DistriMuSe

D4.1
Requirements and design of the distributed platform, design and simulation tools

Lead Author(s):

Main editor(s):	Felipe Gil Castiñeira, Pablo Fondo Ferreiro (UVIGO)
Contributor(s):	Felipe Gil Castiñeira, Pablo Fondo Ferreiro, José Manuel Rúa Estévez, Juan José López Escobar, Daniel A. Rodríguez (UVIGO) Timo Schneider (CIT) Luca Davoli, Veronica Mattioli, Laura Belli, Gianluigi Ferrari (UNIPR) Ljubomir Jovanov (IMEC) Rafael Marín Pérez, Miriam Zambudio Martínez, Alejandro Arias (ODINS) Emily Walter, Marco Gunia (TUD) Irene Bicchierai, Lorenzo Falai, Francesco Brancati (RESIL) Mikko Saajanlehto (ELIVE)

Chips JU

EUROPEAN PARTNERSHIP

Roberto Girau (UNIBO)
Jesús Garrido (UNIGRA)
Álvaro Arco Castillo (SED)
Mehrjoo Masoomi, Roman Ipatkov, Kari Bäckman (BENETE)
Juha M. Kortelainen (VTT)
Ludo Cuypers, Jade Guo (COMMETO)
Pieter Crombez (Televic)
Peter Karsmakers, Reda El Hail (KUL)
David Abia Gutiérrez, Jacobo Domínguez Mosquera (ACORDE)
Federica Viero, Edoardo Pagot (RELAB)
Paloma Jimeno Sánchez-Patón, Raúl Santos de la Cámara (HIB)
Jakub Arm (BUT)
Mike Joosten (PRODRIVE)
Geert Vanstraelen (MACQ)
Andrea Generosi (EMOJ)
Fernando Seco (CSIC)

Reviewer(s):	Idilio Drago (UNITO) Álvaro Arco Castillo (SED)
Version:	1.0
Date:	20/12/2024
Due date of Deliverable:	20/12/2024
Actual submission date:	20/12/2024
Dissemination level:	PU

Revision history				
Version	Date	Modifications	Author(s)	Status
0.1	16/07/24	Initial structure	Felipe Gil Castiñeira, Daniel A. Rodríguez, Pablo Fondo Ferreiro (UVIGO)	Draft
0.2	1/10/24	First contributions	Felipe Gil Castiñeira, José Manuel Rúa Estévez, Pablo Fondo Ferreiro (UVIGO) Jesús Garrido (UNIGRA)	Draft
0.3	18/11/24	Contributions	All partners	Draft
0.4	30/11/24	Contributions review + editing	Felipe Gil Castiñeira, Pablo Fondo Ferreiro, José Manuel Rúa Estévez, Juan José López Escobar, Daniel A. Rodríguez (UVIGO)	Draft
0.5	9/12/24	First final version	Felipe Gil Castiñeira, Pablo Fondo Ferreiro, José Manuel Rúa Estévez, Juan José López Escobar, Daniel A. Rodríguez (UVIGO)	Final draft
0.6	18/12/24	Internal review	Idilio Drago (UNITO) Álvaro Arco Castillo (SED)	Reviewed draft
1.0	20/12/24	Final version	Felipe Gil Castiñeira, Pablo Fondo Ferreiro (UVIGO)	Final release

Table of Contents

List of Figures

List of Tables

1 Preface

This document, titled *"Requirements and design of the distributed platform, design and simulation tools"*, represents the initial deliverable of WP4, specifically addressing tasks T4.1 (*Specification of distribution platforms*) and T4.4 (*Specification of distribution platforms*). It outlines the preliminary version of the *"specification of the requirements that should satisfy the different components provided by WP4 (distributed platform, design and simulation tools)"* as well as the *"first version of the design of the distributed platform, design and simulation tools"*.

At this stage of the project, several tasks have recently been completed, delivering key results such as the initial state-of-the-art assessment, use case definitions, high-level objectives, technical requirements, and the requirement specification for all sensor units. Building on these results, this document serves as a foundational resource that will be refined over the coming months and further detailed in subsequent deliverables. These include D4.2 (*First version of the reference implementation for the distributed platform, design and simulation tools*), D4.3 (*First version of the reference implementation for design and simulation tools*), D4.4 (*Evaluation of AI applications in the distributed platform*), and, notably, the second version of this document: D4.6 (*Second release of the reference implementation for the distributed platform, design and simulation tools*).

The first objective of this document is to provide a more detailed analysis of the requirements derived from the various Use Cases for defining the distributed platform. This involves refining the results of D1.2 (*Use case definitions, high-level objectives, and technical requirements*) through the *"Extraction of requirements from the different Use Cases and partners"*. The document begins with a concise introduction that emphasizes the importance of obtaining system requirements as a critical step in system development, along with the theoretical principles underlying this process. It then outlines the methodology adopted for this task, illustrated through a practical example.

Subsequently, this methodology is applied to a representative set of the demos and pilots of the three Use Cases considered in DistriMuSe. The resulting analysis provides a comprehensive understanding of the characteristics of the data streams generated by various sensors, which is essential for defining the communication requirements of the distributed platform. It also details the algorithms planned for use, categorized as perception, processing, and decision-making algorithms, along with their associated computing resource requirements to evaluate the platform's processing capabilities and the feasibility of distributing these processes. In addition, the analysis specifies the requirements for actuators to ensure seamless compatibility and integration within the system. Finally, it highlights a set of non-functional requirements, including latency, reliability, security, and privacy, which are critical for the selection of suitable technologies for the distributed platform.

The following section, *Section 5: Design of the Distributed Platform*, begins with a high-level overview of the architecture and then delves into the key components: hardware platforms, communications, virtualization, distributed data access, intelligent platform orchestration, application composition, monitoring, observability, benchmarking, and security. Each subsection adopts a consistent approach, starting with a theoretical overview of the component within the context of a distributed architecture to establish an understanding of the fundamental concepts and key trends in the literature. This is followed by an examination of various technologies currently employed in practical implementations. Finally, each subsection concludes with one or several proposals for integrating the component into the DistriMuSe distributed platform architecture.

The last section, *Section 6: Design and Simulation Tools,* offers an overview of the tools and methodologies designed to support developers in implementing and testing their applications for the DistriMuSe distributed platform.

It is essential to emphasize that the purpose of this document is to propose an abstract distributed architecture that can be implemented using different technologies rather than to provide a detailed specification. However, since DistriMuSe aims to deliver a reference implementation, the document also highlights specific technologies that are identified as having promising features for the initial implementation.

2 Introduction

DistriMuSe aims to build an architecture that can handle the diversity of sensors, information types, secure communication over heterogeneous networks, and information processing devices to provide advanced services transparently to end users.

Each use case has particular requirements that can be addressed with different technologies and methodologies. Furthermore, DistriMuSe proposes leveraging the computing power available in the different devices to improve the results and optimize the whole system. Thus, by implementing distributed algorithms (with a special focus on AI) it will be possible to process the information at the optimal location in terms of the requirements of latency of the results, available computing capabilities, power efficiency, minimization of data transmitted, privacy, etc.

“Distributed algorithms” is a broad term that can be interpreted in different ways. It can be a sequential process where the information is processed successively by the elements in a chain (i.e. the sensors provide the raw data, then a second stage filters and fuses the information, a third stage extracts features, a fourth notifies users, etc.); it can be a centralized algorithm that runs alternatively in the cloud or at the edge, depending on the state of the network; it can be a complex AI algorithm implemented among different nodes that work as peers, etc.

In a traditional approach, there is a specific architecture for each of the use cases and an “ad hoc” implementation for each scenario. Thus, a large amount of developer effort is required for this specific design, development and debugging tasks.

DistriMuSe intends to provide an architecture that simplifies this scenario by providing the components that **enable developers to implement new services and applications that utilize various types of distributed algorithms without needing to adapt closely to the low-level specifics of a particular environment**.

This objective requires working in different aspects, such as:

- **Perception Abstraction:** Developers should not need to know how to configure each particular sensor. They should be able just to configure the relevant parameters and receive the data or even directly the events extracted from the data from a sensor, or several sensors combined (perception).
- **Communications Abstraction:** Developers should not need to configure each low-level communication protocol, or even know the details about addressing and routing.
- **Storage Abstraction:** Developers should be able to store information without requiring details about how or where the information is kept.
- **Computing Abstraction:** Developers should be able to provide their algorithms as code or binaries without requiring adapting them to different architectures or operating systems.

In order to provide such abstractions, the distributed platform needs a “control plane” that is able to discover nodes, “understand” the topology and the characteristics of the communication links, “decide” where to deploy the components of the algorithms or where the information should be stored, manage the lifecycle of services and end-user applications, “react” to changes, etc.

This is an ambitious goal. Ideally, the architecture should allow developers to simply describe (“program”) what they want to do and automatically arrange everything. Unfortunately, this is not yet possible, so developers still need to know some details of the particular scenario and adapt their code accordingly. Nevertheless, DistriMuSe can help with tools that simplify the design, development, debugging, and performance analysis of distributed applications.

3 Objectives

As stated before, the main objective for Deliverable 4.1, with title “Requirements and design of the distributed platform, design and simulation tools” is to **complete the first stages in the creation of an architecture** that enable developers to implement new services and applications that utilize various types of distributed algorithms without needing to adapt closely to the low-level specifics of a particular environment.

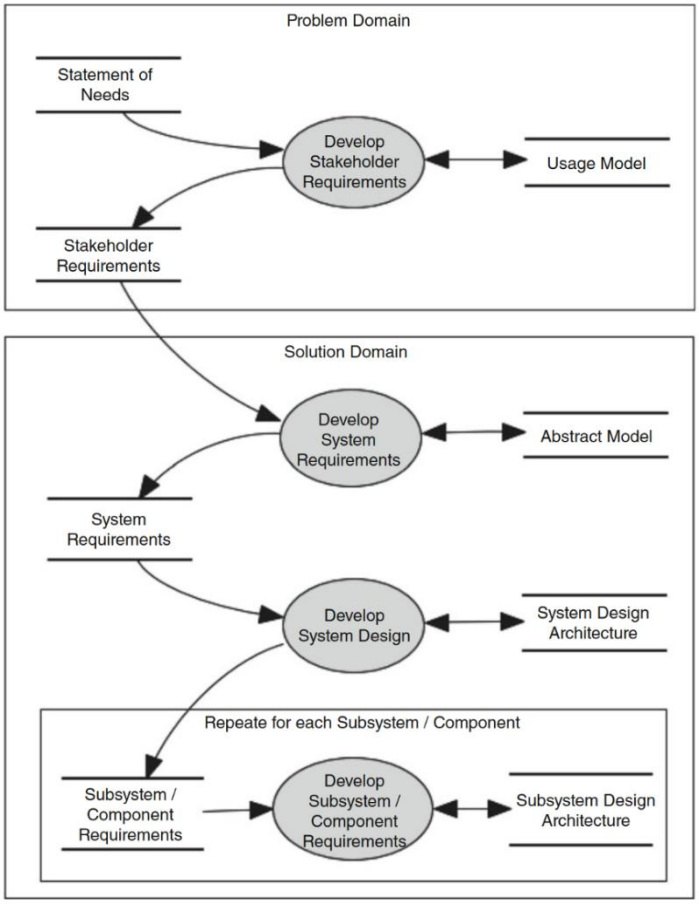

Figure 1: System development process (Dick, 2017)

This is **part of the process for the development of systems,** illustrated in Figure 1. The development process is depicted as a Data Flow Diagram, where ovals denote processes, and rectangles represent data or information that is either read or generated. Arrows indicate the direction of data flow, showing whether data is being read or written. Once a well-defined set of Stakeholder Requirements is established, outlining what stakeholders expect to achieve with the proposed system, consideration of potential solutions can begin. Instead of immediately moving to design, it is beneficial to first identify the essential characteristics that the system must possess, regardless of the final design details.

Thus, this objective involves several steps or subobjectives:

- Establish the stakeholder requirements
- Define the system requirements
- Develop the system design

Stakeholder requirements are used to understand what stakeholders or, in a narrower way, what users need to do (Dick, 2017). Adopting a usage-centric approach increases the likelihood that the solution will meet user needs effectively, focusing on essential capabilities and minimizing development effort on unnecessary features.

The stakeholder requirements have been extracted and defined in WP1. Deliverable 1.2 “Use case definitions, high-level objectives and technical requirements” analysed the e UCs and demonstrators presented for the project and provided a set of:

- User stories.
- The identification of the relevant target users and stakeholders of the system prototyped by the demonstrator.
- High-level architectural details.
- High-level requirements (detailing functional aspects and the overall behaviour of the system as a whole).

D4.1 aims to obtain the system requirements (in the solution domain) in order to develop the system design. In this domain, the requirements should be clear and “well formed”. This task is complex, and the solution is rarely reached in a single step, so it is necessary to complete it in different cycles or levels. At each level, modelling and analysis are performed to first understand the input requirements and, secondly, to establish a solid foundation for deriving the requirements at the next level down. The number of design levels is determined by the nature of the application domain and the extent of innovation required in the development process. Typically, the first level of system development in the solution domain involves transforming the stakeholder requirements into a detailed set of **system requirements defining the necessary functions and capabilities that the system must have** in order to address the problems identified.

With the system requirements in place, alternative design architectures can be explored. **A design architecture is represented as a collection of interacting components that together display the intended properties,** or the system’s “emergent properties” (see section 3, “Requirements”) which should precisely align with the characteristics outlined in the system requirements. This architecture specifies the responsibilities of each component and defines the interactions necessary among components to achieve the overall functionality specified in the requirements. In essence, the **design architecture specifies** the requirements **for each system component** outlining both their **functional responsibilities** and their obligations for **interaction**.

4 System requirements

Requirements form the foundation of every project**, outlining what stakeholders** (users, customers, suppliers, developers, businesses, etc.) **expect from the system**, as well as **specifying the actions the system must perform to meet those needs**.

Once communicated and agreed upon, requirements guide the project's progress. However, stakeholder needs can be diverse, sometimes conflicting, and may not be clearly defined from the outset. These needs may also be constrained by external factors or influenced by shifting goals over time. Without a stable set of requirements, a development project is likely to struggle or lose direction (Dick, 2017).

As stated before, requirements are fulfilled by the characteristics that emerge from the overall behaviour of the system, being the system the *"collection of components which cooperate in an organised way to achieve some desired result"*. At the core of the concept of a "system" is the idea of "emergent properties". This means that the value of a system is not determined by any single part, but rather arises from the interactions between its components. Some "emergent properties" are so fundamental to a system that they are rarely stated as explicit requirements (e.g. the ability of an aircraft to fly emerges from the complex interactions of all its key components, rather than being a single, isolated requirement). Another important concept is that of "systems of systems". This refers to the idea that every system can be viewed as a component of a larger, overarching system.

During the initial design of a solution, it is a good practice to avoid unnecessary solution details, that is, being "design agnostic". This can be expressed in terms of "black boxes". When considering the system as a whole, treating it as a "black box", focusing on:

- External interfaces, including inputs and outputs.
- Externally observable behaviours of the system, described in terms of those inputs and outputs.

As the design process progresses, the system will naturally be decomposed into various components. The initial "black box" view of the system is then expanded to reveal a series of smaller black boxes, each representing an internal subsystem. When considering the entire system as a black box, only the external interfaces are of interest. However, once the system is decomposed into subsystems, the design process also brings internal interfaces to light.

DistriMuSe involves a large number of partners and pilots, and to streamline and standardize the extraction of requirements, we have adopted the following methodology. For each pilot, a high-level system model is created to represent the various components involved, including sensors, algorithms (classified as perception, processing, and decision-making algorithms), and actuators (outputs), along with their interrelations, depicted as arrows. Additionally, a table is provided that includes the detailed requirements relevant to the design of the DistriMuSe distributed platform for each identified component. This table includes information such as data rates, interfaces, expected computing power, maximum latency for generating outputs, and other key parameters.

An example of a very high-level system model for DistriMuSe is represented in Figure 2.

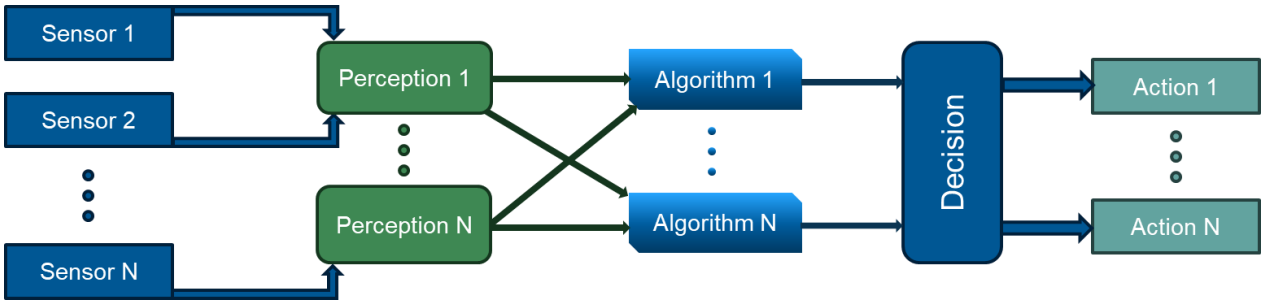

Figure 2: A high-level system model for DistriMuSe

This system model will be used to start defining the internal functionality of each one of the components, to understand their purpose and their interfaces. For example, Figure 3 shows a simplification of the internal functionality of a collaborative robotics cell that avoids harming a human operator when is near a robotic arm.

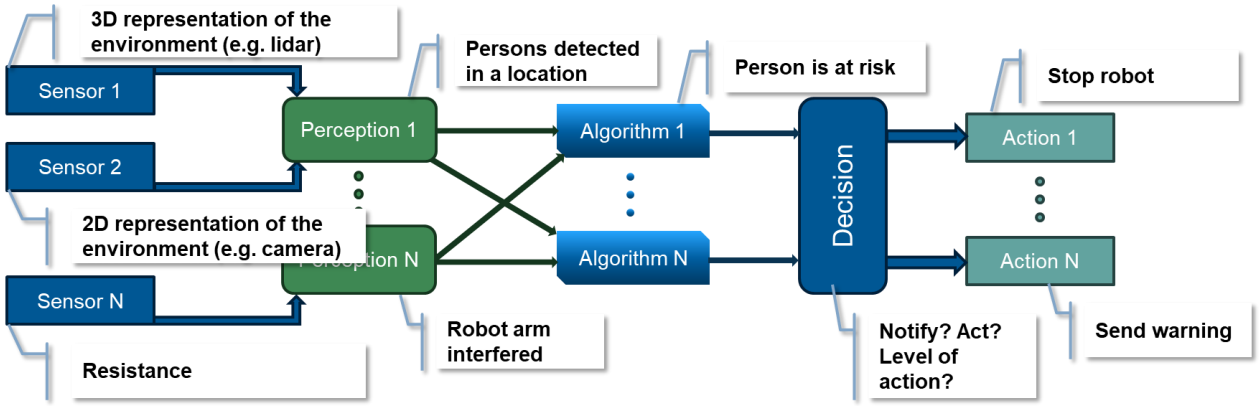

Figure 3: System model applied to a collaborative robotics cell

Requirements can be derived from detailed descriptions of the components, providing a basis for understanding each component's specific needs and roles within the system. The requirements must specify the functionality each component must deliver, the interfaces it must utilize or provide, and any constraints it must adhere to. These constraints may be directly imposed by overall system constraints (e.g., a mandated electronic technology for all components) or derived from them (e.g., distributing the overall system algorithms across components to reduce latency). Essentially, the requirements for each component (or subsystem) are treated as system requirements when that component is considered as an independent system. Figure 4 shows a simplification of some requirements that should be extracted for the example of a collaborative robotics cell. Table 1 shows an example of the detailed requirements that should be extracted for a collaborative robotics cell.

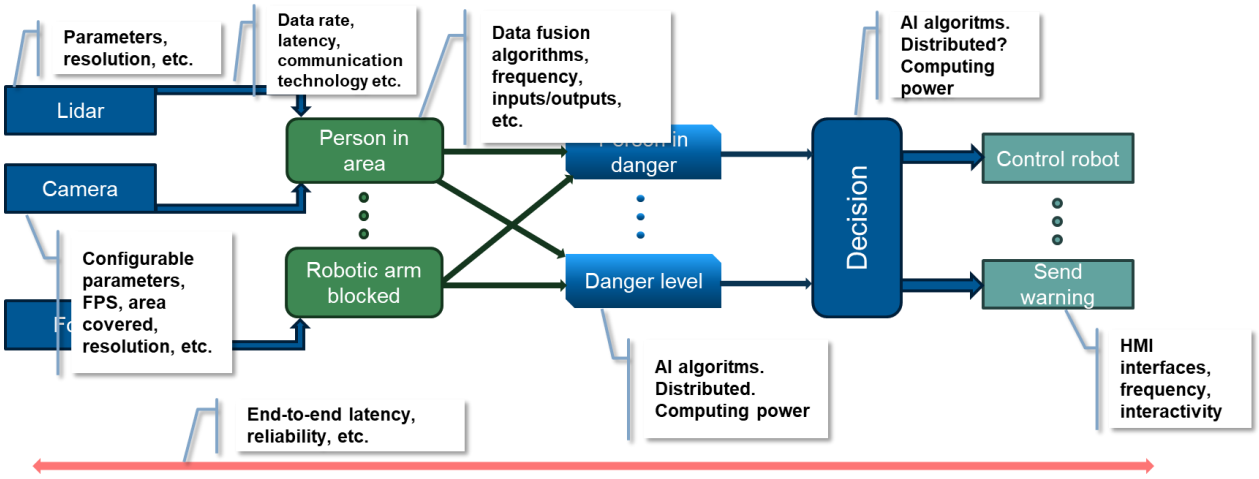

Figure 4: Example of requirements extracted for a collaborative robotics cell

Table 1: Example of the detailed requirements for a collaborative robotics cell

Sensors	

Lidar	• **Model:** LD-MRS820301
	• **Data output:** 3D point cloud
	  ○ Data output rate: 50Hz
	  ○ Interface: Ethernet (IP)
	  ○ Data format: See “Ethernet data protocol” in the annex.
	• **Input:**
	  ○ Data: movement
	  ○ Interface: Can bus
	• **Configuration interface:** Ethernet + RS232
	  ○ Configuration data: See “configuration data” in the annex

Camera	• **Model:** JKL 2D
	• **Data output:**
	  ○ Resolution: 2592×1944
	  ○ Frame rate: 24fps
	  ○ Interface:
	  ▪ Ethernet (IP)
	  ○ Data format: GigE Vision

Torque sensor	• **Model:** FT 300
	• **Data output:** Array of six float values
	  ○ Data output rate: 100Hz
	  ○ Interface: RS485
	• **Configuration interface:** none
	  ○ Configuration data: none

Sensor connection	
Lidar	• RS232, Ethernet, CAN • **Rate:** 100 Mbps
Camera	• Ethernet • **Rate:** 1 Gbps
Torque sensor	• RS485 • **Rate:** 10 Mbps

Perception	

Person in area	• **Input used:**
	  ○ Lidar
	  ○ Camera
	• **Algorithm:**
	  ○ DNN
	  ○ 1 GB RAM
	  ○ C++
	  ○ Dual-core processor
	• **Data output:**
	  ○ Rate: 100 Hz
	  ○ Data format: JSON (see annex)

Robotic arm blocked	• **Input used:**
	  ○ Torque sensor
	• **Algorithm:**
	  ○ Classifier
	• **Data output:**
	  ○ Rate: 100 Hz
	  ○ Data format: JSON (see annex)

Algorithms	
Person in danger	• DNN • 1GB RAM • C++ • Dual-core processor
Danger level	• KNN • 100 MB RAM • C • Single-core processor
Decision	

Stop robot	• Decide if robot should be stopped
	• **Distributed AI algorithm:**
	  ○ Node 1: evaluate probability
	  ▪ C++
	  ▪ 1 GB RAM
	  ▪ Single-core CPU
	  ○ Node 2: evaluate damage
	  ○ Node 3: decide actuation

Actuators	

Robot control	• **Model:** SR Controller SA22
	• **Data:**
	  ○ Interface: Ethernet (IP)
	  ○ Data format: See annex

HMI	• GUI • 1080p • Tactile interface • Qt
Non functional requirements	
End-to-end latency	• 50 ms
Reliability	• 99.99999% • Should work without cloud connectivity

It is relevant to emphasize that at each level in the solution domain, engineers must make decisions that progressively guide the design toward the final solution. These decisions inherently narrow the design space, excluding certain options, yet they are essential for making progress. It should be avoided to dive into excessive detail too early in order to be “design agnostic” and to allow creativity and innovation.

In this context, this document presents the first set of results, which will be revisited and refined through the various stages that will guide the design toward the final solution. A key milestone will be the start of the second cycle of the project, at which point we will review and update the requirements based on the results and experiences gained during the first cycle.

4.1 Distributed platform

This section outlines the requirements for a **representative selection of demos and pilots** considered across the three use cases of DistriMuSe. These selected pilots serve as the basis for identifying the requirements necessary for selecting the technologies and designing the distributed platform architecture. The requirements will be further refined and updated in D4.5, incorporating the outcomes of the first cycle.

4.1.1 UC1 Continuous Hybrid Health Monitoring

4.1.1.1 Demo 1.1 - Human life-style monitoring

Demo 1.1 is focused on human life-style monitoring, aiming to track daily activities of user groups. The different pilots defined for this use case introduce a different set of requirements, since they involve different environments, scenarios and technologies.

Pilot P1-LL: MCI long-term analytics and treatment tracking

Pilot P1-LL focuses on using multimodal sensing and decision-making to monitor motor and cognitive impairments in mild cognitive impairment (MCI) and Parkinson’s patients, as well as frailty status and fail risk estimation. The platform integrates RGB cameras for bradykinesia and tremor detection, microphones for acoustic and linguistic analysis, IMUs and Depth/RGB cameras for gait and posture analysis, and RGB/Thermal cameras with Radar for activity and fall detection.

These inputs are processed using advanced algorithms, including Kalman Filters, AutoEncoders, and statistical classifiers, with edge computing enabling low-latency real-time analysis. A meta-model combines insights to provide metrics like Parkinson’s Disease Level Index and Frailty Characterization, supporting precise, actionable healthcare insights - e.g. to evaluate the progression of disease in patients at several stages of the demo and evaluating their worsening level.

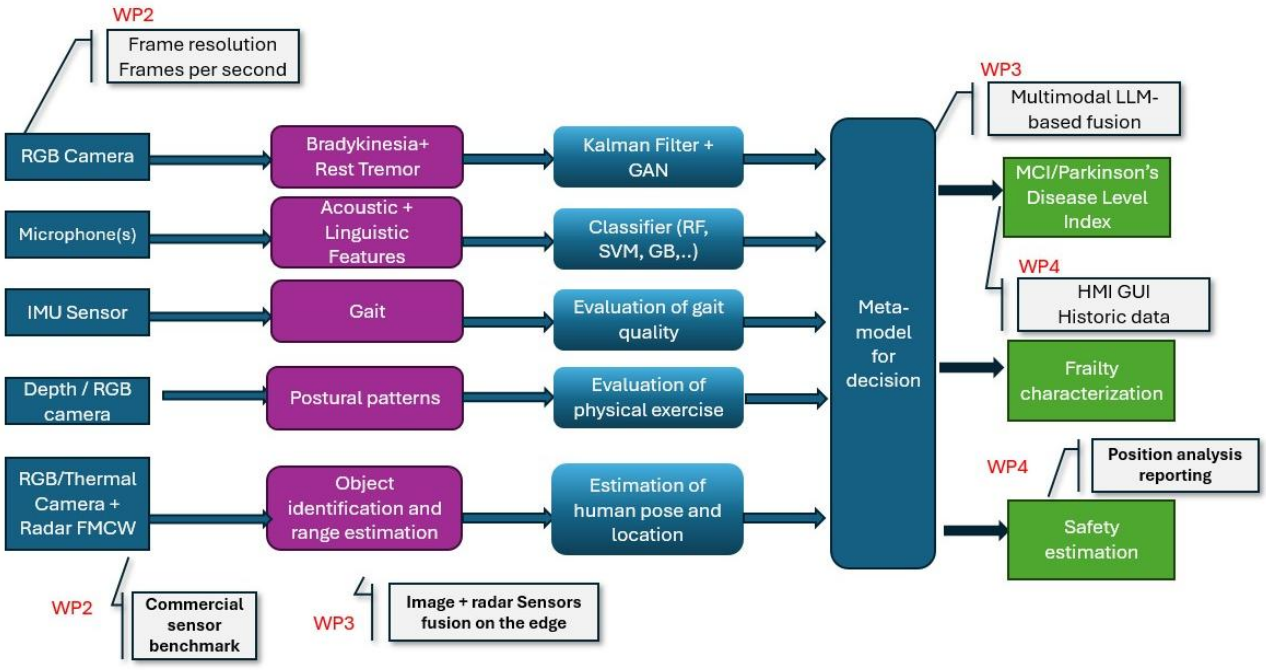

Figure 5: High level system model for Pilot P1-LL

Table 2: Detailed requirements for Pilot P1-LL

Sensors	
RGB Camera	• **Model**: High-resolution camera • **Data Output**: Bradykinesia detection, rest tremors • **Resolution**: Frames with adjustable resolutions 2K(2560*1440) 25fps,1080P(1920x1080) 25fps,720P(1280 x 720), VGA(640 x 480), QVGA(320 x 240) 25 fps and video compression (H.264) • **Frame Rate**: 25 fps • **Interface**: Ethernet (10/100 Mbps)
Microphone+Tablet	• **Model**: Tablet with a built-in microphone or/and lapel microphone • **Data Output**: Speech samples in wav format (non-compressed audio) • **Resolution**:16 bit/sample • **Sampling Frequency**: 16 kHz • **Interface**: Bluetooth/Ethernet
IMU Sensor	• **Model**: Foot-mounted inertial motion unit(s) • **Data Output**: three-axes accelerometer, gyroscope and magnetometer data (raw); step detections and gait parameters (processed) • **Resolution**: depending on application; typically 10 % of measured variable

	• **Frame Rate**: 100 Hz (raw signals); given by walking cadence (processed), i.e. 1-2 fps • **Interface**: Bluetooth
Depth camera	• **Model**: depth and RGB video • **Data Output**: raw images usable for human motion analysis • **Resolution**: 848 x 480 pixels (typical) • **Frame Rate**: 30 fps or lower • **Interface**: USB
RGB/Thermal Camera + Radar FMCW	• **Model**: FMCW Radar (commercial or Distrimuse Partner) + RGB or Thermal camera in edge node • **Output**: 2D image and 2D range estimation • **Resolution**: high precision <10m • **Sampling**: TBD, real-time processing on the edge • **Interface**: Ethernet/Wi-Fi for edge node
Perception	
RGB Hand Motion Features	• **Input**: RGB Camera Data • **Algorithm**: Media Pipe + Kalman Filter • **Output**: Hand Palm Detection + Hand Landmark Model (frequency of motion direction changes and change in distance of hand movement)
Voice Features	• **Input**: Speech wav files • **Algorithm**: Acoustic + Linguistic feature extraction + Statistical Classifier (RF, GB, ..) • **Output**: MCI level
Gait + Posture Features	• **Input:** IMU raw measurements, depth and RGB frames • **Algorithm**: INS (inertial navigation system) + gait segmentation; joint pose estimation • **Output**: gait parameters (step length, cadence, clearance, etc); joint pose 3D coordinates of human body
Radar + Camera Features	• **Input**: FMCW range estimation + Camera images (Thermal or RGB) • **Algorithm**: Image + range late fusion, body pose estimation • **Output**: body object, location relative to sensor
Algorithms	
Bradykinesia+ Rest Tremor	• **Input**: Motion features per frame • **Algorithm**: GAN • **Output**: Tremor Severity Level
Acoustic and Linguistic Analysis	• **Input**: Recording stored in a wav file

	• **Algorithm**: Acoustic + Linguistic feature extraction + Statistical Classifier (RF, GB, ..) • **Output**: estimation of the MCI level
Gait Analysis	• **Input:** IMU estimated gait parameters • **Algorithm:** SVM classifier • **Output:** estimation of frailty status and fall risk
Posture analysis	• **Input:** joint pose 3D coordinates of human body • **Algorithm:** evaluation quality of execution of physical exercises • **Output:** score of physical exercise performance and recommendations
Patient Activity	• **Input**: FMCW range estimation + Camera image (Thermal or RGB) • **Algorithm**: Image + range late fusion, body pose estimation, object identification • **Output**: detection of patient body object, location relative to sensor, pose estimation
Decision	
Meta-model for decision	• **Input**: User Location + Tremor Severity + Gait Quality + MCI level • **Algorithm**: TO BE DECIDED • **Output**: MCI/Parkinson Disease Level Index + Frailty Characterization (JSON)
Human Position	• **Input:** patient body estimator (image + radar), location/acceleration from other partners sensors • **Algorithm**: TBD, safety estimator from body pose, location relative to other objects and acceleration • **Output:** safety estimation
Gait and posture analysis	• **Input:** qualitative data from gait and posture methods • **Algorithm:** not yet fixed • **Output**: combined score for frailty status and physical exercise performance
Actuators	

Overall platform	• **Output data** (JSON Format)
	  ○ MCI/Parkinson Disease Level Index (integer 0-1)
	  ○ Frailty Characterization (integer 0-1)
	  ○ Human Position – safety score
	• Visualization dashboard

Non-functional Requirements	
Overall platform	• End-to-end latency < 200 ms • Data should be not lost if there are connectivity interruptions • Data should be kept stored

Pilot P1-G: Sensor-based human lifestyle monitoring

This pilot uses a decision-making framework for monitoring human activities within a care facility environment or an assisted home. For this, the following sensor modalities are used:

- Either an FMCW or a UWB radar
- A wearable location tag with alarm button (a part of a localization system)

It starts with four perception modules, each interpreting data from various sources: the first perception (*movement classifier*) interprets the incoming stream of radar data to identify short-term low-level movement events, while the second perception block (*within room-level localization*) estimates the positional 3D coordinates of foreground objects inside the room. The third perception block (*room-level localization*) receives information from tags to estimate room-level positions (e.g. person A is in room X). The fourth perception block (person-specific alarm trigger) triggers a person-specific alarm when a specific person has pressed her/his button.

Next, the output of the within room-level localization perception block is processed by the position semantics algorithm that, for example, translates position to “Person is located in the bed”. Once a position is translated to a common context it can be used in a generic (environment independent) algorithm that translates a series of low-level movement events to an activity (e.g. *If a person sits-down at a kitchen table around noon and subsequently makes up/down movements with the arms the person might be eating)*. This position semantics algorithm requires (semi-)automated calibration for each new environment. For example, in the logged data it is observed that a person typically is for a long time at a certain position during the night. The calibration procedure could then automatically assign bed area to this cluster of position as this is likely to be true.

The perceptions and the outcome of the position semantics algorithm are fed into four algorithms that further process the data. The first algorithm (*activity recognizer*) identifies high-level activities by fusing a sequence of low-level movements, contextual semantic position (e.g. kitchen) and temporal (e.g. noon) information. The second algorithm (*visitor detector*) focuses on detecting if there is a visitor. For example, if there are multiple moving persons and there is no caregiver (indicated by the tag-based system) it is likely that there are one or more visitors. The third algorithm (*uncertainty quantifier*) triggers an event when the radar measurements deviate strongly from what was observed during the training of the activity classifier. This could indicate that there is an unusual situation which requires attention. Furthermore, this unusual data could be stored (when buffered sufficiently long at the radar or perception block) in the cloud (at the *decision* block) for retraining the activity recognizer (and uncertainty quantifier) or update the position semantics algorithm (maybe new positions are measured where the person never was before). The last algorithm (*unresolved alarm detector*) is monitoring for situations where a patient recently asked for help while there is no caregiver close to that person at this moment.

Different to all other blocks that only buffer data for a limited time window, in the decision block all received data is stored to enable long-term analysis. This block consolidates the algorithm outputs to generate actionable outcomes, which are categorized as either alarms (indicating critical incidents,

like a potential fall) or warnings (highlighting unusual activity patterns, such as prolonged wandering or changes indicative of mild cognitive decline).

