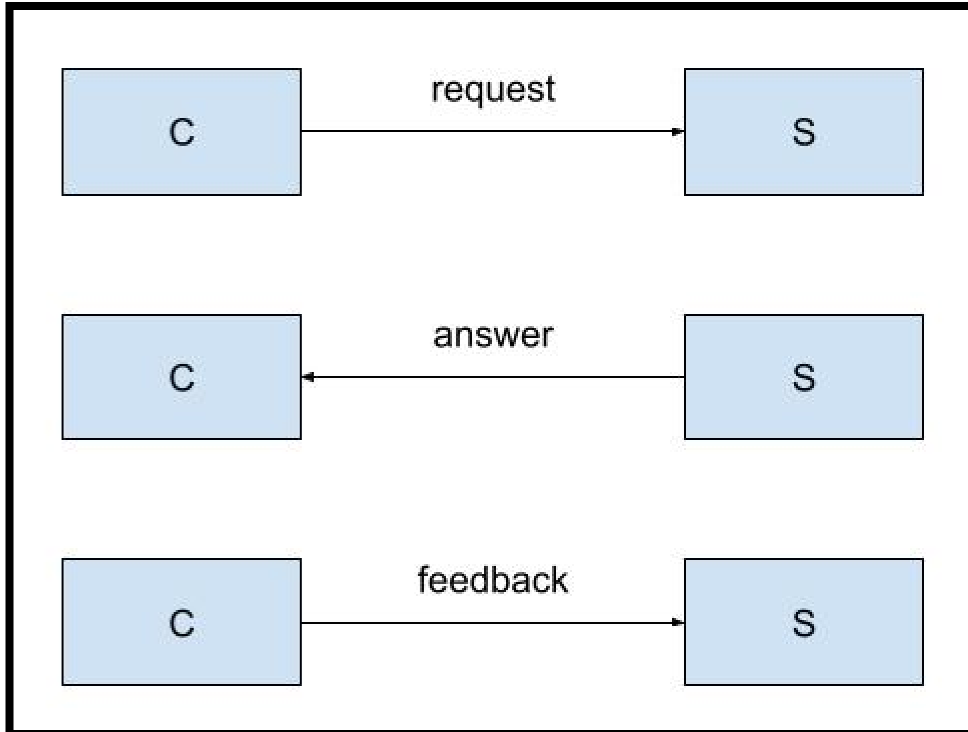
$h(R) = H(\{v_i \mid i \in [0, \#K) \})$ (so called RHI -Record Hash Identifier - of R)
(where H is the function the represents the chosen hashing algorithm)

5.2.1.2.2. Three-steps communication

The actions are performed as follows by each client:

- creation: an immediate message is sent to S and its feedback is awaited
- Insertion: a new record is created locally with data received
- deletion: a local record is removed
- update: a local record is modified according to the information received

These are managed with a three-stage communication protocol: The client requests updates from the server and waits for the response with the data. It then performs local realignment operations and sends completion feedback (with any error reports).



The three-step communication protocol for the synchronisation

In the **request** step, the client sends its own identifier (dependent on the execution instance, not on a user account: the same user could for instance use the application on several devices that must be synchronised independently) and a locally generated authorisation key. The identifier is also present in all the server's replies and is generated by the latter: the first time the client contacts the server, an 'empty' identifier is sent (since it typically consists of an alphanumeric string, the convention is to send an empty string), and the one obtained in the first reply is then used for all subsequent communications.

When the server receives a request:

- if it receives an empty identifier, it generates a new one and associates it with the received security key, effectively generating an account for subsequent communications
- if it receives a non-empty identifier, it checks on the accounting table that the security key matches

With this strategy, the individual execution instances receive an identifier from the server, but independently choose a security key.

In the **answer** step, the server sends the data necessary to provide an answer to the client, along with the identifier: *at this stage it is also possible to introduce a level of encryption by directly using the security key to encrypt the information, which can then, of course, be decoded locally*.

In the **feedback** step, the client updates the server on its status-this provides assurance that the alignment was successful.

5.2.1.3. Actual implementation

In the current working implementation, very complex database models are used (taking advantage of the SQLAlchemy toolkit) including a "Syncing" table for managing communication accounts.

The synchronisation process is managed by the clients with APIs for executing the "request" and "feedback" steps, in the code respectively named "sync/download" and "sync/acknowledge,".

Thus, the process of synchronising data from the server S to a client C is done according to the following steps:

1. C requests an initial synchronisation to S: sends an empty identifier and an autogenerated key
2. S responds with a new identifier and a first block of data: sorts the data according to the GSC and sends the first ones (the quantity is dynamically definable)
3. C receives the data and synchronises it locally: depending on the RHI code for each record, two cases can occur:
    a. the RHI code is not present locally: it means that the record is a "new" record and must be created
    b. the RHI code is already present locally: it means that the record was previously synchronised and it is being updated
4. C sends feedback reporting any errors or correct synchronisation to S

5. S receives such feedback: in case of error, policies of various kinds are applied (one can also simply repeat the operation), but in case of success, the reference of the most recent GSC correctly sent and synchronised is stored locally

The entire sequence repeats cyclically with a variation in step 2: S sorts the data by GSC as before but ignoring all records whose value of that index is no greater than that recorded as the last synchronisation.

Extensive excerpts of the actual code are reported in Appendix 8.4.

Using the described conceptual model and the actual implementation just stated (currently being also updated to increase performance and efficiency), remarkable results have been obtained, some evidence of which is shown below, where a comparison table is displayed for some features with the implementation based on a standard solution (using state-of-the-art technologies with FBP and other tools) versus the one built with this project: obviously where execution or intervention times and costs are shown, the best values are the lowest.

note	feature	FBP+tools	EFA	enhancement
1	pwa's crashes (out of 1000 runs)	234	18	1'300%
2	Recovery time after a crash (average out of 1000 runs)	62'18"	30"	12'460%
3	Force update of records	not available	available	+
4	Maintenance bugs/support	0.33 OTU € 66.00	0.14 OTU € 28.00	236%

benchmarking comparison between FBP and EFA solutions

Notes:

1. "pwa" means "Progressive Web App": it is a web-based application that implements the clients marked as "C" in ∫5.2.1.2.2. Most of the crashes result from the limitations of the web browsers through which these kinds of apps are enjoyed: using the methodology designed for

EFA resulted in easily manageable progressive synchronisation. In addition, the three-step communication technique makes it easy to synchronise at multiple times and to resume from the last correctly synchronised record even after any problem (including the possibility of a hardware crash)

2. in the standard solution there is no certain way to identify records that have already been synchronised, and in particular to determine which ones need to be updated from a previous communication: even if a database provided a specific field with the "last updated date" there would need to be a specific implementation that referred to that field (trivially, the name of the field itself would need to be used in the code, and in case the database was changed the management code would then need to be updated as well). In EFA the use of the Global Sequence Counter and the strategy around it solves this problem.
3. It may be useful to force synchronisation of records for various reasons (even simply in testing): in a standard implementation the only effective way is to "resave" a record, but even then it may not necessarily be seen as "up-to-date" (there must still be some field to account for this). In EFA this action is incredibly simple, and it is also possible to perform targeted actions: for a record to be seen as "up-to-date" (from the standpoint of needing to be resynchronized) it is sufficient to force the update of its GSC. it is also possible to take action on the table that keeps track of the hits provided by individual clients: the server has for each client a list of the last synchronised GSC for each level of complexity, so those with a higher value are extrapolated upon request for data. One can manipulate such a table by artificially inserting lower values as per the following example. Suppose there is a value "x", so at the request for synchronisation the server extrapolates all records with GSC>x: wanting to force a "resync" of all records it is enough to change that value to 0 so with the update of a single value the desired effect is obtained.
4. the estimate is for "living costs" of staff on average: obviously there are notable indirect savings not accounted for in the calculation.

5.2.2. Superpassword

During the execution of work orders imprinted on the new architecture, two special cases happened that stimulated a new line of research and led to the remarkable result of a software patent, as better described below.

5.2.2.1. The case of the lost access

During the development of particularly important evolutions or upgrades, verification scenarios are constructed to test the operation of activities, and on this occasion the event described below happened in Work Case “Cloud” [see ∫4.2.2.], while other samples are reported afterwards..

5.2.2.1.1. Lost decryption key

Two key elements stand out:

- it is necessary in critical contexts to have a system for retrieving access codes through some redundancy of information, but limiting as much as possible the possibility of these codes being available to third parties
- it is necessary that the recovery be absolutely verifiable with total certainty avoiding any issues related to access time but especially to the possibility of loss of information

For Work Case “Cloud” there are currently 14 instances running and each instance is assigned a supervisor. A scenario was tested in which all data is encrypted: the key is symmetric, different for each instance, and known only to the supervisor.
The desired goal - achieved by the strategy described in ∫5-2-2-3 - was to be able to retrieve that password through an action of a qualified majority of the other supervisors, being able to verify that

such retrieval provides the correct data with absolute certainty, because decryption tests are not permitted for the following reasons:

- there is no absolute guarantee in general that one can tell whether a decryption was successful just by looking at the decrypted data: if, for example, such data is a generic sequence of values (e.g., a list of detections from sensors), it is possible that a decryption with an incorrect key will return a list that is "syntactically" valid, but differs from the original data
- the application of decryption on massive amounts of information may require significant resources (particularly computation time)
- some particularly important packets of information are safeguarded with media that admit a limited number of decryption attempts

5.2.2.1.2. Another case: wait for years to access your smartphone

Many current smartphones (and other mobile devices) have a protection system that blocks access to the device and in case of an error imposes an ever-increasing waiting time before a new attempt can be made.

5.2.2.1.3. Another case: "Lost Passwords Lock Millionaires Out of Their Bitcoin Fortunes"

In January 2022, newspaper "The New York Times" published the article "Lost Passwords Lock Millionaires Out of Their Bitcoin Fortunes" about a programmer who has lost the access code for his portable hard-drive containing more than 7000 bitcoins (more than 200 billions of USD). This drive has hardware encryption that allows the user to input the decoding key up to 10 times before erasing everything.

5.2.2.2. The turning point

Information technology has always typically made otherwise burdensome tasks easier, but often by applying concepts that already exist in reality.Building on this, we reasoned about the methodology commonly applied among people to manage confidential personal resources.

In such situations people normally rely on individuals trusted by them, such as close family members or friends. If, for example, a person has a numbered safe deposit box accessible with a (physical) key of which he has only two copies, he might give one to some acquaintance and ask him to keep it for possible emergencies. To create an additional layer of "assurance," he might even not give out the number of the box itself, but tell a third party (but not provide him with any key). Further he could only tell a third party the names of the first two. It should be clear that if the need arises (such as by misplacing the key or forgetting the number of it), action can be taken by piecing together the previous information, noting also that none of the confidants involved can independently access (nor can any two of them) the box itself.

The "digital" transposition of the above example with regard to the storage of a key can be framed in the “Secret Sharing Schemes” themes [22] which address the issue of distributing a key but are by no means conclusive for the previous case since their basic application provides output that is not statically (i.e., without using it directly) verifiable. One of the greatest weaknesses of such schemes is that the stage of recomposing the individual components always generates a syntactically valid and therefore hypothetically correct result.

Surprisingly enough, no existing practical solution has been found in the literature, so much so as to push for a home-made solution. The study then evolved into a paper screened by various experts into a software patent (which obviously focuses on a few innovative concepts while leaving ample room for implementation possibilities), as described in the following paragraph.

5.2.2.3. The invention: superpassword patent

The invention describes a method for managing a secret, nicknamed "superpassword", by generating a set of independent components that can be used to recover the original information using a subset of desired size, having total assurance of the success of the operation through a verification process, with no other data to store.

The components do not contain any information that directly links to each other and there's no external information stored elsewhere, not even for the verification step. Moreover the information on algorithms to be used for the raw decryption are embedded in data.

Technical problems are solved by managing the "superpassword" in such a way that:

- the storage is secure: it is understood that there is no immediate method of gaining access to such information, but only a method that requires knowledge, actions, skills or abilities - or combinations of the foregoing - of the owner and which are hardly pursuable by third parties;
- recovery is possible in an integral manner: the owner is able to fully recover the secret easily and quickly;
- the recovery is verifiable: it's possible to verify the recovery procedure independently of the usage of the secret

(thus analysing the information at that stage and not trivially trying to use the secret itself).

This last feature is of utmost importance as it allows certainty of the effectiveness of the recovery before any possible use. It should be noted that there are systems in which the use of incorrect codes can lead to the loss of the data itself or other problems such as the need to wait for a certain period of time before another code can be tried.

NEWS REFERENCE (sample case). In January 2022, newspaper "The New York Times" published the article "Lost Passwords Lock Millionaires Out of Their Bitcoin Fortunes" about a programmer who has lost the access code for his portable hard-drive containing more than 7000 bitcoins (more than 200 billions of USD). This drive has hardware encryption that allows you to input the decoding key up to 10 times before erasing everything.