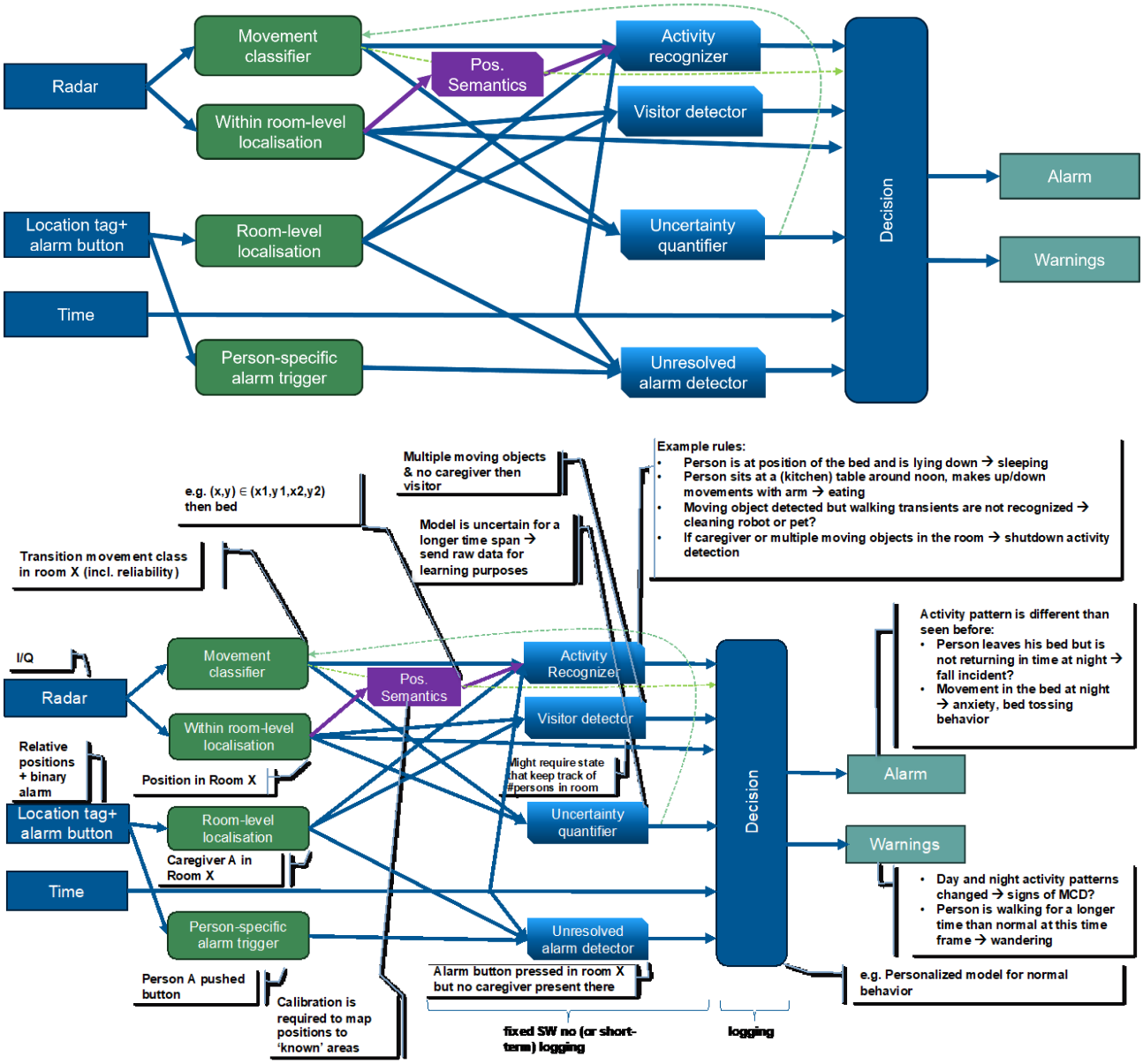

Figure 6: High level system models for Pilot P1-G

Table 3: Detailed requirements for Pilot P1-G

Sensors	
Radar	• **Model:** FMCW radar + UWB radar • **Data output:** o Range resolution: 0.126m o Maximum range: 6.46m o Doppler resolution: 0.093 m/s o Maximum Doppler: 4.471 m/s

	• **Parameter:**
	  o Transmit antennas: 3
	  o Receive antennas: 4
	  o Chirp loops: 96
	  o ADC samples: 64
	  o Frame rate: 50ms
	  o Start frequency: 60.75 GHz
	  o Wavelength: 4.94mm
	  o Slope: 54.71 MHz/us
	  o Bandwidth: 1180. 05 MHz
	  o Sampling frequency:2950 ksps

Location tag + alarm button	• Relative positions • Binary alarm
Time	• Receive data time (< 500ms second)
Sensor connection	
Radar	• **Data rate:** 45.6 kb/s • **Communication technology:** UART • **Baud rate:** 9600 / 115200 bps • **Storage:** Enough storage for the model (2.28 MB), dependencies (Python, TensorFlow/TensorFlow lite, NumPy, serial), input/output data • **RAM:** at least 4 GB • **Model input (Perception 1):** Normalized • Doppler time window with fixed size
Perception	
FMCW radar-Movement classifier	• **Inputs:** IQ radar data with shape (e.g. # samples: 256, # chirps: 100, # channels: 12). Where the first dimension is the number of samples per chirp, the second dimension is the number of chirps per frame and the 3rd dimension is number of antenna pairs. • **Outputs:** Probability value for each of the low level events • **Function:** a trained machine learning model such as a convolutional neural network that maps the radar features to a low-level movement event. • **Internal components:** Digital signal processing pipeline for extracting Doppler profile / Machine learning model
UWB radar-Movement classifier	• **Inputs:** Channel Impulse Response [CIR] (e.g. # sample: 60) • **Outputs:** Probability value for each of the low level events • **Function:** a trained machine learning model such as a convolutional neural network that maps the radar features to a low-level movement event. • **Internal components:** Digital signal processing pipeline for extracting Doppler profile / Machine learning model

FMCW radar- Within room-level localisation	• **Inputs:** IQ radar data with shape (e.g. # samples: 256, # chirps: 100, # channels:12). Where the first dimension is the number of samples per chirp, the second dimension is the number of chirps per frame and the 3rd dimension is number of antenna pairs. • **Outputs:** 3D coordinates of foreground objects. • **Function:** Calculate 3D coordinates to get relative positive with respect to the radar. • **Internal components:** Digital signal processing pipeline to extract point cloud. Group tracking algorithm with Kalman filtering for tracking.
UWB radar- Within room-level localisation	• **Inputs:** 4 Channel Impulse Response [CIR] (e.g. # sample: 60) from four antennas spaced λ/2 apart in 2x2 configuration. • **Outputs:** 3D coordinates of foreground objects. • **Function:** Calculate 3D coordinates to get relative positive with respect to the radar. • **Internal components:** Digital signal processing pipeline to calculate two different angle (azimuth, elevation) and a component which calculates the distance to with respect to the radar. One final component which combines this into a position.
Location tag+ alarm button- Room-level localisation	• **Inputs:** Range, RSSI and time information between tags and anchor nodes • **Outputs:** the room and position (X,Y) of the resident • **Function:** calculate the position of a tag using BLE range and signal strength information and/or ultrasound room detection information • **Internal components:** localisation algorithms and fusion of both pipelines
Location tag+ alarm button-Person-specific alarm trigger	• **Inputs:** button press • **Outputs:** trigger of the button press • **Function:** translate physical button press into a wireless message • **Internal components:** N/A
Algorithms	
Activity recognizer	• **Inputs:** Predictions of movement events, Localisation information, Predefined rules • **Outputs:** Predicted high-level activity • **Function:** Predict high-level activity derived from the combination of inputs and finite state machine corrections • **Type of algorithm/fusion:** Finite state machine • Storage for probability vectors (3.8kb/s)
Visitor detector	• **Inputs:** Position from radar, Position from Tags, Predefined rules • **Outputs:** Differ between nurse and visitor / Get number of persons in the room

	• **Function:** Predict the number of persons with the patient as well as if they are medical staff or visitors • **Type of algorithm/fusion:** fusion of rules and position data for both care giver and patient
Uncertainty quantifier	• **Inputs:** Doppler profile • **Outputs:** Record anomalous data decision • **Function:** Detect novel data unseen by the classification model • **Type of algorithm/fusion:** Out of distribution detection algorithms (Autoencoder, density/distance-based methods …) • Storage for anomalous data (45.6 kb/s)
Unresolved alarm detector	• **Inputs:** Alarm button push, Position caregiver • **Outputs:** Decision to trigger an alarm or not • **Function:** Trigger an alarm if the patient has an urgent care and no caregivers are responding. • **Type of algorithm/fusion:** fusion of alarm and position data
Decision	
Alarms Warnings	• **Inputs:** Activity, nurse / visitor presence, alarm button state (examples: Person is sleeping, disable radar, trustworthiness, Visitor / Nurse Detection, Record anomalous data for investigation, Trigger Alert) • **Outputs:** Alarms and warnings, health index • **Function:** Interpret algorithms output to actionable decisions • **Type of algorithm/fusion:** N/A
Distributed processing	
Edge processing (Gateway)	• Support for various sensor types and protocols • Support for real-time processing and low-latency responses • Collects sensor data in proprietary formats, standardizes, and securely sends to the cloud • Standardize data into a common format for further processing and analysis. • Dynamical deployment of new (AI) algorithms on the edge devices
Security	• Use of secure communication protocols (e.g., TLS, HTTPS) • Authentication and authorization mechanisms to protect access to devices and data • Encryption of sensitive data both at rest and in transit
Gateway communication	• Adherence to USP/TR-369 for device management • Implementation of agent-controller architecture • Secure and reliable communication with cloud-based services
Orchestration& Device management	• Standardized Approach: Leverages USP/TR-369 for device management on different system components

	• Agent-Controller Architecture: Agents on devices communicate with Controllers to exchange standardized management commands. • Enables remote monitoring and configuration of gateway parameters.
Non-functional requirements	
Latency	• model inference + pre-processing + to receive data time (< 500ms second)
Interoperability	• Compatibility with various sensor types and protocols • Integration with different cloud platforms and services
Security and Compliance	• Adherence to industry standards and regulations (e.g., USP/TR-369)
Scalability	• Ensure the system can scale to accommodate a growing number of devices and data streams.

4.1.1.2 Demo 1.2 - Sleep monitoring

Pilots P1-KMPHG/P1-KSL

The pilots in Demo 1.2 focus on developing less obtrusive methods for monitoring sleep and assessing sleep disorders with accuracy comparable to traditional polysomnography. This is achieved by leveraging advanced sensor and computing technologies to enable high-quality monitoring with minimal patient discomfort.

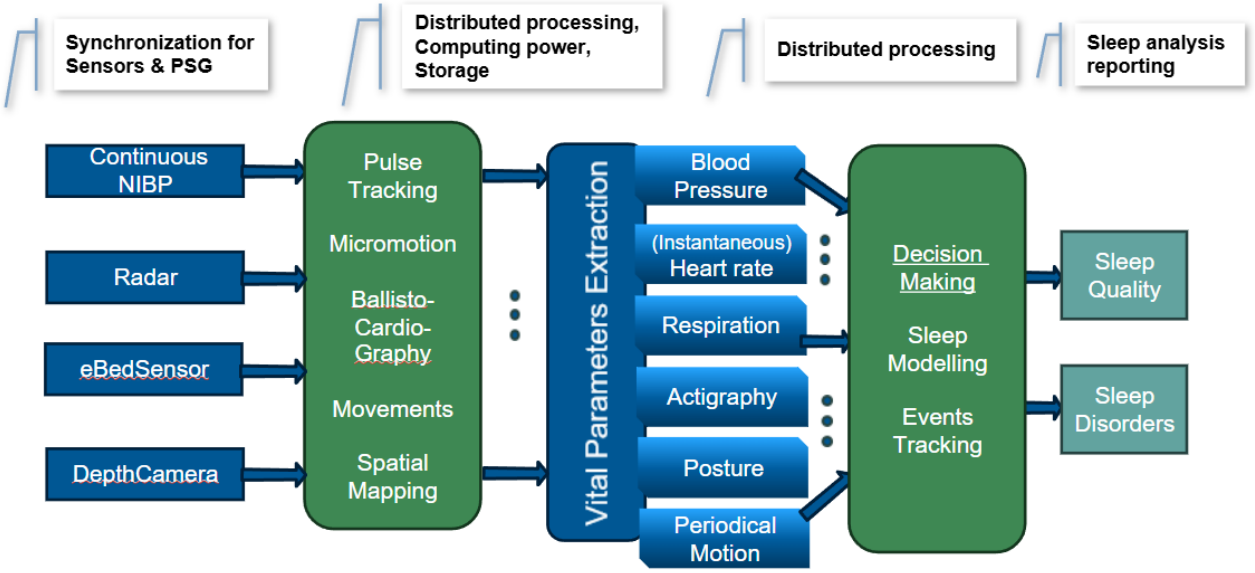

Figure 7: High level system models for Demo 1.2

To achieve accuracy comparable to traditional polysomnography, the Sleep Monitoring platform integrates multiple sensor technologies, distributed computing, and advanced algorithms. These components enable unobtrusive monitoring, real-time data analysis, and personalized insights for both healthcare professionals and patients.

Figure 7 illustrates how data flows from multiple sensors (e.g., blood pressure monitors, ballistocardiographic sensors, sound sensors) to perception layers. These layers process raw data into actionable insights using advanced algorithms. The system then integrates these insights for decision-making, enabling precise actions and recommendations.

Table 4: Detailed requirements for Demo 1.2

Sensors	

Radar	• Model: FMCW 60GHz MIMO prototype:
	cycle1, current device version
	cycle2, DM project developed polarimetric radar. These requirements are given in deliverable D2.1 under section "VTT"
	• Data output: 20 Mb/s with 50Hz sample rate
	• Edge processed output for 2D range-angle image
	  ○ Framerate: 50Hz
	  ○ 8.8 kb/s for each image pixel
	  ○ For typical target area < 1 Mb/s
	• Device Interface: Ethernet (IP)
	• Data format: VTT radar python library specific binary
	• Input:
	  ○ Data: Radar reflection magnitude for object tracking and phase for body micromotion detection (e.g. heartbeat)
	• Configuration interface: Ethernet
	  ○ VTT radar python library .yaml configuration file

eBedSensor	• **Model:** eLive.care (or new brand: ElvySense) 8-channel eBedSensor • **Device Interface:** USB-A (male) • **Data output:** 1,8 Kb/s with 110 Hz samplerate • 8 distinct measurement areas
Unobtrusive blood pressure monitoring (Continuous NIBP)	• **Data output:** Bluetooth LE
Ballistocardiographic sensors	• **Data output:** Bluetooth LE
Sound sensors	• **Data output:** Bluetooth LE
Depth camera	• Stereo imaging + laser-based 3D sensor

	• Main modalities for the use case: real-time 3D point clouds and micromotion maps • SW interfacing: custom C API, ROS, data can be stored to e.g. CSV files, other software interfaces can be decided later • HW interfacing: USB3, ethernet • Data points: ~12k points with 3D coordinates information and micromotion in indoor use • Data output: varies depending on the platform. On embedded device (Rasperry Pi 5) ~10 Hz, on laptop GPU (e.g RTX 3070) up to 100 Hz.
Algorithms	
Feature extraction for vital parameters	• Sensor Device specific methods
Data Fusion	• Heart rate and HRV • Respiratory disorders detection • Sleep posture • Sleep modelling
Continuous risk stratification and adaptive sensor recommendations	• Assess data for subtle changes indicating potential health risks. When elevated risks are detected, advanced sensors like EKG modules can be recommended or activated to provide deeper diagnostics.
High frequency radar data processing	• Local GPU-accelerated HUB processes high-frequency radar signals
Sensor data synchronization	
Definition	• Must provide NTP-level synchronisation signal to sensors
Additional plans	• External synchronization signal connected to reference PSG and sensors, e.g. random pulse signal (as suggested by Infineon)
Accuracy	• Accuracy of 10 milliseconds needed especially for the heart pulse detection methods. • Accuracy of 100 milliseconds is feasible for respiratory motion. • Accuracy of several of tens of seconds is feasible for sleep modelling at epoch phase.

4.1.1.3 Demo 1.3 - Sports performance and health assessment

Pilots P1-BRNO/P1-TOR

Demo 1.3 focuses on assessing sports performance and health status by monitoring the physical activity of participants (i.e., non-professional athletes) and patients (even affected by diseases). The overall goal is to measure the activity and exertion levels and produce an estimation of performance levels and maximum effort. This will be achieved by integrating heterogeneous technologies and sensing systems for evaluating physiological signs.

In detail, two pilots – namely, (i) pilot P1-BRNO taking place in specialized laboratories and sports facilities at Brno University of Technology (BUT), Brno, Czech Republic, in collaboration with the Centre of Sport Activities (CESA), and (ii) pilot P1-TOR, conducted at the University of Turin (UNITO), Turin, Italy, in collaboration with IRCCS "Istituto Auxologico Italiano" (IAI) at the San Giuseppe Hospital, Verbania, Italy – are foreseen and will integrate multiple wearable devices (e.g., smart watches, cameras, wearable bodice, wearable bracelets) to collect vital signs and fuse them (e.g., from IMUs, PPG, ECG, sweat analytics, and devices for measuring blood glucose, oxygen saturation, and blood pressure) to calculate different health features, such as heart rate recovery, heart rate variability, glycaemia variability, perspiration rate, energy expenditure.

Then, data flows (processed both locally and at the edge) will be useful for returning feedback and recommendations to physicians and trainers, also relying on cloud computing-based systems for additional data fusion and aggregation.

The system architectures for the P1-BRNO and P1-TOR pilots are shown in Figure 8 and Figure 9, respectively.

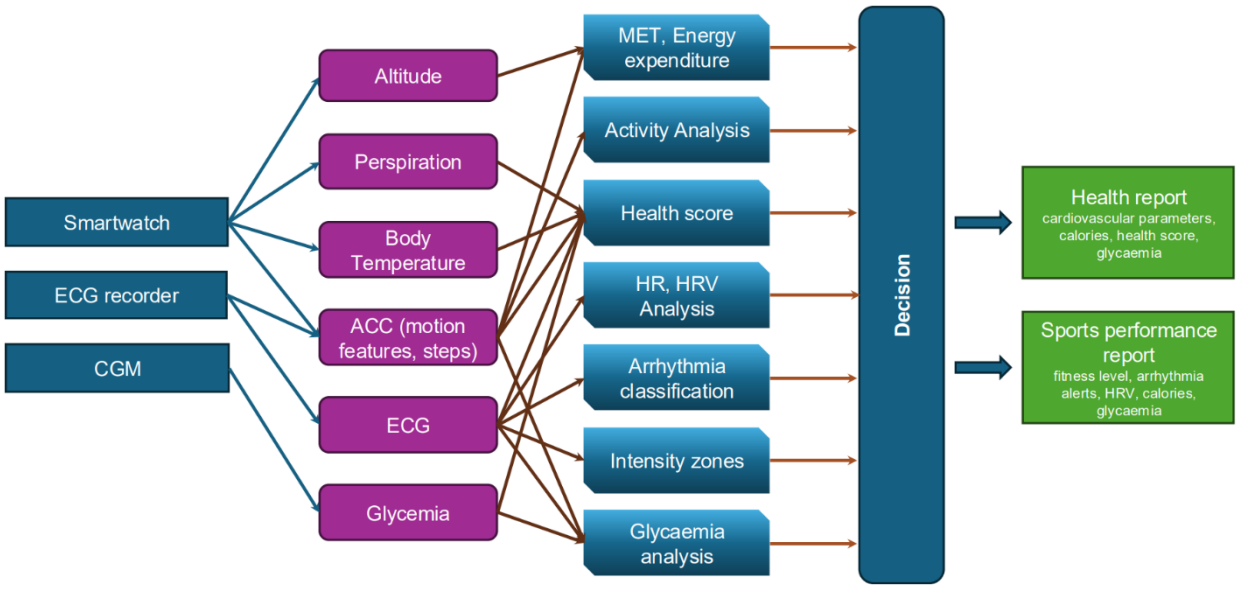

Figure 8: System architecture for the P1-BRNO pilot.

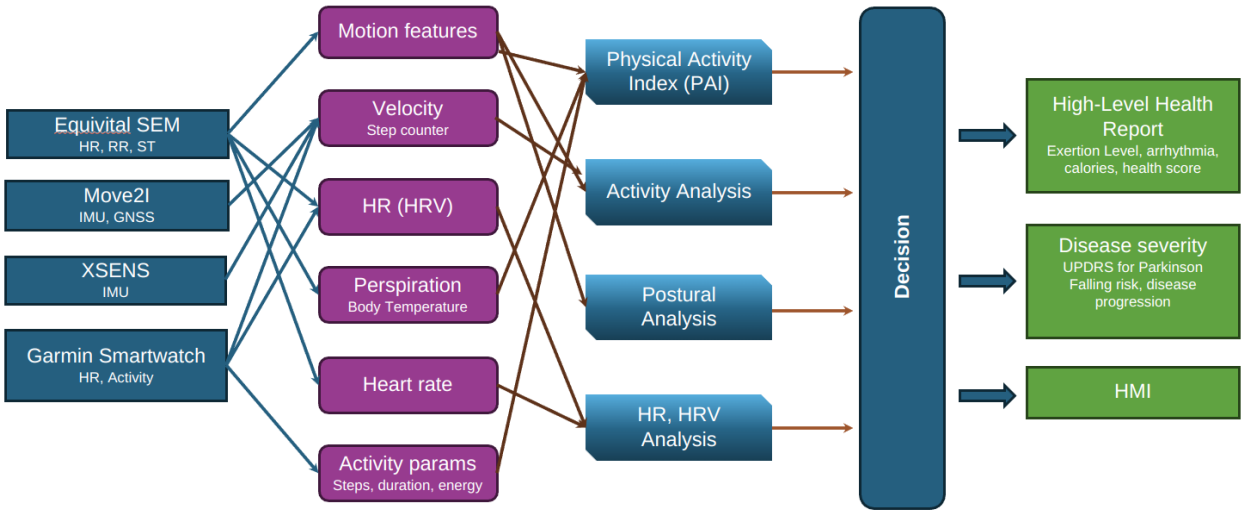

Figure 9: System architecture for the P1-TOR pilot.

Table 5: Main requirements for P1-BRNO pilot.

Sensors	
Smartwatch	• Accelerometer output rate: 100 Hz • Perspiration output rate: 1 Hz • PPG output rate: 1 Hz
ECG recorder	• ECG output rate: 1000 Hz • Accelerometer output rate: 100 Hz
CGM	• CGM output rate: 1 sample/min
Sensor connection	
Wearables with edge device	• Bluetooth Low Energy • Wireless Body-Area Network (WBAN)
Perception	
Altitude	• Altitude difference in 10s packet [m]
Perspiration	• Average perspiration rate in 10s packet [g/m2/h]
Body Temperature	• Average body temperature in 10s packet [°C]
ACC	• ACC features in 10s packet
ECG	• ECG features in 10s packet, raw data
Glycemia	• Glycaemia values [mmol/l]
Algorithms	
MET, Energy expenditure	• Application of prediction equations using classified activities and their intensity

Activity Analysis	• K-NN or DNN • C++ • Single-core processor
Health score	• Algorithm fusing extracted features and questionary information
HR, HRV Analysis	• RR intervals analysis • Features extraction in time/frequency domain
Arrhythmia classification	• CG features extraction • ML/DL methods
Intensity zones	• Detrended fluctuation analysis (DFA) based on RR intervals • Heart rate vs. DFA alpha1 analysis
Glycaemia analysis	• Variability calculation • Correlation with other features
Actuators	
Mobile application	• Android

Table 5: Main requirements for P1-TOR pilot.

Sensors	

Smartwatch	• Model: Garmin Fenix 7S
	• Data output:
	  ○ Interface: BLE then Wi-Fi
	  ○ Data type: heart rate, heart hate variability, respiration rate, body battery, stress level, sleep detection and naps, steps, floors, calories, intensity minutes, active minutes
	  ○ Output rate: 15 s (heart rate), 60 s (respiratory rate), 3 min (stress level and body battery), 5 min (heart rate variability) 24 h (steps, floors, calories, intensity minutes, active minutes)
	  ○ Input: photopletismogram, accelerometer, gyroscope, compass, skin thermometer, barometer, altimeter

Chest sensor	• Model: Equivital SEM + sensor belt
	• Data output:
	  ○ Interface: BLE
	  ○ Data type: heart rate, respiratory rate, skin temperature
	  ○ Output rate: 25 Hz (bunch of data every 15 sec)
	• Input: ECG (sampling 256 Hz), tri-axial accelerometer (sampling 25.6Hz and 256 Hz), respiratory signal (sampling 25.6 Hz)

Wearable sensors set	• Model: XSens MVN Awinda
	• Data output:
	  ○ Interface: proprietary BLE
	  ○ Data type: inertial data
	  ○ Output rate: 60 Hz
	• Input: IMU (accelerometer, gyroscope, magnetometer) with 1000 Hz sampling rate

Wearable device	• Model: move2i®
	• Data output:
	  ○ Interface: BLE, WiFi
	  ○ Data type: inertial data, GNSS
	  ○ Output rate: 10-100 Hz
	• Input: IMU (accelerometer, gyroscope, magnetometer) with 10 Hz sampling rate, GNSS receiver with 1 Hz update rate

Gateway	• Input: Sensor data • Interfaces: BLE, WiFi, USB • Software: Sensors proprietary stacks • Output: connection to the cloud • Battery powered
Sensor connection	
Garmin Fenix 7S	• BLE and Wi-Fi, to be used to send the information forwarded by the smartwatch to the paired smartphone, towards the Garmin servers through the mobile App
Equivital SEM	• BLE • Rate: 25 Hz (bunch of data every 15 sec)
XSens MVN Awinda	• Proprietary BLE Rate: 10 Mbps • Rate: 60 Hz
move2i®	• BLE • Rate: 1 Hz • WiFi • Rate: 10 Hz
Algorithms	
Postural analysis	• PC equipped with Windows 10 OS (at least) • Desktop applications required to receive and decode data from proprietary devices • Storage for collected data • 32 GB RAM • Matlab + Python • ML algorithms to be defined (e.g., *k*-NN or DNN)
Decision	
Disease level	• Distributed AI algorithms o Node 1: classify the cardiac status of the patient

	○ Node 2: classify the quantity of movement of the patient based on inertial information from different sensors ○ Node 3: evaluate to posture of the patient during the monitored exercise ○ Node 4: provide feedback to the doctor on the disease level (e.g., UPDRS index for Parkinsonian patients), to be then compared with the decision of the doctor
Actuators	
HMI	• A user needs to see the results of the evaluation algorithms in a user-friendly way
Non-functional requirements	
End-to-end latency	It can be relaxed in the order of seconds, since the decision is not critical (in terms of the timing it should be returned to the doctor)
Reliability	• 90% (since this system is not meant to replace the medical advisory, it should be an aid for the doctor monitoring the patients) • Might be required to work without Internet connectivity, so with local storage
Simplicity	• The measurement devices should connect easily by a user to the edge device.
Mobility	• At least 4 hours without external power
Flexibility	• Support new devices and protocols

4.1.2 UC2 Situation awareness for VRU and driver safety

4.1.2.1 Demo area 2.2 - Driver distraction detection to avoid accidents (with VRUs)

4.1.2.1.1 Demo 2.2.1 Driver distraction monitoring in a driving simulator

Demo 2.2.1 aims to monitor driver distraction using a driving simulator to assess the impact of different distraction types on driver behaviour. During the first cycle of the project (until M18), the simulator will collect data on driver activity, including facial expressions, eye movements, and other behavioural cues, to build a comprehensive dataset for training AI models. This data will support the development of models that can identify distracted driving events in real-time. In the second cycle (M36), the system will be refined and validated to accurately detect and predict distraction scenarios, providing actionable feedback for reducing driving risks.

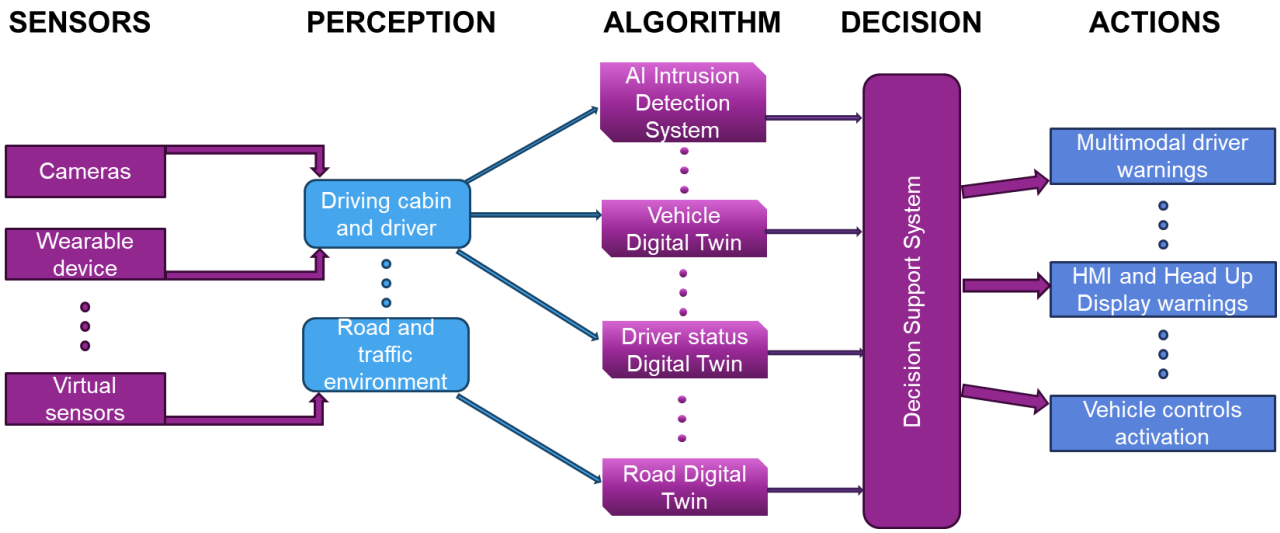

Figure 10: High level system model for demo 2.2.1

Figure 10 illustrates the various sensors planned for use in the demonstrator to gather data on the road and traffic environment, the driving cabin, and the driver. This data will be processed by the Digital Twins, which will analyse it to provide detailed information about the driver's condition, the vehicle's status, and the external context. The Decision Support System will further process this data, applying advanced algorithms to generate actionable insights. These insights will then be translated into specific actions, such as multimodal driver alerts, HMI notifications, and activation of vehicle controls.

Table 6: Detailed requirements for demo 2.2.1

Sensors	
Virtual sensor (ground truth data)	**Output:** • Virtual image of reality • Infrastructure data (e.g. traffic light status...) • Actors pose (e.g. pedestrian position) • Ego vehicle telemetry (e.g. acceleration, velocity, pedals)

RGB camera 1	• **Model:** FLEXIDOME micro 3100i
	• **Data output:**
	  ○ Resolution: 1,920 x 1,080
	  ○ Frame rate: 30fps
	  ○ Interface: Ethernet (IP)

RGB camera 2	• **Model:** Arducam OV2710 IR
	• **Data output:**
	  ○ Resolution: 1952x1092 px
	  ○ Frame rate: MJPG 30fps@1920 x 1080; YUY2 30fps@640 x 480
	  ○ Interface: USB2.0
	  ○ Data format: MJPG/YUY2

Sensor connection	
RGB Camera 1	• Ethernet

	• Rate: 1 Gbps
RGB Camera 2	• USB2.0 • Rate: 480 Mbps
Perception	

Operator Action Detection	• **Input used:**
	  ○ (Virtual) RGB Camera
	• **Algorithm:**
	  ○ CNN
	  ○ 512 MB RAM
	  ○ C++
	  ○ Dual-core processor
	• **Data output**
	  ○ Rate: 25 Hz
	  ○ Data format: JSON or CSV

Event Detection	• **Input used:**
	  ○ (Virtual) Camera
	• **Algorithm:**
	  ○ CNN
	• **Data output:**
	  ○ Rate: 30 Hz
	  ○ Data format: JSON

Anomaly Detection	• **Input used:**
	  ○ RGB Camera
	• **Algorithm:**
	  ○ Anomaly Detection classification
	  ○ Contextual explanation

Driving simulator	• **Input used:**
	  ○ Driver commands (throttle, brake, steering wheel etc.)
	• **Algorithm:**
	  ○ Industrial grade driving simulator software
	• **Data output:**
	  ○ Rate: ~20 Hz - 100 Hz
	  ○ Data format: CSV, MATLAB

Actuators	
Alert	• Alerts when driver distraction exceeds safe thresholds. Alerts shall be visual and/or auditory,
Algorithms	

Late Fusion Algorithm	• **Input data:**
	  ○ Camera-based Sensors
	  ○ Simulator virtual sensors
	• **Output data:**
	  ○ Fitness to Drive index
	  ○ Driver state predictions
	• **Data format:**

	○ JSON
FL for IDS Algorithm	• **Input data for the AI model:** ○ Data from multiple vehicles in CSV format: ○ Sender ID ○ GPS Position (x, y, z) ○ Speed (x, y) • **Output data:** ○ Whether an attack has been detected or not, and what type of attack.
Decision Algorithm	• **Decide if car should break/accelerate/send message.** Stepwise AI algorithm: ○ Step 1: Data aggregation and fusion ○ Step 2: Update digital twin and beliefs ○ Step 3: Decide behaviour
Non-functional requirements	
End-to-end latency	• 50 ms – 100ms • Alerts shall have a latency of less than 200 ms
Reliability	• Emotion recognition: 85%, Attention Detection: 95%
Data collection and storage	• Data collected during each driving session, including video feeds, driver movements, and other sensor inputs; shall be stored securely for further analysis and training of AI models.
Privacy and data security	• Collected data must be anonymized to ensure driver privacy. Secure protocols (e.g., TLS/SSL) shall be implemented for data transmission among the nodes of the system (e.g. between simulator and Digital Twin) and for storage to comply with GDPR.

4.1.2.1.2 Demo 2.2.2 - Driver and passenger mood and distraction monitoring

Demo 2.2.2 will focus on monitoring both driver and passenger mood and distraction levels during simulated driving scenarios. During the first cycle of the project (until M18), the system will gather data using in-vehicle sensors, including cameras and microphones, to evaluate behaviours and emotional states of the driver and passengers. This data will be used to build an AI model capable of recognizing signs of distraction and mood changes. In the second cycle (until M36), the developed model will be further refined and integrated into real-time systems to enhance driver and passenger safety by providing alerts and feedback to minimize risks associated with distractions and negative emotional states.

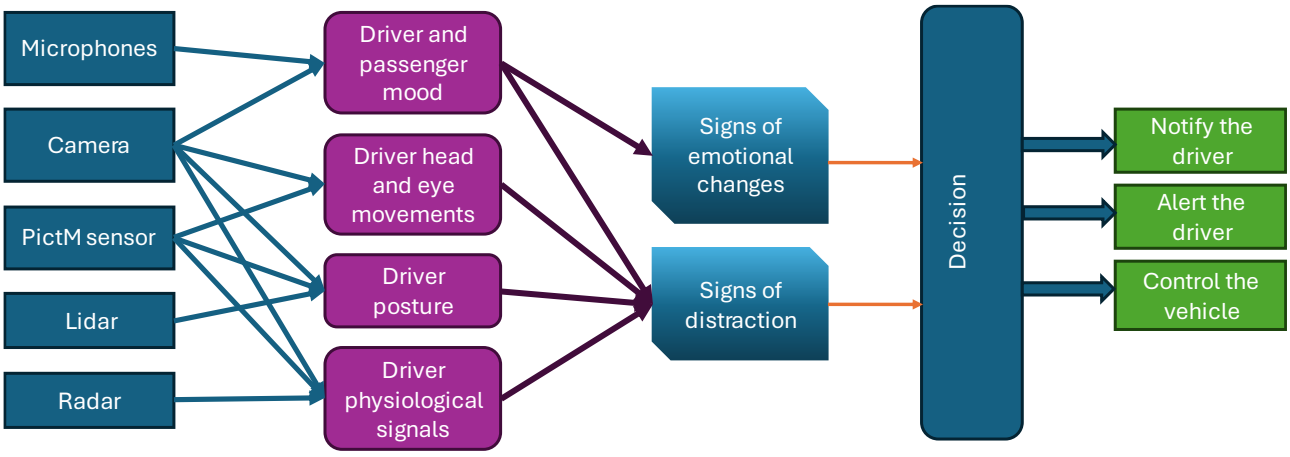

Figure 11 High-level architecture for demo 2.2.2.

4.1.2.1.2.1 Functional requirements

Based on the description of Demo 2.2.2, the following functional requirements can be identified for the distributed platform:

- Driver behavioural monitoring: the system shall capture and analyse real-time data from both the driver and passenger to determine distraction levels and mood states, distinguishing between driver-specific and passenger-specific activities and focusing on distraction detection for the driver and mood analysis for both.
- Alert system: the system shall provide real-time feedback in the form of visual or auditory alerts if driver distraction exceeds safety thresholds and communicate mood-related insights for passengers to the in-vehicle display to inform and support driving comfort and safety.
- Data collection and storage: the system shall collect data during each driving session, including video feeds, driver movements, and other sensor inputs. Data shall be stored securely for further analysis and improvement of AI models.
- Integration with in-vehicle sensors: the system shall integrate with cameras (RGB) and microphones in the vehicle to gather facial expressions etc. for mood and distraction analysis.

4.1.2.1.2.2 Technical requirements

Based on the description of Demo 2.2.2, the following technical requirements can be identified for the distributed platform:

- Sensor Integration: the platform must support integration with RGB cameras and other sensors to monitor facial expressions and body movements of both driver and passengers.
- AI model for distraction detection: DNN models shall run on an edge computing device for low-latency detection.
- Data processing: system must preprocess video feeds (e.g., face and head direction) in real-time ensuring efficient handling of high-volume data.

- Communication and alerting: the platform shall use MQTT or a similar lightweight protocol to communicate data. Alerts shall have a latency of less than 200 ms to ensure prompt notification during distraction events.

- Privacy and data security: collected data must be anonymized to ensure privacy for both driver and passengers. Secure protocols (e.g., TLS/SSL) shall be implemented for data transmission and storage to comply with GDPR.