IMPLEMENTATION SAMPLE. There is a POC for a software application (a kind of "password manager") based on a network of subjects, each with his own superpassword processed several times by the system in order to generate several valid decompositions, distributing the individual components to other subjects, who obviously receive data from several sources and have no way of distinguishing them, not to allow malicious manipulations. Each subject thus has a way of performing a retrieval using any valid subset of one of the decompositions. The distribution could be made to

subjects chosen at random or only considered trustworthy, but in any case none of the recipients will know the identity of the other participants in this action. An additional step may be applied, storing an encrypted version of the actual secret using a method that is easy and comfortable for the owner to be decoded (for example storing a personal question and the encrypted actual secret to be decrypted using the answer to the question itself).

While reverse-engineering is hard, the direct process is quick and easy to apply. In this sense attack actions toward the system are very difficult. Here's an example to have a rough idea of what can be accomplished: let's take a secret and choose 100 as the number of components to generate the main set and 10 as the minimum size of the subset required to retrieve it; let's then generate 9900 more components starting from additional random secrets, so to have 100+9900=10000 in total. Components are spread to different subjects and only the owner knows which are his 100 trusted ones. There are $O(10^{13})$ possible valid subsets in the main set, but $O(10^{33})$ of generic subsets in the total, so there's about 1 chance out of 10^{20} to get a valid one at random (to perform 10^{20} trials each running in 1ms, millions of centuries are required).

5.2.2.3.1. The algorithm

The algorithm basically treats an input digital data, denoted "superpassword" but which can of course be any kind of information, and two integer values of references denoted "n" and "t" generating in output a list of "n" data such that by taking any subset of cardinality at least "t" it is possible to reconstruct the source information by verifying in constant time the correctness of the operation (few more inputs are taken into consideration in the full process as better explained below).

There follows paragraph 5.2.2.4. with some excerpts from the official documentation, with the clarification that the text is an adaptation of the original technical notes that I had originally drafted partly in Italian and partly in english, carried out in collaboration with the consulting firm that supported the patenting steps: this was necessary partly for logistical needs (the patenting process is very long and complex especially for so-called software patents), partly for formal issues (the ligation of patent documents requires some specific formalisms).

5.2.2.4. Patent excerpts, re-edited

The following contents of this paragraph (marked on the side of each page) are extracted from the documents used for the patent production and validation process: they have been partly re-edited to fix obvious typos or complex parts, taking into account that sometimes some formal passages require special language and form for legal reasons.

5.2.2.4.1. Background

In order to safeguarding digital data, there are many protection methods substantially based upon access control (properly said "authentication'") and upon encoding in case even with combined encryption systems (or simply "encryption": during the storing step there is the real encryption, whereas in the data retrieving step there is the "decryption" inverse one). In the second case, information can be used only by decrypting them and in case of network systems this may occur on the user side ("client" side) so that plaintext data are never available on devices out of control of the user itself.

For the authentication and encryption/decryption steps different recognition systems (mnemonic with pin or password, biometric, other… from which hereinafter "accreditation systems" or "accreditation") are used, which usually provide retrieving or restoring methods for emergency situations; in all these cases an alternative identification method (an alternative "channel") has to be provided in order to be able to perform the possible restoring only by the authorised subject.

5.2.2.4.2. Accreditation and password reset

The accreditation systems may be always supported by a secondary system based upon an alphanumeric password (for example, this occurs in almost all access systems on mobile devices in which usually it is compulsory to set a password of this kind as alternative access to other methods, such as the biometric ones).

In case of a simple access control, it is possible to implement "password reset" systems therefore through the alternative channel the authorised subject is identified and a new password is created (for example this occurs in the web systems in which a unique link is sent with time limit on an e-mail address recorded by the user to create a new access password).

In case of an encryption system, no restoring system analogous to the previous one is possible since for the decryption step it is necessary to have available a well precise password and it is not possible to create a new different one thereof.

The present invention defines a method for fully retrieving a password which can be used in any context and in particular in those in which the full retrieving is required as in the information encryption systems.

It is further to be noted that any protection system can be kept in a super system keeping it and which can be managed through a password thereto the present invention is to be applied: this is the case of "password manager" software usable to group several access instruments to several information sets (even with protection methods different to each other). Generally, however, the protection systems can be brought back to digital sequences which may be considered as passwords (for example a common "digital certificate" is nothing but a digital content consisting of a sequence of bytes and displayable directly or indirectly -— through simple encoding — as alphanumeric sequence, and thus even for other protection systems): at the limit it is always possible to represent such sequences as alphanumeric strings by using the hexadecimal representation of the bytes constituting them.

In the description of the invention the term “super password’ is used to represent an alphanumeric password or any digital information for accessing protected data as described above which can even be enriched with additional meta-information as specified hereinafter.

5.2.2.4.3. Technical problem solved by the invention

The object of the present invention is to manage a super password of an owner user (hereinafter “owner”') for information protected through it so that:

1. the storage is secure: it is to be meant that should not be any immediate method to be able to have access to such information, but there should be only a method requiring knowledge, actions, abilities or skills — or combinations of the previous ones — of the owner and that are hardly accessible by third parties
2. the retrieval is possible fully: the owner has to be able to retrieve fully the super password in a simple way
3. the retrieval can be verified: it has to be possible to verify the retrieval procedure independently from the use of the super password (then by analysing the information of such a step and not trivially trying to use the super password itself).

About the above-illustrated problems, the present invention faces and solves important limitations of the state of art thereamong:

1. *storage*: if the information is stored in one single place this piece of information is kept in plaintext locally or protected with an encryption system. In the first case the direct access to the storage place bypasses any protection (for example this is the case of a password stored in plaintext on a server, for an administrator who can enter directly on the server itself), whereas in the second case it simply moves the problem on safeguarding the encryption system (then, the problem remains unsolved). In the present invention the information can be decomposed into several distributable parts at will having the following features:

1.1. it is necessary to reunite several parts — and not necessarily all of them, but a subset of cardinalities, at owner's choice, lower than or equal to that of the set of all parts — in order to be able to retrieve the super password;

1.2. the single parts are not correlatable to each other and they cannot trace back to the super password: there is no direct information allowing to associate several parts to each other, except in case of a general heading, apart from the cunning device described hereinafter. The only procedure with secure detectable effects is the recomposition requiring to have a set as the one indicated in point 1.1;

1.3. having a set of parts as the one indicated in point 1.1 a process is applied leading to the retrieval of the super password (and it is possible to verify that such a process is successful, see also the following points);

2. *retrieval*: by having available a correct set of parts, the recomposition procedure has to be simple and immediate

3. *verification*: by applying a suitable recomposition process to a set of parts if they are correlated but in not sufficient amount or they belong to sets of different origin, no information is obtained, nor even useful for a situation analysis (for example it is not possible to understand if it is the case of an insufficient amount of parts or of not correlated parts). This point is very important: for example, let's consider a system of encrypted and protected data which requires to enter a decryption code: upon entering, data are decrypted and analysed so as to verify if decryption was successful (for example by controlling if the format is correct or with some additional control code) and one counts the decryption attempts so that maybe after 3 failed consecutive attempts the data are removed. In a context of this kind, it is fundamental that once the retrieval of the super password is performed this is certainly the valid one, considering that a direct use for checking would not be possible considering the risk of losing the data themselves.

Ultimately, the verification system is based upon the quality and effectiveness of hashing function: in general, it is designated that there is "reasonable" certainty on the process validation with reference to such effectiveness, considering that — as known —hashing functions can potentially generate conflicts among data (that is they can return the same "hash" value with different starting data: however, this is a rare event by choosing suitable hashing functions as per subsequent examples).

5.2.2.4.4. Process of the invention

The invention defines two complementary processes called of “decomposition” and “recomposition” such that given a super password p and two integers at choice n and t such that $2 \leq t \leq n$ through decomposition one reaches to obtain a set of parts or components (hereinafter even only “components”) of cardinality n such that given any subset of such components of cardinality at least ¢ it is possible through recomposition to obtain the super password p.

The process is such that by applying the recomposition to a subset of components of cardinality lower than t or deriving from different decomposition procedures no useful information is obtained to deduce if the single components are or are not correlated to one another, whereas if it is successful according to the previous conditions the success can be verified.

Cases n=1 and t=1 are not taken into consideration since banal:

- case n=1 (involving even t=1) indicates a decomposition in one single component: in this case there is simply a (in case transformed) copy of the super password and then the case is not interesting, among other things falling as a particular case within the following.
- cases n>1 and t=1 indicate a decomposition in which it is sufficient to have available one single component to reconstruct the super password: actually, it is simply a matter of having n (in case transformed) distributable copies. Even this case then is not interesting considering that it would involve the possibility for whoever has available one of the components to trace back to the super password.

Without losing generality they are considered variables representing digital information as sequences of bytes (integers between O and 255 extremes included).

Hereinafter:

- the lowercase letters designate the “variables” (which represent data) and the uppercase ones the “functions”: the latter are algorithm procedures which given possible input parameters (designated hereinafter with name of the function in round brackets separated by comma) provide output data
- the square brackets and a list of numbers designate the indicated sequences of bytes designate: for example [0, 234, 0] is the sequence of bytes 0, 234 and 0
- the round brackets and a list of variables separated by comma designate the set of variables themselves: for example (a, b, c) is the set of variables a, b and c
- the angle brackets and a list of variables designate the “juxtaposition” of variables: considering that each variable can be represented as sequence of bytes then $< a , b >$ is the sequence which is obtained having at first the representation of a and then of b, then if a=[0, 255] and b=[255, 0] then$<a,b>=[0, 255, 255, 0]$
- literal or numerical indices near variables and functions are used to better represent the elements.

Let’s consider.

- a super password *p*
- an empty set of metadata *d*
- two integers *n* and *t* such that $2 \leq t \leq n$
- a pair of functions $y = J_e(p,d)$ and $(p, d) = J_d(y)$ which, given a super password and metadata, provide a combined representation thereof an vice versa (for example a serialisation);
- a pair of functions $\{ y_i \mid = 1, \ldots, n \} = S_e(x, n, t)$ and $x = S_d(\{ y_i \mid i=1, \ldots, t, \ldots \})$ which, given a string x and the two integers n and t perform the decomposition and recomposition according to the already described specifications
- a hashing function $y = H(x)$ which given a string x returns a “signature” y (also called “hash”: with size typically defined much smaller than the input string)
- two pairs of functions G_{e1}, G_{d1} and G_{e2}, G_{d2} such that G_{e1} and G_{e2} given an input return an output and the corresponding G_{d1} and G_{d2} perform the inverse procedure, therefore if $y=G_{e1}(x)$ then $x = G_{d1}(y)$ and if $y=G_{e2}(x)$ then $x = G_{d2}(y)$

  It means that for any value *k* we have:

  $k = G_{e1}(G_{d1}(k)) = G_{d1}(G_{e1}(k)) = G_{e2}(G_{d2}(k)) = G_{d2}(G_{e2}(k))$

- a pair of functions $y_\alpha=M_e(\alpha)$ and $y_\beta=M_d(\beta)$ and furthermore:

[I] $\alpha=(\ (H(p_1),G_{e2}(p_1))\ )\ \beta=(\ (H(p_2),G_{e2}(p_3))\ )$

$\gamma_k=(\ (\gamma_1,\gamma_2)\ )$ where $\gamma_1=H(k)$ and $\gamma_2=G_{e2}(k)$ (for any k)

Now we define:

[II] $M_e(\gamma_k) = M_e(\ (\gamma_1,\gamma_2)\ ) = G_{e1}(\ (H(k), G_{e2}(k))\ )$

(it is undefined or has a conventional value as 0 if the argument cannot be traced back to a structure like γ_k)

[III] $M_d(\ (H(k_{d1}), G_{e2}(k_{d2}))\ ) = G_{d1}(\ (H(k_{d1}), G_{e2}(k_{d2}))\ ) =$

(we compute $H(G_{d2}(G_{e2}(k_{d2}))=H(k_{d2})$ and then)

[III_a] K_{d2} if $H(k_{d2})=H(k_{d1})$

[III_b] 0 if $H(k_{d2})\neq H(k_{d1})$

So we have:

[IV] $y_\alpha = M_e(\alpha) = M_e(\ (H(p_1),G_{e2}(p_1))\ ) = G_{e1}(\ (H(p_1),G_{e2}(p_1))\ )$

[V] $y_\beta = M_d(\beta) = G_{d1}(\beta) = G_{d1}(\ (H(p_2),G_{e2}(p_3))\ ) =$

[V_a] p_3 if [III_a] is verified

[V_b] 0 if [III_b] is verified

if case [V_a] occurs we then assume that value β comes from a case like [II] as follows: we build a decomposition of α and we have a value β coming from a recomposition, applying the above process we get a 0 for a non valid recomposition or $p_3=p_1$ for valid one. The assumption derives from the goodness of the H-function, of which there is a very large class so that conflicts between applications over non-coincident values are considered essentially impossible ("near-uniqueness" of hashing). [23]

- a 12-byte long string, a string $r = [\ r_a, r_b, r_c, r_d\ ]$ and a pair of functions $A_e(a,r,x_1)=<a,r,x_1>$ and $(a,r,x_2)=A_d(<a,r,x_2>)$. The string *r* represents the punctual choice of the functions applied in

the decomposition step and to be applied in the recomposition step, whereas *a* is an identifying prefix. In substance <a,r> is a kind of "heading" which applies to data *x*.

Security phase

Given *a*, ***p***, ***d***, *n*, *t* and *r*, the decomposition comes from the following sequence:

1. $q_1 = J_e(\mathbf{p}, \mathbf{d})$
2. $q_2 = M_e(q1)$
3. $q_3 = \{ y_i \mid i=1,\ldots, n \} = S_e(q_2, n, t)$
4. $q_{4,i} = A_e(a, r, y_i)$

The latter elements (all the $q_{4,i}$) are the distributable components.

Recovery phase

Given a subset of the distributed $q_{4,i}$ at least of *t* cardinality:

1. $(a,r,y_i)=A_d(<a,r,q_{4,i}>)$
2. $q_3 = S_d(\{ y_i \mid i=1,\ldots,t \})$
3. $q_2 = M_d(q_3)$
4. $(\mathbf{p},\mathbf{d}) = q_1 = J_d(q_2)$

The outcome is either a valid (p,d) couple or an evidence that there has been a problem (due to a wrong input in the first step, usually for a too small subset or for invalid components)

Peculiarities of the functions used in the process:

- Je and Jd may be simple functions to manage the representation of data (for example to build a structured format from a raw one)
- S_e and S_d must adhere to a scheme with a power at least equivalent to a Shamir scheme (any multi-secret sharing scheme is valid [27])
- Hashing function H may be any valid one and there are many available: specific evaluations can be made to select the most appropriate function by approximating the probability of conflicts to zero [28], keeping in mind that a generally convenient choice is to use a widely used function that generates a sufficiently long signature, typically of at least 512 bits or even 1024 bits and above.

5.2.2.5. Mention of automatism under development

The entire process is slavishly applied in such a way that every single entity in the system, except when explicitly requested otherwise through appropriate configurations: in this way each unit protects by default the data that remain stored within it. In the current development, there is a mechanism-still under evolution-whereby the key can then be communicated externally through an explicit request filtered through an authorization system still under study.

5.2.3. Fluid Data

As described in Section 1.4.3. a new data classification model has been developed that is well suited to IoT systems and parallel execution environments.

The basic idea is to be able to parallelize computations as much as possible while avoiding having to manually handle critical sections and synchronizations and limiting unnecessary executions.

Let us consider a simple example in Python to be immediately clear about what we can talk about:

```
import sys

def fact(n: int) -> int:
      return 1 if n<1 else n*fact(n-1)
# #enddef fact

if len(sys.argv)<3: sys.exit(2)
arg1=int(sys.argv[1])
arg2=int(sys.argv[2])
```

```
sys.setrecursionlimit(2*arg1+1) # allow deep recursion
print("T" if fact(arg1)>arg2 else "F")
```

This snippet requires two arguments and it dumps out “T” if the factorial of the first is greater than the second value (e.g. calling with arguments “10000” and “100” it would dump “T” as 10000!>100.

This is a very simple code but it prominently displays a peculiarity as further explained below.

The output of the program depends on the comparison present in the last line where two values are compared, `fact(arg1)` and `arg2`: to make the comparison the interpreter processes the two operands and in particular to compute the first is uses the function (recursive in this case) defined at the beginning of the code with argument "n" which is called n+1 times (the first call is in the main block, the subsequent calls are effect of recursion).
Having as starting arguments `arg1=10000` and `arg2=100` the program makes a call of the type `fact(10000)`: since the argument "n" is not less than 1 the result of the function is `10000*fact(10000-1)=10000*fact(9999)` and this requires a new execution of the function with the new argument.

If we list the list of calls concisely, we get the following table:

step	argument	local value	main value
1	10000	10000*fact(9999)	10000*fact(9999)
2	9999	9999*fact(9998)	10000*9999*fact(9998) = 99990000*fact(9998)
3	9998	9998*fact(9997)	10000*9999*9998*fact(9997) = 999700020000*fact(9997)
...	...	...	...
10001	0	1	10000*9999*...*1 = 10000!

At last we get the final main value as 10000*9999*...*1, that is actually 10000!. Now the engine can compare it against the second operand and finalise the program.
Note that the main value has partial results made up by an integer multiplied by the result of a call to the recursive function (in all steps but the last one where it is a well defined integer).

Let us also consider the following simple arithmetic property:
$x,a,b \in \mathbb{Z}$
$x>0,\ a>x,\ b>0 \Rightarrow a*b > x$

From the previous properties we observe that the partial results computed at each step are multiplications of an increasing value by a positive value (in a complete implementation the “fact” function could be improved by checking the argument so that there is an exception if the argument is negative. In EFA this is possible using Python and type-hinting, see Appendix 8.1)

As a final consideration let’s take the comparison `fact(10000)>100` using the partial results instead of the full computation: at the first step we would have `10000*fact(9999)>100`, but applying the previous arithmetic property with x=100, a=10000 and b=fact(9999) we find that the result would be true without any further step to apply. In conclusion we could perform the comparison in 1 step instead of 10001 steps!

Any programmer certainly understands the importance of this observation on the fly, with the idea of applying it to complex and potentially computationally time-consuming functions and processes.
How is this characteristic realised and generalised? This is described in the following paragraphs.

5.2.3.1. Rigid vs Fluid

The datatype models of the most common programming languages (Python, JavaScript, C, GO, and others) can be succinctly thought of as a cataloguing of information that from models with basic characteristics (e.g., groups comprising data of type "integer" or "boolean"), define increasingly complex ones that extend them. Each piece of data belongs to a certain category and possibly can also

be classified under a specific hierarchy (e.g., in Python the standard Boolean values True and False are also integers, so an expression like `True*3+10*False` where `True` and `False` are respectively evaluated as the values `1` and `0`, is valid).

Let us now consider a reference to a value via an identifier (typically a variable) looking at a simple Python snippet:

```
import time
def func(n:int)->int:
    import time
    time.sleep(n)
    return n-1
# #enddef func
y = 1974
z = func(y)
if z>y: print(z)
```

The assignment to the variable "y" is an action that starts with the evaluation of the "right side" (in this case a simple 1974 value expression) and then continues with the actual assignment (to the variable in the "left side").
Similarly is the case in the next line, where, however, the evaluation of the "right side" requires the execution of a function (which, it may be noted, after a certain delay returns the value received as input decreased by one unit).
The last line, on the other hand, requires the verification of a condition: only if it is met is an output generated.

It can be seen that during the execution of the basic steps, each identifier (in the example there are "y", "z", "func(...)", intended as the function invocation, and "n", the latter only within the function) can only be found in two conditions:

- does not have an associated value (undefined)
- has a well-defined associated value (defined)

This condition may possibly change during the execution of a single step. For example, the line y=1974 is actually composed of the following steps:

- "right side" evaluation
- detection of identifier on the “left side”
- updating value of the identifier of the "left side"

Taking into consideration the identifier "y" and the two steps above, the following conditions occur:

`y ↔ undefined`

step: evaluation of “right side” (the integer 1974 is recognized)

`y ↔ undefined`

step: detection of identifier on the “left side”

`y ↔ undefined`

step: update of identifier found in the “left side”

`y ↔ 1974`

This is repeatable as a concept for each identifier present (obviously with increasing complexity depending on the type of operators and expressions).
In particular, the evaluation of the conditional expression z>y involves the evaluation of the "right side," the "left side," and then the actual comparison.

It is good to highlight right away that the given example could represent a structure of the logic of a node in an IoT system in which, however, the function "func" is externally defined (i.e., its invocation would correspond to sending the argument to another node, waiting for a processing response).
Neither the logic of the main block, nor the logic within the function, nor the type of data involved (in this case simple integers) is particularly relevant: what is important to note here is that every single identifier is at any instant either undefined or well-defined, and that the execution of a single micro-step can vary this condition.

We call this data classification model "*rigid*" precisely because of the fact that each reference is in exactly one of two conditions eventually passing from one to the other (typically from undefined to well-defined, but many environments also allow the reverse transition, as in Python where a variable

is initially undefined - a possible “print” generates an exception - and becomes well-defined on the first assignment, but the reverse transition is possible using the keyword “`del`”).

For the openbridge platform and for EFA, a more complex model has been developed that extends the previous one, termed "*fluid*" where essentially there is no longer necessary such a rigid transition between two states, but there can be a more gradual transition between intermediate states, as better described below.

To understand how it works, it is necessary to be clear about some different methods of expression evaluation, as explained below.

5.2.3.2. Greedy evaluation vs Lazy evaluation

Consider the following simple snippet:

```
def f1():
    print("f1")
    return True
# #enddef f1
def f2():
    print("f2")
    return False
# #enddef f2
print(f1() and f2()) # AND
print(f1() or  f2()) # OR
```

The first “print” triggers an output of the strings “f1” and “f2” and then of the “False” value.
The second “print” triggers an output of the string “f1” and then of the “True” value.

Boolean expressions are often evaluated with a “short-circuit” technique, so that operands are evaluated until the whole result cannot be inferred. As the function “f1” returns True, the first boolean

expression being an “and” requires the evaluation of the following operand to determine the final result, so also “f2” is called, while the second boolean expression being an “or” doesn’t need it.

A possible evaluation of all operands is referred to as "*greedy*" while an evaluation of the type shown (much more common as far as boolean logic is concerned) is referred to as "*lazy*".

It is necessary here to point out an incredibly important but often underestimated aspect of this behaviour: the final result is usually well calculated, but it takes absolutely no account of two aspects regarding operands that are ignored (not evaluated):

- there may be "side-effects" (this is the case in the previous example where the function "f2" should generate an output regardless of the result in which it is used)
- may "crash"

The basic idea is to extend this technique to any type of operator and function.

Picking up on the example in paragraph 5.2.3.1, when arriving at assignment `z = func(y)` the execution of the main block pauses waiting for the "right side" to be evaluated (thus for the "func" function to complete its logic) and then proceeds: exploiting the idea of "lazy evaluation," you can run the "func" function in competition with the main block (possibly in parallel) and proceed with the next step.
The next step requires a comparison in which the value of z is required, and in one of the two alternative branches an output is provided. It is only at this point that the execution engine places the main block on hold so that the comparison can be performed correctly.

Parallel initiation of "func" logic and putting the result on hold when it is used (in the next line where the comparison is made) is precisely the logic used in EFA.

A first aspect that highlights the advantage of this methodology can be found by thinking about the fact that between the action of calling "func" and its use there can potentially be many other activities, and thus this simple strategy allows for maximum exploitation of parallelism. This is even more important when considering that an IoT system the nodes are often independent: in a real system (as

in the concrete case of Work Case “Energy” [→∫4.2.1]) the definition and logic of "func" can be defined and executed in a different node from the one where it is invoked, in fact.

A second aspect concerns the possibility of defining complex data by exploiting the analogy with Boolean operators and truth tables.

Consider again the simple "func" function already seen: it has a signature where it is specified that an "int" is accepted as the argument and an "int" is returned as the result. If, however, we marked it by specifying that the input is again an "int" but for the output we specified that it is of the same type as the input and that the value will be less, we would have a "finer" marking of the same function.

By doing so, the execution engine would know that a call of the type "func(n)" will produce not only an "int" value but more precisely an "int" value less than “n”.
How can this feature be exploited?

If the function used as an operand appears in a comparison operation, it is possible to apply the "short circuit" technique under certain circumstances.
Again taking the example in 5.2.3.1, when comparing `z>y`, the identifier `y` has value `1974`, while `z` is the result of applying the function func to the same value. But if the function is marked in such a way as to assert that its result will be a value less than its argument then we already know - regardless of the point result computed within the function itself - that `z<y`. From what has been said, it is possible to determine that the comparison produces a "false" result, and this is the case even if the function is meanwhile still running (having executed it in parallel, as already explained) and therefore it is possible to proceed with the main block (with any further instructions present).

5.2.3.3. Fluid model

The new “fluid” model defined in EFA extends the “rigid” one (as described in previous paragraphs) allowing short-circuit operations against any type of data.