- Environmental adaptability: the system should be robust to changes in lighting conditions (e.g., day/night, sunlight exposure) and ambient noise in the vehicle, ensuring reliable detection performance.

4.1.2.1.3 Demo 2.2.3 - Driver attention and traffic situation matching

Demo 2.2.3 will focus on synchronising the situation inside and outside of the vehicle. On-vehicle sensors monitor the vehicle's surroundings, detecting other road users and traffic infrastructure to create an understanding of the traffic situation. To include our vehicle and its driver as one of the road users in that situation, driver metrics from Demo 2.2.2 are used to provide driver's state and intention. In the first cycle of the project, data collection will take place from simple traffic scenarios. Later on, this data will be used as a basis for creating the system which will recognise the relevant traffic elements regarding the driver and their intention. During the second cycle of the project, the system will be further developed to operate in more complex traffic scenarios and to provide results in real-time possibly warning the driver about traffic users and elements they have missed.

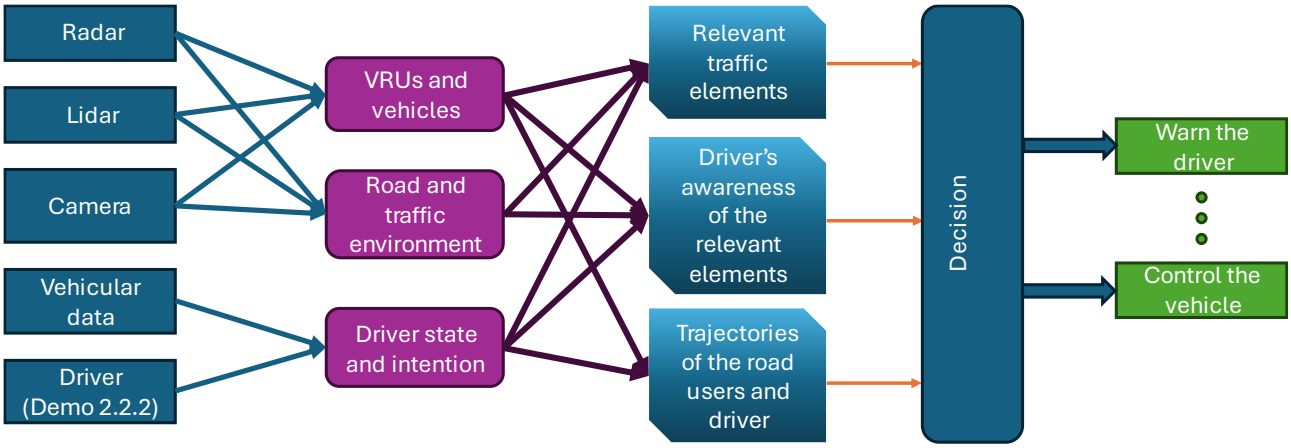

Figure 12 High-level system architecture for demo 2.2.3.

4.1.2.1.3.1 Functional requirements

Based on the description of Demo 2.2.3, the following functional requirements can be identified for the distributed platform

- Traffic scenario monitoring: the system shall capture and analyse real-time data from on-vehicle sensors observing the surrounding traffic. The system shall detect and track traffic elements such as road users and traffic signs.
- Matching driver attention and traffic situation: from the observed traffic elements, the system shall recognise those elements which are relevant for the driver's intention, e.g. VRU crossing the road the driver is driving. Furthermore, the system shall evaluate if the driver has noticed these elements and warn them in case they have not.

- Data collection and storage: the system shall collect data during each driving session, including camera images, lidar and radar data, odometry of the vehicle, as well as data from the driver provided by Demo 2.2.2. Data shall be stored securely for further analysis and development.
- Integration with on-vehicle sensors: the system shall integrate with the sensor setup installed on the test vehicle collecting traffic situation data which is later used when estimating driver's situational awareness. Sensor setup includes at least cameras (RGB), lidar, radar, GPS and IMU. Additional data sources may include odometry and map data.
- Integration with in-vehicle sensors: the system shall integrate with cameras (RGB) and other sensors monitoring the driver (Demo 2.2.2) inside the vehicle to include driver attention in analysing their situational awareness.

4.1.2.1.3.2 Technical requirements

Based on the description of Demo 2.2.3, the following technical requirements can be identified for the distributed platform:

- Sensor integration: the platform must support integration with various environmental perception sensors such as cameras, lidar, radar and GPS.
- Data processing: the system must process sensor data in real-time providing results in time for the driver to be made aware of a possible danger, all while the accuracy of the results must maintain at a high level.
- Privacy and data security: the collected data must be anonymised to ensure privacy for the driver and other road users. Data transmission and storage shall follow the GDPR.
- Environmental adaptability: the system shall function within normal traffic conditions. Severe adverse weather is excluded from the ODD (Operational Design Domain): heavy rain (44 mm/h) and dense fog (visibility < 50 m).

4.1.3 UC3 Safe interaction and cooperation with robots

The objective of UC3 is to develop a certifiable safety system designed to detect and respond when a human enters a designated safety zone. The system must accurately identify the human's position within the workspace, including the location of upper body limbs, and adjust the robot's operational speed based on proximity and the level of hazard present. A critical capability is distinguishing humans from non-human objects to ensure precise and reliable safety responses. Furthermore, the monitoring and control software must be robust and adaptable to dynamic, unpredictable environments, ensuring consistent performance under varying conditions.

UC3 will establish a common framework of UC3, with a single set of scenes under representation, where three demos will address different approaches to develop certifiable safety system. Demo 3.1 will represent an integrative approach including the physical system with all the sensors systems and controlling the real robot. The learning systems here will be previously trained with simulated data generated via 3D representations of the scenes obtained as part of Demo 3.2. Finally, Demo 3.3. will create simplified framework to address the more specific challenge of detecting AMR driving backwards and humans.

From the point of view of the distributed platform, a single overall architecture of modules can be considered in common for all the demos in UC3, as it is described in Figure 13.

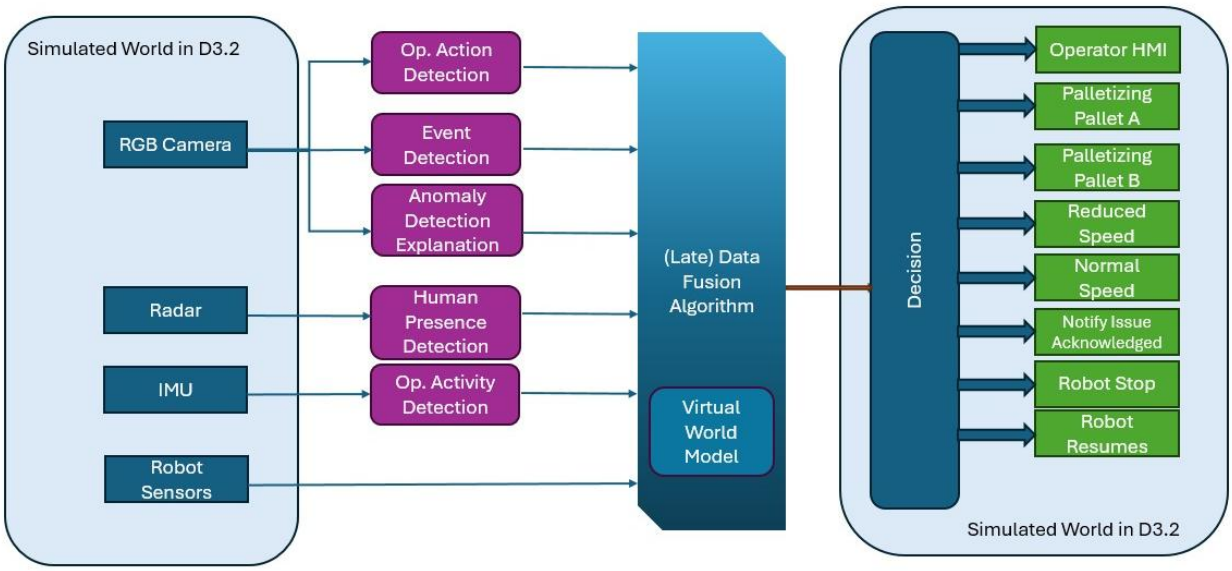

Figure 13: High level system model for UC3.

The detailed requirements for the modules integrated in this architecture are described in Table 8.

Table 7: Detailed requirements for UC3

Sensors	

Radar	• **Model:** LD-MRS820301
	• **Data output:** 3D point cloud
	  ○ Data output rate: 50Hz
	  ○ Interface: Ethernet (IP)
	• **Input:**
	  ○ Data: movement
	  ○ Interface: Can bus
	  ○ **Configuration interface:** Ethernet + RS232

RGB Camera	• **Model:** JKL 2D
	• **Data output:**
	  ○ Resolution: 2592×1944
	  ○ Frame rate: 24fps
	  ○ Interface:
	  ▪ Ethernet (IP)
	  ○ Data format: GigE Vision

IMU	• **Model:** Nicla Sense ME
	• **Data output:**
	  ○ Data output rate: 30Hz
	  ○ Interface: Bluetooth
	  ○ Data format: Numeric
	• **Input:**
	  ○ Data: Operator movement
	  ○ Interface: Bluetooth
	• Configuration interface: none
	  ○ Configuration data: none

Robot sensors	• **Model:** UR20
	• **Data output:** Gripper 6 DoF pose with respect to the robot base
	  ○ Data output rate: 125Hz
	  ○ Interface: ROS
	• Configuration interface: none
	  ○ Configuration data: none

Simulated sensors (D3.2)	• **Model:** N/A. Software simulation of the 3D environment, generating the output of the sensors above. Radar, IMU, RGB cameras and robot sensors will be simulated.
	• **Data output:**
	  ○ Data output rate: Same as the real sensor (different output for each simulated sensor).
	  ○ Interface: ROS
	• Configuration interface: none
	  ○ Configuration data: none

Sensor connection	
Radar	• RS232, Ethernet, CAN • Rate: 100 Mbps
RGB Camera	• Ethernet • Rate: 1 Gbps
IMU	• Bluetooth • Rate: 30Hz
Robot sensors	• Ethernet • Rate: 100 Mbps
Simulated sensors (D3.2)	• Ethernet • Rate: 1Gbps.
Communication TSN Switches	• Ethernet, SFP ports • Rate: 1 Gbps
Perception	

Operator Action Detection	• **Input used:**
	  ○ Lidar
	  ○ Camera
	• **Algorithm:**
	  ○ DNN
	  ○ 1 GB RAM
	  ○ C++
	  ○ Dual-core processor
	• **Data output:**
	  ○ Rate: 100 Hz
	  ○ Data format: JSON

Event Detection	• **Input used:**
	  ○ Camera
	• **Algorithm:**
	  ○ Classifier
	• **Data output:**
	  ○ Rate: 100 Hz
	  ○ Data format: JSON

Anomaly Detection	• **Input used:**
	  ○ RGB Camera
	• **Algorithm:**
	  ○ Anomaly Detection classification
	  ○ Contextual explanation
	• **Data output:**
	  ○ Rate: same as camera
	  ○ Data format: JSON

Radar Detection	• **Input used:**
	  ○ Radar
	• **Algorithm:**
	  ○ Classifier
	• **Data output:**
	  ○ Rate: 100 Hz
	  ○ Data format: JSON

Operator Activity Detection	• **Input used:**
	  ○ Accelerometer + Gyroscope
	• **Algorithm:**
	  ○ Deep Neural Network
	• **Data output:**
	  ○ Activity recognition + anomaly detection
	  ○ Data format: JSON

Algorithms	

Data Fusion Algorithm	• **Input data:**
	  ○ Action Detection
	  ○ Event Detection
	  ○ Anomaly Detection
	  ○ Robot Sensors
	• **Output data:**
	  ○ Danger level (1-5) per area
	• **Data format:**
	  ○ ROS

Virtual World Model	• **Input used:**
	  ○ Robot sensors
	  ○ Operator position
	  ○ Operator Pose
	• **Algorithm:**
	  ○ 3D reconstruction of the elements.
	  ○ Detection of collisions between meshes.

	○ GPU, CPU and memory intensive. • **Data output:** ○ Rate: ~1 Hz ○ Data format: ROS message (timestamp, risk level, collision point).
Decision	
Decision Algorithm	• Decide robot behaviour • Stepwise AI algorithm ○ Step 1: Data aggregation and fusion ○ Step 2: Update digital twin and beliefs ○ Step 3: Decide behaviour
Actuators	
Robot control	• Handled by SR Controller
HMI	• **Input:** level of danger and type of situation detected • **Output:** visual/haptic/auditory feedback to the operator
Non functional requirements	
End-to-end latency	• 50 ms
Reliability	• Processing of sensor data must occur locally at the edge to ensure low latency for real-time safety measures.
Compatibility with other platforms	• Compatibility with ROS (OpenRobotics, 2024) would be highly beneficial for the integration of robotics devices since most hardware manufactures provide controllers for specific software platforms (such as ROS2).

Considering this general architecture of modules, as well as the technical requirements for each module, the following sections will go more in detail on the particularities of each demo in UC3. In addition to these requirements, other relevant functional and technical requirements have been discussed for each particular demo.

4.1.3.1 Demo 3.1 - Sensor fusion as a reliable safety measure

Demo 3.1 aims to implement sensor fusion as a reliable safety measure in an industrial setting involving human-robot interaction. During the first cycle of the project (until M18), a series of sensors, including RGB cameras, motion capture devices, and force sensors, will be integrated to monitor human-robot interactions in real-time. The collected data will be used to build sensor fusion models that enhance the accuracy of detecting safety-critical events, such as proximity violations and unsafe human postures. In the second cycle (until M36), the system will be tested in real-world scenarios to assess its ability to dynamically respond to safety risks by generating alerts and controlling the robot's actions to prevent incidents.

4.1.3.1.1 Functional requirements

- Human-Robot interaction monitoring: the system shall integrate multiple sensor modalities (e.g., cameras, motion capture, force sensors) to continuously monitor human-robot

interactions in real-time, detecting critical events such as proximity breaches, unsafe postures, and potential collisions between humans and robots.

- Sensor fusion for safety enhancement: the system shall combine data from different sensors to improve detection accuracy and reliability in identifying hazardous conditions. Fusion algorithms must provide insights such as distance measurements between the robot and the operator, operator pose recognition, and force analysis.

- Real-time alert and control mechanisms: the system shall generate alerts when predefined safety thresholds are crossed, such as when a human enters a danger zone and be capable of communicating with the robot control unit to trigger emergency actions (e.g., stop or slow down) in hazardous situations.

- Real-time communications: Implement deterministic communications for applications that need to work with low latencies and close to real time, with technologies such as Time Sensitive Networking (TSN).

- Communications control: Deterministic communications should be managed and monitored from a central controller.

- The robot must be equipped with an emergency stop button that allows an operator to manually stop its operation. Robot stopping functions must operate in real time.

- Monitoring of AI system alarm levels must operate in real time.

4.1.3.1.2 Technical requirements

Based on the description of Demo 3.1, the following technical requirements can be identified for the distributed platform.

- Devices and platforms connecting with TSN networks should do so via fibre optic or Ethernet interfaces. It is recommended that the connecting interfaces have network configuration capabilities such as VLAN assignment.

- The system must operate reliably under varying environmental conditions, including changes in lighting (e.g., day/night) in the workspace.

4.1.3.2 Demo 3.2 - Enhancing safety with virtual reality

Demo 3.2 will use virtual reality reconstructions of the scenes to provide valuable information for collaborative robot tasks. During the first cycle of implementation (until M18) of the project, the virtual environment, reproducing the scenes considered in Demo 3.1, will simulate all the physical devices under consideration, including the robotic arm, the unattended vehicle, conveyor belt or even simulated operators (Figure 13). Based on these simulated scenes, databases will be generated (including video sequences, raw data of the sensors and additional information describing what is happening in the scene) that will be used to train data fusion modules that can be embedded in the control loop in Demo 3.1. Then, during the second cycle of the project (M36), the virtual environment will be fed with data from the sensors in the real devices to recreate the current state on the system, providing events information such as collision risk alerts.

4.1.3.2.1 Functional requirements

Based on the description of Demo 3.2, the following functional requirements can be identified for the distributed platform:

- The distributed platform must support simulated sensor sources and arbitrary evolution of time variable. Since the evolution of simulated environments is mainly constrained by the computational load and associated resources, simulated sensors might require computation time faster (or slower) than real-time. It might require some synchronization mechanism when multiple simulated processes are present in the system.
- The distributed platform must support database generation with the information provided from the nodes in the network. This feature would allow that the same implementation of the nodes can be used to generate database for training AI models and later, in a successive stage, to be embedded in the control loop.
- Similarly, the distributed platform should support using previously generated databases to feed the platform nodes, so that previous actions can be re-played to study the behaviour of the modules embedded in the platform.
- Since some hardware devices (such as robotic arms) might require low-level communication or controllers, the distributed platform must allow workers/nodes/processes to be allocated to specific resources.

4.1.3.2.2 Technical requirements

Based on the particular description of Demo 3.2, the following technical requirements can be identified for the distributed platform.

- The distributed platform must provide an application programming interface (API) with C/C++/C# compatible languages for sending/receiving messages.

4.1.3.3 Demo 3.3 - Dynamic factors robots-human safe interaction

In demo 3.3. concepts will be developed to detect humans when driving backwards in a dynamic factory environment. Major challenges are the unpredictable nature of the humans in the factory as their movement patterns are unknown, and that the factory is updated often to accommodate new product introductions, changes in market demand, and stock availability. During the first cycle the partners will focus on an AGV/AMR driving backwards and detecting humans while ensuring complete safety in an operational environment. The first ideas imagine a scope around the AGV/AMR in which it detects humans from other objects in its way. This envisioned system features sensor diagnostics, optimizes the adaptability of mobile robots to safely cope with dynamic and changing environments, monitors the safety status of the mobile robots to track when the actions of the robots will result in unsafe situations, creates a safety zone overview with visualization of unsafe situations of the factory floor. In a second cycle (M36) it will become possible to track humans further, possibly implementing more sensors and more scenarios within the dynamic factory environment taking gait patterns into account. An upgrade of the SLAM/global map could also present opportunities when the first demo is successful.

Because the environment is different from demo 3.1. and demo 3.2. the requirements are also slightly different. Even though both demo's aim to develop and test a certifiable safety system designed to

detect and respond when a human enters a designated safety zone, the demo 3.3. will immediately encompass a physical world. This is why there are different requirements and data fusion steps needed. The architecture is shown in Figure 14 and in Table 8: Detailed requirements for UC 3.3 and demo 3.3.

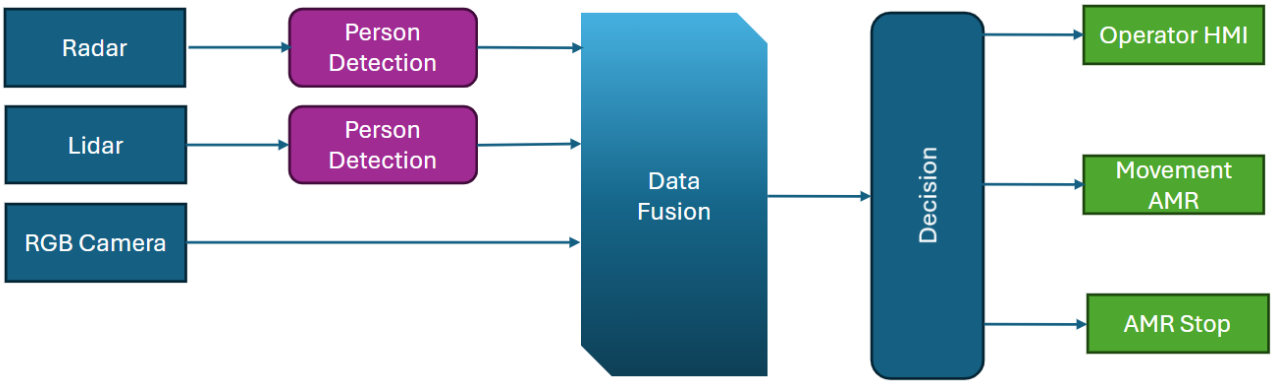

Figure 14: First draft of the dataflow architecture of UC 3.3. and demo 3.3

Table 8: Detailed requirements for UC 3.3 and demo 3.3

Sensors	
Radar	• **Model:** 120 GHz FMCW Radar and aperture. 20 GHz bandwidth, 20 x 30 x 30 mm
Lidar	• **Model:** OS32C-BP-DM • **Detection Angle:** 270 • **Angular Resolution:** 0,4 • **Range:** Up to 10 m • **Interface:** EtherNet/IP & IO
RGB Camera	TBD
Sensor connection	
Radar	• Wired USB connection via FCC. 200 Hz measurement rate
Lidar	• Ethernet, M12/4P
RGB Camera	TBD
Perception	

Person Detection	• **Input used:**
	  ○ Radar
	  ○ Lidar

Actuators/Output	
Operator HMI	• Output: Visual and acoustic feedback
AMR	• **Model:** Mercury VH100-60/30

4.1.3.3.1 Functional requirements

The functional requirements identified for demo 3.3 are:

- Human-Robot interaction monitoring: the system will integrate different sensors (e.g. Lidar and Radar) to monitor humans in the environment of the AGV/AMR to detect unsafe events such as a collision with a human.
- In the data fusion step, multiple data sources will be combined to improve control, trustworthiness and accuracy of the detection.
- The solution should work real-time and will communicate the findings with the AGV/AMR to trigger emergency actions, such as slowing down or stopping completely.
- The solution will map a human in a detection field to assess the situation in real time, knowing where the human is and what danger level this poses. The distinction between moving objects and still objects is important for this requirement.

4.1.3.3.2 Technical requirements

The technical requirements identified for demo 3.3 are the following:

- Real-Time Responsiveness: The platform must support low-latency communication to ensure timely interactions by the AMR.
- Indoor Operation Reliability: Must function effectively in indoor environments with potential signal reflections or other noise.
- Radar Format: Compatibility with I/Q, Range-Doppler, and PointCloud radar data formats.
- Efficient Processing: Optimized for energy-efficient processing on robots to support autonomous operations.
- Accuracy: Provides sufficient accuracy for operational needs.
- Explainability and Interpretability: Outputs and decisions should be understandable, explainable and interpretable by users.
- Data Interfacing: Must facilitate seamless data exchange between multiple radar units.

5 Design of the distributed platform

The aim of this chapter is to introduce the main considerations adopted for the design of DistriMuSe's distributed platform. There is a strong need for flexibility and adaptability in the platform, in order to support the wide range of heterogeneous requirements identified in the previous section, arising from the different use cases.

This section is organized in 9 parts, devoting the first one to introduce the high-level overview of the architecture, also including a theoretical background for justifying the proposed design. Then, the following 8 sections analyse the different components of the distributed platform, including hardware platforms, communications, virtualization, distributed data access, intelligent platform orchestration, application composition, monitoring, observability, benchmarking and, finally, security. We follow a similar structure inside each section, first contextualizing the relevance of the component in the distributed platform, then reviewing the state-of-the-art of the technologies related to that component and finally introducing the main design considerations.

5.1 High level architecture

A software architecture of a system is *"the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" (*Ostrowski, 2021*).* A distributed system is "*an information-processing system that contains a number of independent computers that cooperate with one another over a communications network in order to achieve a specific objective*" (Puder, 2006).

In distributed applications, components execute across multiple physical locations, which brings several advantages: tasks can align with geographic requirements, fault tolerance is enhanced through replication, and performance gains are achievable through task parallelization, among other benefits. However, the execution environment presents significant heterogeneity, with varied hardware platforms, network technologies, operating systems, and programming languages. This diversity poses substantial challenges in developing distributed applications.

We first introduce the well-known alternatives for distributed system architectures to provide a broader context of the challenges involved in their design. Then, we review the state-of-the-art to better understand the latest solutions for distributed architectures and understand their features and limitations. Finally, we present a high-level overview of our proposed distributed architecture and its components.

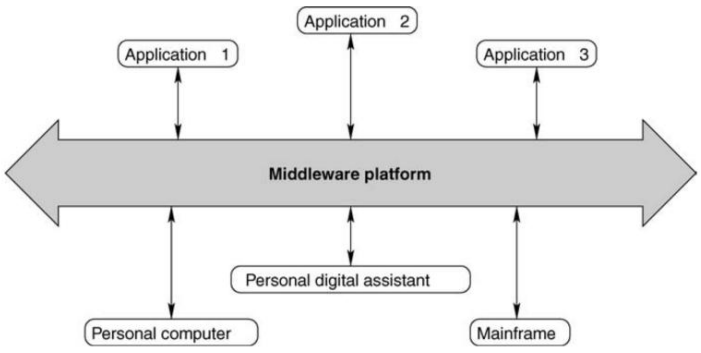

Figure 15: Traditional middleware for distributed systems

Traditional distributed systems used a middleware infrastructure to bridge the applications and the network (*Figure 15*). A middleware platform provides this infrastructure by acting as a buffer, isolating applications from the complexities of the network. While the network primarily serves as a transport mechanism, access to it varies significantly based on technological factors and the underlying physical platforms.

DistriMuSe is a more ambitious and complex scenario that needs more than just network isolation. In DistriMuSe we have a heterogeneity of the components (different types of technologies) that should be abstracted for the application programmers. The literature describes this abstraction characteristic as “transparency”, making the complexity resulting from the distribution transparent (invisible) to the applications programmer.

A middleware for DistriMuSe should offer all the required services for supporting the distributed execution of applications. Applications should be implemented as a set of objects, functions, modules or components that encapsulate their state and behaviour, so they should be only accessed via well-defined interfaces. The interfaces abstract away the implementation details, effectively encapsulating various technologies. As a result, an object serves as a unit of distribution and interacts with other objects by exchanging messages.

Our middleware should implement the following tasks (Ostrowski, 2021):

- Operational interaction: The middleware should allow the operational interaction between two components.
- Remote interaction: The middleware should allow the interaction between two components located in different locations.
- Distribution transparency: From the standpoint of the program, interaction between components is identical for both local and remote interactions.
- Technological independence: The middleware supports the integration of different technologies.

With this middleware, an application would be represented as a set of interacting components (Figure 16). Each component may be allocated to a specific hardware platform with a specific operating system, architecture, and network connectivity; being implemented in different programming languages; and storing the persistent information in different locations.

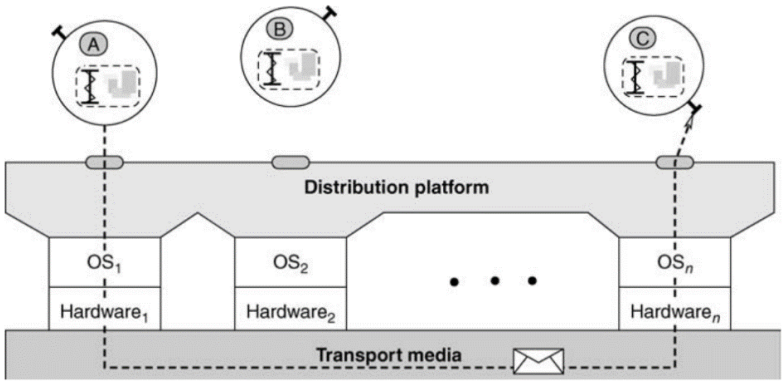

Figure 16: Middleware supporting the distribution of components (Ostrowski, 2021)

The middleware usually overcomes the heterogeneity by offering application programmers an application programming interface (API). This way, application programmers typically see middleware as a program library and a set of tools.

This API can be seen as a horizontal interface between an application and the middleware, defining how an application can access the functionality of the middleware. It is also necessary a vertical interface that defines the interface between two instances of a middleware platform. This can also be seen as the “control plane” that allows the distributed platform to coordinate the different nodes.

There are different architectures and paradigms that we should have into account to implement the distributed platform:

- **Service-Based Architecture (SBA)**: is an architecture style that features a distributed macro-layered structure with independently deployed user interfaces and remote coarse-grained services, but that relies on a single monolithic database. While it adopts microservices principles like domain-driven design and bounded contexts, SBA simplifies database management by avoiding database-per-service separation, addressing the complexity of distributed databases but trading off service autonomy and scalability. See Figure 17.

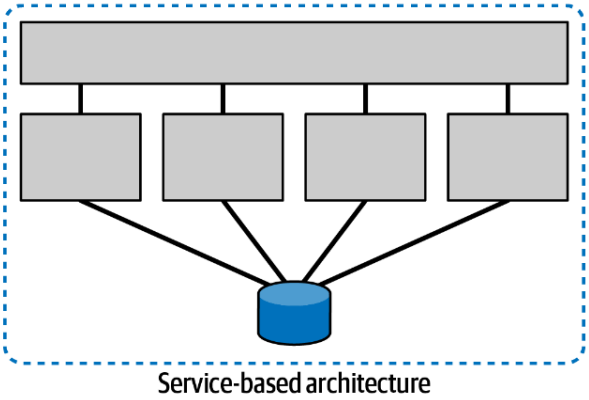

Figure 17: Service based architecture (Ford, 2021)

- **Brokered or mediated architecture** (Figure 18): is an architecture where a broker, mediator or orchestrator handles all the request from the application or user layer.

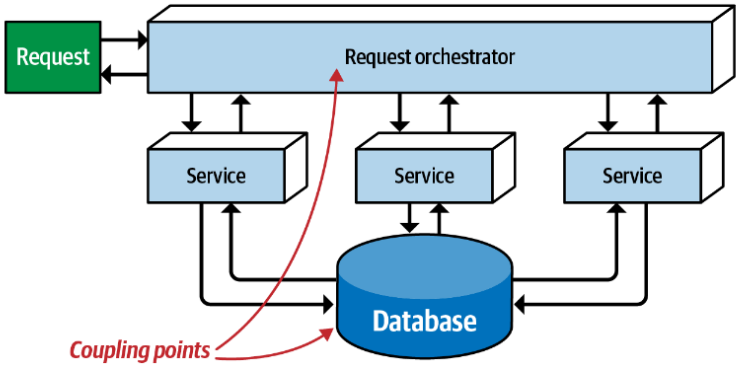

Figure 18: Mediated architecture (Ford, 2021)

- **Event-Based Architecture (EBA):** is a software design pattern focused on generating, detecting, and responding to events within a system. Events signify important occurrences or state changes, making EBA ideal for building responsive, real-time, and loosely coupled distributed systems. See Figure 19.

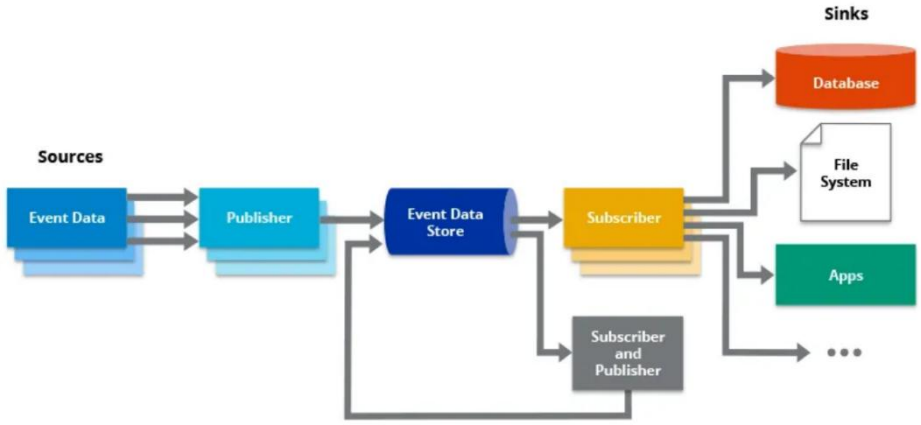

Figure 19: Event based architecture (Peiris, 2023)

- **Microservices Architecture:** is a design style where highly decoupled services, each acting as a bounded context, operate independently, including their data dependencies. Each service can have unique architectural characteristics, such as scalability or security, tailored to its needs. This approach enables teams to work rapidly and autonomously, minimizing the risk of disrupting other services. See Figure 20.

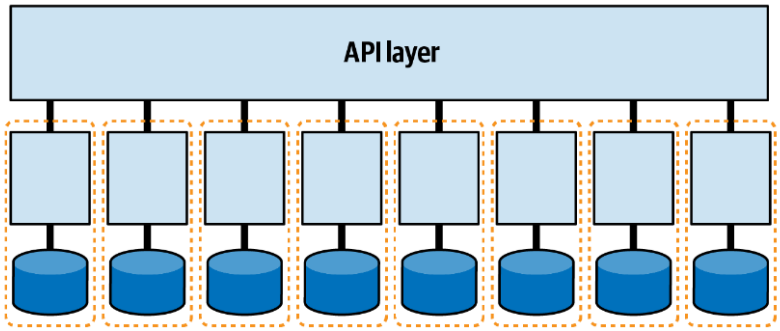

Figure 20: Microservices architecture

When we have a large architecture, we will have different components interacting among them. When two services communicate, we must decide whether the communication should be synchronous or asynchronous:

- **Synchronous communication:** requires the requestor to wait for the response from the receiver. The calling service makes a call and blocks until the receiver returns a value or status.

- **Asynchronous communication:** allows the sender to continue processing without waiting for a reply, enabling loose coupling and better scalability but introducing complexity in handling responses and ensuring message delivery.

Another important dimension when designing an architecture is how to handle the coordination among modules, that is, how to manage the workflow between services during communication. It becomes increasingly important as workflow complexity grows. Two common patterns address this:

- **Orchestration:** A central service controls and directs the interactions between other services, ensuring tasks are completed in a defined sequence.

- **Choreography:** Services interact directly, reacting to events in a decentralized manner without a central controller.

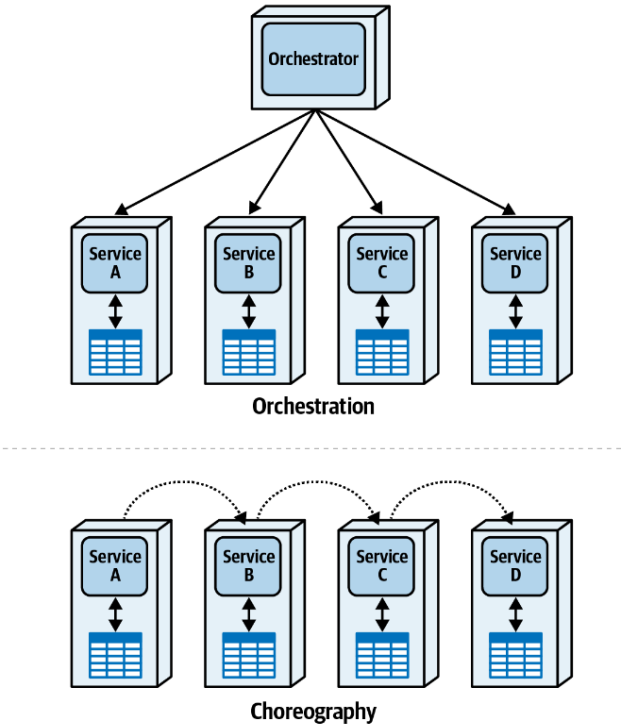

Figure 21: Orchestration versus choreography in distributed architectures (Ford, 2021)

While simple workflows (e.g., a single service responding to a request) require minimal coordination, more complex workflows demand thoughtful design to manage dependencies and ensure reliable execution.

Until now, we have focused on the design of a distributed architecture from a "static perspective," considering an architecture tailored for a specific application. However, **DistriMuSe** requires a more dynamic approach—an architecture capable of supporting the flexible deployment of applications. This allows developers to introduce new applications to address emerging use cases, update existing ones, or add additional functionality, all while leveraging the same underlying infrastructure.

Such architecture requires a “control plane”, to manage all facets of module execution such as deployment, scheduling, monitoring, accounting, etc. This control plane should also manage the lifecycle of the infrastructure, as resources may be volatile (nodes may disappear because of instability in the connectivity or battery limitations).

Ideally, in the initial formation phase, the control plane should identify potential nodes that are part of the network and that can implement distributed applications. It should discover all the available resources suitable for participation. These resources are then filtered based on the specific application. It should create a model documenting key details such as software and hardware capabilities, battery levels, network connections, and the historical behaviour of resources regarding availability and stability (Ferrer, 2023). Figure 22 illustrates the lifecycle of a temporary infrastructure, including the creation and adaptation stages.

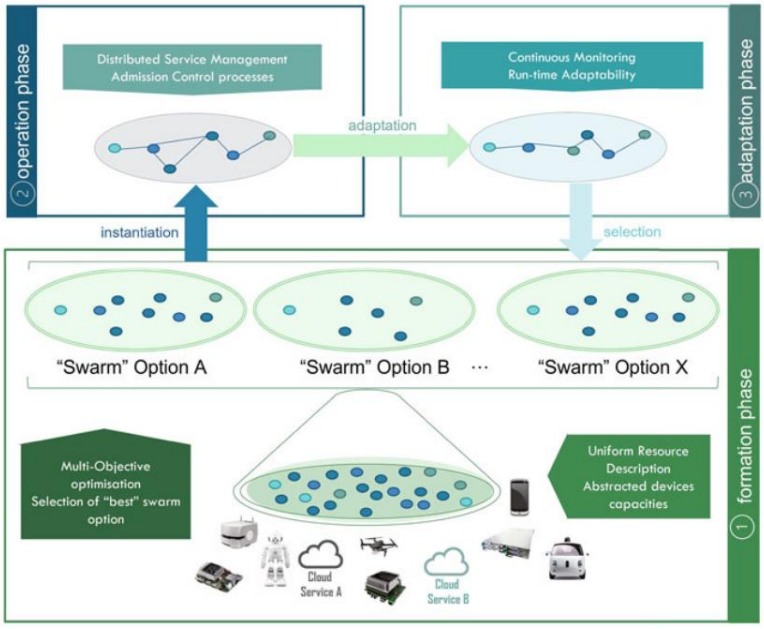

Figure 22: Lifecycle of a temporary distributed infrastructure

The control plane is also responsible of the admission of requests for executing services in the infrastructure. It should decide if the available resources are enough for the service, and to place the components in the adequate nodes. Furthermore, during the execution of the distributed application the overall behaviour will be observed to understand if the requirements are being satisfied (bandwidth, latency, etc.) and readapt the infrastructure accordingly.

5.1.1 State of the art

In the previous section we have presented the main concepts related to the design of a distributed architecture for simplifying the deployment and execution of applications in a set of heterogeneous nodes.

There are different proposals and implementations of architectures with similar objectives. This section provides a brief review.

5.1.1.1 ROS2

The initial version of the robot operating system (ROS) was originally conceived as a unified platform for developing single-robot applications. It provides a standardized suite of tools, libraries, and conventions designed to support the creation of a wide range of robotic systems.

ROS was not originally designed to support multi-robot systems, a capability that has become increasingly essential in recent years. This necessity drove the development of ROS 2.

ROS 2 was designed as a distributed system where nodes communicate directly with one another, leveraging the capabilities of the underlying Data Distribution Service (DDS) middleware. This decentralized architecture provides robust, efficient, and reliable communication for large-scale and distributed robotic applications.

ROS 2 design objectives are the following (Macenski, 2022):

- Distribution: Robotics problems are addressed using a distributed systems approach, where functionality is divided into independent components (e.g., device drivers, perception systems, control systems). These components operate in their own contexts and communicate explicitly, requiring a decentralized and secure setup.
- Abstraction: Communication between components is governed by well-defined interface specifications. These abstractions ensure data exchange semantics and promote interoperability by avoiding overfitting to specific components, enabling flexibility in substituting hardware or software.
- Asynchrony: Components communicate asynchronously in an event-based system. This allows applications to handle the differing time domains of physical devices and software components, each with its own data and command frequencies.
- Modularity: Emphasizing simplicity, modularity is implemented at all levels (e.g., APIs, message definitions, and tools). The software ecosystem is decentralized, organized into federated packages rather than a monolithic codebase.

To achieve these objectives, the software ecosystem is divided into three categories:

- Middleware: Abstracts the communication among components, including network APIs and message parsers.
- Algorithms: Offers commonly used robotics algorithms, such as perception, SLAM (Simultaneous Localization and Mapping), and planning.
- Developer Tools: Provides command-line and graphical tools for tasks like configuration, debugging, simulation, visualization, and logging. Additionally, it includes tools for source management, build processes, and distribution.

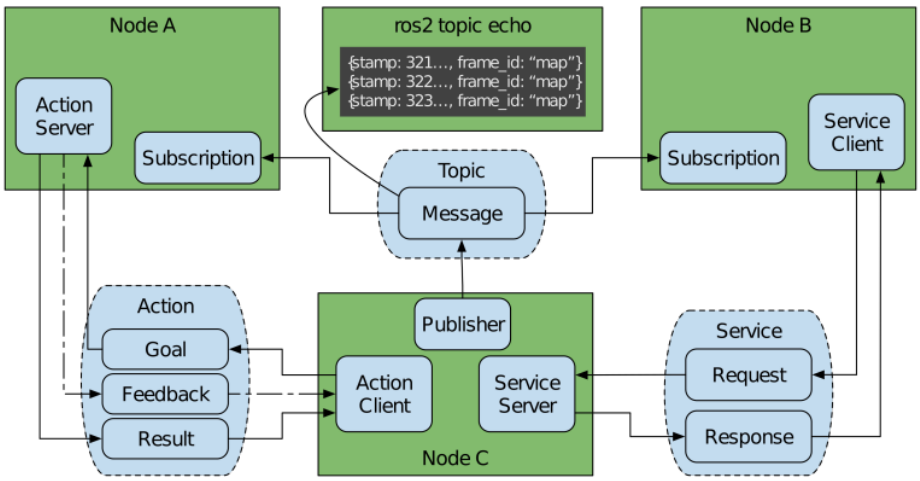

Figure 23: ROS 2 node interfaces: topics, services, and actions (Macenski, 2022)

The ROS 2 architecture is built on abstraction layers and modular design, distributed across decoupled packages to enhance flexibility and replaceability. Client libraries, allow developers to work with core communication APIs for various programming languages, enabling seamless distribution across processes, machines, and even cloud resources. These layers include an interface for common functionality and a middleware abstraction, supporting interchangeable middleware implementations for adaptability (including DDS and Zenoh support).

The ROS 2 communication APIs provide tools for various patterns, including topics, services, and actions, all organized under the concept of a node (Figure 23). Topics enable asynchronous publish-subscribe messaging with strongly typed interfaces, allowing flexible many-to-many communication and easy system introspection. For synchronous communication, services use a request-response model that ensures data association without blocking the client process. Actions, designed for long-running tasks, offer asynchronous goal-oriented interfaces with support for periodic feedback and cancellation. These patterns, along with additional tools like parameters and timers, allow developers to design organized and introspective robotic systems. Communication interfaces are defined using IDL and compiled into code for the supported programming languages, ensuring consistency and extensibility.

ROS 2 also offers architectural patterns like node lifecycle management, allowing system integrators to control states like Active or Inactive. Nodes can be flexibly assigned to processes during deployment, optimizing system resources and adapting to development needs.

Regarding security, ROS 2 employs the DDS-Security standard for secure communication, supplemented by SROS2 tools for simplified security management. Key security concepts include Authentication, which uses public key cryptography for identity verification; Access Control, enabling fine-grained policies to manage network participant interactions and communications; and Encryption, which safeguards data against eavesdropping and replay attacks through AES-GCM symmetric-key cryptography. SROS2 provides command-line utilities for managing digital signatures, configuring access policies, and setting up encryption, streamlining the implementation of these security measures.

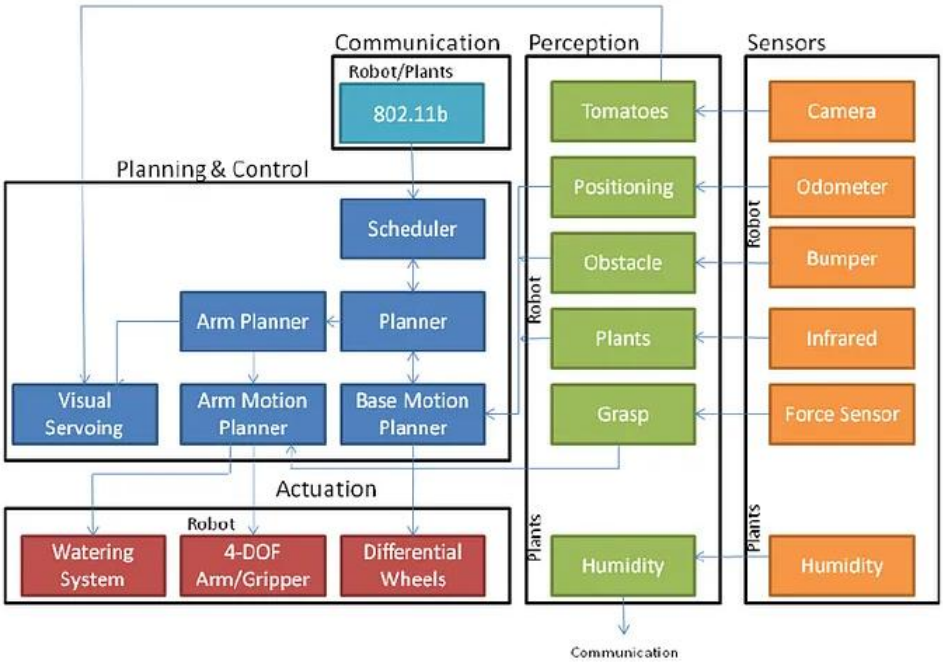

Figure 24: Example of ROS2 components in a robot application (Kutluca, 2020)

Figure 24 shows an example of a robot application. It presents how in the ROS 2 distributed architecture, components such as sensors, motion controllers, detection algorithms, artificial intelligence, and navigation algorithms function as nodes. These nodes communicate seamlessly in

a distributed environment using the DDS middleware, which facilitates a simple and flexible data exchange among them.

5.1.1.2 Kubernetes based architectures

On recent years, it has been significant efforts towards creating distributed platforms that could leverage on the cloud-edge-continuum paradigm. These efforts focus on bridging the gap between the cloud, edge and mist layers. Part of this effort is dedicated to adapting the open-source orchestration platform Kubernetes to this particular scenario.

Kubernetes is an open-source platform for orchestrating containers. Originally developed by Google and open sourced by them in 2014, now the coordination of the development falls over the project Cloud Native Computing Foundation (CNCF) of the Linux Foundation. Kubernetes allows for the deployment, automatic scaling, and management of containerized applications. These are applications bundle containers forming standalone, and portable executable packets that contain all the dependencies for the application to ensure the correct execution of the application. These applications are expected to run in a secure and isolated environment on top of the host operating system (Kubernetes, 2024).

The challenge with Kubernetes is that is heavily tailored to operate in datacentre environments. On these environments the hardware is homogeneous, and communication are carried in a wired and high-performance network. While this is expected on traditional cloud computing. The scenario for DistriMuSe is highly heterogeneous, on both, the hardware and the communication technologies being used.

5.1.1.2.1 KubeEdge

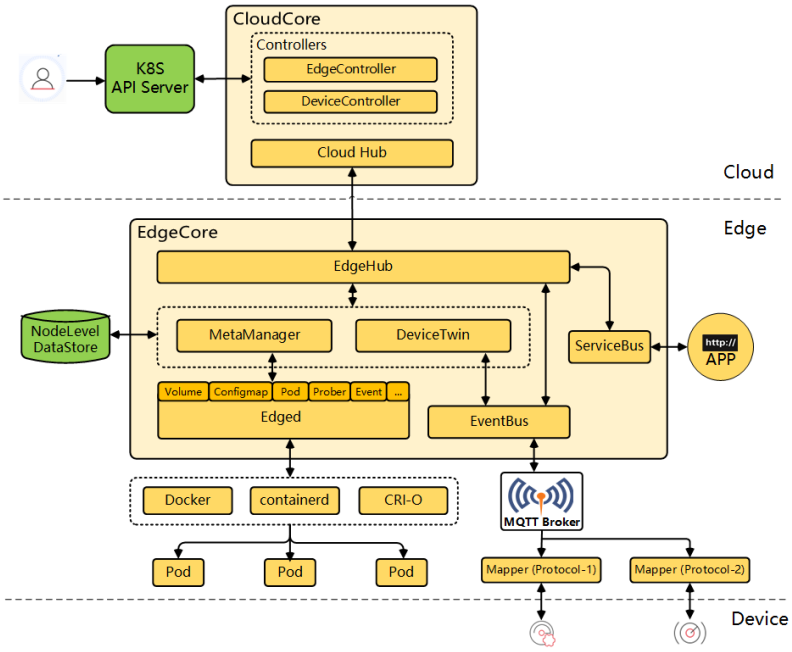

Figure 25: KubeEdge architecture (KubeEdge, 2024b)

This project, supported by the CNCF of the Linux Foundation, seeks to extend the orchestration of containerized application to the edge. It is based on Kubernetes and provides an architecture, shown in Figure 25. This architecture seeks to provide the adaptability to this scenario, as well as

introducing features to simplyfy the development of applications while maintaining the support to the massive ecosystem of application and tools of Kubernetes (KubeEdge, 2024a).

5.1.1.2.2 OpenYurt

OpenYurt, another project developed under the support of the CNCF, also provides an edge computing platform based on upstream Kubernetes, extending cloud-native capabilities to the edge.

Given the particularities of the edge environment, this platform implements a series of features to solve some of the inherent challenges. Due to the heterogeneity of the communication technologies used to connect the site edge with the cloud infrastructure (5G, WiFI, etc...), it is anticipated that the link may not be stable or, even, there may be occasional loss of connection. In this scenario, as the communication link cannot be considered reliable, and the performance cannot be foreseen, a certain type of autonomy is needed at the edge sites, so part of the time-critical process and coordination could be done local to the site.

OpenYurt enhances edge performance and availability by leveraging a strong autonomy capability of the edge sites introducing local caching and heartbeat proxy reporting mechanisms that allow continuous and reliable operation even in the event of a failure of the communication link with the cloud control plane (OpenYurt, 2024a).

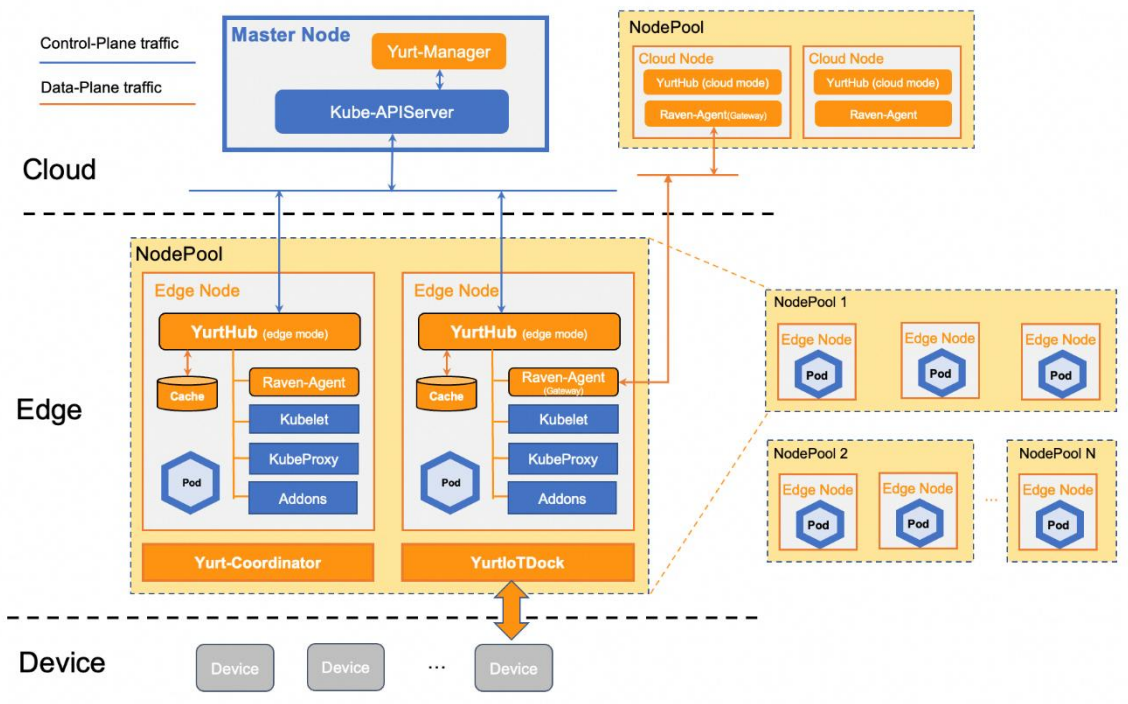

Figure 26: OpenYurt architecture (OpenYurt, 2024b)

Figure 26 depicts the architecture of OpenYurt. While the blue boxes represent the different Kubernetes components, the orange boxes represent the components implemented by the OpenYurt to implement the additional features and the adaptability to the edge environment provided on top of Kubernetes.

5.1.1.3 ROS with Kubernetes

There are different efforts to make ROS work in a Kubernetes environment, especially to leverage the support by Kubernetes of edge architectures to deploy ROS applications on a large-scale system. The objective is to abstract all computing devices, including the robot's on-board computers, edge servers/virtual machines (VMs), and cloud VMs, as a unified computing infrastructure.

5.1.1.3.1 KubeROS

KubeROS is a platform created to simplify the development and deployment of ROS 2 applications across the mist, edge and cloud environment. To achieve this, KubeROS leverages containerized ROS 2 applications and utilizes Kubernetes to deploy them on the appropriate hosts. KubeROS aims to provide automated scaling, fault tolerance, and resource optimization even in large scale deployments, for example, for the control of a large robot fleet.

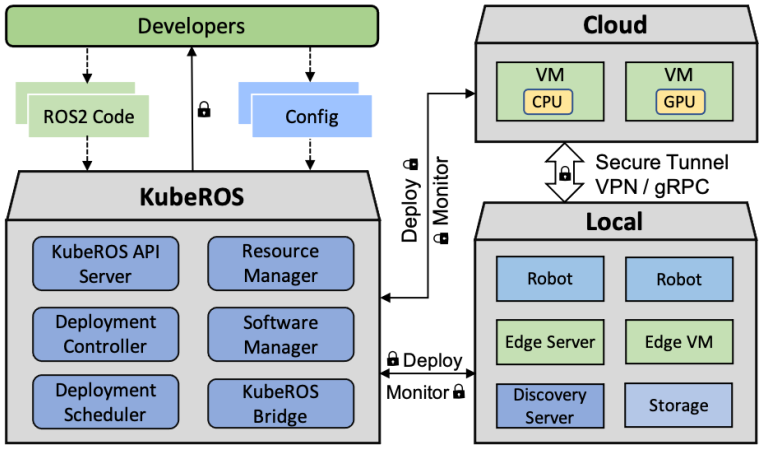

Figure 27: High-level overview of KubeROS system (Kuberos, 2024)

Figure 27 depicts the high-level architecture of the KubeROS system. The platform is comprised of the control plane, which includes the API server for handling API requests and system interactions, the scheduler for allocating ROS 2 applications across the nodes available in the platform, and the controller for monitoring and adjusting the system state. Additionally, the platform features a resource manager for the supervision of resource availability and provide system state information, and a software manager to manage the lifecycle of the software. With this architecture KubeRos aims to provide simplify the deployment of the ROS 2 Applications abstracting the hardware and the resource management complexity.

5.1.1.3.2 ROS with KubeEdge

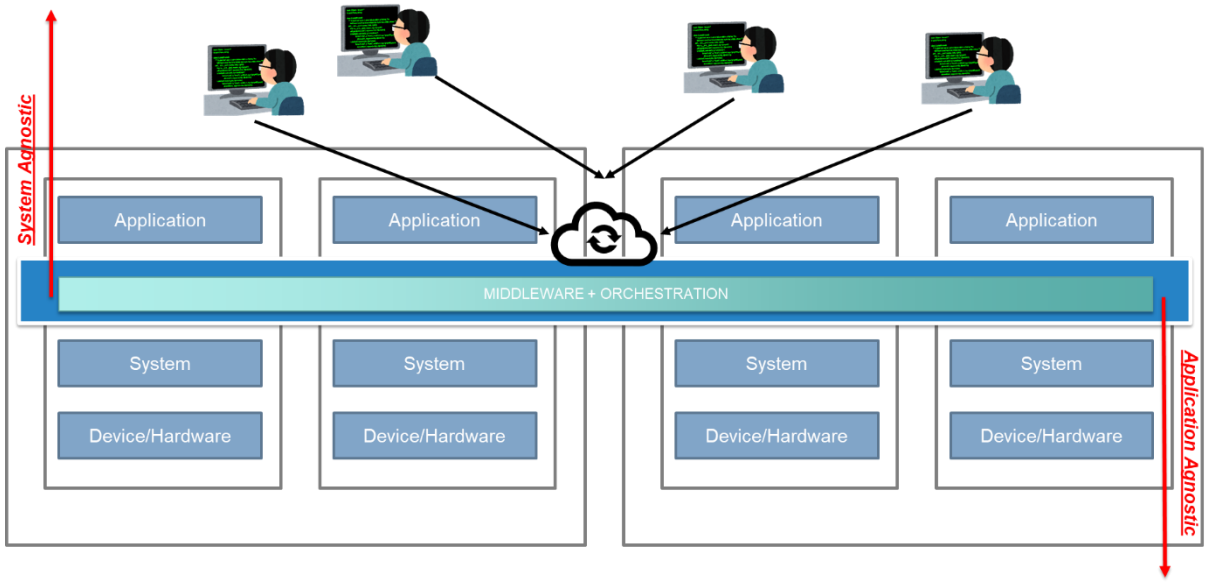

Figure 28: Kubernetes Robotic Edge Cluster System (Tomoya Fujita)

The integration of KubeEdge with ROS 2 provides a framework for building distributed robotics systems, as shown in Figure 28. KubeEdge enhances the flexibility of ROS 2 by enabling dynamic deployment of nodes across edge and cloud environments. Leveraging Kubernetes' orchestration capabilities, developers can optimize computation placement, ensuring efficient use of resources while maintaining system performance.

Security is a cornerstone of this architecture, with the combination of DDS-Security in ROS 2 and KubeEdge's secure communication protocols enabling zero-trust environments. Tools like SROS2 further enhance security by providing robust authentication, access control, and encryption, ensuring data integrity and privacy across the distributed system.

KubeEdge's support for application-agnostic network configuration allows ROS 2 nodes to communicate seamlessly, regardless of changes in network topology or deployment environments. This ensures that communication patterns such as topics, services, and actions remain unaffected, fostering a highly adaptable and resilient system.

By extending the capabilities of devices, KubeEdge allows ROS 2 nodes—representing sensors, actuators, or computational algorithms—to integrate with a diverse range of hardware. This integration expands the functional scope of the system, enabling more complex and versatile robotics applications.

Global observability is achieved through KubeEdge's monitoring and logging tools, which complement ROS 2's introspection capabilities. Together, they provide comprehensive visibility into node states, communication patterns, and overall system health, empowering operators to make informed decisions and maintain system reliability.

5.1.1.4 WasmCloud

WasmCloud is an open source CNCF project that focus on the creation and management of WebAssembly application components. WasmCloud allows to build, execute and orchestrate WebAssembly binaries. WebAssembly (abbreviated as wasm) is a standardized binary instruction

format that can be used as compilation target for a wide variety of languages, including, C, C++,C#, Python, Go, Rust and more. These compiled binaries can be executed within wasm-compatible runtimes or directly in browsers that support WebAssembly. Figure 29 shows the binary compilation and execution process.

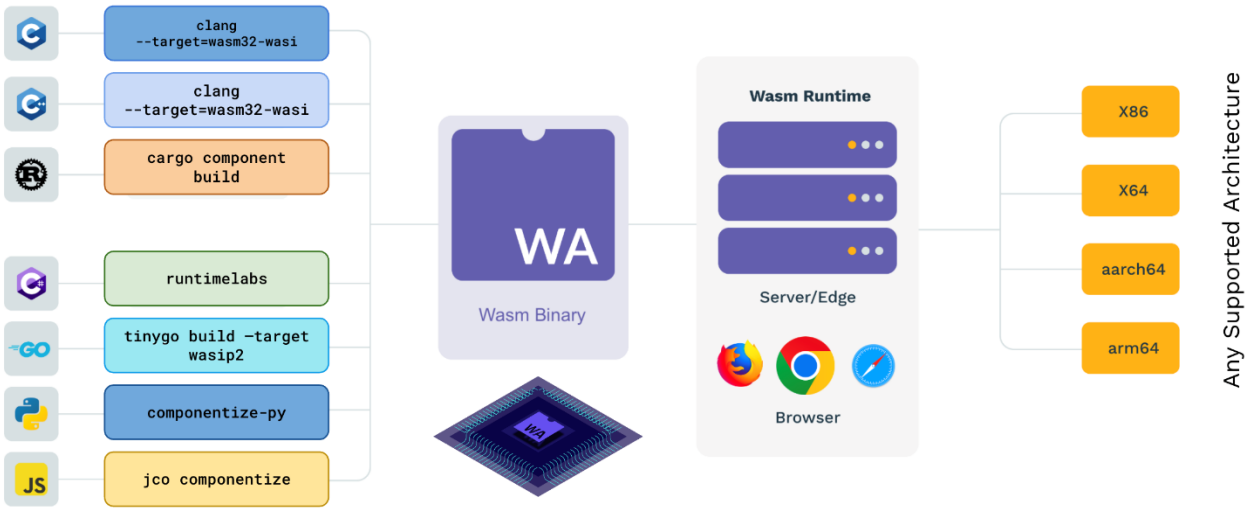

Figure 29: WebAssembly binaries compilation and execution (WasmCloud, 2024a)

The components are:

- **Portable**: The platform-independent design of wasm allows for the execution of the different components in any WebAssembly runtime, independently of the operating system or hardware underneath.
- **Interoperable**: After being converted to WebAssembly, a library written in one programming language can seamlessly integrate with and be utilized by code originally developed in another language.
- **Composable**: Multiple components can be linked and combine in more complex applications, allowing to reuse components
- **Reactive**: Components follow the principles of reactive programming. The components run when they are invoking by another entity

Additionally, the components are stateless, so no information about the state of the application is store in the component. In order to provide storage or any reusable service wasmCloud considers the use of providers. Providers are executables that run in the host and deliver common functionalities or capabilities, such as, a frontend process for accessing a key-value database like Redis or serving content over HTTP (WasmCloud, 2024c).

To provide connectivity between the nodes, wasmCloud introduces the concept of lattice. A lattice is a self-forming, self-healing mesh network that provides connectivity between the different entities (WasmCloud, 2024d).

Figure 30 illustrates the architectural design of wasmCloud, which incorporates the concepts of component, provider, host, and lattice.

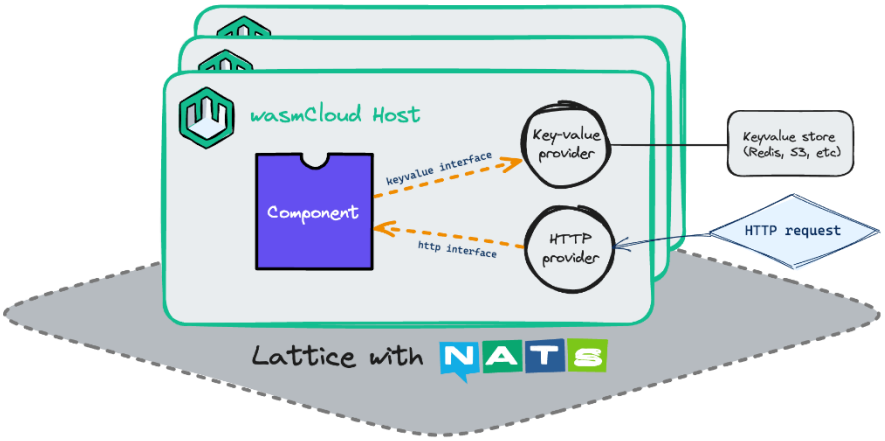

Figure 30: wasmCloud architecture (WasmCloud, 2024b)

5.1.1.5 Function as a Service (FaaS) architectures

Serverless computing is often associated with Function as a Service (FaaS). This novel cloud computing model rethinks how applications are developed, deployed, and scaled. This paradigm seeks allowing developers to concentrate on writing code, by abstracting the complexities of the management of servers or infrastructure. In the serverless model, applications are composed of independent functions designed to perform specific tasks in response to various events or triggers. Despite its name, "serverless" does not imply the absence of servers but rather the abstraction of their management. Cloud providers handle provisioning, scaling, and maintenance dynamically, enabling a more agile, scalable, and cost-effective approach to application development.

A defining feature of serverless computing is its event-driven architecture. Functions are triggered by events such as HTTP requests, database updates, or file uploads, promoting the creation of modular and loosely coupled components. This modularity enhances flexibility and maintainability in application design (Aluvalu, 2023).

Proposals such as Dyninka (Fortier, 2021) combine FaaS and dataflow programming to rapidly prototype FaaS-based distributed applications. The system consists of three key components: a DSL (Domain-Specific Language) to define dataflow applications as deployable microservices by generating the necessary code; a deployment module to package and deploy services in an infrastructure-agnostic way, interacting with container orchestration systems; and a container orchestration runtime to manage deployments, adapt to constraints, and provide service-level insights into deployment states.

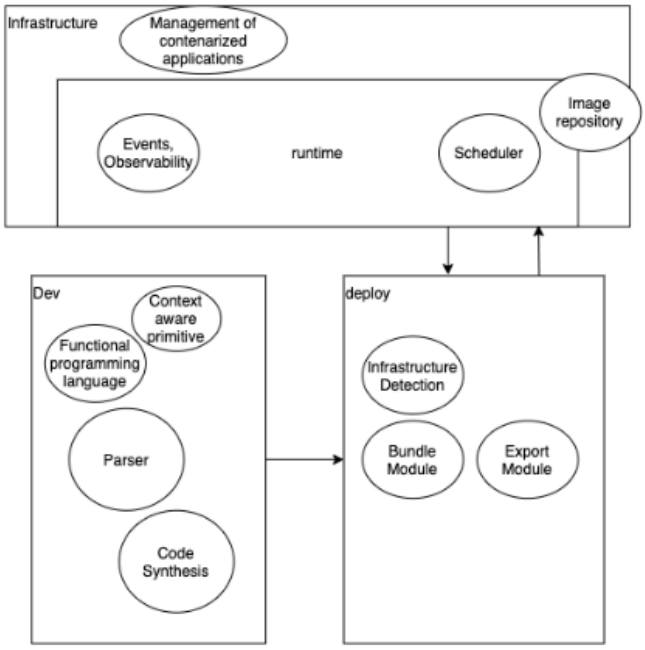

Figure 31: Dyninka architecture (Fortier, 2021)

5.1.1.5.1 Apache OpenWhisk

Apache OpenWhisk is a popular opensource platform for the deployment of event-driven, serverless, and stateless applications, which are referred to as "Actions". The event-driven paradigm enables these actions to be triggered by events, such as messages in queues, data from sensors, or other entities within the system. This approach allows the creation of pipelines of actions, where multiple actions can be chained to achieve the process of data, enabling more complex workflows.

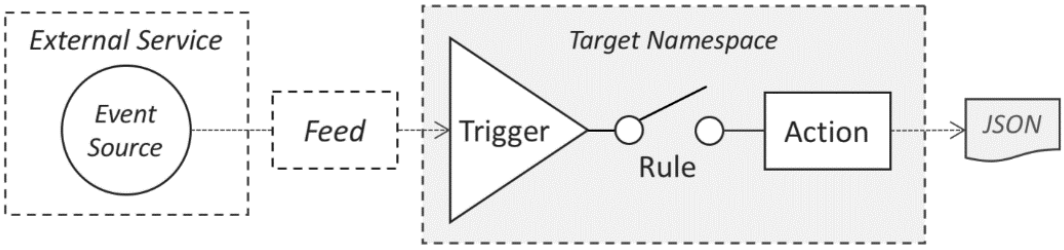

Figure 32: OpenWhisk event-driven programming model (OpenWhisk, 2024)

The Programming model is depicted in Figure 32, illustrating the architecture of an Apache OpenWhisk action deployment. The relevant element for an action deployment are the event sources, triggers, rules and actions. Event sources are entities that generate events, such as, sensors, or messages queues. These events are routed to the triggers, the triggers are named channel that filter the events coming from the different event sources and send the relevant event to the rules associated with the triggers and specific actions, after a desired event reaches the rule, the associated action will be invoked.

Regarding the development of these actions, the Apache OpenWhisk platform supports a variety of built-in runtimes, including .Net, Go, Java, JavaScript, PHP, Python, Ruby, and Swift. Additionally,

OpenWhisk also provides the possibility of creating new custom actions using Docker to support additional languages and libraries.

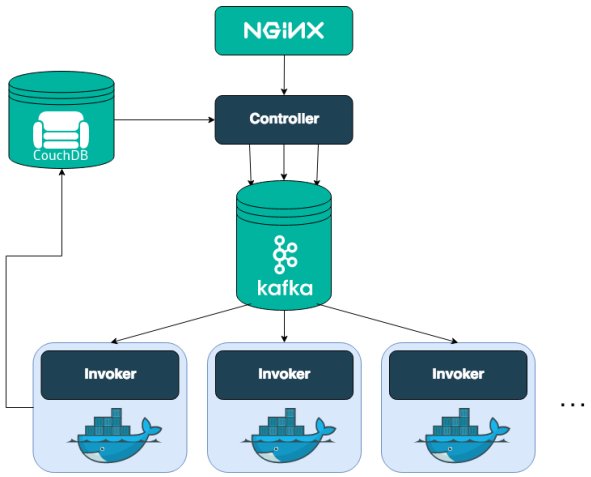

Figure 33: OpenWhisk high level architecture (OpenWhisk, 2024)

Figure 33 illustrates the high-level architecture of OpenWhisk. It leverages multiple open-source technologies, such as, Nginx, Kafka and Docker to provide an open-source serverless cloud.

Regarding the deployment of the platform, OpenWhisk supports a wide variety of deployment options suitable for both local and large-scale deployment on public cloud provider, such as, Google Cloud or Amazon Web Services. It supports a range of scenarios, including the deployment of a standalone instance, ideal for development, and larger deployments using Kubernetes and Helm for a large-scale deployment.

5.1.2 DistriMuSe distributed platform architecture

The wide range of requirements previously identified for the different use cases, involve the interoperation of a set of application running on top of geographically distributed heterogeneous devices and platforms. In this scenario, DistriMuSe aims to build an **architecture that enables developers to implement new services and applications utilizing various types of distributed algorithms without needing to adapt closely to the low-level specifics of the different supported environments.** To achieve this, the architecture must include a **middleware layer** that abstracts these details, allowing developers to design applications using a **Domain-Specific Language**. The architecture itself should handle the deployment of components and ensure the delivery of services to end users.

Developers should be abstracted from the complexities of various underlying systems, enabling them to focus on higher-level tasks. For perception abstraction, they should configure only relevant parameters and seamlessly receive data or events from sensors or combined sensor systems without worrying about individual sensor configurations. Communications abstraction ensures developers are not burdened with configuring low-level communication protocols or understanding addressing and routing details. With storage abstraction, developers can store information without concern for how or where it is maintained. Finally, computing abstraction allows developers to provide their algorithms as code or binaries without needing to adapt them to different architectures or operating systems.

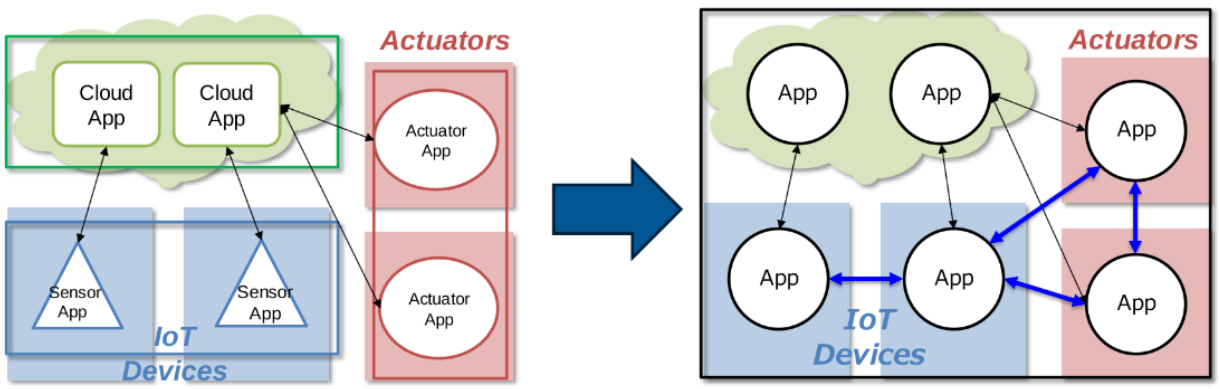

Figure 34: Transitioning from ad-hoc application designs to high-level architectural design

As shown in Figure 34, our objective is to deliver an abstract platform that conceals underlying complexities, simplifying the processes of design, development, and maintenance. This abstraction enhances versatility, enabling the creation of more sophisticated applications that can be easily relocated or adapted to different environments.

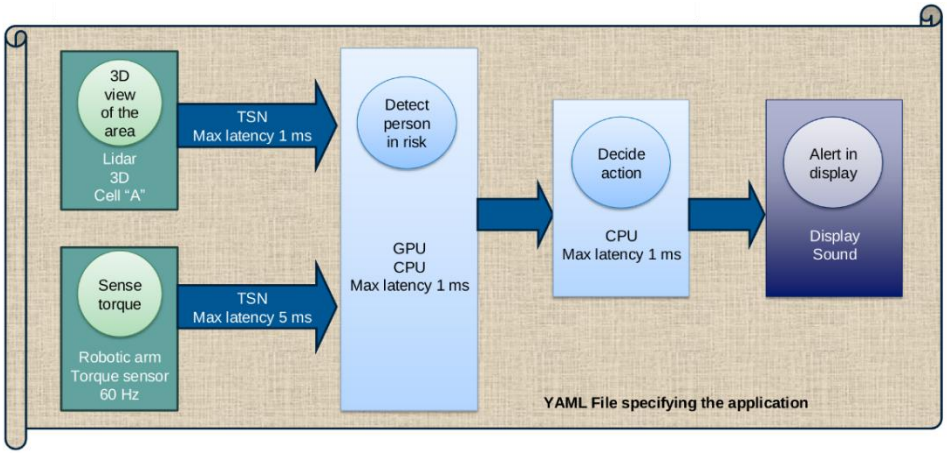

Figure 35: Representation of the implementation of an application using a high-level DSL

The first component required is a Domain-Specific Language (DSL) to define applications as high-level dataflow programs that manage the inputs and outputs of various components within the application. As depicted in Figure 35, application developers will be responsible for implementing the different components (represented as circles in the figure) as microservices (containers that can utilize various technologies, as discussed below). Developers will need to deploy prebuilt containers to an "image registry" or utilize previously built containers already available in the registry. Developers will also need to provide the configuration parameters for these microservices (shown as text inside the boxes), along with defining the flow of information between them (represented by the arrows).