It has already been mentioned that each reference is simply matched with no value or a precise value.
In the new model more features are available:

- name: the name of this datatype

- status: it may be either “stable” or “unstable”
- value: this is the actual (stable or not) value (if any) of the reference
- parent: the datatype from which it is derived
- constraints: specific inherited characteristics with possible additional limitations, dynamically set
- operands: declare how to use own’s value when applied as an operator to a specific operand

Consider, for example, a simple “positive integer” basic data type with a precision of two bytes. This could be characterised by:

- `name: posint`
- `status: stable` (as a starting point any value is considered stable unless involved in some computation)
- `value: empty` (as a starting point any value is considered empty, kind of “undefined”)
- `parent: none` (in this example we consider this datatype as a “primitive” one)
- `constraints: min = 0, max = 65535` (we simply define the extremes of validity)
- `operands`: in a comparison exploits the notion that the value is contained in the range min..max

We can also then define some convenient conventions, such as the fact that the return values of a function - unless otherwise specified - can refer to the function's arguments for data type settings

We could define a new derived type to represent a value that can only decrease from the initial value:

- `name: shrinkingposint`
- `status: stable`
- `value: (parent’s value)`
- `parent: posint`
- `constraints: min = 0, max = (parent’s value - 1)`
- `operands: (same as parent)`

The “magic” is in the constraints: based on definition and conventions adopted, if we declare a function to take a “posint” input and returning a “shrinkingposint”, we are stating that if we pass a value “n” (range 0..65535) , the result will be in the range 0..n-1.
Performing a comparison we can use the information of the range to verify if the compared value is within the range or not, even if it is not yet the actual “final” computed value.

This characteristic was actually applied in Work Case “Energy” (see ∫4.2.1): each tank’s water level is described in the inner logic with a variable with a fluid datatype whose value is set as an “ever growing integer”. The function that returns this level depends on some external sensors, so it is not always accurate, but we know that its value can never decrease. By assigning a data type that declaredly can only grow, a "range" of values is forced that has the level of the monitored tank as a maximum and starts from a minimum at 0 that increases with each new detection, with a state set as "unstable" until verification (when it becomes "stable": this happens every 3 detections, of which the average is calculated).

Detection	Detected	Level (avg)	Min	Max
1	0mc	0mc	0mc	230mc
2	1mc	0mc	1mc	230mc
3	**2mc**	**1mc**	**1mc**	**230mc**
4	3mc	1mc	1mc	230mc
5	3mc	1mc	1mc	230mc
6	**3mc**	**3mc**	**3mc**	**230mc**
7	2mc	3mc	3mc	230mc
8	3mc	3mc	3mc	230mc
9	**3mc**	**~3mc**	**3mc**	**230mc**
10	5mc	3mc	3mc	230mc
11	5mc	3mc	3mc	230mc
12	**5mc**	**5mc**	**5mc**	**230mc**

sequence of level detection for the work case analysed

Thus this is exactly what was exemplified at the end of the previous paragraph and is what is actually done in an EFA-based application.

Every 3 detections, an average is taken, although then the calculated level is also modified with additional correctives. However, one notices in particular in line 7 a detection with an error: if the standard calculation were used, it would result in a reference value of 2 (surveyed value) or 2.66 (calculated mean) instead of 3. Since it has been mandated that the value can only "increase", once the reference range has reached the lower limit of 3 (this occurred in the step corresponding to line 6), it remains fixed.

Nodes that request this value and check whether it is greater than a certain comparison value "n" use that lower limit instead of the calculated value. In this case this choice is made for safety reasons: in other cases different choices could be made, of course. In ∫5.2.3.4 some metrics are reported.

A more complex example can be had by thinking of the following case:

```
1  import sys
2
3  n = 10000
4  m = n**10
5  sys.set_int_max_str_digits(100000)
6  sys.setrecursionlimit(20000)
7  def fact(x:int)->int:
8      if x<1:
9          r = 1
10     else:
11         r = x * fact(x-1)
12     # #endif
13     return r
14 # #enddef fact
15 print "T" if fact(n)>m else "F"
16
```

The example is similar to the previous one, but in this case the function used in the experiment given a positive integer argument "x" requires a number of iterations equal to just "x." Thus in the example fact(10000) requires 10000 iterations.

The comparison in the last line has $m=n^{10}$ as the comparison term, i.e. 10000^{10}.

In a standard computation, it is necessary to wait for the function to be fully computed (as already explained in an IoT system such an invocation may be a request to another node and it is often desirable to be able to maximise parallelism between operational units).

Taking advantage of the fluid model of EFA, it is possible to declare the function so that it accepts a positive integer as an argument, but specifying that it returns an "increasing" value, that is, the internally computed partial value is an increasing number.

In practice, we declare that the function returns a positive integer and that therefore its value will be in the range `1..maxint` (meaning by "maxint" the maximum integer value that can be represented) and that it is a growing value: when making the first call in the main body (i.e., where `fact(n)` appears), the execution engine can execute the function in parallel and know that the final value will be between `1` and `maxint`. Since, however, it has to check whether that value is greater than `m`, the result of the comparison cannot yet be determined at that stage, so the process that handles that comparison waits for more information (all of this transparently without the user having to handle synchronizations or semaphores).
The body of the function is executed and we arrive at the line r = x * fact(x-1) and at this point the execution engine makes a new call (recursive in this case) once again in parallel. In the current process the identifier r is assigned the result of the product between x, which is worth 10000, and the function invoked with parameter 9999 and then continues and then arrives at "return r." The value of r is returned to the calling process and at this time consists of a value being processed that will become the product between 10000 and the result of fact(9999). In summary at each iteration the range of what will be the final result sees its lower bound increase (see details at the beginning of paragraph 5.2.3). After only 11 iterations the main body process is awakened and executes the corresponding conditional branch.
An example of gain in time (and of course computing resources) is given in the following paragraph.

5.2.3.4. Current implementation

In the current implementation [→6] the “fluid data” model is implemented in some specific cases using python classes and taking advantage of operator and function overloading.

Fluid datatypes have been defined as Python classes in which "magic" methods such as "__eq__," "__lt__," "__gt__," and similar others are implemented, so as to allow intercept access to objects when used as operators in expressions even with infixed operators.

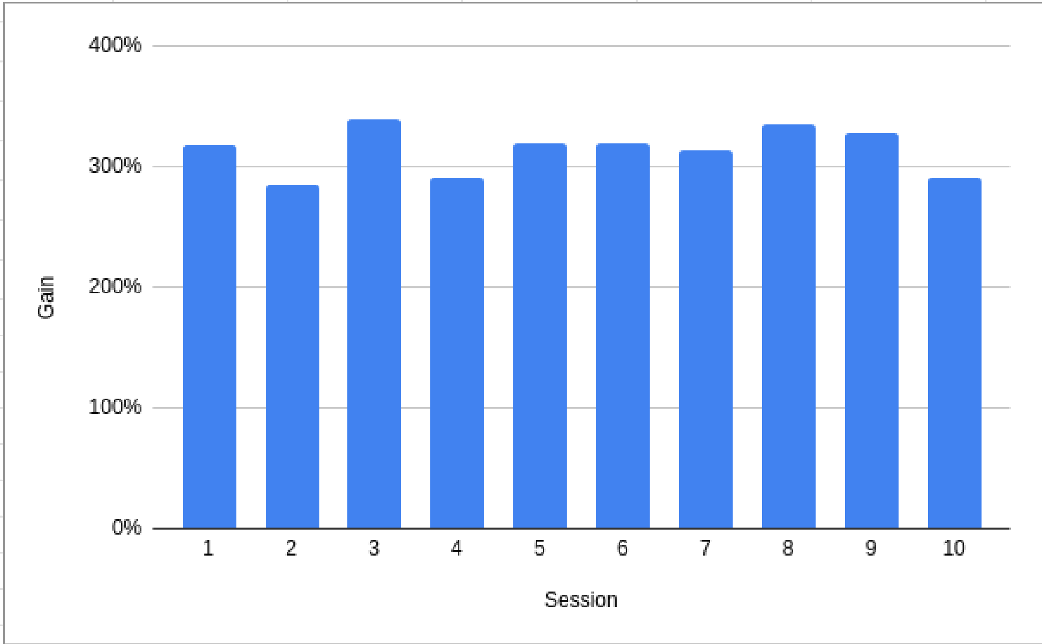
In Work Case “Energy” [→∬4.2.1] an update is under development and has been tested: one gateway unit interfaces with multiple subunits connected with water level sensors. This gateway plays the part of the main block of the previous examples, while subunits run the body of the function.
The gateway collects the results of all subunits and compares their value with certain control thresholds: in a standard implementation such control needs to wait for the full computation coming from the peripheral units, which perform checks with different sensors by making partial estimates and then refining the value until the final result is returned (in analogy with the previous example in which the factorial is computed by successive multiplications, getting closer and closer to the final value). Using a "smooth" model, analogous to the example, the gateway is able to determine some comparisons without having to wait for the entire processing.

In the following tables readings taken on the model under development are reported: 10 control runs were made, each with 100 executions comparing a standard *rigid* implementation with a *fluid* one, of which the readings were then averaged. In each run, time-consuming activities within each subunit were detected by highlighting after how long the useful value for computation was available. In the case of the standard implementation this value is available at the end of the entire processing, while with the *fluid* implementation the time is reduced because the computed partial value at some point is sufficient to proceed with the main logic (as in the examples in the previous paragraphs). The gain in execution time in the gateway unit is then shown.

Session	Rigid	Fluid	Gain
1	17.75”	5.59”	318%
2	17.04”	5.98”	285%
3	17.90”	5.28”	339%
4	15.96”	5.48”	291%
5	18.03”	5.65”	319%
6	18.16”	5.70”	319%
7	16.68”	5.32”	314%
8	18.20”	5.43”	335%
9	17.46”	5.32”	328%
10	16.54”	5.68”	291%

Comparison of rigid vs fluid implementation for Work Case “Energy”

(run time in seconds)



6 Applied research: actual developed work

Right at the beginning of the research journey, the general concept of setting up an operational solution as a generalist platform was sketched out: given that all the works were framed as IoT systems and in any case as a heterogeneous set of subsystems to be put in communication with each other and orchestrated appropriately, I came up with the name "openbridge" comprising the term "bridge", which excellently represents the idea of a connecting element between distant but connected shores, and the term "open" to indicate freedom of access and configuration.

Initially, some works had partial solutions already differentially set up, but the basic concepts were already there: these varied solutions were marked with the term "openbridge 1.0." Subsequently - as the present more punctual research journey began - the Node-RED application and a number of additional tools were adopted for which the FBP paradigm became basic: all the concepts discussed here were punctually applied through custom libraries, basic engine patches, special configurations and other adaptations, resulting in the solution currently operating on dozens of companies, thousands of users and millions of data, marked as "openbridge 2.0" reflecting the new EFP paradigm. At the same time, the design and development of a fully EFA-based environment called "openbridge 3.0" is being pursued, also evaluating the implementation of an ex-novo programming language that natively offers the EFP paradigm.

The operating environment actually implemented and currently operating 24/7 is described below.

6.1. Actual implemented components currently running

The structure described in ∫4.4 is currently built by exploiting a mix of technologies with numerous custom adaptations:

- the “Core” uses “docker” and “swarm” technologies and leverages a framework developed in Python and Javascript with several very common supporting libraries to simplify some settings (including Flask, SQLAlchemy, and jQuery). It works in a distributed environment with different layers for the “app” and the “db” sub-components

- the “dash” is implemented as a web application with dynamic HTML+JavaScript contents
- the “shell” is an interactive advanced cli where Python and JavaScript actions may be run at will: this is used mainly for maintenance and support
- the “api” is an intermediate layer that implements a full “REST” API system
- the “edge” is an independent application that can run on devices with very few resources (low CPU power and low RAM)

All components are currently set to run under Linux O.S., but as they are set using a containerized engine, a supporting tool has been developed allowing the entire system to be run under a host with Linux, Windows, MacOS and also others supporting docker.
The latter (edge) is optimised to run on small devices such as single-board computers like the Raspberry series. Many airpim’s customers have been provided with Iono devices, that are industrial PLC modules embedding a raspberry.

Since the main languages used are Python and JavaScript, specific solutions have been implemented for these languages (there are some minor additional tools made with other languages as well, such as C and GO).

For Python, support functions have been built to automate various operations, such as strong type checking (see also Appendix 8.1). Most implementations were made by creating dedicated classes and setting guidelines for using all and only the methods of those classes or objects instantiated by them.

For JavaScript some patches have been set in NodeJS and Node-RED such as the one that allows general error checking (see also Appendix 8.2). A fully customised support framework was then implemented that encapsulates in a single namespace, under a structured object that exposes numerous functions, a number of features that can be used directly in the code: again, there are guidelines that indicate to exploit such an object for interactions with other nodes and for data management.

6.2. System popularity, evaluation and personal contribute

The current platform is in operation for about 50 different customers, some with multiple active instances.