Components and data flows may be subject to specific constraints. For example, a component designed to capture information from a Lidar sensor must be deployed on a node equipped with such a sensor, which might also need to be positioned in a precise location. Likewise, data flows could have strict latency requirements, necessitating the use of specific networking technologies and

configurations to guarantee timely data delivery. In these scenarios, developers must thoroughly understand the low-level details of the environment and explicitly specify these requirements when defining the application.

Once developers submit their applications, the DistriMuSe distributed platform architecture takes charge of matching application components to the characteristics of the available nodes and execute the application.

From this description, we can deduce the functionalities that should be supported by the DistriMuSe distributed platform:

- **Application Interpretation:**
  - It should interpret applications described using a DSL file.
- **Requirement Understanding:**
  - It must analyse and comprehend the specific requirements of each application.
- **Node Management:**
  - The platform should include tools and procedures to maintain accurate, up-to-date information about the characteristics, resources, and availability of each node.
- **Deployment Decision-Making:**
  - The platform must determine if an application can be deployed, evaluating whether the system's available resources meet the application's requirements. This decision should also include applying architecture and security policies.
- **Application Deployment:**
  - If deployment is feasible, the platform should handle the following:
    i. **Resource Allocation and Configuration:** Reserve and configure resources such as sensors, actuators, communication channels, computing power, and storage.
    ii. **Container Deployment:** Deploy the application's components as containers.
    iii. **Lifecycle Management:** Synchronize and manage the lifecycle of all deployed components to ensure proper operation.

Such functionalities are implemented by what we can call the **CONTROL PLANE** of the DistriMuSe distributed platform. The control plane is responsible for managing and orchestrating the overall operation, acting as the "brain" by handling decision-making and coordination tasks, in contrast to the **DATA PLANE**, which executes the actual operations like data forwarding or processing.

5.1.2.1 Control plane

Figure 36 provides a high-level diagram of the implementation of the DistriMuSe distributed platform.

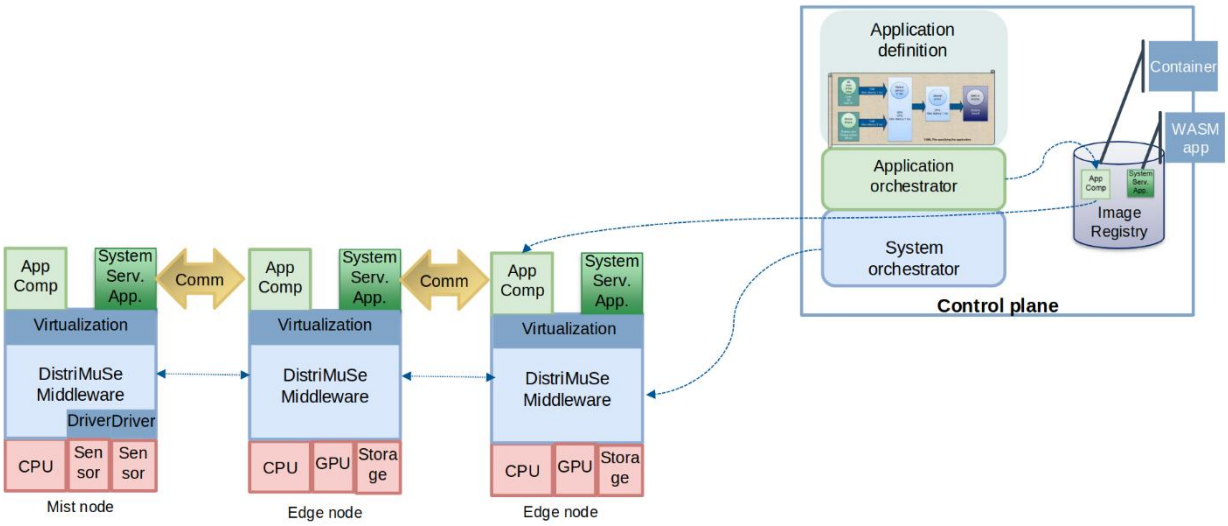

Figure 36: High level view of the DistriMuSe architecture

This Figure illustrates the different **components** we need to implement the **functionalities of the control plane.**

- **System orchestrator**: The resource manager of the system, responsible for collecting information about the presence of nodes, their features, and their availability. It interacts with the DistriMuSe middleware components running on each node to discover nodes, gather data on their features, and assess the availability of resources. The orchestrator uses a registry to store and manage this information, ensuring that the system has an up-to-date view of all available resources.

- **Application orchestrator:** Serves as the interface between the system and the applications. It receives application definitions, either in the form of a DSL file or through a higher-level development tool, and uses the system orchestrator services to deploy the components to the appropriate nodes. The orchestrator evaluates factors such as policies, required features, and available resources to decide the best location for deploying the components. It also interacts with the DistriMuSe middleware components running on the different nodes to configure elements of the data plane, including sensors, communication networks, and other operational resources.

- **Image registry:** Repository that stores components of applications as containers. It allows for the efficient management, versioning, and distribution of container images, which can be deployed across various nodes within a system. The registry acts as a central hub where prebuilt or custom containers are stored and retrieved for deployment in the appropriate environments.

- **DistriMuSe middleware:** Software running on each node that enables the services required by the DistriMuSe distributed platform. It provides functionalities for both the control plane and the data plane. In the **control plane**, it assists with tasks such as node discovery, resource management, notification, and managing the lifecycle of application components. In the **data plane**, it offers abstractions to the applications, including low-level details of hardware and devices, communication between application components, storage management, and more. This middleware plays a crucial role in bridging the various system layers and enabling seamless operation across the distributed platform.

DistriMuSe addresses scenarios involving a diverse array of devices, including advanced sensors and actuators, embedded devices, gateways, edge infrastructure, and cloud infrastructure. Based on the state-of-the-art approaches previously studied, devices can be orchestrated using either a **choreography pattern** or an **orchestrated pattern**. While a choreography pattern would result in a fully distributed system, the complexity of this approach, combined with the intricate scenarios considered in DistriMuSe, makes it impractical. Therefore, we have opted for an orchestrated pattern, where a central node or set of nodes implements the **system orchestrator**, **application orchestrator**, and **image registry**.

In line with Kubernetes terminology, this central node is referred to as the **master node**. Depending on the specific use case scenario, the master node can be deployed in the cloud, edge infrastructure, or even in a mist infrastructure. For example, scenarios for use case 1, involving cloud-based processing without stringent latency requirements, may leverage cloud infrastructures to deploy the master node in a centralized location. On the other hand, scenarios for use case 2 may not involve cloud infrastructures, and thus seem more suitable for the deployment of master nodes in edge infrastructures. However, strict latency constraints involved in use case 3 may need to deploy the master node at the mist layer. The master node may be exclusively dedicated to the control plane or may share responsibilities between the control and data planes, offering flexibility to adapt to different deployment scenarios.

Again, in alignment with the Kubernetes architecture, the nodes responsible for executing the various components of an application in DistriMuSe are referred to as **worker nodes**. These nodes are required to run the **DistriMuSe middleware**, which abstracts the underlying hardware and communication details for the application components. This abstraction simplifies the interaction between applications and the distributed infrastructure, as detailed in the upcoming “Data Plane” section. DistriMuSe identifies three distinct types of worker nodes:

- **Mist/Sensor:** This level is closest to the sensors and actuators. It focuses on collecting sensor data but offers limited processing capabilities due to constraints on power and computational resources.
- **Edge:** Positioned near the mist/sensor level, this layer operates on gateways and edge servers. It provides a low-latency computing environment suitable for tasks such as data fusion, data aggregation, and other intermediate data processing activities.
- **Cloud:** The final level consists of nodes located in data centres, either owned by a partner or provided by a cloud computing service. These nodes are designed to handle the most computationally intensive tasks, leveraging their high processing power and scalability.

5.1.2.2 Data plane

The **data plane** is the part of the DistriMuSe distributed platform architecture responsible for the actual processing, forwarding, and execution of the distributed applications. It directly handles the capture, storage, movement and transformation of data as it flows through the system, often interacting closely with sensors, actuators and devices.

By separating the data plane and the control plane, we can achieve greater scalability, flexibility, and reliability, as each plane focuses on distinct aspects of system functionality.

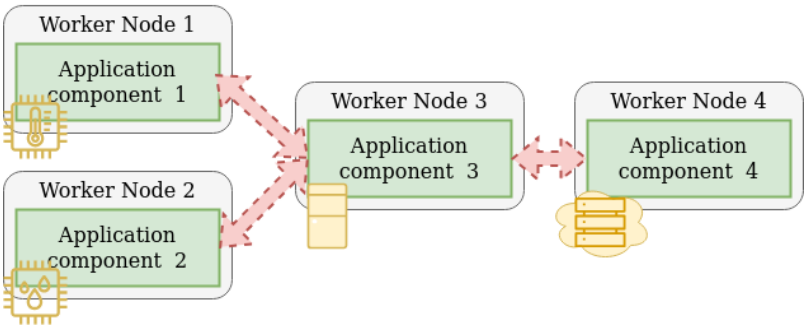

Figure 37: Distributed application from the point of view of the data plane

Figure 37 illustrates a view of a distributed application from the perspective of the **data plane**. In this generic application example, the various application components are executed on different worker nodes, with the data plane abstracting the underlying details of the nodes from the application components. These components exchange flows of information without needing to be aware of the specifics of the networking infrastructure or whether the other components are running on the same node or a different one. This abstraction simplifies application development and operation within the distributed platform.

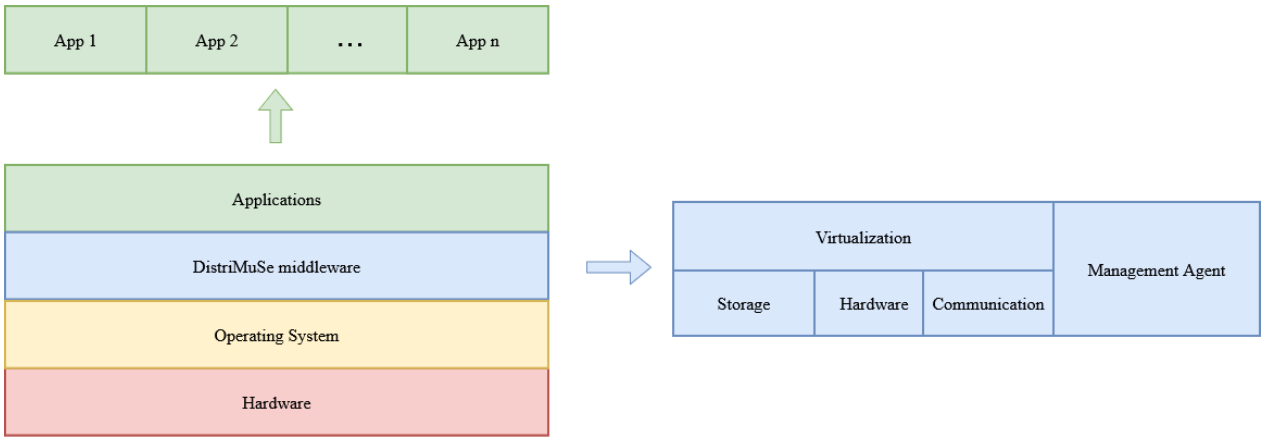

Figure 38: Internal architecture of a worker node

In the context of the worker node architecture, the **DistriMuSe middleware** functions as a management and abstraction layer positioned between the node's operating system and the various application components. This layer facilitates hardware abstraction, provides control over the execution of application components, and enables resource monitoring within the system.

As stated before, the middleware provides applications with different abstractions such as:

- **Hardware Abstraction**: A virtualization runtime ensures that the same application component can run across different types of nodes and architectures. This layer also provides standardized interfaces to access specific hardware, such as accelerators, sensors, and actuators, ensuring uniform functionality regardless of the underlying hardware.
- **Storage Abstraction**: Applications can utilize permanent storage without needing to know the physical location of the stored information. This abstraction simplifies data management and ensures seamless access across the distributed platform.

- **Communication Abstraction**: Application components communicate by exchanging messages without requiring knowledge of their physical locations or the low-level details of the communication infrastructure. This enables a highly flexible and transparent communication mechanism within the platform.

As noted earlier, the control plane relies on an agent that interacts with the system and application orchestrators to facilitate key tasks. These tasks include node discovery, resource management, sending notifications, and managing the lifecycle of application components.

5.2 Hardware platforms

Intelligent systems that are considered in the use cases of the DistriMuSe project rely on the Cloud, Edge, and Mist Computing paradigms to shape a distributed architecture composed of multiple heterogenous and decentralized hardware nodes according to the operated workflows. Indeed, the variety of demonstrators and pilots show a wide range of requirements in terms of computational power, storage capabilities or communication interfaces. Therefore, a proper dimensioning of the needed resources in the available hardware infrastructure is key to provide the demanded services.

5.2.1 State of the art

The use of *cloudy* architectures is the current preferred solution in the industry to leverage flexible and efficient computing platforms. From traditional Cloud Computing to modern Edge Computing solutions, they are based on the transparency principle to scale up services according to the dynamic service workloads, and they enable ubiquitous computing capabilities with minimal interaction for global operation coverage.

- **Cloud layer**: it is based on robust and reliable clusters of powerful servers that provide elastic and on-demand service operation located in remote areas, thus enabling vertical scalability. Cloud service providers (CSPs) offer different service accessibility tiers (SaaS, PaaS, IaaS, etc.) depending on the resource granularity. Besides, they may constitute public datacentres for generic services, private ones for custom enterprise solutions, or hybrid approaches to share data between multiple communities. Cloud nodes are a broad range of devices, from generic multi-core CPU and GPU servers to specialized accelerator for high-demanding AI operations (TPUs and NPUs), including FPGAs for specific-purpose optimizations and efficient storage systems. Vendors, like Google Cloud, Amazon Web Services (AWS) or Microsoft Azure, integrate a disaggregated hardware infrastructure to decouple computing, networking and storage functionalities, to enhance the scalability and utilization of the overall system, while reducing costs and increasing availability degree.

- **Edge layer**: it brings the computations closer to the data sources, enabling real-time processing, reducing latency, and minimizing the bandwidth usage in the core network. Additionally, we achieve horizontal offloading along different geographic areas in context-aware situations, in keeping privacy issues safe. The Edge supports advanced capabilities for dynamic load balancing and customized user profiling for that enhance responsiveness and autonomy of the system. Regarding the infrastructure, it includes a very diverse collection of devices to meet the specific constraints and requirements of such environment, from tiny resource-constrained, battery-powered microcontrollers to customized high-performance servers for intensive time-critical operations. This way, it leverages sophisticated applications that demand collaborative, multi-modal and intelligent solutions,

for instance continuous health monitoring, situational awareness for smart driving and safe human-robot cooperation.

The selection of suitable resources to execute the services depends on the nature of the application, as well as the required computational power to provide a suitable quality of service. Especially, in AI applications which may require specialized hardware for inference, including light AI-based workloads that may be run directly by the microprocessor or microcomputers.. As well as the increasing complexity of the applications and the need to perform intensive operations in the nodes while providing different concurrent services, make multi-core platforms an optimal choice for the implementation of the nodes. In addition, a proper dimensioning of every kind of node is essential to provide a balanced solution in terms of the SWaP-C (Size, Weight, Power and Cost) principles.

5.2.2 DistriMuSe distributed infrastructure

The proposed distributed platform for DistriMuSe draws upon all existing layers to leverage the Cloud-Edge Continuum concept. The Continuum Computing unifies all layers into a single integrated framework that ensures seamless interaction for dynamic workload balancing, resource allocation and autonomous operation. In fact, the proposed platform combines heterogeneous hardware to handle different AI/ML tasks, real-time analytics and data aggregation and fusion, it customizes each setup configuration to optimize a modular architecture, and it integrates Cloud and Edge resources in a fluid pool that must support similar-design applications.

- **Cloud level infrastructure:** it is composed of private Cloud instances equipped with large-scale AI solutions that can process large amounts of data in a centralized hub. Either for training machine learning models from the gathered data from sensors or aggregating, fusing, delivering analytics and inference results, or even just persisting the information in storage systems. Specifically, Cloud nodes include a wide variety of computing resources from self-managed or third-party providers that are deployed on-demand to adjust to the dynamic workload and are flexible enough to handle high-intensive AI applications which require specialized hardware, like GPUs, TPUs or NPUs.

- **Edge level infrastructure (specialized AI hardware):** it includes an heterogenous group of nodes dedicated to special low latency processing with location-awareness capabilities that bridges all layers of the architecture (Fog), and sensing, actuating and very lightweight processing that interacts with the humans' environment and manages fault tolerance resiliency (Mist). Besides, the modularity of the hardware platform is key to easily adapt it to new scenarios, allowing to extend their capabilities with the support of pluggable modules daughterboards/shields, new communication technologies or extra computational capabilities. In addition, Edge nodes will support OS functionalities to launch concurrent and virtualized services that give the platform flexibility enough to play a different role depending on the specific needs of the application scenario.

- **Fog level infrastructure**: it has two main tasks, offering time-critical computing and connecting access networks. First, it provides geo-distributed computational resources to handle latency-sensitive operations, empowering also mobility and load balancing through the effective utilization of near resources. Second, it acts as an intermediary to facilitate communication between different technology networks, such as Wi-Fi, 5G, V2X or LPWAN. Indeed, gateway devices play a pivotal role in enabling smooth data transmission across devices and layers, maintaining a reliable and interoperable connectivity in diverse and dynamic conditions. In this sense, we propose to develop a hardware platform that supports

both high-intensive computing for critical AI applications and multi-communication interfaces for the simultaneous existence of different access networking. As in the case of the Portenta X8 + Max Carrier Board, a set from the professional line of Arduino that plugs modules into a carrier board, i.e. a 4 x ARM Cortex A53 microprocessor with 2 GB of RAM and 16 GB of eMMC, whereas the carrier board includes LoRa and cellular transceivers and exposes the following set of interfaces: 10/100/1000 Ethernet, CAN, RS232/422/485 and USB; providing also a mPCIe connector for connecting other modules (like a TPU or extra storage).

Figure 39: Portenta Max Carrier

- **Mist level infrastructure** (also known as IoT layer): it includes all devices where data is generated from and consumed by, but also intermediate nodes that do some lightweight processing to filter, aggregate or inference information. The Mist is intended to maximize the exploitation of resources to ensure the desired quality of service in a constrained environment, at the same time it guarantees a high availability degree enabling autonomous operation of the local network. The set of devices in the Mist layer consists of a wide range of sensors and actuators that interact with the environment of each use case, as well as microcontroller and microcomputers that have the capacity to launch tiny tasks, and even lightweight AI algorithms. On the one hand, there are any kind of sensor, wearable or robots that interact with the environment to produce raw data, such as foot-mounted IMU sensors which provide acceleration, angular speed and magnetic field measurements for the establishment of motion patterns or daily routines, to detect possible deviations caused by possible patient cognitive problems, or Self-Guided Vehicles (SGVs) that assist workers in logistics operations in the industry. On the other hand, we can find compact processing devices that are able to run sophisticated, yet lightweight, AI models to fuse data and extract meaningful information directly at the local environment, for instance NVIDIA Jetson Xavier to enable on-board monitoring of the driver's interactions and its real-time analysis to warn about dangerous situations.

Figure 40: Foot-mounted IMU sensors.

5.3 Communications

The DistriMuSe distributed platform, like any other distributed system, heavily depends on robust communication mechanisms to facilitate the exchange of information between nodes. These mechanisms are used for ensuring seamless coordination, data transfer, and synchronization across the distributed components.

Thus, communication protocols are essential for both the control plane and the data plane. In the control plane, these protocols facilitate coordination among orchestrators and nodes, ensuring management and resource allocation. In the data plane, they enable the flow of information and synchronization among application components.

The middleware layer includes communication protocols, marshalling of data, naming protocols, security protocols, and scaling mechanisms like replication and caching. These protocols provide essential services that enable various applications to communicate and interact effectively, regardless of their specific requirements or architecture (Tanenbaum, 2021).

The different aspects refer to the varying needs of individual applications, which may require customized or specialized protocols for their unique purposes. Middleware bridges these differences by offering a set of standardized services that can be reused across diverse applications.

Thanks to the middleware, developers can focus their efforts on defining application-specific protocols for communication, rather than dealing with low-level communication details. With the support of standardized protocols provided by the middleware, developers can often simplify their tasks, sometimes only needing to submit information when required, leaving the complexities of communication, data handling, and security to the middleware itself (Figure 41).

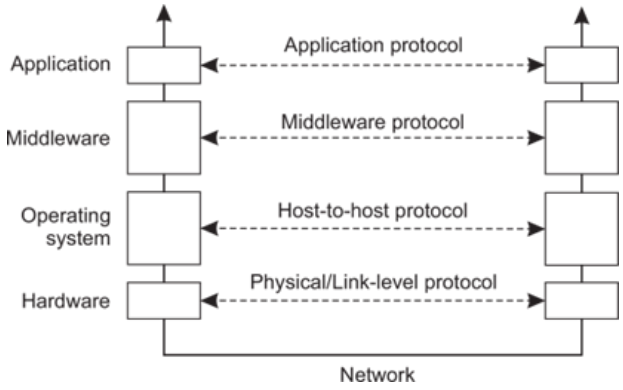

Figure 41: Communication layers in a distributed application (Tanenbaum, 2021)

5.3.1 State of the art

Middleware communications generally do not differentiate between data plane and control plane protocols. However, some protocols are specifically designed for the control plane. Accordingly, this state-of-the-art review is organized into two sections: the first examines the generic exchange of information applicable to both planes, while the second delves into control plane-specific protocols, which address tasks like coordination, management, orchestration, and control.

5.3.1.1 Middleware communications

Middleware communications can be categorized based on various criteria. For instance, whether the middleware ensures the delivery of information even when the receiver is not actively connected.

- **Transient communication**: Communication server discards message when it cannot be delivered at the next server, or at the receiver.
- **Persistent communication**: A message is stored at a communication server as long as it takes to deliver it. In this case, messages can be stored at the request of the submission, at the delivery or after request processing.

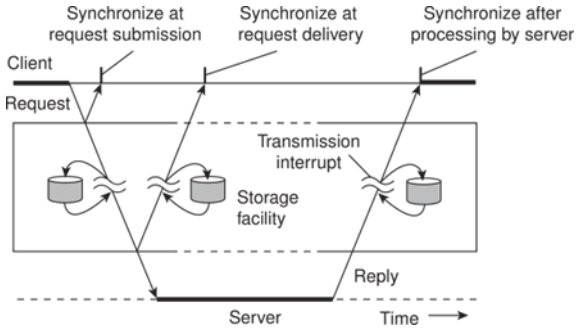

Figure 42: Persistent communications (Tanenbaum, 2021)

An alternative way of classification could be based on the synchronization strategy:

- **Asynchronous communication**: A sender continues operating immediately after it has submitted its message for transmission. The message is (temporarily) stored immediately by the middleware upon submission.
- **Synchronous communication**: The sender is blocked until its request is known to be accepted.

By combining these types of communication, we can derive a higher-level classification of middleware communications:

- **Client/server:** Generally based on a model of transient synchronous communication, where client and server must be active at the time of communication. In this scenario, clients issue a request and block until it they receive the reply. Servers essentially wait for incoming requests and subsequently process them. Clients cannot do anything while they wait, and failures must be handled immediately.
- **Messaging oriented:** Based on a model of persistent asynchronous communication, where processes send each other messages, which are queued. Senders don't wait for immediate reply, so they can do other functions, being the middleware layer the responsible of ensuring fault tolerance.

Both models can be used by developers to connect distributed applications. Nevertheless, they still would need to implement a large amount of logic to allow the exchange of information. To avoid this, Birrell and Nelson introduced the **Remote Procedure Call** or RPC concept in 1984. RPC enables programs to execute procedures that reside on remote machines as if they were local. When a process on machine A invokes a procedure on machine B, the process on A is paused while the

procedure runs on B. Data can be transferred between the two processes through procedure parameters, with results being returned once the execution is complete. This approach abstracts away the complexities of message passing, providing a seamless interface for the programmer, who does not need to manage communication details explicitly.

The middleware that implements RPC abstractions must manage significant complexities. One challenge arises from the fact that the calling and called procedures operate on different machines, each with its own address space, which complicates their interaction. Additionally, transferring parameters and results between these machines can be difficult, particularly when the machines differ in architecture or system configurations. Another critical issue is fault tolerance.

This model has also been applied to object-oriented programming. In this case, instead of just calling remote functions, object based remote invocations allowed clients to interact with remote objects as if they were local, using the same object-oriented concepts (such as inheritance, polymorphism, and encapsulation) that were prevalent in object-oriented programming languages. A popular implementation is the Common Object Request Broker Architecture (CORBA) standard, which was published in 1991.

Despite the advantages of Remote Procedure Calls and Remote Object Invocations, those mechanisms are not always appropriate because of their synchronous nature and the problems caused by link failures. **Message-oriented communication** is a good alternative to build **robust distributed systems.**

Many distributed systems and applications are built using the basic message-oriented model provided by the transport layer (sockets), but usually it is necessary a middleware that makes network programming easier.

ZeroMQ (Hintjens, 2013) provides enhancements to regular sockets for the different patterns that most of the distributed applications follow. First, ZeroMQ supports the one-to-one, many-to-one (a server can listen in multiple ports using a single blocking receive operation), and one-to-many (multicast) operations. Communications are asynchronous, as senders will normally continue after submitting a message.

ZeroMQ supports request-reply, publish-subscribe, and pipeline communication patterns:

- Request-reply pattern: Commonly used in traditional client-server communication, such as in remote procedure calls. The client application uses a request socket to send a request message to the server, expecting a corresponding response. The server, in turn, uses a reply socket to send the appropriate response back to the client. This simple communication model ensures that the client waits for a reply before proceeding with further actions.

- Publish-subscribe pattern: Clients subscribe to specific messages that are published by servers. Only the messages a client has subscribed to will be transmitted to them. If a server publishes messages that no client is subscribed to, those messages will be discarded and effectively lost. This pattern is useful for scenarios where multiple clients are interested in receiving updates, but not all messages are relevant to every client.

- Pipeline pattern: A process generates results and pushes them out, assuming that other processes are available to pull in those results. The process generating the information does not care which specific process consumes them. Similarly, processes that pull results from multiple sources will do so from the first available producer. The primary goal is pushing

results through a series of processes as quickly as possible, ensuring continuous processing without idle time.

A more advanced concept is **message-queuing systems**, also known as **message-oriented middleware (MOM)**. These systems provide intermediate-term storage for messages, allowing communication between processes without requiring the sender or receiver to be active at the same time. This decouples the components, enabling asynchronous message delivery and ensuring that messages are stored temporarily until the recipient is ready to process them. The asynchronous persistent communication is achieved through the support of **middleware-level queues**, which correspond to buffers at communication servers.

Messages can contain any type of data. From the middleware's perspective, the key requirement is that messages are correctly addressed. In practice, this is achieved by assigning a unique system-wide name to the destination queue. This way, the basic interface offered to applications can be simple, requiring only a few operations:

- PUT: Append a message to a specified queue.
- GET: Block until the specified queue is nonempty and remove the first message.
- POLL: Check a specified queue for messages and remove the first.
- NOTIFY: Install a handler to be called when a message is put into the specified queue.

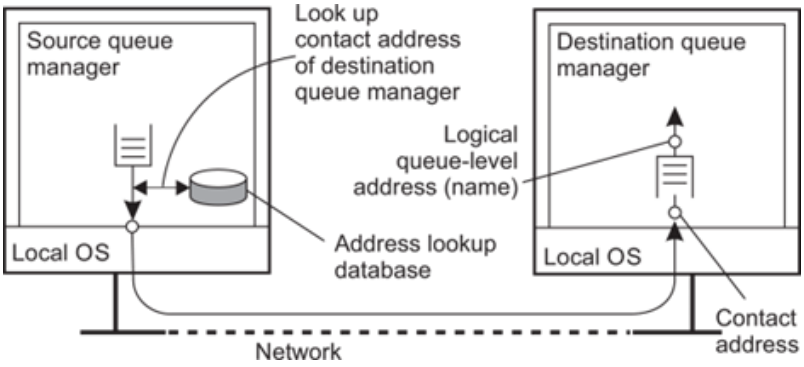

Figure 43: Routing in a queue manager by matching queue names to lower-level protocol mechanisms (Tanenbaum, 2021)

Queues are overseen by queue managers. An application can only place messages into a local queue, and messages can only be retrieved by extracting them from a local queue. As a result, queue managers are responsible for routing messages between queues (Figure 43).

When a queue manager knows the lower-level address of the destination queue, it can directly contact the server to transfer messages. This requires that each queue manager's contact address is known to all other managers. However, this approach can become problematic in large-scale message-queuing systems due to scalability concerns. To address this, specialized queue managers functioning as routers, are used to forward messages to other queue managers. This allows the message-queuing system to gradually form a comprehensive, application-level overlay network. In such systems, only a few routers need to be aware of the entire network topology. The source queue manager, for example, only needs to know to which adjacent router should send a message to for a given destination queue. The router in turn, only needs to know its adjacent routers to forward the

message accordingly, and so on. While name-to-address mappings are still needed for all queue managers, including routers, the resulting routing tables are much smaller and easier to manage.

Message-queuing systems rely on a shared messaging protocol, where all applications adhere to a common message format. This ensures that there is consistency in both the structure and data representation of the messages exchanged between different applications.

However, it is possible to add an application-level gateway, known as **message broker**, that can convert incoming messages to the format understood by the destination application. Such message can be used to enable communication between different applications, services, or systems. It serves as an intermediary, managing, routing, and translating messages to ensure smooth and efficient data exchange across diverse platforms. If the cases where we have a **centralized message broker,** it can be used also for routing, directing messages to the correct destinations according to predefined rules or patterns, simplifying the routing problem compared to pure distributed queue-manager architecture.

With such brokered or unbrokered queue systems we need to resolve a name to an address where the information should be transferred. This brings up the **Information-centric networking** or **ICN** paradigm, and one of its implementations the **named-data networking (NDN).** It proposes a shift from the actual host-centric IP based approach of the actual Internet into a Content-Centric Networking (CCN) paradigm (Zhang, 2014). In this new architecture, the data is accessed without prior knowledge of its physical location or the identity of host containing it. Instead, each node that seeks to retrieve a given data element, sends a special type of packet to the network, an Interest packet, to notify to the network its desire. The rest of the nodes should retrieve the data element requested if the given data element is available on the contacted node. On other case, the node should, forward the Interest to the rest of nodes in the network. After the Interest packet reaches a node that has the data element, this node sends a Data packet through the inverse path taken by the Interest packet. This Data packet contents the information requested and additional metadata, e.g. content type and structure. Figure 44 depicts the packet format for both packet types.

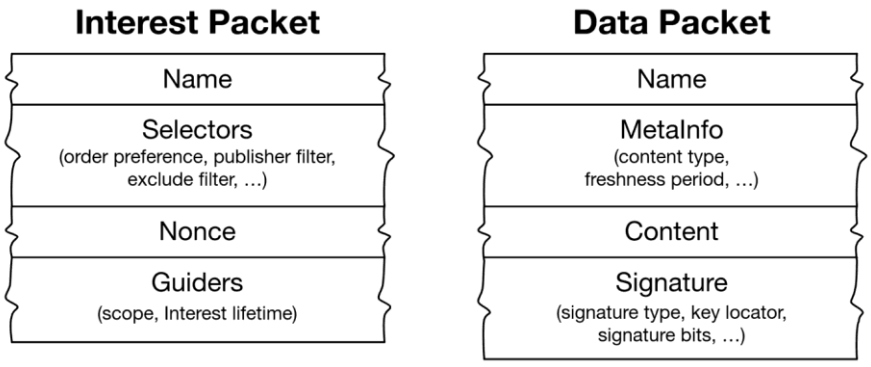

Figure 44: Packets in the NDN Architecture

DistriMuSe must accommodate a diverse range of use cases, some of which have stringent real-time requirements that need to be met by the underlying communication technology as well. In this aspect, the communication protocol to be used has a major impact in the performance of the overall system. These characteristics are stated in some published studies (Barzegaran, 2021), and the possibility of implementing Time Sensitive Networks (TSN) as such communication method it also addresses.

TSN is a collection of standards developed by the IEEE 802.1 working group which describes data transmission mechanisms for time-sensitive communication over deterministic Ethernet networks. TSN enables real-time communications with mechanisms that ensure bounded latency, frame pre-emption, scheduled priority-driven traffic and reliability via redundant transmission, making TSN a key factor in an industry growing into greater data flow and processing.

The previously mentioned study illustrates the benefits of including TSN in distributed platforms with the utilization of Gate Control Lists (GCLs) and Time Aware Shapers (TAS) to optimize the data flow. Those mechanisms allow assigning priority to frames, and time-multiplex the delivery of different priorities. Because of the topology of distributed platforms, mixed-critically applications will be executed at the network nodes, and ensuring enough bandwidth for each priority level, prevents the starvation of any priority flow, and therefore guarantees that all applications meet their deadlines even approaching real-time capabilities when needed.

Other studies (Navaratman, 1987), remark the necessity in distributed platforms of ensuring data atomicity (all nodes receive the messages or none of them) and that all nodes receive the information sent from a source in the correct order, i.e., data coherence. At the time of this study, TSN was not widely available yet, and it was not considered in its proposed solution, but TSN can improve the settling of these needs more efficiently and solve most of them.

In Best Effort Ethernet, information can get lost or delayed in situations where the network is highly congested, and therefore atomicity cannot be guaranteed, or it is possible that encounters delay that lower the overall system efficiency. TAS and GCL mechanisms combined with transmission reliability of TSN provide mechanisms for ensuring the reception of data by all nodes in the system with bounded latencies, allowing nodes to get information in time. Also, clock synchronization eases the ability of the system to coordinate the nodes, and to achieve data coherence between the nodes in the system.

With this background, the implementation of real-time application-oriented communication technologies is very beneficial for distributed platforms, and the TSN example used is no exception. In addition, its traffic division allows using the same network for data and control traffic of the distributed platform, simplifying the network architecture and facilitating the monitoring and control of the whole.

5.3.1.2 Control plane communications

The control plane of the DistriMuSe distributed platform will need to gather information and manage the various nodes within the system.

There are different standards with this objective, for example the TR-369, also known as the User Services Platform (USP), is a standardized remote device management protocol. It offers management and monitoring functions for a wide range of connected devices, including access gateways, WiFi routers, telephones, IoT devices, and virtual services like containerized applications. Especially the IoT device management and the edge gateway device management are important in the context of DistriMuSe.

The TR-369 represents a significant advancement in managing networked devices by optimizing the separation of the data and control planes. In TR-369 the data plane is dedicated to ensuring seamless data flow. Leveraging techniques like Quality of Service (QoS) and protocols such as HTTP/2 and MQTT, the data plane guarantees reliable service delivery. Additionally, the integration of edge computing capabilities within the data plane helps minimize latency and improve responsiveness for time-sensitive applications. The control plane, on the other hand, is focused on

the management and control of devices. It enables the configuration, monitoring, and management of networked devices using a standardized framework. The control plane is responsible for communicating commands, retrieving status information, and managing the overall operation of devices. This standardized approach fosters interoperability among diverse device types, enabling seamless configuration and real-time adjustments based on network conditions. By separating these two planes, USP enhances scalability, resilience, and simplicity. This separation allows for independent management of each plane without compromising overall performance.

5.3.2 Data plane DistriMuSe communications

In our review of the state of the art, we examined the key concepts related to middleware communications. The DistriMuSe distributed platform is designed to support a variety of use cases with diverse requirements, implemented by different partners, each with their own interests and expertise. Consequently, this document on the communication layer of the distributed architecture offers recommendations that will be adapted to different implementations in the demonstrators.