While there are no absolutely valid metrics for objectively evaluating a software project with source code analysis alone, some can give a rough idea of certain parameters. Among these a very popular one is the analysis of the number of lines of code, known as "SLOC": this is certainly a reasonable reference to represent the size of a project and having the notion of the type of work can also be indicative of the economic value. [33]

Running a simple SLOC analyser on the “openbridge” project around the end of the last quarter of 2023 gave the following results:

Totals grouped by language (dominant language first):

```
javascript:    656911 (48.52%)
python:        474340 (35.04%)
ansic:         178659 (13.20%)
sh:             36135 (2.67%)
Others:          7789 (0,58%)
```

```
Total Physical Source Lines of Code (SLOC)                = 1,353,834
Development Effort Estimate, Person-Years (Person-Months) = 388.31 (4,659.67)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 5.16 (61.93)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 75.24
Total Estimated Cost to Develop                           = $ 52,454,809
 (average salary = $56,286/year, overhead = 2.40)
```

In addition to the general estimate just provided to give an idea of the size of the project, other analysis tools can be applied on the code to estimate the specific contribution of the individual authors who worked physically on the source code. Through the open source tool "git-fame" applied

on the last branch of the repository with code developed almost entirely during 2023, the following data is highlighted:

Author	**contribute distribution**
A. Naimoli	90.3/33.8/81.1
(other 1)	4.6/29.5/12.8
(other 2)	2.8/24.5/ 2.0
(other 3)	1.9/ 8.5/ 2.5
(other 4)	0.2/ 1.0/ 0.9
(other 5)	0.1/ 0.9/ 0.2
(other 6)	0.0/ 1.7/ 0.4
(other 7)	0.0/ 0.0/ 0.0
(other 8)	0.0/ 0.0/ 0.0

(first 9 contributors of the "openbridge" platform: author identities are hidden but for the author of the present thesis)

Looking at the above table, "contribute distribution" estimates the contribution of a single author as a percentage of "locs" (line-of-codes), "commits" and "files" for surviving code. Above all, "locs" and "files" are very important as they somehow represent the overall actual contribution share of a single author, while "commits" is less important as a single commit may go from a single character to a bunch of updates for many files.

representative image of the openbridge dashboard in the industrial environment

(AI processed image from real elements)

7. Conclusion

The development of IoT systems and in general systems that integrate multiple units that communicate with each other is increasingly relevant in modern society. The development of software that must manage such systems must address specific issues being able to integrate realities with different technologies. The choice of an effective design and programming model, i.e., an appropriate architecture with a good paradigm, is crucial to achieve efficient, upgradeable and manageable results. The currently popular standard models are based on paradigms that have been set up for several decades now and that suffer from some important limitations, in particular regarding some difficulties in synchronising data, regarding privacy and the possibility of appropriately encrypting data with the chance, however, of being able to recover the protection key (verifying the goodness of the recovery process), regarding flexibility in coordinating asynchronous processes. In this research, principles are proposed for a new architecture evolving from the historically established ones, with some deeper investigations leading to solutions with respect to the above difficulties, implemented in a fully working actually released platform.
The research has been practically tested on numerous operational cases in industry and deepened with various investigations in medical problems: this has led to the publication of some papers, a speech at an international conference, and the realisation of a software patent, but also to the actual implementation of the aforementioned platform that is currently operational and used by various companies on a daily basis.

The entire course took place almost entirely during the COVID pandemic period, causing significant difficulties in the possibilities of interacting with other work cores and physically operating within the companies. To overcome this issue, remote collaborations were strengthened, including one with one of the pioneers of programming for IoT systems (as described in the introduction, ∬1.1.1) who welcomed the developments achieved in this work.

On the one hand, collaborations in the field of telemedicine have stimulated ideas and comparisons with important players, leading up to a presentation at an international congress (as stated in ∬1.1.4.2) with attendance by personalities from academia, industry and institutions at the highest level. These relationships enabled further dissemination with some publications for the field.

With a view to effective applicability in the industrial field, the practical project was then made to focus on defining general principles accompanied by some specific solutions, so as to build a reference model for a software architecture and programming paradigm that would be truly effective in reality. A truly usable model was obtained and implemented in the "openbridge" platform, currently fully operational, which stimulated the creation of some software patents, one of which has already been registered internationally (∫1.1.4.4): it is a solution composed of hundreds of thousands of lines of code (see ∫6.2) and which manages on a daily basis, without interruption, services of major companies operating in the context of "Industry 4.0" projects.

Looking forward, the aim is to delve even more deeply into each and every aspect enunciated here and to realise a development environment (with complete libraries for the most popular programming languages or even by implementing an ad-hoc programming language) that will make available in a simple and transparent way all the features described here.

8. Appendix

For effective application lifecycle management, I have designed and developed many strategies, development tools, configurations and procedures. The entire openbridge application is managed inside a dedicated environment: this enables convenient management of operating stations for collaborative activities by simply installing and activating that environment. Many tools are used within it to conveniently manage recurring and sensitive operations (such as build chains and release procedures) and some specially developed proprietary frameworks.
The following are partial examples of such capabilities.

8.1. Python function signature type check

Recent Python releases (after 3.5) allow type hinting in function declarations: the signature of a function may state the type for each argument so as to better describe what is expected.
The interpreter doesn't perform any check, anyway as declared in the official website: "*The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc.*" [24]
Figure 8.1 shows an early version of a Python function to be used as a decorator: it automates the checking of argument types by generating a dedicated exception in case of inconsistencies.
Updates to that function were later developed to handle more complex cases as well.

Here is an example on how to use the decorator:

```
1   @signcheck
2   def y(z: int = 0, b: str = "", c: int = 10):
3       print(f"z={z} (int), b={b} (str), c={c} (int)\n\n\n")
4   # #enddef y
```

A function y is defined, that expects three arguments: an integer, a string and another integer,
Calling the function with y(-1, "test") or y(), no exception is raised, while y(c="wrong") would raise the exception signmismatch: Signature type mismatch. <class 'NoneType'>/<class 'int'> (ko), <class 'NoneType'>/<class 'str'>

(ko), <class 'str'>/<class 'int'> (ko), and a call like y(1, 1, 1974) would rise signmismatch: Signature type mismatch. <class 'int'>/<class 'int'> (ok), <class 'int'>/<class 'str'> (ko), <class 'int'>/<class 'int'> (ok).

```python
def signcheck(decorated):
    class signmismatch(Exception):
        def __init__(self, reference):
            message = "Signature type mismatch. "
            infos   = []
            for key,val in reference.items():
                infos.append(f"{val.get('got', '?')}/{val.get('hint', '?')} ({'ok' if val.get('flag', None) else 'ko'})")
            # #endfor
            message += ", ".join(infos)
            super().__init__(message)
        # #enddef
    # #endclass
    def decorator(*args, **kwargs):
        reference   = {}
        #
        def check(arg, hint):
            got  = type(arg)
            flag = (got==hint)
            result = {"hint": hint, "got": got, "flag": flag}
            return result
        # #enddef
        #
        try:
            annotation  = decorated.__annotations__
            keys        = list(annotation.keys())
            for key in keys:
                reference[key] = {
                    "hint": annotation[key],
                    "got":  type(None),
                    "flag": None
                }
            # #endfor
            for pos,arg in enumerate(args):
                key  = keys[pos]
                reference[key] = check(arg, reference[key]["hint"])
            # #endfor
            for key,val in kwargs.items():
                arg = val
                reference[key] = check(arg, reference[key]["hint"])
            # #endfor
        except Exception as e:
            print(f"?ERROR {e}")
        #endtry
        #
        flags = [val.get("flag") for key,val in reference.items()]
        if flags.count(False)>0:
            raise signmismatch(reference)
        # #endif
        #
        decorated(*args, **kwargs)
    # #enddef decorator
    return decorator
# #enddef signcheck
```

Figure 8.1. Python function to be used as a decorator for automatic type checking

8.2. JavaScript catch-all handler

JavaScript allows managing exceptions with the *try-catch* construct but one pitfall is a possible “Syntax Error” as it comes from the parser, thus breaking the whole code where the error resides.

This has proven to be one of the most insidious problems while using some software and in particular Node-RED because, as already discussed, a single (even trivial) syntax error in some third function completely crashes the entire application!

As a temporary solution (while waiting for a stable version of the framework currently under development) custom patches were applied to Node-RED: specifically, any "external" code block (also meaning any individual logic written in the visual programming environment) is wrapped in a specially developed code that allows it to catch and handle any error, *including syntax errors*.

Consider, for example, a block of code that needs to be processed: instead of processing it directly it is handled as in the following example (where *...original_code...* is the code in question):

```
1   // not runnable mock-up of a wrapper to handle any error/exception, including SyntaxError, of a piece of code
2   try {
3       console.log("[EFP] Marker.Begin");
4       eval( /*…original_code…*/ );  // original code passed as a string
5       console.log("[EFP] Marker.End");
6   } catch (e) {
7       let caught = null;
8       if (e instanceof SyntaxError) {
9           caught = "SyntaxError";
10          /*… handling code for SyntaxError …*/
11      };
12      // other cases...
13      if (caught==null) {
14          /*… handling code for more... …*/
15      };
16  }
```

There are two tremendous advantages to applying this technique: it is possible to catch any "unhandled exception" (i.e., any exception for which a dedicated try-catch construct has not been used), and more importantly, it is also possible to catch an exception of type "SyntaxError" that otherwise cannot be caught in any way.

The mock-up above is just a glimpse of the actual code developed which is much more complex since it allows for automatic handling of special cases and also more information about the origin of the exception. In particular, errors are communicated to the container in which the code is executed so as to realise the “error handling system” described in paragraph 3.2.2.

8.3. Node-RED flow sample

Here follows the code of the sample reported in ∫1.2:

```json
[
    {
        "id": "a25debe8b2bd24d1",
        "type": "tab",
        "label": "Flow 2",
        "disabled": false,
        "info": "",
        "env": []
    },
    {
        "id": "b422f89eb46108b0",
        "type": "inject",
        "z": "a25debe8b2bd24d1",
        "name": "",
        "props": [
            {
                "p": "payload"
            },
            {
                "p": "topic",
                "vt": "str"
            }
        ],
        "repeat": "",
        "crontab": "",
        "once": false,
        "onceDelay": 0.1,
        "topic": "",
        "payload": "",
        "payloadType": "date",
        "x": 180,
        "y": 80,
        "wires": [
            [
                "2cf3a6ace2692975"
            ]
        ]
    },
    {
        "id": "2cf3a6ace2692975",
        "type": "function",
        "z": "a25debe8b2bd24d1",
        "name": "sample",
        "func": "let y = \"ciao\";\nlet z = WRONG(y);\n\neval(\"SYNTAXERROR\");\n\nmsg.payload = {\n    \"thru\": msg.payload,\n    \"crnt\": z\n}\nreturn msg;",
        "outputs": 1,
        "timeout": 0,
        "noerr": 1,
        "initialize": "",
        "finalize": "",
        "libs": [],
        "x": 380,
        "y": 80,
        "wires": [
            [
                "54bf94e751186e1e"
            ]
        ]
    },
    {
        "id": "54bf94e751186e1e",
        "type": "debug",
        "z": "a25debe8b2bd24d1",
        "name": "OUTPUT",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "payload",
        "targetType": "msg",
        "statusVal": "",
        "statusType": "auto",
        "x": 580,
        "y": 80,
        "wires": []
    },
    {
        "id": "154b1b39d51e663a",
        "type": "catch",
        "z": "a25debe8b2bd24d1",
        "name": "CATCH",
        "scope": null,
        "uncaught": false,
        "x": 190,
        "y": 260,
        "wires": [
            [
                "22bfcec8edead234"
            ]
        ]
    },
    {
        "id": "22bfcec8edead234",
        "type": "debug",
        "z": "a25debe8b2bd24d1",
        "name": "ERROR",
        "active": true,
        "tosidebar": true,
        "console": false,
        "tostatus": false,
        "complete": "true",
        "targetType": "full",
        "statusVal": "",
        "statusType": "auto",
        "x": 480,
        "y": 260,
        "wires": []
    }
]
```

There is a “function” node with two issues: a call to a missing function and an instruction that raises a syntax error. A “catch” node is also set but it cannot be used to create an alternative path as it would be too complex.

Moreover there also could be “uncaughtException errors” and the documentation states:
“*The typical cause will be that a node has kicked off an asynchronous task and that task has hit an error. A well-written node will have registered an error handler for that task, but if there isn’t one, the error will go uncaught.*”

It is not acceptable to rely on the fact that a node is "well written" or in any case it is often necessary to be able to handle such situations: especially in very complex systems it is always possible that bugs of various kinds will be present so that being able to intercept even these kinds of errors is absolutely essential (just think of critical situations such as an industrial production line-perhaps of hazardous materials, for example in a power plant-or a medical system where it is unthinkable to have software that can openly crash in case of unexpected errors)

8.4. Basic classes to model tables for starsyncing

The following is basic class modelling useful for handling data tables in Python.

Necessary imports include system and support libraries based on SQLAlchemy:

```
import datetime
from sqlalchemy.ext.hybrid import Comparator, hybrid_property
from sqlalchemy.schema import ForeignKeyConstraint
from sqlalchemy import or_, asc, desc, func, inspect
from sqlalchemy.orm.attributes import flag_modified
from sqlalchemy.orm.collections import InstrumentedList

from .storage import ExternalDatabaseManager

import os, io
from PIL import Image
from pathlib import Path

import time
import json

from operator import attrgetter
from decimal  import Decimal
```

These are supplemented by additional custom library imports:

```
from app import app, db
from app.system.models.BaseModel import BaseModel
from app.system.tools import vclass
```

Where:

- "app" is a Flask initialised application
- "db" the main SQLAlchemy object
- "BaseModel" is a specialisation of SQLAlchemy's "db.Model" to expose some convenient generalist methods
- "vclass" is a on-the-fly class generator as reported below

```
def vclass(classname, args=None, id=None):
   args = args or {}
   args['label'] = '' if 'label' not in args else args['label']
   args['description'] = '' if 'description' not in args else args['description']
   args['copy'] = lambda : vclass(classname, args, id) # (*)
   args['__repr__'] = lambda _ : '<{} {}>'.format(classname, id)
   result = type(classname, (object,), args)
   return result
   # (*) mimic "copy" method of dictionaries as this class is used
   #     to build custom attributes in db's tables and "flask db"
   #     wants somewhere dictionaries for this, for example in
   #     /sqlalchemy/sql/schema.py [156] when performing
   #     flask db migrate ...
```

("app", "db" and "BaseModel" implementation are not of special interest here so their reference above is enough to understand how the rest of the code here works).

The following classes in addition to defining tables for "starsyncing" [→∫5.2.1] are also enriched to handle a mapping between the defined model and an external one, as explained in the comment inside the code.