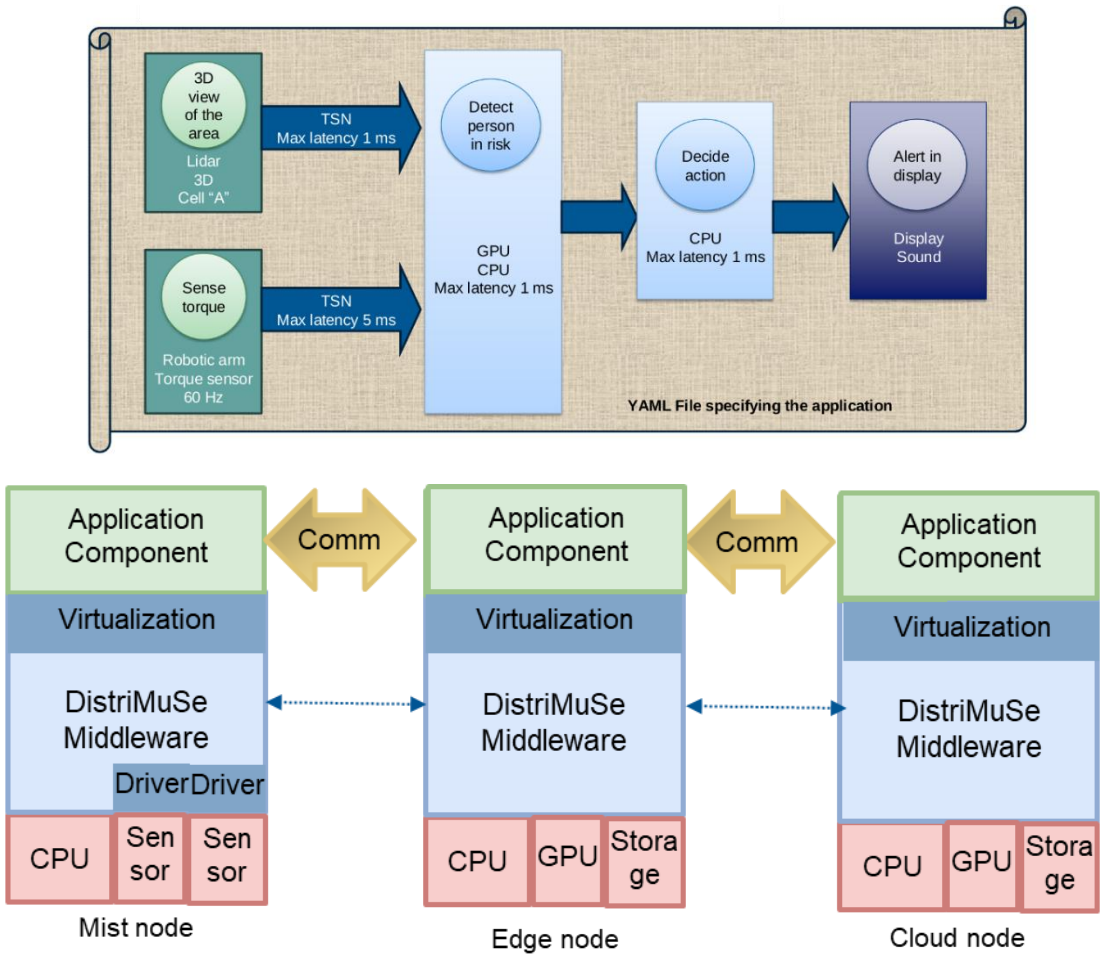

Figure 45: Data plane communications of a distributed application (Top: DSL description of an application. Bottom: Implementation model).

Figure 45: Data plane communications of a distributed application (Top: DSL description of an application. Bottom: Implementation model). shows an example of data plane communications in an illustrative scenario. At the top of the figure, we depict how a developer would describe a distributed application, with the blue arrows representing the flow of information through the data plane. At the

bottom, we present an implementation model, where application components exchange information through the data plane (represented by yellow arrows). This model also highlights the distinction between the data plane and the control plane, where the latter involves the exchange of information among instances of the middleware layer.

Any of the technologies presented in the state of the art are applicable for the data plane communications in DistriMuSe. However, it is important to note that a **message-oriented middleware** provides clear advantages for implementing distributed applications in the DistriMuSe distributed platform:

- Sender and receiver don't require to be synchronized.
- By following the ICN pattern, low-level addressing is not required.
- Messages can contain any type of data.

There are multiple implementations such as MQTT, DDS, Kafka, etc. that can be used:

- Eclipse Cyclone DDS
- eProsima Fast DDS
- RTI Connext
- Gurum DDS
- Open DDS
- Zenoh
- Zenoh-Pico
- MQTT
- ZeroMQ
- nanomsg
- nng
- LCM
- IceOryx
- OPC-UA
- eCal
- ROS 1
- Kafka
- GRPC
- Websockets
- Subspace
- XMLRPC
- CAN bus
- YAMI4
- SNMP
- Mavlink
- Cyphal (libcanard)

However, Zenoh includes several features that can be used by the DistriMuSe distributed platform

as we will discuss. Yet this protocol is not mandatory for all the communications of the platform, it is considered a good candidate for the first reference implementation.

Zenoh was designed for distributed services requiring data to be transparently shared, stored or even computed on-demand (Sabella, 2021). It utilizes a Named Data Networking (NDN) paradigm to enable access transparency in distributed environments, handling both data at rest (e.g., databases) and data in motion (e.g., pub/sub). Its routing infrastructure is designed for Internet-scale operation, ensuring data is delivered to the right location based on the available applications. With this approach, applications focus on the data they need rather than its location. Publishers assign a key to data and associate it with a value, while subscribers express interest in specific data using key expressions.

Zenoh uses Uniform Resource Identifiers (URIs) for keys, enabling hierarchical data organization. This design allows transparent data access through queries and key selectors. The routing infrastructure manages key matching and retrieves data from the nearest point in the network, so applications don't need to worry about data location, focusing instead on their logic.

By utilizing transparent data access, application developers can effortlessly build distributed applications. Additionally, the DistriMuSe distributed platform aims to simplify this process by enabling developers to define the input and output endpoints of distributed application components and the data flows between them, while abstracting the underlying middleware.

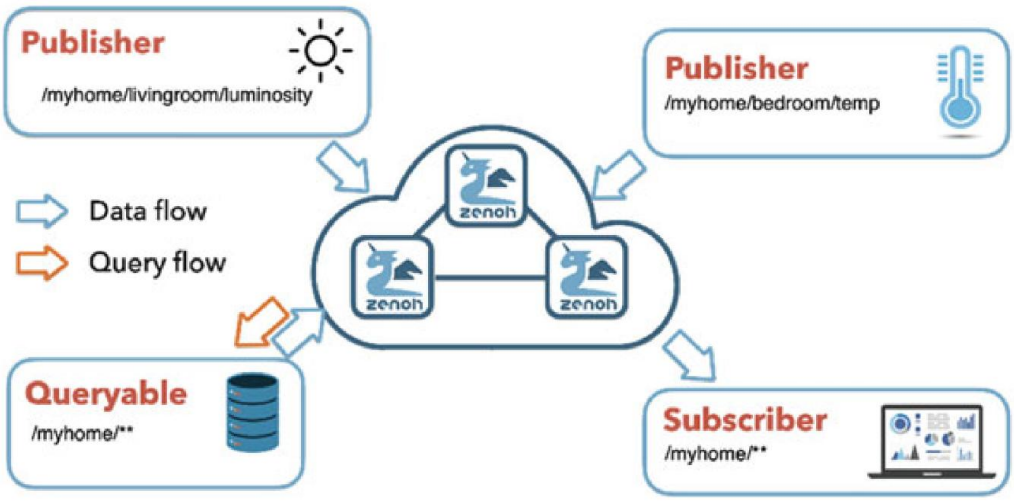

Figure 46: Zenoh abstractions (Sabella, 2021)

Zenoh offers four key abstractions that suit our needs (Figure 46):

- Resource: A named piece of data, represented as a key-value pair, such as (/house/room/temp, 25°C).
- Publisher: A source that provides values for a specific key expression, like a sensor publishing the temperature value for /house/room/temp.
- Subscriber: A recipient of values for a particular key expression, such as a monitoring application subscribing to /house/*/temp to track the temperature in any room of the house.

A subscriber is linked to a specific key expression and automatically receives all publications that match that key.

- Queryable: A data source that can be queried on demand, like a sensor node that returns the current temperature when queried at /house/room/temp.

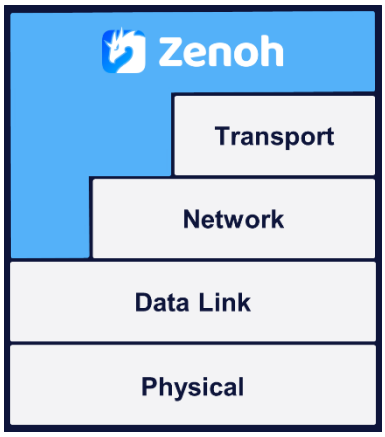

Figure 47: Zenoh protocol stack

Zenoh is designed to operate on the Data Link layer or higher in the networking stack, supporting a wide range of network technologies (Figure 47). It currently functions over protocols such as TCP/IP, UDP/IP, QUIC, Serial, Bluetooth, OpenThread, Unix Sockets, and Shared Memory. This versatility allows Zenoh support the different DistriMuSe use cases, where different low-level protocols are used (Bluetooth, OpenThread, Wi-Fi, TCP/IP, etc.).

DistriMuSe use cases also considers a wide range of network topologies: real-time local only networks, interconnected networks (local area networks combined with edge and cloud nodes), etc. In this regard, Zenoh supports such variety of network topologies to ensure efficient communication across diverse environments (Figure 48).

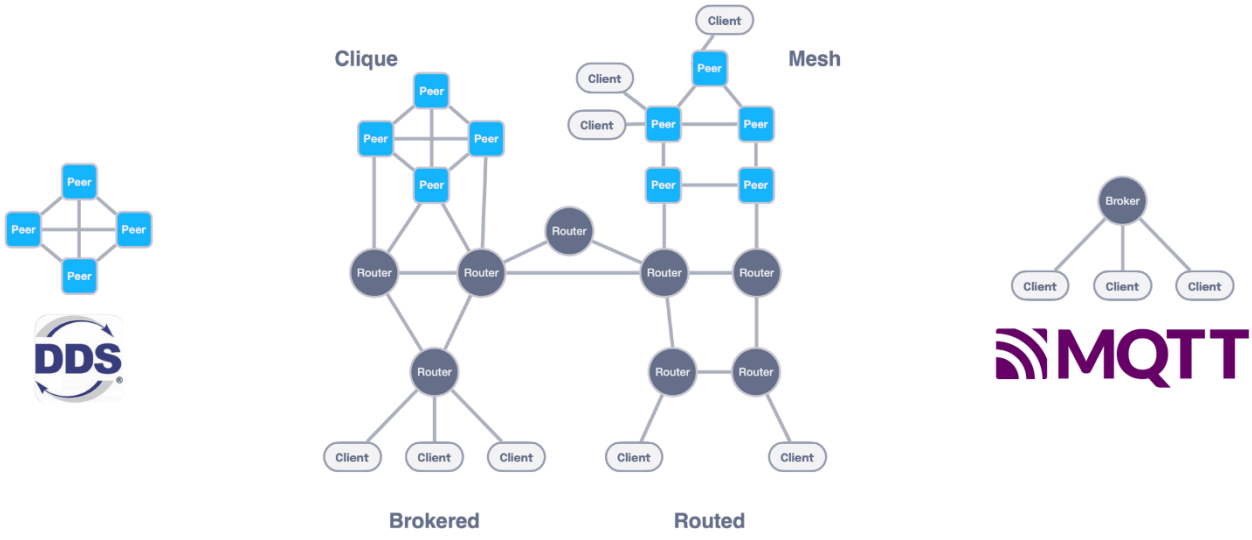

Figure 48: Zenoh topologies compared with DDS and MQTT

The first topology is the peer-to-peer (P2P) model, where nodes communicate directly with each other without relying on a central intermediary. This setup is particularly advantageous for scenarios requiring low-latency communication and direct data exchange between nodes.

Zenoh also supports a brokered topology, where a centralized broker manages and routes messages between nodes. This model simplifies communication management and is especially useful when nodes cannot directly connect due to network limitations or when a central coordination point is beneficial. For these scenarios, MQTT could also be used.

For large-scale deployments, Zenoh can operate in a hierarchical topology, where intermediate nodes act as routers or aggregators. This structure enables efficient data routing and aggregation across distributed regions. In addition to these, Zenoh supports a hybrid topology that combines aspects of both P2P and brokered models. For example, nodes in a local network may communicate directly using P2P, while remote nodes connect through a broker, offering flexibility based on application needs.

Zenoh also supports mesh topologies, where nodes are interconnected to form a resilient network with multiple data transmission paths, enhancing fault tolerance and scalability. Finally, in a star topology, a central hub node acts as the focal point for all communication, with other nodes connecting to this hub. This approach works well for smaller systems or those requiring centralized control.

Furthermore, it is proposed to implement TSN as an optional communication mechanism for certain segments of the network in the distributed platform, including both internal communication between nodes and external communication with systems outside the platform. TSN ensures deterministic data exchange in these segments, which is critical for applications requiring high reliability and precise timing. The inclusion of TSN as a communication option provides the flexibility to deploy it where needed, ensuring compatibility with existing non-deterministic segments and making it easier to transition toward high-performance communication.

Finally, Zenoh can seamlessly integrate with other technologies, such as DDS. In the context of the robotics use case (UC3), ROS stands out as one of the most widely adopted frameworks in the research community for operating robotic systems. It offers low-latency communication between nodes within the same local-area network, along with exceptional flexibility for incremental development using a modular node-based architecture. Furthermore, many robotic system manufacturers provide low-level interfaces compatible with ROS, simplifying the implementation of complex control loops and enhancing system interoperability.

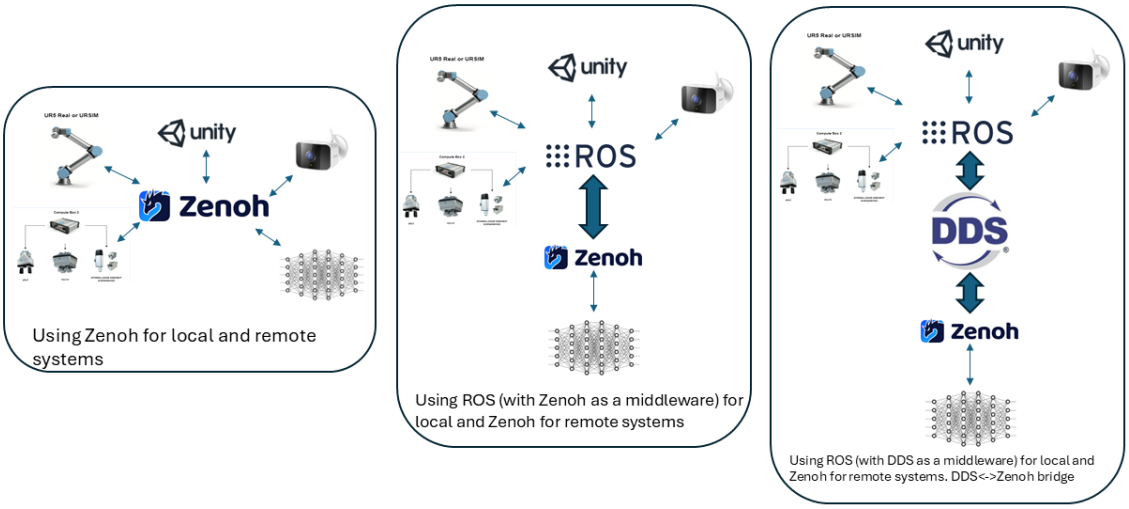

Figure 49: Different approaches for connecting robotic systems in UC3 with Zenoh

Zenoh can be integrated with existing ROS robotics systems following different alternatives, as shown in Figure 49.

Zenoh (or the distributed platform) can manage all communication between nodes, offering the advantage of a unified platform for development and management. However, this approach requires the implementation of low-level access for each device, which would fall under the responsibility of the respective technology groups.

An alternative approach involves using ROS with Zenoh as the underlying middleware. This strategy allows full utilization of ROS's extensive toolset and its existing interfaces with robotic devices. Additionally, with Zenoh as the middleware, the distributed platform can be extended to include remote nodes, such as cloud-based AI systems, enabling seamless integration beyond the local-area network (LAN).

Another option is to use ROS with DDS as the middleware while bridging DDS with Zenoh. This approach retains compatibility with current robotic systems and their DDS-based communication while enabling remote connectivity to cloud services through Zenoh, providing a flexible and scalable solution for distributed robotics applications.

5.3.3 Control plane DistriMuSe communications

The control plane of the DistriMuSe distributed platform is responsible for managing the system's overall functionality and resource utilization. Its responsibilities include the identification of available nodes within the platform, along with their specific functionalities and resources at any given moment, maintaining an up-to-date understanding of the infrastructure. Additionally, the control plane handles the deployment of application components to the most suitable nodes based on resource availability and application requirements. Moreover, the control plane monitors the health and performance of the system, providing real-time oversight and diagnostics to detect and troubleshoot issues as they arise.

The master node in the DistriMuSe distributed platform is likely to incorporate various communication protocols tailored for management and supervision tasks, such as TR-369, protocols for downloading container images, node discovery, and other essential operations. These protocols facilitate efficient coordination and oversight within the platform.

However, alongside these standard protocols, the master node must also support specific functionalities uniquely required by DistriMuSe. Zenoh can play a pivotal role, offering several features that align well with these needs. For example, the **scout** functionality performs a discovery on the network looking for other Zenoh entities (e.g. routers or peers) in order to establish a Zenoh session. Such functionality can be used for discovering nodes participating in a DistriMuSe scenario.

5.4 Virtualization

Lightweight virtualization technologies are one of the main building blocks of the distributed platform, providing the required flexibility and robustness for the deployment of applications. They are a central element of modern distributed systems due to their ability to streamline application deployment, enhance scalability, and ensure efficient resource utilization.

Virtualization is essential in a distributed cloud-edge-mist platform as it allows to optimize resource utilization by enabling multiple workloads to share the same physical infrastructure, ensuring efficient use of CPU, memory, and storage. It provides strong isolation for enhanced security, minimizing interference between workloads, and simplifies the deployment, scaling, and management of services to dynamically adapt to the demands. It also simplifies workload portability, enabling the seamless migration of an application to different physical infrastructures.

5.4.1 State of the art

Virtual machines (VMs) are the most consolidated technology for providing a robust and secure virtualization solution. VMs provide near native performance and have been widely used in cloud computing scenarios and large datacenters (Giallorenzo et al., 2021). However, they are not suitable for resource-constrained environments such as edge clouds where resources are scarce (Tao et al., 2019). This is due to the large demand of resources caused by the need to deploy a guest operating system (OS) kernel on top of the hypervisor. In the last few years, containers gained popularity as a lighter virtualization technology based on sharing OS kernels. Docker provides a container-based approach to virtualization that isolates applications in lightweight environments, ensuring compatibility across diverse infrastructures. Nevertheless, even containers can introduce a significant footprint for edge infrastructure and mist devices, where resources are strongly constrained. Thus, WebAssembly (Wasm) emerged as a solution for providing ultralightweight virtualization, in which applications not only share OS kernels but also system interfaces and libraries. WebAssembly offers a lightweight virtual environment that allows applications to run efficiently across browsers and native systems, enhancing interoperability and performance in distributed contexts.

Unlike VMs and containers, Wasm uses a compact binary format, which allows for efficient and optimized performance. This technology is especially suitable for enabling serverless computing, by reducing cold-start latency up to a 99.5%, improving memory usage 5 times and overall increasing execution throughput more than 4 times, when compared to Docker containers (Gackstatter, 2022).

Both Docker and WebAssembly have been applied in the three critical areas within European projects: iHealth, vulnerable road user (VRU) safety through digital twins, and safety in industrial robotics. These domains leverage virtualization to enhance system resilience, optimize real-time response, and facilitate interoperability across complex and distributed systems.

Containers have revolutionized distributed computing by providing isolated environments that encapsulate application code, dependencies, and runtime configuration. Docker, as a leading container platform, enables efficient application deployment across various systems, from edge

devices to cloud environments. Studies demonstrate Docker's ability to support scalable, distributed applications with high portability and consistency, significantly benefiting industries that require rapid prototyping and deployment across heterogeneous systems (Merkel, 2014). Containers have also shown strong promise in supporting edge applications, essential for iHealth and industrial robotics, by reducing latency and enabling faster response times compared to centralized cloud approaches (Morabito et al., 2018).

WebAssembly has emerged as a high-performance, secure, and ultra lightweight runtime environment initially designed for web applications. However, recent advancements have positioned Wasm as a viable option for IoT and edge computing applications, particularly due to its low memory footprint and execution speed (Haas et al., 2017). In the context of digital twins, Wasm offers a platform-agnostic way to deploy interactive, real-time simulations for applications such as vulnerable road user safety, ensuring high-performance computing capabilities while maintaining security and platform independence. The article by Kakati et al. discusses the potential of Wasm in non-web environments, particularly in the edge-cloud continuum, highlighting its suitability for applications that require low-latency execution and isolated environments across diverse platforms (Kakati & Brorsson, 2023).

In healthcare, containers and Wasm enable flexible and secure deployment of iHealth applications, supporting a modular approach to data processing, analytics, and patient monitoring (Ray, 2023). This modularity is essential for personalized healthcare, where applications must adapt to a wide range of user requirements and rapidly evolving data streams. For vulnerable road user safety, digital twin applications leverage containerized microservices and Wasm environments to simulate real-time scenarios, supporting predictive analytics and decision-making frameworks that can prevent accidents and enhance situational awareness (Xu et al., 2023). Meanwhile, industrial robotics relies on virtualized environments to enhance safety protocols and facilitate remote control and monitoring of robotic processes in hazardous environments (Yun & Jun., 2022). Containers and Wasm both contribute to these scenarios by providing secure, scalable, and real-time solutions that align with the project's focus on distributed cloud-edge-mist systems.

5.4.2 Lightweight virtualization

DistriMuSe's distributed platform will leverage lightweight virtualization technologies such as WebAssembly or Docker for the deployment of distributed applications. This will enable the portability of the applications between different locations, independence of the underlying infrastructure and near-native performance.

The virtualization layer includes runtimes deployed on the computing nodes, an interface with the orchestration layer and an Open Container Initiative (OCI) image registry for the storage of the applications (e.g., Wasm modules, Docker images).

The nodes of the infrastructure will be equipped with runtimes (such as Wasmer, WasmEdge or Wasmtime for WebAssembly workloads or Docker engine) that will be deployed across cloud, edge and mist computing nodes. This will serve as the main execution layer for running application components (e.g., compiled as Wasm modules or embedded as Docker containers). These modules will be able to interact with native system functionalities. For example, in the case of WebAssembly, these functionalities will be provided through the WebAssembly System Interface (WASI) for seamless usage of available host resources. This interface can be used to access the information captured by the different sensors available in specific nodes of the DistriMuSe's platform.

The interface with the orchestration layer will enable the management and orchestration of the workloads in the distributed system. This will allow functionalities such as the deployment, tear down, status and lifecycle management, in general. For efficient deployment and scaling, virtualized images will be stored in a registry, using the OCI specification. This enables a standardized approach to storing, sharing, and distributing application components across the system. By adopting the OCI image format, the platform ensures compatibility and integration with existing container-based environments (e.g., Kubernetes) while maintaining the unique advantages provided by the lightweight virtualization framework.

5.5 Distributed data access

In many monolithic systems that rely on a single database, developers typically don't need to worry about the process of reading from database tables or other means of storage. However, when data is distributed across separate databases or storage spaces managed by different services, accessing the data becomes more complex.

Hence, distributed data access is a critical component of any distributed platform. Distributed applications need the ability to access necessary information, regardless of its storage location. The DistriMuSe platform must enable seamless and efficient sharing, retrieval, and synchronization of data across all system nodes, spanning the mist, edge, and cloud layers.

In other to provide a robust distributed data access architecture, the architecture must meet the following requirements:

- **Scalability**: The system must be able to scale with both the number of nodes, hosts and processes, and with amount of traffic and network usage.
- **Efficiency**: The system must be able use the underlying resources efficiently, including both the storage and network infrastructure, to ensure the performance of the system.
- **Transparency**: The data access architecture must be designed in such a way that the developer is not required to know where the data is located and/or where it is generated.
- **Consistency**: The system must guarantee the reliable and uniform data access independently of the location of the node.

This section aims to explore and analyse data access mechanisms in modern distributed systems, including the distributed access to information generated by sensors, the application image distribution procedure and the distributed storage. Additionally, a state-of-the-art subsection explores distributed data access patterns used in modern distributed systems.

5.5.1 State of the art

In a distributed system, services often need access to data that they do not own or control, which falls outside their bounded context. There are several ways services can gain read access to such data.

We will discuss four patterns: Inters-service Communication pattern, Column Schema Replication pattern, Replicated Cache pattern, and the Data Domain pattern (Ford, 2021).

The **Interservice Communication pattern** is one of the most widely used methods for accessing data in distributed systems. When a service needs to read data that it cannot directly access, it

typically requests the data from the service or system that owns it. This is achieved through a remote access protocol, allowing one service to communicate with another, retrieve the necessary information, and continue processing without having direct access to the data source.

This pattern, while common and simple, has significant drawbacks. When a service makes synchronous remote calls to another service for data, performance suffers due to several types of latency. These include network latency, security latency, and data latency. This can result in delays of up to one second for fetching required data.

Another major disadvantage is service coupling. The service requesting the data is tightly dependent on the service owning it, meaning that if the owner is unavailable, the first service cannot function either.

The **Column Schema Replication pattern** involves replicating columns across tables, which allows data to be available to other bounded contexts. In this pattern, a column from one table is replicated in another table. This approach makes the replicated data directly accessible, eliminating the need for the service to request the data from the remote system.

This pattern addresses performance, fault tolerance, and scalability by replicating data across tables in different services. However, it introduces two significant challenges: data synchronization and data consistency. When the data in the original service changes, it must notify the other services that replicate the data, typically using asynchronous communication methods such as queues or event streaming. Another issue with this pattern is data ownership governance. Because data is replicated in another service's table, that service can modify the data even if it does not own it, potentially causing consistency problems.

Despite these challenges, the Column Schema Replication pattern offers significant benefits in performance and scalability, as the service needing the data can access it directly from its own table, avoiding costly remote calls. While it may not be suitable for all cases, it can be useful for scenarios like data aggregation, reporting, or cases where large data volumes and high responsiveness or fault tolerance are crucial.

The **Replicated Caching pattern** is a technique for improving responsiveness by storing data in-memory, reducing retrieval times from milliseconds to nanoseconds. However, caching can also play a key role in distributed data access and sharing. The Replicated Cache pattern uses in-memory caching where data needed by one service is made available to others without needing to make separate requests. Unlike other caching models, a replicated cache stores data in-memory across multiple services and continuously synchronizes it, ensuring that all services always have the same data. This enables faster data access across the distributed system while maintaining consistency.

The Replicated Caching Pattern offers a way to improve distributed data access by ensuring that each service has its own in-memory cache, which is kept in sync across services. This allows services to share data without needing to ask other services for it, avoiding the latency and dependencies associated with interservice communication. Unlike other caching models, such as single in-memory caches and distributed caches, replicated caches synchronize data across services, allowing fast data access and fault tolerance, even if the owning service is down.

However, this pattern does come with trade-offs. The first is the service dependency on the cache data and startup timing. If the owning service is down when the dependent service starts, the dependent service will wait for the cache data. The second trade-off involves memory usage, particularly if the volume of cached data is high. For services with multiple instances, this can quickly lead to high memory consumption. The third challenge is keeping data synchronized when the data

changes frequently. Finally, configuration and setup management can be complex in dynamic environments, which can delay the establishment of connections between services. Despite these trade-offs, replicated caching is highly effective for relatively static data.

The **Domain Data Pattern** resolves data access challenges by creating a shared schema between services that need access to common data. This pattern is particularly useful when other data access patterns, like Interservice Communication, Column Schema Replication, or Replicated Caching, are not feasible due to issues like performance, reliability, or data consistency.

In this pattern, multiple services access shared tables, where the data is no longer owned by any specific service. This eliminates the need for complex interservice communication, reducing latency and improving data access.

However, there are trade-offs. Unlike other patterns that abstract data access, the Domain Data Pattern directly uses a common database to store all the data. Additionally, the pattern might expose security vulnerabilities by giving services more access to data than necessary.

5.5.2 Application image repository

As previously discussed, the DistriMuSe distributed platform provides application developers with abstractions to create distributed systems using a Service Description Language (SDL). This language is used to describe and configure the various components and the data flows between them. Developers are required to implement their application components as containers or WebAssembly applications.

The orchestrator is responsible for assigning each component to the appropriate node in the system. To achieve this, the orchestrator must ensure that the component images are readily available for download onto the respective nodes. Consequently, a reliable storage solution is needed to store these component images, making it possible for each node to download the necessary images for deployment.

The images are stored in image repositories or image registries, which serve to store, manage, and distribute images. Since the application components of the DistriMuSe platform adhere to the Open Container Initiative (OCI) specification, it is essential that the repository supports this image format.

A single repository or registry represents a single point of failure and, depending on the physical location and distribution of the hosts, its performance may degrade. To mitigate the inherent risks of relying on a centralized registry, replication can be employed, along with the use of multiple registries distributed across different points in the system topology.

5.5.3 Distributed access to the sensed information

Sensor nodes gather data from the physical environment and are connected to the whole system through a network. While many sensor nodes are designed for minimal resource consumption, some may have onboard storage capabilities. This storage allows for local buffering, ensuring data is retained temporarily during connection issues or when aggregation is necessary. In such setups, node storage reduces data transmission demands and helps maintain operational resilience. Technologies like EEPROM, SD cards, or eMMC storage modules are common here. Emerging trends also include lightweight distributed file systems optimized for low-power devices, such as FUSE-based systems or TinyDB.

Data can be accessed in two primary ways: when it is sensed, which is referred to as "data in motion" by the Zenoh creators, or it can be recovered from local or remote storage, known as "data at rest." Data in motion involves real-time data streaming, where data is processed and transmitted as it is being generated, while data at rest refers to stored data that is retrieved for use when needed, typically from databases or storage systems.

The DistriMuSe distributed architecture proposes adopting a mechanism for unified data management that provides unified abstractions for data in motion, data at rest, and computations in the mist-edge-cloud continuum

For example, Zenoh protocol defines publishers and queryable network entities. A publisher acts as the source for resources matching a key expression. For example, it could publish data for a specific key like home/kitchen/sensor/CO2, or for a set of keys like home/kitchen/sensor/* or home/kitchen/**. A queryable is a resource that holds data matching a key expression. For example, a queryable for home/kitchen/** will be able to respond with matching resources when queried for keys that fit the specified expression.

A subscriber can request to receive resources that match a key expression. For example, it could subscribe to a specific key like home/kitchen/sensor/C2O2, or to a range of keys such as home/**/sensor/*. In this way, the subscriber would receive the "data in motion" generated by any sensor that fits the specified key pattern.

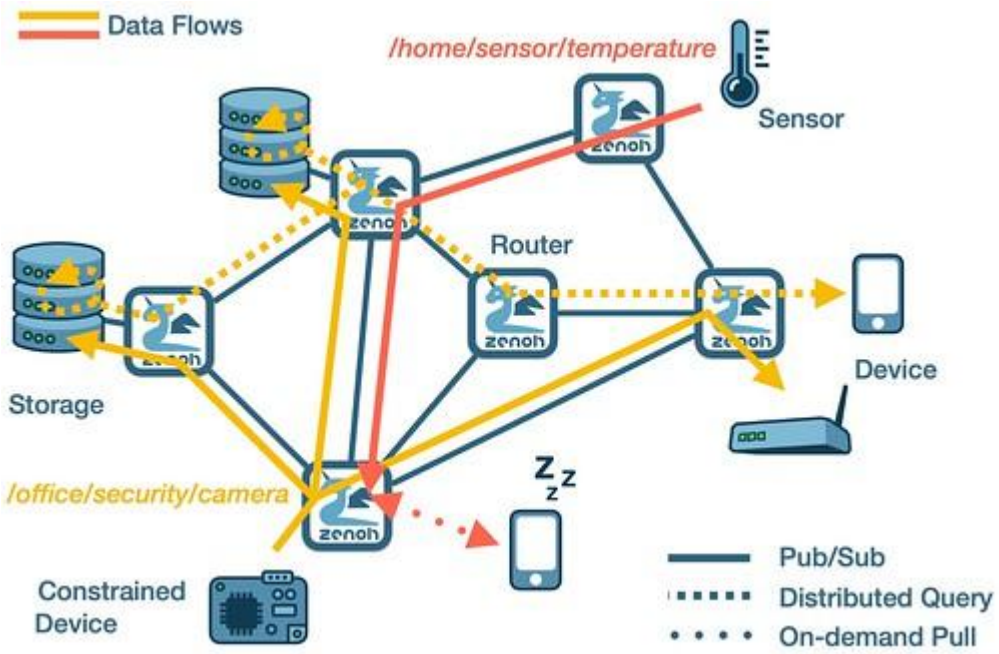

Figure 50: Zenoh deliver of data at motion (stream) and data at rest (stored)

The combination of a subscriber and a queryable allows for the abstraction of any storage (Figure 50). Each publication matching the subscriber's expression is ingested into the storage, while every query matching the queryable results in a query to the storage. This setup enables seamless interaction with both real-time data (through the subscriber) and historical data (through the queryable), allowing efficient management and retrieval of information from sensors.

5.5.4 Distributed data storage

Certain characteristics of the data, such as granularity, fidelity, frequency, and sensitivity, directly influence decisions regarding its placement. Data placement must balance key factors such as latency, throughput, security, cost, locality, and compliance, which have proven to be particularly impactful. Keeping data closer to users or systems that access it frequently reduces latency and enhances performance, addressing the urgency of required results. Partitioning plays a crucial role in distributing data evenly, improving scalability and efficiency.

Security measures, such as encryption and access controls, safeguard sensitive information, while costs can be minimized by selecting economical storage options and reducing data transfers. Locality ensures data is readily accessible when needed, while compliance with regional regulations like GDPR or HIPAA avoids legal complications.

A well-thought-out data placement strategy achieves a balance among these factors, resulting in a reliable and efficient system.

As previously noted, in the DistriMuSe distributed architecture, data can be stored close to the source, such as within the sensor device itself or an embedded system connected to it. Data can also be stored at the edge, where it can be aggregated, filtered, and processed near its generation point. Edge servers or nodes are often equipped with advanced storage options to support real-time analytics, AI model execution, and data pre-processing. The main goal at the edge is to minimize latency and bandwidth usage while ensuring critical data is processed and retained when necessary. The cloud serves as the backbone of the distributed storage system, enabling large-scale data processing, long-term storage, and connectivity across different layers.

Data is typically organized into several storage types based on usage:

- **In-Memory Storage**: For scenarios requiring extremely fast access, systems like Redis or Memcached are used to cache frequently accessed data or maintain session information, enabling high-speed retrieval.
- **Hot Storage**: This tier holds the most recent and operationally significant data, ensuring minimal latency for access. Technologies such as Amazon S3 with Intelligent-Tiering, Google Bigtable, and Azure Blob Storage (Hot Access Tier) are ideal for handling these workloads.
- **Cold Storage**: This tier stores archived and batched data, including system assets such as machine learning models. Solutions like Amazon Glacier, Google Coldline, and Azure Archive Storage offer cost-effective long-term storage options.

To achieve the requirements identified at the beginning of this section, we propose to extend the Information-Centric Networking paradigm to seamlessly integrate distributed storage using a uniform interface to access the data, regardless of whether the information is stored somewhere in the network or is sent by a node in real-time.

The result should be the access to a data element via a structured identifier or key to retrieve stored data or real-time data being generated by the corresponding sensor. In the same manner, to modify the content of stored data the action would be equivalent to publishing information on the same key.

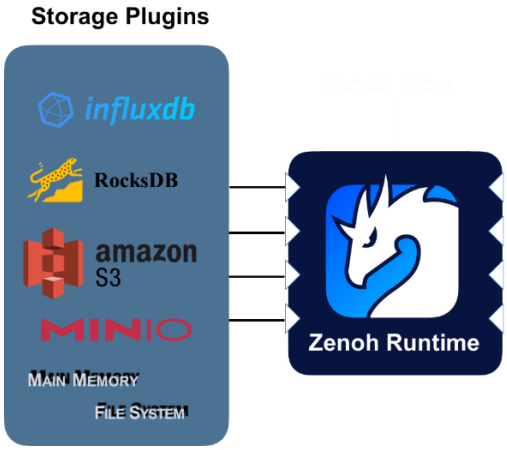

Figure 51: Zenoh storage plugins

For example, protocols such as Zenoh, allows to integrate the communication layer with the distributed storage by supporting various storage plugins (Figure 51), following the Interservice Communication pattern. While it has some limitations, this pattern provides a straightforward implementation that facilitates the integration of different storage types. This allows for the seamless management and sharing of data across the distributed architecture, accommodating various storage tiers (e.g., hot, cold, in-memory) while ensuring that data can be accessed within the mist-edge-cloud continuum.

5.6 Intelligent platform orchestration

The increasing complexity of distributed applications and the need for scalable, reliable infrastructure have led to the rise of intelligent platform orchestration tools. These tools enable automated deployment, management, and scaling of containerized applications across diverse environments, from cloud to edge computing. Orchestrators are vital for handling the complexities of modern applications, including service discovery, load balancing, self-healing, and resource management. While Kubernetes has become the most widely adopted orchestrator, many alternatives offer unique features and optimizations for specific use cases, infrastructure constraints, or operational preferences.

In this section, we explore the landscape of platform orchestrators, each with distinct approaches to workload management and deployment orchestration. DistriMuSe will follow the state-of-the-art proposals to design and develop its system and application orchestrators.

5.6.1 State of the art

5.6.1.1 Docker Swarm

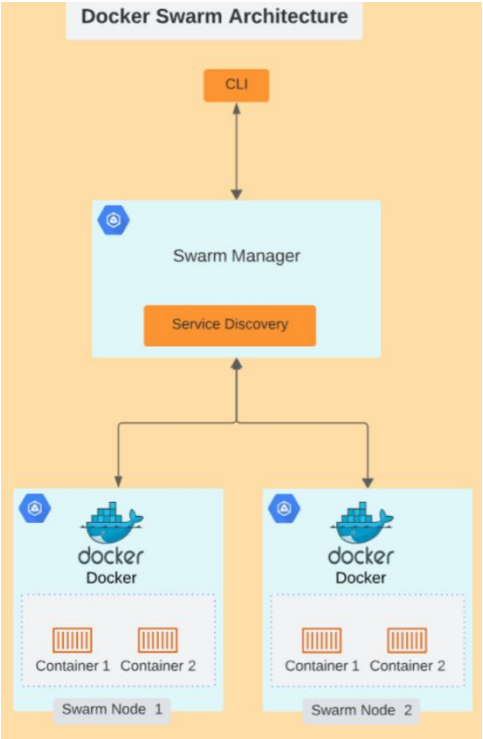

Figure 52: Docker Swarm Architecture: Centralized Swarm Manager orchestrates containers across multiple nodes, using service discovery for inter-container communication and the CLI for cluster management.

Docker Swarm (Docker, 2024; Soppelsa & Kaewkasi, 2016), developed by Docker Inc., is a native clustering and orchestration tool integrated with Docker. Unlike Kubernetes, Docker Swarm prioritizes simplicity and ease of use, offering a more straightforward setup and configuration process. It provides basic features such as load balancing, scaling, and service discovery but lacks the extensive modularity and customization options found in Kubernetes. Docker Swarm is often chosen by organizations seeking a lightweight orchestrator with minimal overhead, particularly in smaller-scale environments where Kubernetes' complexity may be unwarranted. While Docker Swarm does not have native support for WebAssembly, it is possible to run Wasm applications within Docker containers using tools like **Wasmer** and **Wasmtime**, which encapsulate Wasm inside a Docker container. However, this approach requires more manual configuration and lacks the optimized support for Wasm available in some other orchestrators.

Figure 52 shows the architecture of Docker Swarm. The Swarm Manager is the central component of Docker Swarm responsible for managing the cluster and orchestrating containers. It handles key tasks such as scheduling services, managing network configuration, scaling applications, and ensuring high availability. The manager includes a **Service Discovery** mechanism that enables automatic detection and resolution of services within the Swarm. This allows containers to communicate with each other using service names rather than specific IP addresses. Users interact with the Swarm Manager through the Docker CLI, allowing them to issue commands to create, scale, and manage services across the cluster. The CLI provides an interface to deploy services, inspect

system state, and configure various aspects of the Swarm. Docker Swarm clusters consist of multiple nodes, which can be designated as **Swarm Nodes**. These nodes can serve as either manager nodes or worker nodes. In the architecture shown, the Swarm Nodes host Docker containers (Container 1 and Container 2 in each node). The Swarm Manager orchestrates these containers across nodes, distributing workloads and ensuring that containerized applications run reliably. Each node is equipped with Docker's runtime environment to manage the lifecycle of containers, facilitating container deployment, scaling, and load balancing. Docker Swarm provides built-in load balancing and scaling. If a container fails or a node goes down, the Swarm Manager redistributes workloads to ensure continuous operation, maintaining the specified service level.

5.6.1.2 Nomad

HashiCorp's Nomad (HashiCorp, 2024) is an orchestration tool designed for simplicity and flexibility, supporting not only containerized applications but also virtual machines and standalone binaries. Nomad's single binary architecture makes it lightweight and easy to deploy across multiple infrastructures, including on-premises, cloud, and hybrid environments. It is particularly suitable for organizations with a diverse range of workloads that include both containerized and non-containerized applications. Nomad's integration with HashiCorp's ecosystem, including Vault for security and Consul for service discovery, allows for a cohesive infrastructure management solution, albeit with fewer native features for container orchestration compared to Kubernetes. Nomad offers native support for Wasm through its **wasm runtime**, allowing users to schedule and orchestrate Wasm workloads without requiring additional tools. This makes Nomad a compelling choice for organizations looking to adopt Wasm in multi-cloud or hybrid environments where a lightweight and versatile orchestrator is needed.

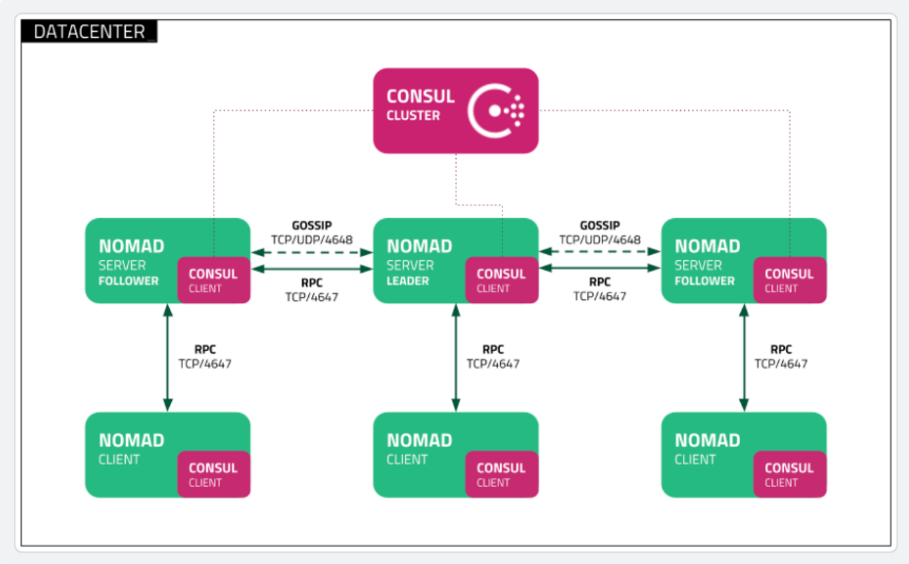

Figure 53: Nomad reference architecture: Nomad servers and clients coordinate via RPC for workload orchestration, with Consul providing service discovery and health checks. The Gossip protocol enables efficient failure detection across the cluster.

Figure 53 shows the reference architecture for HashiCorp's Nomad. Nomad works closely with Consul, a service mesh and distributed configuration solution, to manage network discovery, service health checks, and configuration management within a data centre. The Nomad servers handle cluster coordination, including scheduling, resource allocation, and maintaining state. Among the servers, one acts as the **Leader**, responsible for making scheduling decisions, while the others are

Followers, which maintain a consistent copy of the state but defer to the leader for instructions. Servers communicate via **RPC (Remote Procedure Call)** over TCP on port 4647 to synchronize state and respond to client requests. Nomad Clients are responsible for executing and managing workloads (jobs) on individual nodes within the cluster. They interact with Nomad Servers to receive task assignments and report the status of their jobs. Each Nomad Client also includes a Consul Client for service registration and health checking, facilitating network discovery and ensuring workloads are reachable across the data centre. The Consul Cluster is a set of Consul servers responsible for maintaining a consistent service registry and managing health information for each service in the cluster. Consul enables service discovery and provides health monitoring, which is essential for ensuring that Nomad can place tasks on healthy nodes and reroute traffic if necessary.

Nomad and Consul servers communicate using the Gossip protocol (on TCP/UDP port 4648) for lightweight membership and failure detection. This protocol allows nodes to exchange information about the cluster's health and membership status. Nomad servers and clients communicate over TCP on port 4647 using RPC, which is used for task allocation, scheduling, and client-server coordination.
In this architecture, Nomad and Consul work together to provide a resilient, distributed orchestration environment. Nomad handles workload scheduling and resource management, while Consul supports service discovery and health checks. The Gossip protocol ensures efficient failure detection, and RPC enables seamless communication across servers and clients, ensuring high availability and robust performance within the data centre.

5.6.1.3 Rancher

Rancher (SUSE, 2024) is a comprehensive container management platform that provides a unified interface for managing multiple Kubernetes clusters as well as support for other orchestration engines. While Rancher has evolved to focus heavily on Kubernetes, it initially supported its own orchestrator, Cattle, which was designed to be simpler and more user-friendly than Kubernetes. Cattle offered fundamental orchestration capabilities suited for less complex environments, focusing on ease of use and fast setup. However, Rancher's current focus on multi-cluster Kubernetes management does not preclude its relevance in multi-orchestrator environments, as it can manage different Kubernetes distributions and offers integrations for non-Kubernetes workloads. Rancher does not natively support Wasm but can manage Kubernetes clusters that use Wasm-compatible tools like Krustlet.

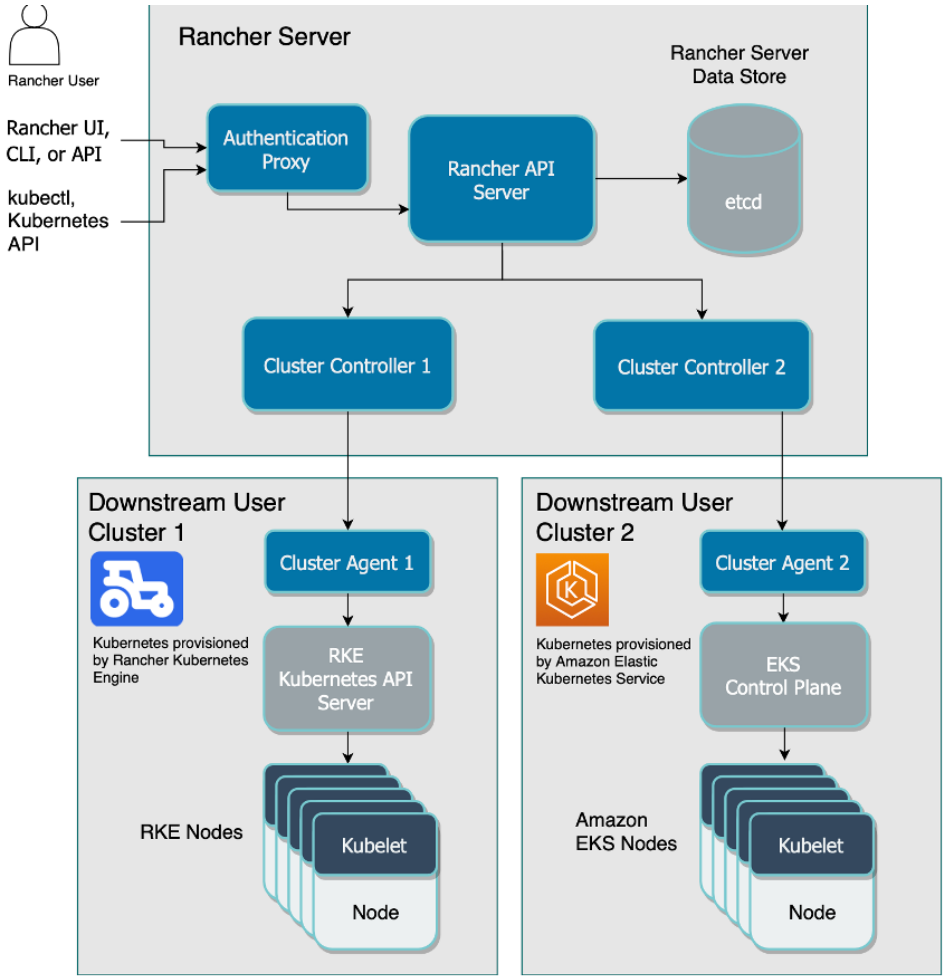

Figure 54: Rancher Architecture: Centralized management platform with Rancher Server overseeing multiple downstream Kubernetes clusters. Includes authentication, API server, and controllers for cluster synchronization. Supports clusters with Rancher Kubernetes Engine (RKE) and Amazon EKS.

Figure 54 shows the architecture of Rancher. The Rancher Server is the core component of Rancher, responsible for managing and orchestrating Kubernetes clusters. It includes several key elements:

- Authentication Proxy: this layer manages user authentication, ensuring secure access to the platform. Users can interact with Rancher via multiple interfaces, including the Rancher UI, CLI, or API. Kubernetes-native tools like kubectl can also be used to interface with the Rancher-managed clusters.
- Rancher API Server: the API server coordinates with the various internal components of Rancher to handle user requests and cluster management tasks. It also interfaces with the data store.
- Data Store (etcd): Rancher relies on etcd, a distributed key-value store, for storing the configuration and state of the Rancher server. This includes cluster configurations, user permissions, and other essential information for cluster management.

Rancher utilizes **Cluster Controllers** to manage and control downstream clusters. These controllers are responsible for synchronizing the state between the Rancher Server and each individual downstream Kubernetes cluster. They also facilitate communication with the respective Cluster Agents in the downstream clusters.

Rancher supports multiple downstream Kubernetes clusters, which are referred to as "user clusters" in the Rancher environment. These clusters are managed and monitored by Rancher but can be provisioned using different Kubernetes engines, such as the **Rancher Kubernetes Engine (RKE)** or managed Kubernetes services like **Amazon Elastic Kubernetes Service (EKS)**.

Each downstream cluster is configured with a **Cluster Agent**, which communicates with the Rancher Server's Cluster Controllers, providing updates on the cluster's state and executing Rancher's instructions within the cluster. This enables centralized management of various clusters, regardless of the underlying Kubernetes distribution or infrastructure.

5.6.2 DistriMuSe intelligent orchestrator

The DistriMuSe distributed platform considers a centralized orchestrator in the architecture. Its main features include:

- Management and distribution of application components to worker nodes.
- Monitoring and lifecycle management of applications components.
- Resource monitoring and control.

For the deployment of a distributed application in the platform, the developer must create a distributed application definition file using the SDL. This file defines each of the components, their configuration and the links connecting the components. The orchestrator then divides the application into the different components and determines the communication needs between them to ensure that the application can perform the required tasks effectively. After the application components are created, the requirements for each component are determined. For instance, the need for special hardware requirements (e.g., sensors for certain sensing components, cameras for video streaming, GPUs or NPUs for hardware accelerated AI training or inference). The orchestrator streams the application components to a set of workers nodes that fulfil the requirements and have enough resources to perform the task.

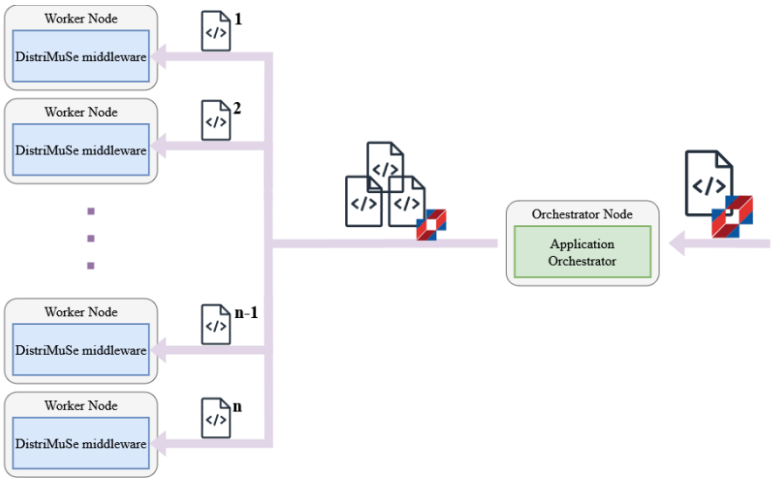

Figure 55: Application distribution

Once the components have been deployed, the orchestrator monitors their execution, as well as continues the monitor of the resources and the status of each node. The orchestrator also provides

the capability to fully control the life cycle execution of the application components, allowing to start, stop, delete or restart, one or more components, either by user command, or in response to the needs of the operation. For example, in the event of a deployment failure, it can deallocate the resources taken by the different components of this application.

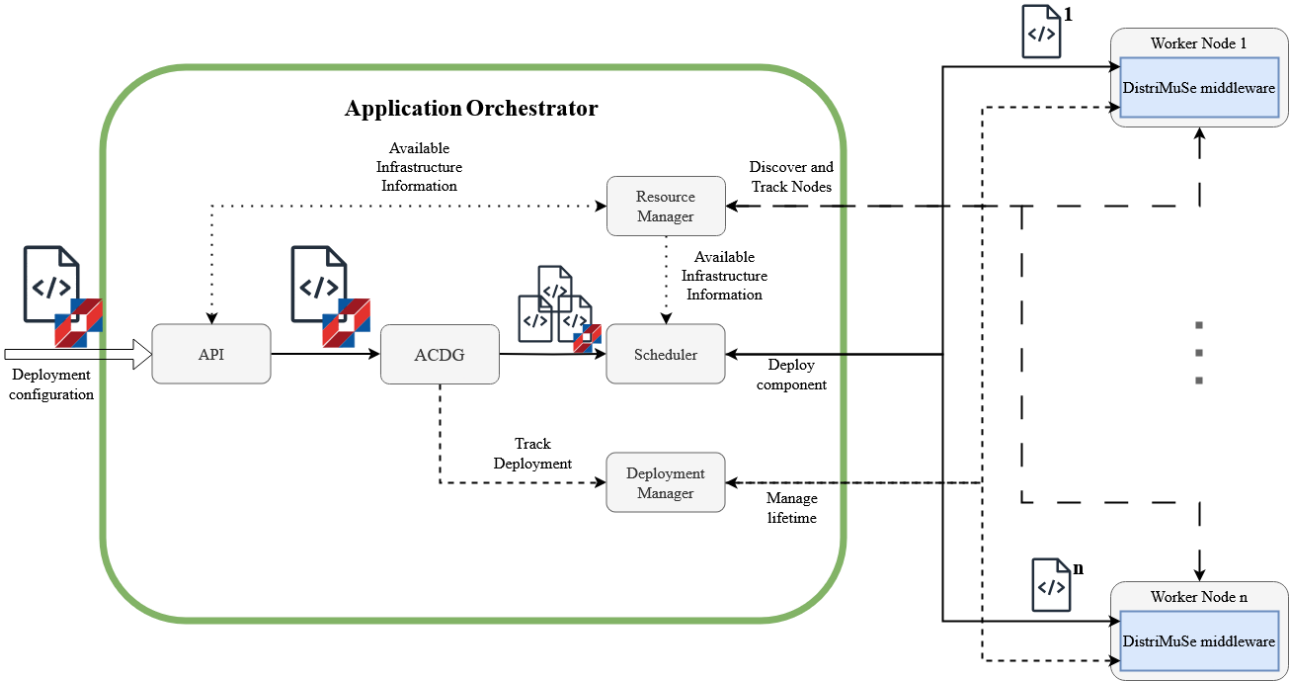

Figure 56: Internal architecture of the orchestrator

The orchestrator will interact with the worker nodes through the control plane communications. This communication channel is reserved and restricted to be used only for the platform orchestrator (i.e., it cannot be used by the applications directly). We identify two types of data flows:

- **Application streaming flow:** Used by the orchestrator to send the application components to each worker. This same flow could also be used to load AI models or any large size file or group of files dependencies
- **Control flow:** Used between worker management entities and the orchestrator to perform monitoring of the resources, life cycle control commands, allocation and deallocation of resources.

The orchestrator internal architecture, shown in Figure 56, contains the following elements, from a functional point of view:

- **Orchestrator API**: This element enables the control of the orchestrator to an external entity, mainly, a human operator.
- **Application Component Definition Generator (ACDG)**: This element is responsible of the creation of the different application components definition files for a given deployment. These files contain the subset of the information needed for each application component deployment.
- **Scheduler:** The scheduler has the task of assigning the application components to nodes with the hardware and computational requirements needed for the application components.

- **Deployment Manager**: Controls the application components and deployment lifetime. It communicates with the workers' agent in other to check the state of the different application components.

- **Resource Manage**r: This element discovers the worker nodes available at a given time and their capabilities. After discovering, the Resource Manager starts to track the usage of resources of the discovered nodes and the status of the node, ensuring that the list of the nodes considered for deployment are healthy.

The flow of a new deployment starts with an application description and a command being sent to the API. The API server validates the structure of the definition and if it is valid, the definition is sent to the Application Component Generator to create the application components. These application components are then assigned to a healthy node that complies with the requirements estimated for the specific component. Lastly, the Deployment Manager is informed about the new deployment and after that this last element should track the state and control the lifetime of this deployment. Additionally, the Resource Manager is deployed at the start of the Orchestrator to be able to detect the available worker nodes and track their resources (including connected sensors, and specialized hardware, such as, GPU or NPUs).

A discovery protocol will be used to detect the nodes (e.g. the gossip protocol implemented by Zenoh). After discovery, the capabilities of the node will be queried using a predefined path.

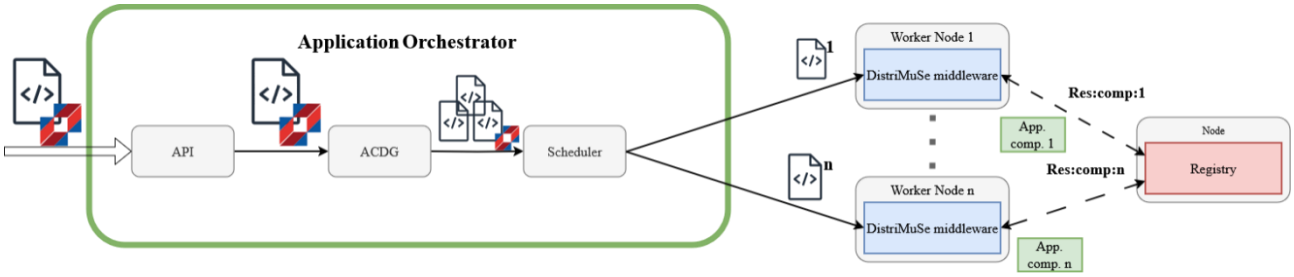

Figure 57: Distribution of application components

Regarding the creation of the distribute application components, they are expected to be created and stored on a separate entity on the network prior to the deployment of the application. This entity, known as repository or registry, allows for upload and retrieval of application components with a unique identifier. This identifier must be included into the deployment definition file. This process is depicted in Figure 57.

5.7 Application composition

The pipes-and-filters programming paradigm, commonly used in IoT applications, organizes tasks into a sequence of independent processing steps called filters. While effective for single-machine applications, the emerging scenarios considered in DistriMuSe need more flexible approaches. These scenarios require computations to span across multiple locations, such as on-board systems, edge, and cloud environments, and involve complex, dynamic data interactions. DistriMuSe use cases involve complex distributed applications with components that need deployment across various nodes in the system. To address this, it is essential to understand how these applications can be decomposed into smaller, manageable components and designed effectively. This

decomposition ensures that each component can operate independently while maintaining the overall functionality of the application.

5.7.1 State of the art

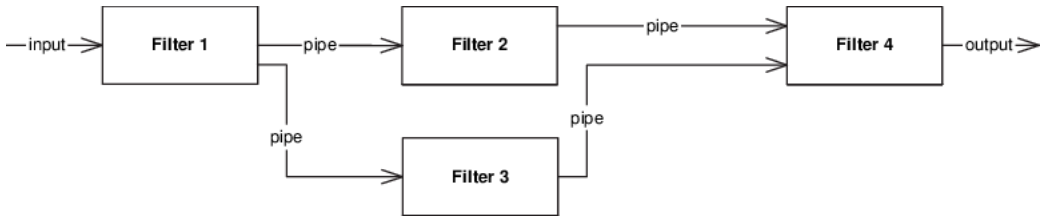

Figure 58: Pipe-and-filter pattern

The pipe-and-filter pattern (Figure 58) organizes systems that transform streams of discrete data items into reusable, loosely coupled components connected through simple interaction mechanisms. This design enables flexible combinations, parallel execution, and component reusability. Each filter performs a specific transformation, consuming data from input ports and outputting the processed data via output ports. These filters are connected through pipes, which pass data between filters, and a filter can have multiple inputs and outputs (Bass, 2012).

In a pipe-and-filter system, pipes buffer data during communication, allowing filters to execute asynchronously and concurrently. This decoupling ensures that a filter does not need to know the identity of its upstream or downstream filters. Consequently, the system exhibits a property where the overall computation can be treated as the functional composition of the computations performed by individual filters.

Data transformation systems often adopt the pipe-and-filter structure, with each filter responsible for a specific part of the overall transformation of input data. The independence of processing at each step facilitates reusability, parallel execution, and simplified reasoning about the system's behaviour. Such systems frequently serve as the front end of signal-processing applications. For example, initial filters might process sensor data by compressing and smoothing it. Further downstream, filters reduce the data further and perform synthesis across inputs from different sensors. The final filter in the pipeline typically delivers its processed data to an application, such as a modelling or visualization tool.

Despite its advantages, the pipe-and-filter pattern has several weaknesses. It is generally unsuitable for interactive systems because it disallows cycles. Additionally, managing a large number of independent filters can lead to significant computational overhead, as each filter typically operates as its own thread or process. Furthermore, the pattern may not be ideal for long-running computations unless augmented with checkpoint or restore functionality. Without such mechanisms, a failure in any filter or pipe can cause the entire pipeline to fail, posing challenges for fault tolerance.

Because of these limitations other patterns were proposed, specially designed for IoT applications, each offering unique approaches to managing and processing data:

- Dataflow Programming: Focuses on modelling the flow of data between operations, enabling asynchronous data processing across multiple nodes, such as those handling data from IoT sensors.
- Event-driven Programming: Is centred on applications whose control flow is triggered by events, such as user actions or incoming data from IoT sensors.

- Functional Programming: Treats functions as fundamental values, allowing them to be passed as parameters or returned from other functions. This approach can simplify and reduce the complexity of source code for processing and interpreting IoT sensor data.
- Aggregate Programming: Employs the concept of computational fields, providing a unified abstraction for managing large self-organizing device networks. This paradigm supports the creation of complex distributed services while ensuring encapsulation, modularity, and safe service composition in IoT environments.

One of the most interesting paradigms is Data Flow Programming (DFP), which addresses the challenges of the pipes-and-filters patterns by generalizing it into a directed graph of components, called operators, rather than a linear sequence. DFP shifts developers' focus toward the logic of the operators while the framework handles communication and data exchange. This abstraction simplifies application development and ensures seamless operation across diverse deployment environments (Baldoni, 2023).

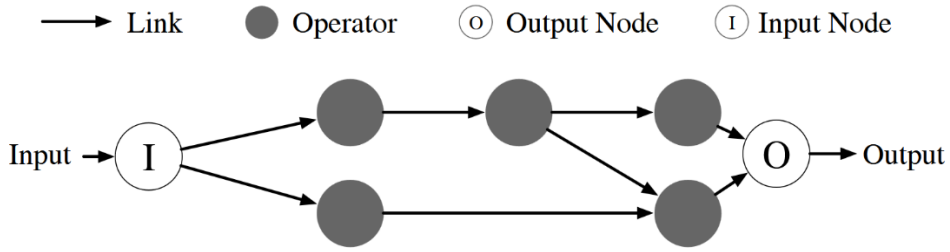

Figure 59: Generic Data Flow program graph (Baldoni, 2023)

In DFP, applications are represented as directed graphs. In this model, each node in the graph, called an operator, performs a computational task, and the links (unbounded FIFO streams) between operators carry the data. Links are categorized as inputs or outputs, and the connection between an operator and a link is referred to as a port. Operators in DFP are executed concurrently, and they can be triggered based on specific strategies, such as requiring all inputs to be present or only a subset. The execution of an operator is termed firing. This model allows developers to decompose complex applications into simpler components and focus on the logic of each operator, while the communication between them is abstracted away. However, DFP does not specify how to handle real-time processing, deadlines, periodic computations, progress tracking, or multi-location deployments.

Kahn Processing Networks (KPNs), a variant of DFP, impose stricter execution rules, requiring that all inputs for an operator must be present before it can fire. This ensures data dependencies are respected but limits flexibility, especially in applications that need to process data incrementally or require partial data processing. For example, an operator processing sensor data might need to generate outputs as soon as any input is available, which is not possible under KPN's strategy.

Dataflow Process Networks (DPNs) offer more flexibility than KPNs by allowing operators to fire when only a subset of inputs is available, enabling partial processing of data. This approach supports more dynamic processing but introduces limitations, particularly in scenarios that require tracking historical or stateful information. For instance, in applications like object tracking or prediction, operators may need to remember past states, which DPNs cannot handle effectively due to their inability to maintain local state. Thus, while DFP, KPN, and DPNs provide powerful abstractions for dataflow-based computation, each has its own trade-offs in terms of flexibility, state management, and suitability for real-time or stateful applications.

Zenoh-Flow follows the DFP pattern, having in mind the following principles for its design (Baldoni, 2023):

- Cloud-to-Thing Native: Facilitates the deployment of applications across the continuum, with automatic allocation of application operators.
- Declarative: Explicit application definition, enabling analysis and optimization of the application graph.
- Composition: Support the reuse and combination of operators, allowing developers to build more flexible systems.
- Deterministic and Time-Aware: It incorporates features like time-stamping, deadlines, logging, and replay to meet the requirements of time-sensitive applications.

Zenoh-Flow extends the traditional DFP model by allowing nodes to have “side-effects”, such as maintaining internal states and performing I/O operations. This capability enables more complex applications, such as video pipelines and AI/ML models, which require historical data for predictions and learning. In Zenoh-Flow, nodes are categorized into three types: Sources, Sinks, and Operators. Sources perform input operations, feeding data into the system, while Sinks handle output operations, sending results out. Operators, on the other hand, do not handle I/O but can maintain internal states. One challenge of allowing side-effects is the potential loss of determinism. To address this, Zenoh-Flow uses Hybrid Logical Clocks (HLC) to timestamp all data, ensuring a total order of events and allowing the system to reproduce executions as long as the node states are deterministic. Finally, Zenoh-Flow supports time-triggered execution, enabling periodic wake-ups for nodes to execute at regular intervals.

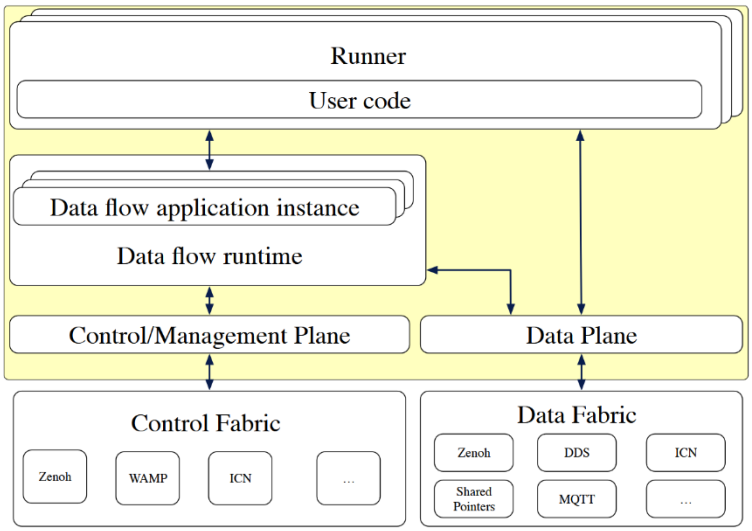

Figure 60: Zenoh-Flow high-level architecture (Baldoni, 2023)

The architecture of Zenoh-Flow (Figure 60) is very similar to the one proposed for the DistriMuSe distributed platform, although it uses custom developed components and runtimes, limiting the compatibility with programming languages and platforms. The architecture consists of:

- Runners: Execute user code and manage individual operators.

- Data-Flow Instance: Represents an application running across the continuum. This instance tracks Runtime information and oversees operator execution.
- Dataflow Runtime: Maps applications to infrastructure, manages operator lifecycle, and configures the data plane.
- Control/Management Plane: Manages communication and control across different runtimes within the continuum. It stores information about the state of the infrastructure and any applications running.
- Data Plane: Moves data between operators, configured by the runtime.

5.7.2 DistriMuSe application and algorithm distribution

We propose following the Data Flow Programming pattern and the Zenoh-Flow model to implement our distributed applications.

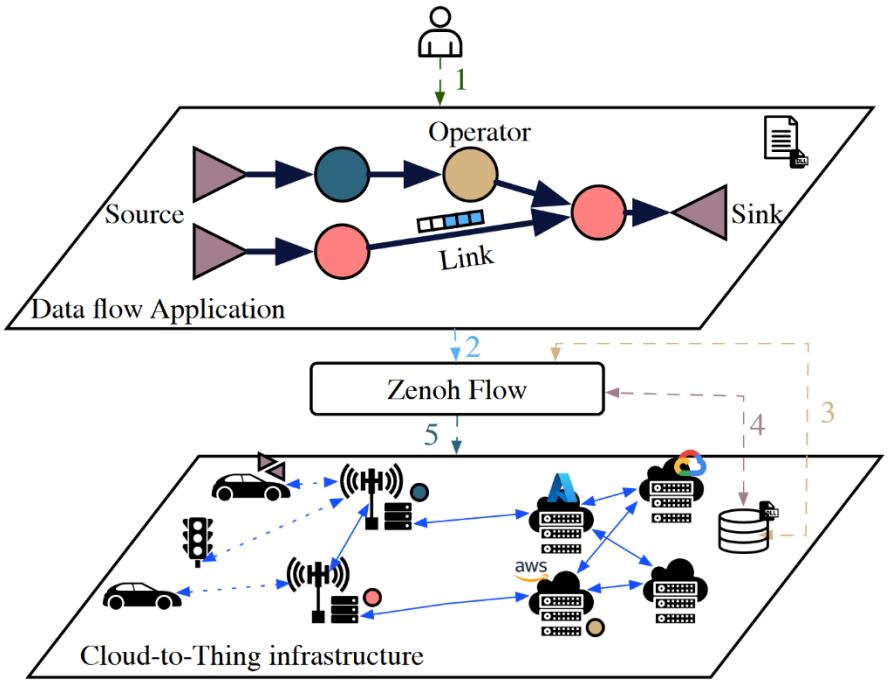

Figure 61: Zenoh-Flow end-to-end application deployment workflow (Baldoni, 2023)

In the DistriMuSe distributed platform, similar to the Zenoh-Flow workflow (Figure 61), application developers start by defining and implementing the nodes, specifying their ports, configurations, and implementation locations (containers). These implementations adhere to the DistriMuSe API, which provides helpers for receiving inputs and delivering outputs in a generic manner, without needing to be aware of the topics where the information is published or subscribed. The next step is to create a data flow graph by assembling the operators in a descriptor file using the SDL. This file serves as a contract that defines how nodes are interconnected, their requirements (e.g., hardware access), and how to automatically deploy the application across the DistriMuSe scenario.

The platform encourages code reuse by allowing operators to be shared and composed across applications, speeding up development and fostering collaboration. The descriptor and operators are then onboarded into the DistriMuSe control plane, where they are stored in a registry. Upon receiving an instantiation request, the platform loads the operators from the registry, creates the runtime graph, and establishes communication between the operators, all while following the rules specified in the descriptor.

5.8 Monitoring, observability, and benchmarking

The use cases envisioned for DistriMuSe involve complex scenarios featuring numerous devices that may dynamically join or leave the network due to factors such as power loss, disconnections, or other constraints. These scenarios also account for varying device capabilities and the potential for application failures, making robust and adaptive system design essential.

Therefore, it is crucial to monitor all components comprehensively, spanning from the application layer to the underlying infrastructure. This includes sensors and actuators, edge hubs, servers, hosts, network elements, edge/mist applications, and AI/ML models.

5.8.1 State of the art

Monitoring is critical for ensuring that devices are functioning correctly and that the applications and models deployed on them are operating as intended. This involves two key aspects: the physical hardware and the deployed software. Both must be observed to maintain system reliability and performance.

The term monitoring is commonly used, but the “observability” paradigm became popular recently. Monitoring focuses on reactive data collection, such as logs, while observability takes a proactive approach by interpreting the collected data to derive insights. A log provides human-readable information about an event, while a metric quantifies the event in its respective unit. Tracing captures the relationships between multiple events. Together, logs, metrics, and traces form the foundation of observability. Observability can be considered a superset of monitoring, offering a deeper understanding of system behaviour and performance (Iyengar, 2024).

In summary, while monitoring collects logs and metrics to track system health, performance, and failures, requiring a balance in what to log; observability extends this by analysing data to diagnose issues, with advanced platforms suggesting or implementing fixes to maintain service-level objectives (SLOs).

The network infrastructure considered in the different DistriMuSe uses cases are highly complex, combining hardware and software components that may generate vast amounts of data, making full human observability unfeasible. Intelligent observability platforms are essential for maintaining comprehensive situational awareness. Network observability focuses on monitoring network performance, predicting traffic trends, and ensuring reliable service delivery. Network monitoring may analyse different parameters such as the number of devices, the bandwidth available, the throughput, and the network latency.

Network nodes must also be monitored and observed to assess their availability and resource capacity. This information allows the DistriMuSe distributed platform orchestrators to make informed decisions regarding the deployment of application components. By monitoring nodes, it is possible to have real-time insights into the health of the system, enabling the identification of potential issues before they result in failures.