Specifically the "Syncing" class defined in the queue, models the management table of accounts used for synchronisation, as described in section 5.2.1.2.2

```
class MappedMeta(type):
   """
       manage unique instances per "mapped" reference of derived classes
       - "mapped": reference to mapped object (if any)
       - "force:   if set, ignore checking
   """
   #
   def __call__(cls, *args, **kwargs):
       obj = None # setup
       try:
           """
           retrieve "mapped" and "force"...
           ...and if the former is set and latter is not set...
           ...try to get "reversed" reference...
           ...and use it if found instead of a fresh new one
           """
           mapped  = kwargs.get("mapped", None)  # check if setting/looking a "mapped"
```

```
            force   = kwargs.get("force",  False) # retrieve "force" flag (True: no checking, otherwise: check)
            if force==True:
                obj = None
            else:
                reverse = getattr(cls, "reverse")     # try to retrieve ".reverse" method
                if mapped and callable(reverse):      # if there's a "mapped" reference AND there is a ".reverse" method...
                    obj = reverse(external=mapped)    # ...get reversed reference (if any)...
                # #endif
            # #endif
        except Exception as e:
            obj = False                          # ...default to False in case of error
        # #endtry
        #
        # mock-up for possible custom "__setattr__"
        # -----------------------------------------
        # _cls__setattr__ = cls.__setattr__
        # def _wrapper__setattr__(self, key, val):
        #     old     = getattr(self, key, None)
        #     changed = old!=val
        #     print(f"(_wrapper__setattr__) key={key}, old={old}, val={val}, changed={changed}")
        #     _cls__setattr__(self, key, val)
        #     #if changed:
        #         # _syncupdate = getattr(self, "_syncupdate", False)
        #         # if _syncupdate: _syncupdate(self, update=True, force=False, DOSAVE=False)
        #     # #endif
        # # #enddef
        # cls.__setattr__ = _wrapper__setattr__
        #
         self = obj if obj not in [False, None] else cls.__new__(cls, *args, **kwargs) # use existing instance if found or create a
breand new one
        try:
            if self: cls.__init__(self, *args, **kwargs)  # initialize
        except Exception as e:
            print(f"\n\n?ERROR: cls={cls}, args={args}, kwargs={kwargs}\nEXCEPTION: {e}\n\n")

        # #endtry
        #
        return self
    # #enddef __call__
    #
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
    # #enddef __init__
    #
# #endclass MappedMeta

# following two definitions are needed to avoid metaclass conflict in MappedModel
class MappedModelDerived(metaclass=MappedMeta):
    pass
class metaMappedModel(type(EntityModel), type(MappedModelDerived)):
    pass

class MappedModel(EntityModel, MappedModelDerived, metaclass=metaMappedModel):
    """
    To map an internal model with an external one
    INIT:
    At initalization a single external object may be passed under "mapped" parameter to create a linked reference.
    To note that ".mapped" property performs a query, so if some data needs to be preserved they must be stored
    in jsonconfig and/or custom fields. Example follows:
    # retrieve a single customer from external database (use primary key set):
    cli_cod="40800629"; cod_soc="001" # primary references, then retrieve object below
    mapped  = ExDB_Customer.query.filter(ExDB_Customer.cli_cod==cli_cod, ExDB_Customer.cod_soc==cod_soc).first()
    # create ob's object mapped to previous one:
    cliente = Cliente(mapped=mapped) # this has a ".mapped" property to retrieve original object via primary keys
    syncing (hash, date, status) contengono hash, timing e state utili al syncing che sono automaticamente valorizzati:
    - alla creazione di un nuovo oggetto
    - al salvataggio quando un qualunque valore rende l'oggetto "modified"
    - in caso di chiamata diretta (anche attraverso il metodo di classe che aggiorna tutti i record)
    ATTRIBUTES/METHODS:
    object:
```

```
    - mapping      : JSON list to store mapping infos such as external (primary) keys to identify record
                       - mapped.class : external class to look for, try _classmapped if none given
                       - mapped.keys  : external keys to use
    - mapped       : to retrieve mapped (external) record using mapping.mapped.keys
class:
    - reverse(...) : try to retrieve internal reference of an external one
metaclass is MappedMeta, that returns a unique instance per mapped reference

NOTE: attributes named like "__cache_...__" are reserved and generated on-the-fly!!!
      for example "__cache_mapped__" is generated inside "mapped" method to temporary
      store the result and use it in subsequent calls

"""
__abstract__ = True
_classname   = 'MappedModel'
_classmapped = None          # to be overridden as a single class or a list of classes
#
# level (used for "weight" of data)
raw_level_min   = 0
raw_level_max   = 9
raw_level_range = [n for n in range(raw_level_min, raw_level_max+1)]
@classmethod
def raw_level_value(cls, level=None, default=None):
    def castInt(txt):
        result = None
        try:
            result = int(str(txt).strip())
        except:
            result = False
        # #endtry
        return result
    # #enddef
    if isinstance(level, str) and level=="min": level = cls.raw_level_min
    if isinstance(level, str) and level=="max": level = cls.raw_level_max
    level = castInt(level)
    level = level if isinstance(level, int) and level in cls.raw_level_range else default
    return level
# #endif
#
 jsonmapping = db.Column(db.JSON(),      default={"mapped": [{"class": None, "keys": {}}]}, info=vclass(_classname, {'label':
'mapping', 'description': '', 'size': 6}))
 jsonsyncing  = db.Column(db.JSON(),      default={"hash": None, "date": None, "status": None, "rnd": None, "linked": None},
info=vclass(_classname, {'label': 'mapping', 'description': '', 'size': 6})) # md5 hash, timestamp as "%Y-%m-%d,%H:%M:%S.%f"
#
@property
def synchash(self):
    value = self.jsonsyncing.get("hash", None)
    return value
# #enddef
@property
def syncdate(self):
    value = self.jsonsyncing.get("date", None)
    return value
# #enddef
@property
def syncstatus(self):
    value = self.jsonsyncing.get("status", None)
    return value
# #enddef
@property
def synclinked(self):
    # None | "unmapped" | "linked" | "deleted" | "restored"
    value = self.jsonsyncing.get("linked", None)
    return value
# #enddef
#
def save(self, NOLOOP=False, **kwargs):
    modified = db.session.is_modified(self) or self.id is None
    if modified and not NOLOOP: self._syncupdate(update=True, DOSAVE=False)
    saved = super(MappedModel, self).save(kwargs) if modified else None
```

```
        return saved
    # #enddef
    #
    def _syncdata(self, update=False, force=False, DOSAVE=True):
        import hashlib, datetime, random
        result  = {}
        update  = update or not self.syncdate # always update if never done before
        changed = False
        #
        jsonsyncing = self.jsonsyncing
        #
        if update or force:
            sdt = datetime.datetime.now().strftime("%Y-%m-%d,%H:%M:%S.%f") # current timing
        else:
            sdt = self.syncdate
        # #endif
        self.jsonsyncing["date"] = sdt
        #
        if force:
            rnd = str(int(random.random()*100)).rjust(2, "0")[-2:]         # 00% <= rnd <= 99%
            md5 = hashlib.md5(f"{rnd}|{sdt}".encode()).hexdigest()         # unique hash
            self.jsonsyncing["rnd"]    = rnd
            self.jsonsyncing["hash"]   = md5
            self.jsonsyncing["status"] = "init"
            changed = True
        # #endif
        #
        # compare current linking (linked) with previous (synclinked) and possibly update it
            linked, synclinked = self.linked, self.synclinked # current, previous: None | "unmapped" | "linked" | "deleted" |
"restored"
        self.jsonsyncing["linked"] = {
            (True, None)          : "linked",
            (True, "unmapped")    : "linked",
            (True, "linked")      : "linked",
            (True, "deleted")     : "restored",
            (True, "restored")    : "restored",
            (False, None)         : "unmapped",
            (False, "unmapped")   : "unmapped",
            (False, "linked")     : "deleted",
            (False, "deleted")    : "deleted",
            (False, "restored")   : "deleted"
        }.get(tuple([linked, synclinked]), "linked" if linked else "unmapped") # default shouldn't happen as all cases are covered
        changed  = changed or (synclinked != self.jsonsyncing["linked"])
        #
        if DOSAVE: flag_modified(self, "jsonsyncing")
        #
        changed  = changed or update
        saved    = self.save(NOLOOP=True) if DOSAVE else None
        modified = saved is not None
        #
        result["saved"]    = {"code": saved.code, "message": saved.message, "status": saved.status} if saved else None
        result["modified"] = modified
        result["changed"]  = changed
        result["synchash"] = self.synchash
        result["syncdate"] = self.syncdate
        #
        result = dictObj(result)
        return result
    # #enddef
    #
    @property
    def syncdata(self):
        return self._syncdata()
    # #enddef
    #
    def _syncupdate(self, update=False, force=False, DOSAVE=True):
          return self._syncdata(update=update, force=not self.synchash or force, DOSAVE=DOSAVE) # always update and force hash if
currently None (e.g. first time)
    # #enddef
```

```
    def update(self, *args, mapped=None, linked=None, **kwargs):
        if linked is not None:
            jsonsyncing = self.jsonsyncing
            if not isinstance(jsonsyncing, dict): jsonsyncing={}
            jsonsyncing["linked"] = linked
            self.jsonsyncing = jsonsyncing
            flag_modified(self, "jsonsyncing")
            self._syncupdate(update=True, DOSAVE=False)
        # #endif
    # #enddef update

    def __init__(self, *args, mapped=None, force=None, **kwargs):
        mapped_class, mapped_keys = None, {}
        self.jsonsyncing = {"hash": None, "date": None, "status": None} ; flag_modified(self, "jsonsyncing")
        self._syncupdate(DOSAVE=False)
        #
        mappedlist = mapped if isinstance(mapped, list) else [mapped] # manage as a list
        self.jsonmapping = {"mapped": []} ; flag_modified(self, "jsonmapping")
        # build mapping-mapped --------------------------------
        jsonmapping_mapped = []
        for mappedobj in mappedlist:
            """
            mappedobj may be an object of a class pointing to the external db or a dictionary
            with at least the name of the class in "tclass" and a sub-dictionary with key:val
            of primary keys-values
            """
            if isinstance(mappedobj, dict):
                mapped_class = mappedobj.get("tclass", None)
                mapped_keys  = mappedobj.get("pval", {})
            else:
                   if mappedobj and hasattr(mappedobj, "primary_key_column_list") and isinstance(mappedobj.primary_key_column_list,
list):
                    try:
                        keylist      = mappedobj.primary_key_column_list
                        vallist      = [getattr(mappedobj, key, None) for key in keylist]
                        mapped_class = str(mappedobj.__class__.__name__)
                        mapped_keys  = dict(zip(keylist, vallist))
                    except:
                        mapped_class = None
                        mapped_keys  = {}
                    # #endtry
                # #endif
                mapped_keys = dict(sorted(mapped_keys.items())) # keep keys sorted
            # #endif
            jsonmapping_mapped.append({
                "class": mapped_class,
                "keys":  mapped_keys
            })
        # #endfor
        #
        # sort whole mapped reference by class name
        jsonmapping_mapped = sorted(jsonmapping_mapped, key=lambda val: val.get("class"), reverse=False)
        # -----------------------------------------------------
        self.jsonmapping["mapped"] = jsonmapping_mapped
        super(MappedModel, self).__init__(*args, **kwargs)
    # #enddef
    #
    @classmethod
    def syncupdate(cls, update=False, force=False, DOSAVE=True):
        # update/set sync fields (e.g. if not set after an upgrade) of ALL records
        #
        # - update : force "update" (of syncdate) keeping hash if exists
        # - force  : force full update (syncdate+synchash) changing hashes too
        #   (with both flags set to False - default - only previously unsynced records are updated)
        #
        result = {"tot": 0, "old": 0, "new": 0}
        #
        records = cls.query.all()
        result["tot"] = len(records)
        for record in records:
```

```
            if not record.jsonsyncing: record.jsonsyncing = {"hash": None, "date": None, "status": None} ; flag_modified(record,
"jsonsyncing")
            ret = record._syncupdate(update=update, force=force, DOSAVE=DOSAVE)
            result["old"] += 0 if ret.changed else 1
            result["new"] += 1 if ret.changed else 0
        # #endfor
        return result
    # #enddef
    #
    @property
    def mapping(self):
        result = dictObj(self.jsonmapping)
        return result
    # #enddef
    #
    @property
    def mapped(self):
        # return mapped object based on jsonsyncing informations, possibly cached
        cachedattr = "__cache_mapped__"
        iscached   = hasattr(self, cachedattr)
        #
        if iscached:
            result = getattr(self, cachedattr, None)
        else:
            gg = globals()
            ll = locals()
            #
            result           = None
            ex_records       = []
            classmapped      = self._classmapped # should be single if given this way
            jsonmapping      = self.jsonmapping
            jsonmapping      = jsonmapping if jsonmapping else {}
            jsonmappedlist   = jsonmapping.get("mapped", [])
            for jsonmappedobj in jsonmappedlist:
                jsonmappedobj = jsonmappedobj if isinstance(jsonmappedobj, dict) else {}
                  ex_class      = jsonmappedobj.get("class", classmapped)                                          # use default if
none found in json
                 ex_class      = gg.get(ex_class, ll.get(ex_class, None)) if isinstance(ex_class, str) else False # retrieve actual
class from string
                ex_keys       = list(jsonmappedobj.get("keys", {}).items())
                if len(ex_keys)>0:
                    filters = []
                    for k,v in ex_keys:
                        r = k.lower() # TODO: verify this part... needs convertion to lowercase to get attribute
                        filters.append(getattr(ex_class, r, None)==v)
                    # #endfor
                    ex_record   = ex_class.query.filter(*filters).first()
                    ex_records.append(ex_record)
                # #endif
            # #endfor
            #
            # return either any single value found or a not empty list, None otherwise
            if len(ex_records)==0:
                result = None
            elif len(ex_records)==1:
                result = ex_records[0]
            else:
                result = ex_records
            # #endif
            #
            setattr(self, cachedattr, result)
            #
        # #endif
        ##app.logger.info(f"[MappedModel.mapped] iscached={iscached}")
        return result
    # #enddef
    #
    @property
    def linked(self):
        # return True/False if there's any actual mapped reference
```

```
        result = None
        mapped = self.mapped
        result = bool(mapped) # True : single record or not-empty array, False : None or empty aray
        return result
    # #enddef
    #
    @classmethod
    def reverse(cls, external=None, internals=None):
        # may pass an "internals" list to search for, otherwise all records are retrieved
        result = None
        if external:
            if isinstance(internals, list):
                all = internals
            else:
                try:
                    all      = cls.query.all() # try to retrive all records of class "cls"...
                except:
                    all      = []              # ...falling back to an empty list in case of problems
                # #endtry
            # #endif
            #
            found = None
            for element in all:
                mapped = getattr(element, "mapped", None) # possibly get "mapped" reference of current record in loop
                if mapped and isinstance(element, cls):
                    if isinstance(external, list) and isinstance(mapped, list): # if looking for a list and a list is found...
                        if len(external)==len(mapped):                          # ...and both length match...
                            found = True
                            for single in external:
                                found = found and (single in mapped)            # ...verify if all searching objects are mapped...
                            # #endfor
                        else:
                            found = False                                       # ...otherwise fails
                        # #endif
                        found = element if found else False            # get full list if found (instead of boolean)
                    else:
                        found = element if external==mapped else False # simple comparison if a single element is given
                    # #endif
                # #endif
                if found: break
            # #endfor
            result = found
        # #endif
        return result
    # #enddef
    #
    def _keys(self, full=False, level=None):
        # retrieve keys (fields) references for representations:
        #
        # take "all" fields, BUT (excluding):
        # - the ones starting with a "_"
        # - some special ones ("jsonmapping", "created", "updated")
        # - some "heavy" ones based on level
        #   uses:
        #    . "vars" for basic fields
        #    . "_extra_vars" attribute for special fields: they start with a "." (automatically added if not present)
        #                    and are taken even as nested references (e.g. "mapped.id" would take self.mapped.id)
        #
        cls     = self.__class__
        my_keys = [my_key for my_key in vars(self) if my_key[0]!="_" and my_key not in ["jsonmapping", "created", "updated"]]
        # possible extra vars: ----------------------------------------------------------
        _extra_vars = getattr(self, "_extra_vars", [])
         _extra_vars = [("." if _extra_var[0]!="." else "")+_extra_var for _extra_var in _extra_vars] # automatically prepend a dot
if not present
        my_keys += _extra_vars
        # ------------------------------------------------------------------------------
        #
        level = cls.raw_level_value(level)
        if level is not None:
            exclude = [] # starting with no exclusion
```

```
        # exclusion based on level<->datatype:
        if level < cls.raw_level_max:
            exclude += [db.LargeBinary]
        # #endif
        cols    = [col for col in cls.__table__.columns if col.type.__class__ in exclude] # check if needs to be excluded
        for col in cols:
           if col.key in my_keys: my_keys.remove(col.key)
        # #endfor
    # #endif
    return my_keys
# #enddef
#
def _raw(self, full=False, relations=False, level=None):
    cls = self.__class__
    # return raw data of object:
    # - full==True  : including "mapped" reference representation
    # - full==False : with a dictionary reference for "mapped" (if any)
    # - relations   : True/False = include/exclude relations
    # - level       : min..max (light..heavy) = exclude/include "light&heavy" fields (e.g. int/binarydata-blobs)
    result = {}
    #
    my_id, my_classname, my_dict, my_mapped = None, None, None, None
    #
    my_keys = self._keys(full=full, level=level)
    mapped  = getattr(self, "mapped", None)
    if full:
        my_mapped = str(mapped) # take string representation of raw object
    else:
        my_mapped = {}     # class -> keys->value reference
        mapped = [mapped] if not isinstance(mapped, list) else mapped # reference as list
        for submapped in mapped:
            if submapped:
                submapped_classname = submapped.__class__.__name__
                submapped_keys      = submapped.primary_key_column_list
                my_mapped[submapped_classname] = {}
                for submapped_key in submapped_keys:
                    submapped_val = getattr(submapped, submapped_key, None)
                    submapped_val = submapped_val if isinstance(submapped_val, str) or submapped_val==None else False
                    my_mapped[submapped_classname][submapped_key] = submapped_val
                # #endfor
            else:
                my_mapped[None] = None
            # #endif
        # #endfor
    # #endif
    #
    my_dict = {}
    for my_key in my_keys:
        try:
            # possible special keys (possibly given with _extra_vars) including a "dot"
            if "." in my_key:
                if my_key[0]==".": my_key=my_key[1:]          # remove possible leading ".", just used as a identifier
                attr_key = my_key                             # keep reference
                my_key = f"extra__{my_key.replace('.', '_')}" # "mapped.id" would become "extra__mapped_id"
                   my_val = attrgetter(attr_key)(self) if "." in attr_key else getattr(self, attr_key) # try to retrieve value:
if an exception is raised, my_key has already been updated
            else:
                my_val = getattr(self, my_key)
            # #endif
            # -------------
            # special types:
            if type(my_val) in [bytes]:
                my_val = list(bytearray(my_val))
            # #endif
            if type(my_val) in [Decimal]:
                my_val = float(my_val)
            # #endif
            if type(my_val) in [datetime.datetime]:
                my_val = my_val.strftime("%Y-%m-%d,%H:%M:%S.%f")
            # #endif
```

```
            # --------------
            # final check, verify if jsonable:
            try:
                jsonized = json.dumps(my_val)
            except Exception as e:
                app.logger.info(f"[MappedModel] _raw: {e}. {my_key}:{my_val}")
                my_val = f"!{my_val}"
            # #endtry
            # --------------
        except:
            my_val = None
        # #endtry
        my_dict[my_key] = my_val # this may be - correctly - overridden by a "relationship" afterwards
    # #endfor
    #
    my_id        = self.id
    my_classname = self._classname
    #
    # if only "dict" (not full) is required, embed "mapped" data (as dict) and possibly relationship inside
    if not full:
        my_dict["mapped"]        = my_mapped
        if relations:
            relationships = inspect(cls).relationships
            for relationship in relationships:
                related_class  = relationship.mapper.class_
                related_name   = related_class.__tablename__
                related_key    = relationship.key
                related_object = getattr(self, related_key, None)
                related_raw    = None
                try:
                    if not isinstance(related_object, InstrumentedList):
                        # recurse for non-list attributes
                          related_raw    = related_object._raw(full=False, relations=False, level=level) if related_object else
None
                    else:
                        # list "id"s for list-like attributes
                        related_raw    = [item.id for item in related_object]
                    # #endif
                except Exception as e:
                    related_raw    = False
                # #endtry
                my_dict[related_key] = related_raw # may override a previous set field with better informations
            # #endfor
        # #endif
    # #endif
    #
    result = {
        "id":        my_id,
        "classname": my_classname,
        "dict":      my_dict,
        "mapped":    my_mapped
    }
    result = result if full else result.get("dict")
    #
    return result
# #enddef
#
@property
def rawlight(self):
    return self.rawmid(level="min")
# #enddef
def rawmid(self, level=None):
    my_raw = self._raw(full=False, relations=True, level=level)
    return my_raw
# #enddef
@property
def rawheavy(self):
    return self.rawmid(level="max")
# #enddef
#
```

```
@property
def flatlight(self):
    return self.flatmid(level="min")
# #enddef
def flatmid(self, level=None):
    cls = self.__class__
    # level may be:
    # - string "min" for level_min
    # - string "max" for level_max
    # - integer in min..max (forced if outside range) for chosen level
    # - something else: default
    level_min     = cls.raw_level_min
    level_max     = cls.raw_level_max
    level_default = level_max
    if isinstance(level, str):
        if level == "min": level = level_min
        if level == "max": level = level_max
        if level.isnumeric(): level = int(float(level))
    # #endif
    if isinstance(level, int):
        if level<level_min: level = level_min
        if level>level_max: level = level_max
    # #endif
    #
    if not isinstance(level, int): level = level_default
    #
    my_raw = self.rawmid(level)
    my_raw["remote"] = my_raw.pop("id", None)
    my_flat = dictFlat(my_raw)
    return my_flat
# #enddef
@property
def flatheavy(self):
    return self.flatmid(level="max")
# #enddef
#
def _dict(self, format=True):
    import json
    def jsonize(obj):
        import datetime
        jdata = None
        try:
            if isinstance(obj, datetime.datetime):
                jdata = obj.strftime("%Y-%m-%d,%H:%M:%S.%f")
            else:
                jdata = str(obj)
            # #endif
        except:
            jdata = None
        # #endtry
        return jdata
    # #enddef
    #
    my_raw  = self.raw
    my_dict = json.dumps(my_raw, indent=4 if format else None, sort_keys=True, default=jsonize)
    return my_dict
# #enddef
#
@property
def dict(self):
    my_dict = self._dict(format=False)
    return my_dict
# #enddef
#
def __repr__(self):
    my_raw = self._raw(full=True, relations=False, level="min")
    #
    my_repr = f"<{my_raw.get('classname')} {my_raw.get('id')}: {my_raw.get('dict')} {my_raw.get('mapped')}>"
    return my_repr
# #enddef
```

```
    #
# #endclass MappedModel
class UnmappedModel(MappedModel):
    """
    Not to map an internal model with an external one
    (simpy acts as a "wrapper" to align a mapped structure for both mapped and unmapped data)
    """
    __abstract__ = True
    _classname    = 'UnmappedModel'
    _classmapped = False
    _syncable     = False
    #
# #endclass UnmappedModel

class Syncing(MappedModel):
    __bind_key__ = 'custom'
    _classname    = 'Syncing'
    _classmapped = None
    _default_fields = ["id", "device"]
    #
    synced_level_min    = MappedModel.raw_level_min
    synced_level_max    = MappedModel.raw_level_max
    synced_level_range = [n for n in range(synced_level_min, synced_level_max+1)]
    #
    # device  : uniqe id (hash), automatically set, of a client-device:
    #           - at first (or anonymous) synchronization it is generated and sent to the client
    #             that should give a "passkey" to pair (kind of "password")
    #           - subsequent calls (not anonymous, with a given "id") must be found on the server
    #             and passkey must match
    # passkey : a "password" to match against device (id), set by client
     # synced   : latest synchronization as a dictionary with table's name as a key and a subdictionary with level's name as a key
and latest as a value
    #            e.g. synced = { "tablename" : { "0": ..., "1": ... } }
    #
     device    = db.Column(db.String(255), unique=True,   default="", info=vclass(_classname, {'label': 'Device', 'description': ''
}))
     passkey   = db.Column(db.String(255), unique=False, default="", info=vclass(_classname, {'label': 'Device', 'description': ''
}))
    synced   = db.Column(db.JSON(), default={}, info=vclass(_classname, {'label': 'Synced', 'description': '', 'size': 6}))
    #
    def __init__(self, passkey=None, *args, **kwargs):
        super(Syncing, self).__init__(*args, **kwargs)
        self.device  = self.jsonsyncing.get("hash")
        self.passkey = passkey
        self.synced  = {}
    # #enddef
    #
    def mark(self, tablesORlatest=None, levels=None, latest=None):
        """
        tablesORlatest may be:
        - a dictionary key-val for table-levels/latest
        - a list of table names, so extra levels/latest argument are required (see below)
        #
        tablesORlatest : may be a single name, a list or a dictionary (see above)
                         - as a list: names of tables
                         - as a dict: table's name as a key + tuple(level, latest) or {"level": {"latest": ... }, ...}
                           (if level is missing - both in tuple or dict - default is set)
         levels          : may be a single value or a list (should be same length of tables, otherwise is padded with default, i.e.
level_maximum)
                         - None is an alias for True (see below)
                         - True for default_level
         latest          : may be a single value or a list (should be same length of tables, otherwise is padded with default, i.e.
current)
                         - None to skip
                         - True for default_latest
        #
        - tables are identified by their name
        - levels are an integer in closed range (weight of marking)
        - latest are timestamp reference of marking
        """
```

```
        import datetime
        cls = self.__class__
        #
        result         = None
        current        = datetime.datetime.now().strftime("%Y-%m-%d,%H:%M:%S.%f") # current timing
        #
        default_latest = current
        default_level  = "max"
        #
        if not isinstance(self.synced, dict): self.synced={}
        #
        levels = []; latests = []
        if isinstance(tablesORlatest, dict):
            tables          = list(tablesORlatest.keys())
            levelsANDlatest = list(tablesORlatest.values())
            for levelANDlatest in levelsANDlatest:
                if isinstance(levelANDlatest, dict) and len(list(levelANDlatest.items()))==1:
                    level  = list(levelANDlatest.keys())[0]   # key of the only item of dictionary is level
                    latest = list(levelANDlatest.values())[0] # val of the only item of dictionary is latest
                elif isinstance(levelANDlatest, tuple) and len(levelANDlatest) in [2, 1]:
                      level   = levelANDlatest[0] if len(levelANDlatest)==2 else True              # first  element if both present
or default otherwise
                      latest  = levelANDlatest[1] if len(levelANDlatest)==2 else levelANDlatest[0] # second element if both present
or the only one otherwise
                else: # skipping values in case of error
                    level   = None
                    latest  = None
                # #endif
                levels.append(level)
                latests.append(latest)
            # #endfor
        else:
            tables = tablesORlatest
            latest = latest
        # #endif
        latest = latests
        #
        if not isinstance(tables, list): tables=[tables] # force data as a list
        if not isinstance(latest, list): latest=[latest] # force data as a list
        num    = len(tables)                                  # number of elements
        levels = levels + [None]*num ; levels = levels[0:num] # auto-padding level
        latest = latest + [None]*num ; latest = latest[0:num] # auto-padding latest
        #
        result = True
        #
        for idx in range(0, num):
            table = tables[idx]
            if isinstance(table, str):
                if table not in self.synced: self.synced[table] = {} # possibly set table's subdictionary
                timing = latest[idx]
                level  = levels[idx]
                if timing is not None: # skip "None" items
                    timing = default_latest if timing is True         else timing
                     level  = cls.raw_level_value(level, default_level if level in [True, None] else None) # compute level possibly
setting default
                    self.synced[table][level] = timing # create or update subdictionary
                    flag_modified(self, "synced")
                        subresult = bool(self.synced.get(table, {}).get(level) == timing) and bool(timing) # regardless of saving's
outcome check if data match
                    result    = result and subresult
                # #endif
            # #endif
        # #endfor
        self.save()
        return result
    # #enddef
    #
# #endclass Syncing
```

Bibliography

bibliography and webography

[1] Intelligenza Artificiale per la Pediatria 5P: LLM, GPT e una nuova architettura / Artificial Intelligence for Pediatrics 5P: LLM, GPT and a new architecture (2024, forthcoming in quaderni ACP, ISSN: 2039-1374)

[2] La classifica CENSIS delle università italiane, edizione 2023/2024. CENSIS (Centro Studi Investimenti Sociali), July 2023

[3] "airpim, riconosciuta la prima pmi innovativa in Trentino", Trentino quotidiano online, 06 March 2016
https://www.giornaletrentino.it/cronaca/trento/airpim-riconosciuta-la-prima-pmi-innovativa-in-trentino-1.613551

[4] Naimoli, A.E. (2015). Web Technologies in Oncology. In: Gatti, G., Pravettoni, G., Capello, F. (eds) Tele-oncology. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-16378-9_8

[5] Capello, F., Naimoli, A.E., Pili, G. (2014). Conclusions. In: Capello, F., Naimoli, A., Pili, G. (eds) Telemedicine for Children's Health. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-06489-5_12

[6] Naimoli, A.E. (2014). Connectivity, Devices, and Interfaces: Worldwide Interconnections. In: Capello, F., Naimoli, A. E., Pili, G. (eds) Telemedicine for Children's Health. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-06489-5_6

[7] Capello, F., Naimoli, A.E. (2014). eLearning: Distant Learning for Health Professionals That Work with Children. In: Capello, F., Naimoli, A., Pili, G. (eds) Telemedicine for Children's Health. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-06489-5_10

[8] Capello, F., Naimoli, A.E., Pili, G. (2014). Perceived Needs in Pediatrics and Children's Health: Overview and Background. In: Capello, F., Naimoli, A., Pili, G. (eds) Telemedicine for Children's Health. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-06489-5_1

[9] Naimoli, A.E. (2014). Technology and Social Web: Social Worldwide Interactions. In: Capello, F., Naimoli, A.E., Pili, G. (eds) Telemedicine for Children's Health. TELe-Health. Springer, Cham. https://doi.org/10.1007/978-3-319-06489-5_7

[10] Shiers, J.D., Naimoli, A.E. (2014). The High-Tech Face of e-Health. In: Gaddi, A., Capello, F., Manca, M. (eds) eHealth, Care and Quality of Life. Springer, Milano. https://doi.org/10.1007/978-88-470-5253-6_9

[11] Naimoli A.E., Devices for data production: which specificities in the pediatric field, speech at the International Bologna Consensus Assembly on Telemedicine, Bologna, March 2023

[12] Berte,D.(2018).Defining the IoT. Proceedings of the International Conference on Business Excellence,12(1) 118-128. https://doi.org/10.2478/picbe-2018-0013

[13] Sulkowski, Adam. (2019-2020). Industry 4.0 Era Technology (AI, Big Data, Blockchain, DAO): Why the Law Needs New Memes. Kansas Journal of Law and Public Policy Online, 29, 1-12.

[14] Sicari, S, Rizzardi, A, Coen-Porisini, A. How to evaluate an Internet of Things system: Models, case studies, and real developments. Softw: Pract Exper. 2019; 49: 1663–1685. https://doi.org/10.1002/spe.2740

[15] "Samsung Automation - Open source"
https://developer.samsung.com/automation/open-source.html

[16] Gabbrielli, M., & Martini, S. 2010. Programming Languages: Principles and Paradigms. Undergraduate Topics in Computer Science. Springer London. https://doi.org/10.1007/978-1-84882-914-5.

[17] Naimoli, A.E., EFA – Event Flow Architecture: an hybrid programming paradigm for distributed systems. Progress Report for PhD course. 2022.

[18] Tokas, Shukun & Owe, Olaf & Ramezanifarkhani, Toktam. (2020). Language-Based Mechanisms for Privacy-by-Design. 10.1007/978-3-030-42504-3_10

[19] Mena, M., Corral, A., Iribarne, L. and Criado, J., 2019. A progressive web application based on microservices combining geospatial data and the internet of things. IEEE access, 7, pp.104577-104590.

[20] Zach Mccormick and Douglas C. Schmidt. 2012. Data synchronization patterns in mobile application design. In Proceedings of the 19th Conference on Pattern Languages of Programs (PLoP '12). The Hillside Group, USA, Article 12, 1–15.

[21] Nakatani, Kazuo; Chuang, Ta-Tao; and Zhou, Duanning (2006) "Data Synchronization Technology: Standards, Business Values and Implications," Communications of the Association for Information Systems: Vol. 17 , Article 44. DOI: 10.17705/1CAIS.01744

[22] N. Al Ebri, Joonsang Baek and C. Y. Yeun, "Study on Secret Sharing Schemes (SSS) and their applications," 2011 International Conference for Internet Technology and Secured Transactions, Abu Dhabi, United Arab Emirates, 2011, pp. 40-45.

[23] Czech, Z. J., Havas, G., & Majewski, B. S. (1997). Perfect hashing. In Theoretical Computer Science (Vol. 182, Issues 1–2, pp. 1–143). Elsevier BV. https://doi.org/10.1016/s0304-3975(96)00146-6

[24] "*typing — Support for type hints — Python 3.12.1 documentation*" (official Python website documentation) https://docs.python.org/3/library/typing.html

[25] U.Farooq, M., Waseem, M., Mazhar, S., Khairi, A., & Kamal, T. (2015). A Review on Internet of Things (IoT). In International Journal of Computer Applications (Vol. 113, Issue 1, pp. 1–7). Foundation of Computer Science. https://doi.org/10.5120/19787-1571

[26] Statista 2024, StackOverflow surveys 2020/2021

- https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages
- https://insights.stackoverflow.com/survey/2020
- https://insights.stackoverflow.com/survey/2021

[27] Pang, L.-J., & Wang, Y.-M. (2005). A new (t,n) multi-secret sharing scheme based on Shamir's secret sharing. In Applied Mathematics and Computation (Vol. 167, Issue 2, pp. 840–848). Elsevier BV. https://doi.org/10.1016/j.amc.2004.06.120

[28] M. Singh and D. Garg, "Choosing Best Hashing Strategies and Hash Functions," 2009 IEEE International Advance Computing Conference, Patiala, India, 2009, pp. 50-55, doi: 10.1109/IADCC.2009.4808979.

[29] Singh, S., Sharma, P. K., Moon, S. Y., & Park, J. H. (2017). Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions. In Journal of Ambient Intelligence and Humanized Computing. Springer Science and Business Media LLC. https://doi.org/10.1007/s12652-017-0494-4

[30] Z. Chaczko and R. Braun, "Learning data engineering: Creating IoT apps using the node-RED and the RPI technologies," 2017 16th International Conference on Information Technology Based Higher Education and Training (ITHET), Ohrid, Macedonia, 2017, pp. 1-8, doi: 10.1109/ITHET.2017.8067827.

[31] Ferencz, K., & Domokos, J. (2019). Using Node-RED platform in an industrial environment. XXXV. Jubileumi Kandó Konferencia, Budapest, 52-63.

[32] "11 Most In-Demand Programming Languages" from Berkeley Boot Camps, UC Berkeley Extension, education branch of the University of California, Berkeley.
https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages

[33] Nátz, K., Orosz, T., & Szalay, Z. G. (2020). Methods of functional measurement of software. methods, 4(8), 9.

[34] La cura delle informazioni digitali in telepediatria alla luce della normativa europea, 2022, Quaderni acp (ISSN: 2039-1374), DOI: 10.53141/QACP.2022.162-163

[35] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2023. Could Tierless Languages Reduce IoT Development Grief? ACM Trans. Internet Things 4, 1, Article 6 (February 2023), 35 pages. https://doi.org/10.1145/3572901

[36] Ortiz, G., Castillo, I., Garcia-de-Prado, A., & Boubeta-Puig, J. (2022). Evaluating a Flow-Based Programming Approach as an Alternative for Developing CEP Applications in IoT. In IEEE Internet of Things Journal (Vol. 9, Issue 13, pp. 11489–11499). Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/jiot.2021.3130498

[37] R. Young, S. Fallon, P. Jacob and D. O. Dwyer, "A Flow Based Architecture for Efficient Distribution of Vehicular Information in Smart Cities," 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 2019, pp. 93-98, doi: 10.1109/IOTSMS48152.2019.8939233.

[38] S. Delcev and D. Draskovic, "Modern JavaScript frameworks: A Survey Study," 2018 Zooming Innovation in Consumer Technologies Conference (ZINC), Novi Sad, Serbia, 2018, pp. 106-109, doi: 10.1109/ZINC.2018.8448444.

[39] Morrison, J. P. (2010). Flow-based Programming: A new approach to application development. North Charleston, SC: Createspace Independent Publishing Platform.

©2020-2021-2022-2023-2024