There are many tools that can be used for observability:

- Fluentd (https://github.com/fluent/fluentd)
- Grafana (https://grafana.com/grafana/)
- Graphite (https://github.com/graphite-project)
- Nagios (https://www.nagios.org/)
- Prometheus (https://prometheus.io/)
- SkyWalking (https://skywalking.apache.org/)
- Zabbix (https://www.zabbix.com/)

5.8.2 Monitoring in DistriMuSe

Monitoring and observability are essential tools for the DistriMuSe distributed platform. Nodes and network must be monitored and observed to assess their availability and resource capacity. This information allows the DistriMuSe distributed platform orchestrators to make informed decisions regarding the deployment of application components.

The DistriMuSe middleware, running on network nodes, will leverage a publish-subscribe communication paradigm (e.g., Zenoh) to publish data about resource availability and status. Additionally, it will integrate Kubernetes architecture and tools to manage this process. Metrics such as CPU usage, memory consumption, sensor availability, and application component utilization will guide the scheduling and deployment of applications.

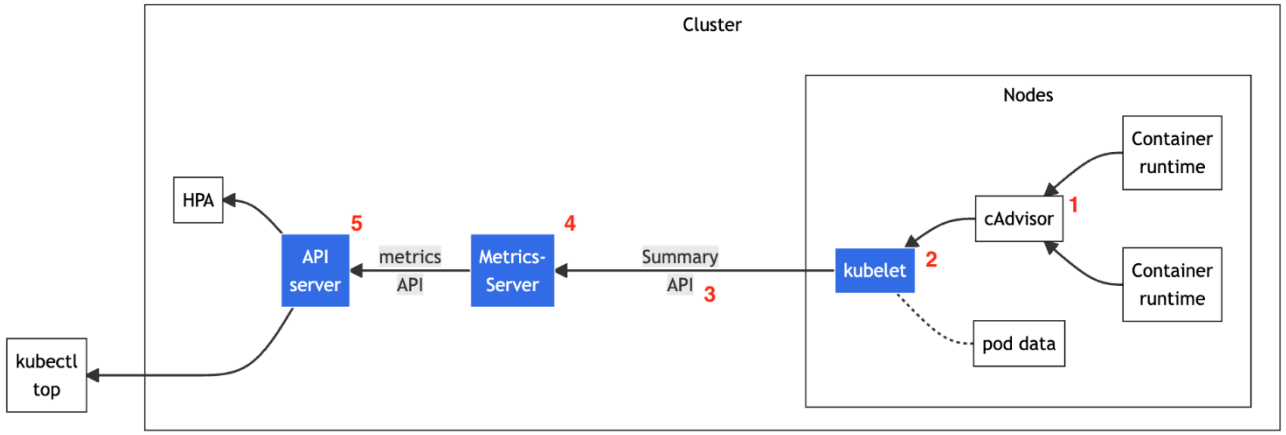

Figure 62: Metrics pipeline in Kubernetes

The metrics will be sent to a metrics server endpoint, running in the master node, to make scheduling decisions (following the Kubernetes model in Figure 62). Moreover, they will be used to predict resource demands and prevent over-provisioning in infrastructures used for more than one specific application.

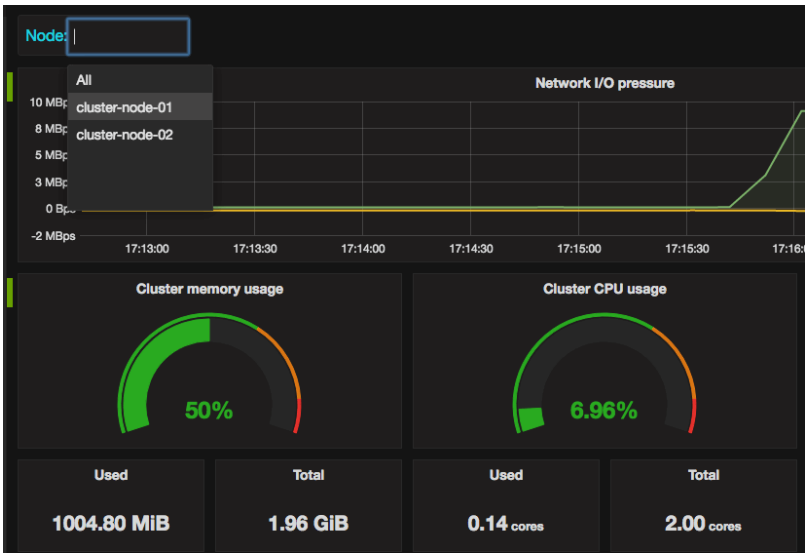

Figure 63: Metrics in Grafana

The metrics will be also offered to platform managers using tools such as Prometheus or Grafana, creating interfaces such as the shown in Figure 63.

The information will also be accessible through APIs, enabling system managers to benchmark their applications. Benchmarking, a structured method for evaluating system performance, involves executing targeted tests and workloads to gather quantitative data on how well a system performs under various conditions. This allows system managers and developers to create controlled scenarios to assess their implementations and compare them against alternative solutions.

5.8.3 Pilot specific monitoring and benchmarking

Monitoring for TSN Communications

Monitoring plays a pivotal role in distributed platforms, ensuring the smooth operation and maintenance of all system components. Particularly in high-performance communication networks, monitoring is indispensable for identifying issues, optimizing resource allocation, and maintaining seamless communication between devices.

An effective monitoring system provides a comprehensive understanding of network behavior in real time, that involves visualizing traffic flows, identifying the type of data being transmitted, monitoring system utilization, and understanding the network's topology. Additionally, it ensures detailed insights into device connections, interface statistics, and other operational parameters that are crucial for effective management. Such real-time awareness not only facilitates troubleshooting but also enables proactive optimization of network performance.

For efficient management, the integration of the monitoring system with the orchestrator is vital, allowing the orchestrator to have a centralized view of the system, enabling it to allocate resources efficiently, balance loads, and synchronize operations seamlessly. The centralization of monitoring and control becomes especially critical in deterministic and configurable networks like TSN, where numerous parameters can be adjusted to enhance performance.

An example of a centralized monitoring tool could be a Central Network Controller (CNC), that is a specialized tool designed for monitoring and managing networks, and in some cases TSN networks. While the CNC is more important for deterministic networks, the methodology it employs can serve

as a reference for other types of systems, but the advanced capabilities make it particularly effective in managing configurable network parameters and optimizing performance.

The CNC provides a range of functionalities, including:

- Network Monitoring: The CNC collects and displays critical data from devices, offering a clear view of the system's status. It enables users to monitor metrics such as synchronization, traffic flow, and device health in real time.
- Centralized Configuration: Users may directly configure the TSN network through the CNC interface. This includes managing VLANs, TAS (Time-Aware Shaping), CAM tables, and QCI (QoS Class Identifier) parameters.
- Latency and Packet Loss Reports: By utilizing probes integrated into devices, the CNC could generate detailed performance reports, including measurements of network latency and packet loss to identify potential bottlenecks.
- Network Topology Visualization: The CNC could offer a graphical representation of the network topology, showcasing devices, interfaces, and their connections. This visualization is instrumental in understanding and optimizing the network's architecture.
- Exporting Monitoring Data: The CNC could support data export to external time-series databases, enabling deeper analysis or integration with external systems.

Figure 64: Representation of a network topology created using a CNC.

Figure 64 provides a representation of a network topology that could be created using a CNC. This visualization highlights some TSN devices and the connections between them.

In addition to topology monitoring, the CNC might enable the observation of many other parameters, including metrics at the system level, device level, interface statistics, latency measurements, and various communication parameters between devices. These capabilities are critical for ensuring that high-performance communication systems operate reliably and efficiently.

Benchmarking in UC1

In this use case, benchmarking serves as a crucial step for evaluating the performance of our human activity recognition system. For example, there are demonstrators utilizing data from six wirelessly connected IMU sensors. These sensors are fused together to capture complex, multi-dimensional

movement data from individuals operating in collaborative human-robot environments. Through benchmarking, we define baseline patterns of standard human actions within these environments, creating a reference framework against which the system can detect deviations that may signal anomalies or hazardous situations. By establishing these benchmarks, we can objectively assess the model's effectiveness in identifying abnormal movements or unsafe behaviours, essential for maintaining safety and workflow continuity in production and assembly settings. The benchmarking process also allows for iterative improvements in system accuracy and reliability, ensuring the model's responsiveness to real-time dynamics in human-robot collaboration.

Benchmarking in UC2

Benchmarking of methods in use case 2 is a mandatory step to ensure the safety of traffic participants based on reliable detection and classification using AI/ML methods. For benchmarking purposes of use case 2, we will utilize hardware platforms described in section 4.2. We are planning to use semi-automatically annotated real-world data from busy intersections and crosswalks. The data used will be coming from multiple sensors: FMCW radar, RGB camera, and LiDAR. We will employ common detection metrics to ensure the accuracy of detections. We will also benchmark the inference time on the selected platforms, such as NVIDIA Jetson.

Benchmarking in UC3

Using robots, as intended for the use case 3, represents a challenging test-bench for the distributed platform. Accurate control of robotic arms requires communication of data in the range of 1kHz. A single robotic arm will be generating several streams of raw data with the actual state of every joint. Considering a single robotic arm with 7 degrees-of-freedom will transmit the current position, velocity, acceleration and even applied torque for every joint at a frequency of 1kHz. In parallel to this communication, the RGB cameras covering the scene will transmit streams of RGB images (at frequencies of tens of Hz). These images will have to be analysed in real-time to track the position and movement of the operators in the area surrounding the robot arm. Finally, all these analysis modules will generate control commands to operate the robot in a safe way. High communication latencies are not allowed as they could represent a risk for the integrity of the operator and the facility.

5.9 Security

In the last few years, the unstoppable digitization process brings new challenges in terms of security and privacy, with thousands of millions of IoT devices connected all around the world (IoT Analytics, 2024). This has led organisations to face many challenges related to security such as authentication, authorisation, availability, identity management, confidentiality, data integrity and privacy (Belapurkar et al., 2009). Distributed platforms must be carefully designed to address these challenges.

Upcoming European regulations, specifically the Cyber Resilience Act (CRA), aim to enforce secure and maintainable products in the Information technology (IT) and Operational technology (OT) markets, not only by focusing on the security-by-design principle, but considering all the phases of the product lifecycle, which include the tracking of cybersecurity threats, the implementation of mitigation measures and intuitive and robust in-field delivery of patches to the customer's devices (European Commission, 2024).

Thus, in addition of the consideration of security guidelines during the design and development of the products at different levels, the implementation of local and remote secure firmware/software updates (ensuring the authenticity and integrity of the firmware/software) are critical within the

context of the CRA, in order to protect the systems against new vulnerabilities discovered along their lifetime. This is especially relevant for distributed platforms composed of heterogeneous elements (cloud, edge, IoT nodes…), like the distributed platform proposed in DistriMuSe.

5.9.1 State of the art

Authentication ensures that only legitimate users, devices, or applications can access the platform by verifying their credentials through secure methods such as passwords, tokens, or biometrics. Building on this, authorization determines the permissions and access levels granted to authenticated entities, restricting their actions based on predefined roles or policies (Belapurkar et al., 2009).

Availability ensures that services and resources remain accessible to authorized users, even under challenging conditions like hardware failures or cyberattacks, emphasizing system reliability and fault tolerance. Identity management serves as the foundation for secure interactions, enabling distributed platforms to handle user identities and credentials efficiently across cloud, edge, and mist layers. By integrating these elements, distributed platforms can maintain secure, resilient, and trusted operations across diverse and geographically distributed environments.

Confidentiality ensures that sensitive data is protected from unauthorized access through encryption and secure transmission protocols, while data integrity guarantees that information remains accurate and unaltered during transit, storage, or processing, preventing unauthorized tampering or corruption. Privacy emphasizes safeguarding personal or sensitive data, ensuring compliance with legal frameworks and respecting user rights by implementing data minimization and secure storage practices.

Security modelling

The DistriMuSe platform integrates various sensors to monitor human interactions with the environment and make decisions to ensure safety. The platform's architecture should also prioritize cybersecurity. To achieve this, the system will be modelled using the ArchiMate tool and The Open Group Architecture Framework (TOGAF) to ensure that security goals derived from relevant standards are met.

ArchiMate (OpenGroup, 2024a) is an open and independent enterprise architecture modelling language that supports the description, analysis, and visualization of architecture within and across business domains. It is fully aligned with the TOGAF framework, making it a powerful tool for enterprise architects. ArchiMate is a high-level architecture language that helps stakeholders understand the impact of design choices and changes and provide a clear and structured way to model complex systems.

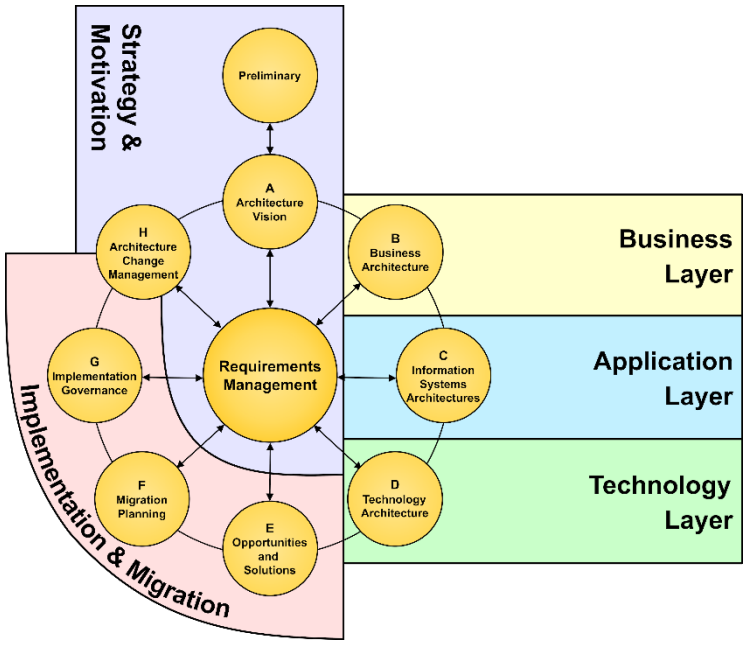

Figure 65: Integration of the TOGAF architecture development method and the ArchiMate language (OpenGroup, 2024a)

ArchiMate can be effectively used to model the security architecture of the DistriMuSe. By incorporating security concepts into the architecture, it ensures that security is considered from the outset, rather than as an afterthought. This approach is known as “security by design.” ArchiMate allows for the integration of security requirements into the overall architecture, ensuring that all aspects of the system are secure. By modelling the security architecture using ArchiMate and integrating it with the TOGAF framework, as shown in Figure 65, the platform can achieve several benefits. First, security is built into the system from the beginning to reduce the risk of vulnerabilities. Second, ArchiMate’s visual models help communicate security requirements and architecture to all stakeholders, ensuring a common understanding. Third, aligning with security standards and regulations ensures that the platform meets legal and industry requirements. Finally, the architecture can be easily adapted to accommodate new security requirements or changes in the environment.

The Open Group security guide (OpenGroup, 2024b) regards security architecture as a cross-cutting concern that interacts with all four of the enterprise architecture (i.e., Business, Data, Application, and Technology). Also, the Open Group’s white paper for modelling enterprise risk management and security with ArchiMate (OpenGroup, 2015) provides specific guidance for risk and security modelling by extending the TOGAF standard and using the ArchiMate language. Following this guideline, high-level steps to the security architecture of DistriMuSe are as follows:

- Identify Security Requirements
- Define Security Policies.
- Model Security Architecture
- Validate and Refine

We put a special focus on innovate technologies for data protection. Hyperledger Fabric and Decentralized Identifiers are two technologies that address several of the main issues related to data security, data privacy, and data integrity. Hyperledger Fabric is a distributed and permissioned

blockchain, which means that all participants are authenticated and authorised, thus reducing the risk of unauthorised access and potential breaches. Furthermore, Hyperledger Fabric's ledger consists of an immutable blockchain, which records transactions permanently, thus ensuring that data remains consistent, reliable, and trustworthy over time. On the other hand, Decentralized Identifiers offer methods to establish digital identities that do not require a central authority, which ensures that sensitive data is only accessible with proper authorisation, while also enhancing privacy by avoiding central data storage. Together, Hyperledger Fabric and Decentralized identifiers provide a robust framework that addresses the data security, integrity and privacy issues.

Lightweight Blockchain

Hyperledger Fabric (Hyperledger, 2024) is an open-source and permissioned DLT platform designed for use in enterprise-level applications, as its modular architecture provides high degrees of confidentiality, flexibility, resiliency, and scalability, among other capabilities that make Hyperledger Fabric differentiate from other popular distributed ledger or blockchain platforms. Furthermore, Fabric is the first distributed ledger platform to support smart contracts authored in general-purpose programming languages such as Java, Go, and Node.js. Another differentiating feature of Fabric is its permissioned platform, which means that participants are known to each other, rather than anonymous, and therefore untrusted as other DLT platforms such as Bitcoin or Ethereum, where it is not possible to track the identity of the responsible of a specific transaction. Moreover, Fabric supports two consensus protocols: Crash Fault-Tolerant (CFT) when Fabric is deployed within a single enterprise, and Byzantine Fault Tolerant (BFT) when Fabric is deployed in a multi-party, decentralized use case. In terms of cryptocurrency, Fabric leverages protocols that do not require a native cryptocurrency, which significantly reduces processing and transaction confirmation latency and the absence of cryptographic mining operations.

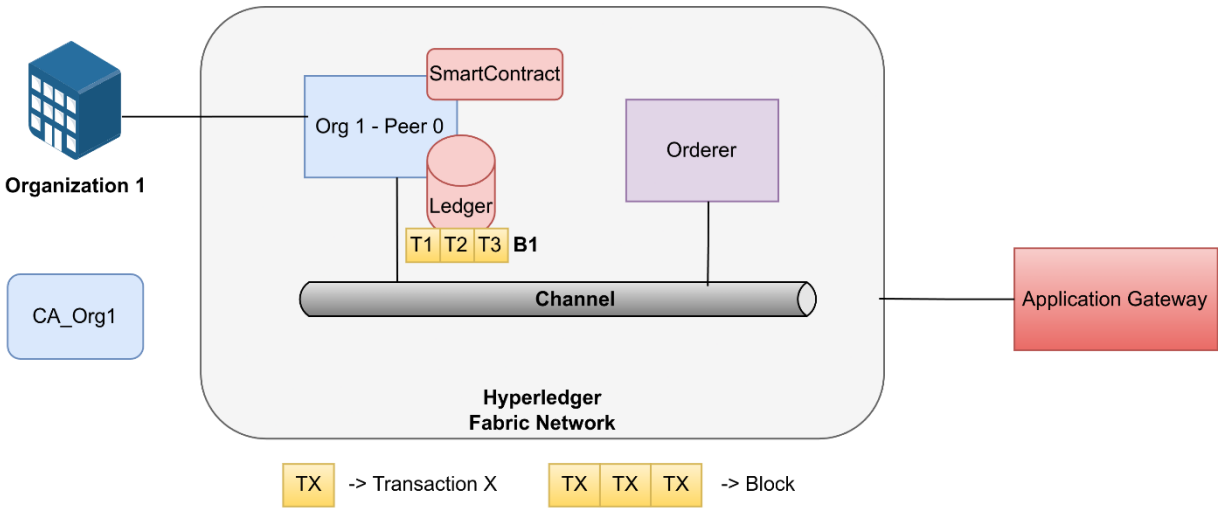

Figure 66: Hyperledger Fabric

Figure 66 shows an overall architecture of the components that participate in a Hyperledger Fabric deployment. One of the most crucial components is the Smart Contract, also called 'Chaincode', which is code — invoked by a client application external to the blockchain network — that manages access and modifications to a set of key-value pairs in the World State via a transaction. A Chaincode defines the transaction logic that controls the lifecycle of a business object contained in the world state. The World State is one of the two parts that form the ledger, and it is a database that holds the current values of a set of ledger states, which facilitates external applications to directly access the current value of a state instead of having to calculate it by traversing the entire transaction log. The

other part that forms the ledger is the blockchain, a transaction log that records all changes that have resulted in the current world state. The main difference between the two parts that form the ledger is their data structure, as the blockchain is immutable, which means that once a transaction is written, it cannot be modified. On the other hand, states in the world state can change frequently (states can be created, updated, and deleted).

Decentralized Identifier (DID) (Reed et al., 2020) is a standard from W3C (W3C, 2024) to implement decentralized authentication systems. DID is a new type of identifier that allows verifiable, decentralized digital identity. A DID refers to any subject (e.g., a person, organization, thing, data model, abstract entity, etc.) as determined by the controller of the DID. DID is a globally unique identifier, auto-generated or issued by a decentralized system (i.e. Blockchain), that acts as a proof of property for a digital identity. Unlike traditional identifiers that require a centralized authorization registry, DIDs are cryptographic verifiable universal unique identifiers, which mean that DIDs can identify organization, abstract entities, data models, or IoT devices. In terms of structure, DIDs are text strings divided into three parts. The first part is the DID's URI identifier, which indicates that the text string is a DID. Then, the second part is the DID Method identifier, which indicates the method used to create, resolve, and manage the DID, e.g., if Hyperledger Fabric is the Verifiable Data Registry that stores and manages DIDs, then the DID Method would be 'fabric'. Finally, the third part is the DID method-specific identifier, which is the identifier that uniquely identifies the DID's subject within the specified DID method. Moreover, DIDs are associated with DID documents, which are JSON-LD objects that contain data describing the DID subject, including a mechanism to obtain the public keys associated with that DID, which are used for verification and proof of possession purposes. Figure 67 shows the overall view of the architecture of DIDs.

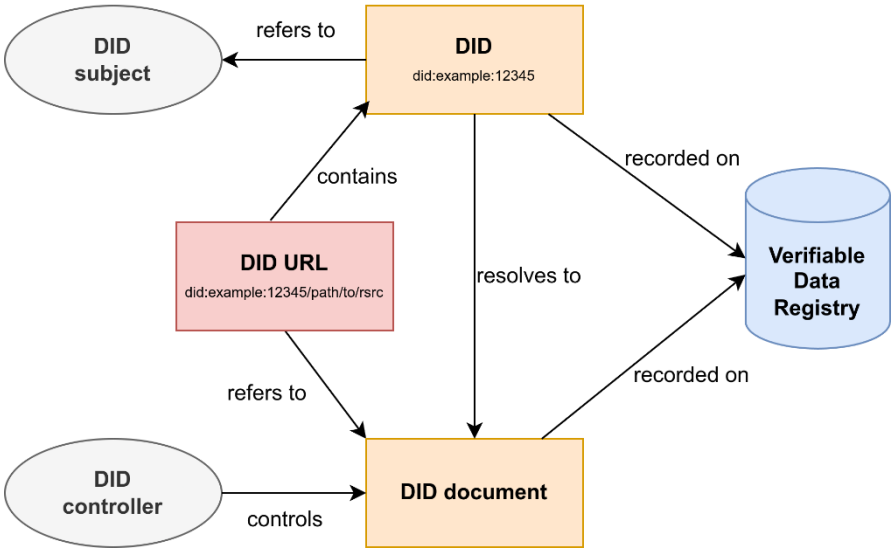

Figure 67: DID overall architecture view

In-vehicle intrusion detection systems

In the automotive industry, the raise in the use of increasingly connected vehicles opens the door to new ways of cyberattacking. This is extremely serious considering that the vehicle's internal network interconnects various critical modules, such as the braking system, steering, engine control, and

driver assistance systems, as well as external interfaces like Wi-Fi, Bluetooth, and mobile networks. Given this context, there arises the need to develop and improve the so-called in-vehicle intrusion detection systems (IVIDS), which are specialized systems designed to detect and prevent unauthorized access and cyberattacks through continuous monitoring of the vehicle's internal network or Controller Area Network (CAN). Moreover, with the advent of not-so-distant autonomous cars, where the complexity and interconnection of systems is even greater, the role of an IVIDS is even more crucial to ensure the integrity of communications between different systems and to protect the vehicle from potential intrusions that could jeopardize both its operation and the safety of its occupants.

The contemporary advancements on in-vehicle IDS focus on the enhancement of security protocols for CAN, which is intrinsically susceptible due to the absence of integrated security features. Recent innovations encompass the formulation of lightweight software-based IDS that employ message cycle time analysis and plausibility assessments to identify anomalies and malicious activities within the CAN bus (Bresch & Salman, 2017). These methods are designed to minimize overhead and at the same time they maintain robust detection capabilities by monitoring the timing and sequence of CAN messages to detect unusual patterns that may indicate the presence of an attack.

Moreover, pioneering methodologies such as VehicleEIDS exploit external voltage signals to monitor for intrusions without reducing CAN bandwidth, reaching an accuracy rate exceeding 97% in the detection of abnormal signals (Xun et al., 2021). Additionally, strategies utilizing density ratio estimation in conjunction with neural networks have been introduced to identify and elucidate the injection of malicious packets, thereby providing a more sophisticated comprehension of attack vectors (Tanaka et al., 2019). Alternative approaches concentrate on entropy analysis of CAN message frequencies to pinpoint potential assaults, thereby delivering timely notifications to drivers (Kim et al., 2017).

On the other hand, other studies include the development of Binarized Convolutional Neural Networks (BCNN) that enhances detection accuracy (94.19% to 96.82%) while drastically reducing memory usage and processing time compared to traditional CNNs (Zhang et al., 2024). This reduction in resource utilization renders BCNNs particularly advantageous for application in environments with limited resources, such as in-vehicle systems. Furthermore, unsupervised methods utilizing autoencoders and fuzzy C-means have shown promising results, achieving up to 84.05% accuracy across various datasets, eliminating the need for labelled data (Kabilan et al., 2024). Multiscale histogram approaches have also been proposed, capturing message identifier frequencies to improve traffic analysis and anomaly detection (Baldini, 2023). This method allows for the capture of both short-term and long-term patterns in the data, making it more adaptable to a wide range of attack scenarios. Moreover, lightweight models with significant parameter reductions have been developed, maintaining high accuracy while minimizing resource consumption (Kristianto, 2024).

Custom-quantized neural networks have demonstrated exceptional performance, achieving 99.9% accuracy. These networks are meticulously tailored to identify a multitude of attack vectors with minimal latency, thereby facilitating the prompt recognition and alleviation of real-time threats. The implementation of quantization methodologies diminishes the precision of model parameters, thereby enhancing the computational speed and reducing power consumption, all while maintaining the capability to accurately detect intrusions (Khandelwal & Shreejith, 2023).

Another interesting solution presented in Campos et al. (2024) proposes a method for misbehaviour detection in intelligent transportation systems using Federated Learning (FL). This approach allows multiple vehicles to collaboratively learn from data without sharing sensitive information, addressing

privacy concerns associated with centralized systems, while achieving an overall accuracy of 93% with an optimized Multilayer Perceptron (MLP).

Secure software updates

Distributed platforms require reliable mechanisms for delivering remote software updates to the nodes of the system. This is critical for a secure and sustainable long-term operation to deliver the patches required for fixing bugs or protecting the system against new vulnerabilities discovered once the system is already deployed in the field. These updates apply not only to the high-level services and applications running on the nodes but to the operating system itself.

Nowadays, the most common approach consists in using Private Key Infrastructure (PKI) for providing the updates, using a secure bootloader that supports cryptographic operations for authentication and signature checks.

An example of the update procedure is shown below:

1. The patch is signed at the server side.
2. The device validates the identity of the server to assure that it is the actual server.
3. Optionally, the server may validate the identity of the connected device.
4. The patch is retrieved by the device and stored in its secondary storage.
5. The system is automatically restarted in bootloader mode, that will do the appropriate checks before applying it.
    - The integrity (checksum) of the patch is checked.
    - The signature of the patch is checked.
    - The patch is applied.

Several open-source initiatives aiming to ease the implementation of more sophisticated update methods, with a focus on Linux systems, have appeared in the last years, such as RAUC - Robust Auto-Update Controller (RAUC, 2024), SWUpdate - Software Update for Embedded Linux Devices (SWUpdate, 2024) and OSTree (OSTree, 2024). They consider additional layers of security, given the different elements that compose a Linux distribution: a trust of chain in which one element of the distribution authenticates the next element in the booting sequence (bootloader → kernel → filesystem → software packages). Some improvements can also be incorporated over the base solution such as incremental updates to save bandwidth.

Recent works in the literature explore novel mechanisms consisting in using a peer distribution strategy for the patches by using blockchain (Witanto, 2020).

5.9.2 DistriMuSe security mechanisms

DistriMuSe's distributed platform should support the following features for providing a proper level of security:

- Authentication: The platform should support the authentication of both users and devices, through well-known authentication mechanisms, such as passwords, certificates or tokens.

- Authorisation: The platform should support Role-Based Access Control, to regulate the permission to access the different resources depending on the role assigned to the user or device.

- Identity management: The platform must manage user, device and application identities, to be used by authentication and authorisation mechanisms. Standard centralized token-based solutions for identity management, such as JSON Web Tokens (JWT, 2024) or OAuth 2.0 (OAuth, 2024), can be used.

- Availability: The platform should include mechanisms to ensure a higher level of availability demanded by some use cases, such as firewalls to avoid Denial of Service (DoS) attacks and redundancy to mitigate the impact of failing or compromised devices.

- Confidentiality: The platform must provide confidentiality by means of secure communications. End-to-end encryption, such as Transport Layer Security (TLS), should be provided in every communication between different elements of the distributed platform.

- Secure software updates. The platform will include mechanisms to support secure software updates. Robust solutions such PKI certificates will be used, leveraging secure a bootloader that supports cryptographic operations for authentication and signature checks.

Security modelling for the DistriMuSe platform

A high-level security architecture model of DistriMuSe is shown in Figure 68. This figure depicts an abstraction of the distributed platform and indicates how security elements interact with the platform's architecture. Every element in the architecture can be treated as an “asset at risk”, associated with some vulnerabilities determined through vulnerability assessment. Vulnerabilities are associated with the loss event and threat event, triggered by the threat agent. The risk of this loss event is estimated through risk assessment, which yields some security goals, policies, and requirements. Security requirements are in turn translated into control measures implemented on the architecture at different levels, reducing the vulnerability of the asset related to that threat.

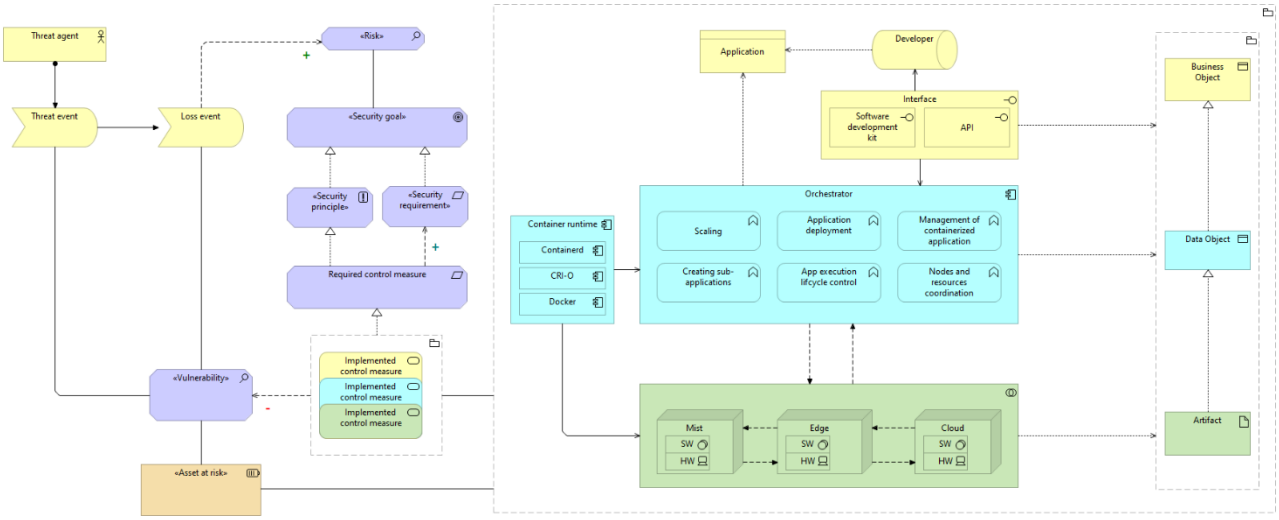

Figure 68: A high-level security architecture for the distributed platform through the ArchiMate language

The proposed architecture will grow in further detail as the platform design becomes more detailed. The motivation level of the architecture (purple elements) may also be used to model the security

requirements derived from security standards to describe how these requirements are related to each other and to the system. Moreover, the threat elements in the model will be used to visualize the threat modelling results, indicating how they impact the system and how control measures can mitigate their risk to the system.

5.9.3 Pilot-specific security mechanisms

Blockchain-based lightweight authentication for medical IoT devices

A blockchain-based lightweight authentication security mechanism consists of the use of DIDs to uniquely identify IoT devices, applications, or any other element of the Distrimuse platform, and the use of Hyperledger Fabric as the Verifiable Data Registry to store and manage DIDs (Lux et al., 2020). Using Hyperledger Fabrics offers several benefits as it is a DLT platform more optimized for IoT networks than other DLT technologies (Pajooh, 2021). One of the main reasons that makes Fabric more optimal for IoT networks is its modular architecture, which allows Fabric to be integrated with various IoT devices and applications. Also, the consensus mechanisms offered by Hyperledger Fabric are more efficient compared to the energy-intensive Proof of Work used by some other DLTs, which makes Fabric consensus mechanisms more suitable for IoT environments. In this way, DIDs are stored in the blockchain as key-value pairs, where the DID identifier acts as the key, and the DID document acts as the value. The use of DIDs allows the authentication of entities (IoT devices, applications, etc.) with cryptographic proofs such as digital signatures. To get to the point of being able to authenticate some entity using DIDs, there is a series of steps that need to be performed:

1. The entity needs to obtain a DID, alongside a DID document, a public key, and a private key. This can be automatically generated or provided by a decentralized platform.

2. The entity needs to register the DID and DID Document in the Verifiable Data Registry, which in this case is Hyperledger Fabric. The DID Document will not contain the public key itself, but rather the methods to obtain it. The private key is stored within the entity itself and must not be shared.

3. The entity starts an interaction with another entity, including a signature for authentication. To authenticate the entity that is making the request and to verify that the request is valid, the second entity will need to obtain the public key of the requester entity, which will be done by retrieving the DID Document associated with the requester's DID from the blockchain and deriving the public key with the specific methods indicated in the DID Document.

4. The other entity verifies that the signature is valid. If it is valid, the second entity can assure that the requester entity is trustworthy and that the request has not been tampered with. If it is not valid, the second entity must not trust the requester entity and discard the request.

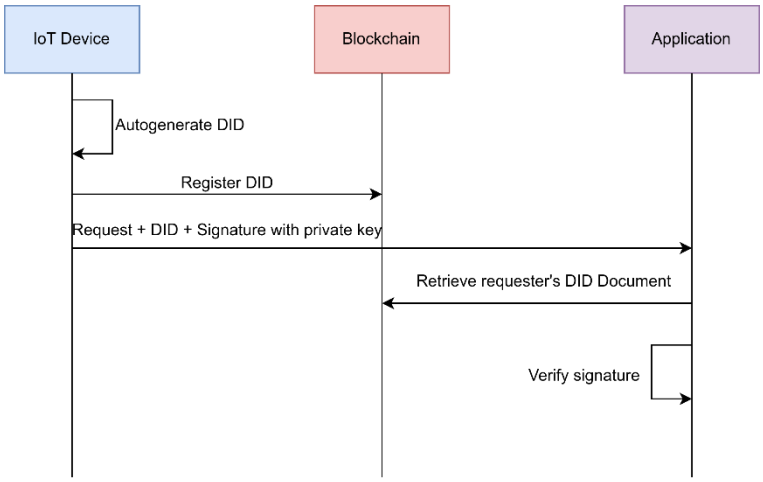

Figure 69: IoT device authentication using DIDs and Hyperledger Fabric.

Figure 69 shows an example of how the authentication process is performed. Storing DIDs in Hyperledger Fabric provides a secure, scalable, and decentralized way to manage the identity and authentication of devices. Furthermore, public and private key cryptography offers confidentiality, authenticity, and data integrity. As the digital signature can only be generated by the entity that holds the private key, any alteration in the message after signing will invalidate the signature, and therefore no trusted channel can be established between the entities. In this way, the storage of DIDs in Hyperledger Fabric, a decentralized system, eliminates the need for a central authority, giving IoT devices more control over their identities while maintaining a high level of trust and security.

Federated learning-based in-vehicle intrusion detection system.

For the design of the IDS, a Federated Learning approach will be used. This technique employs a distributed learning model, which allows training a global artificial intelligence model, such as an Artificial Neural Network (ANN), without the need to centralize all vehicle data on a server or common point. Instead of collecting data from the different vehicles, then creating a dataset, and then using it to train the neural network; the next steps are followed. First, the global neural network to be used is defined and initialized with random weights on the server. These weights are then sent to each vehicle, which trains its own version of the neural network with its own data for a small number of epochs. Once this is done, the modified weights from each vehicle are sent back to the server, which uses an aggregation strategy, such as the average of the weights, to integrate them all, establishing this average as the new weights of the global neural network. This process is repeated for multiple rounds until the global model converges. With the aim of visually showing the interaction between the different components of the system, Figure 70 is proposed. In this figure, two road-side stations are included as an intermediate step between the communication of the vehicles and the server.

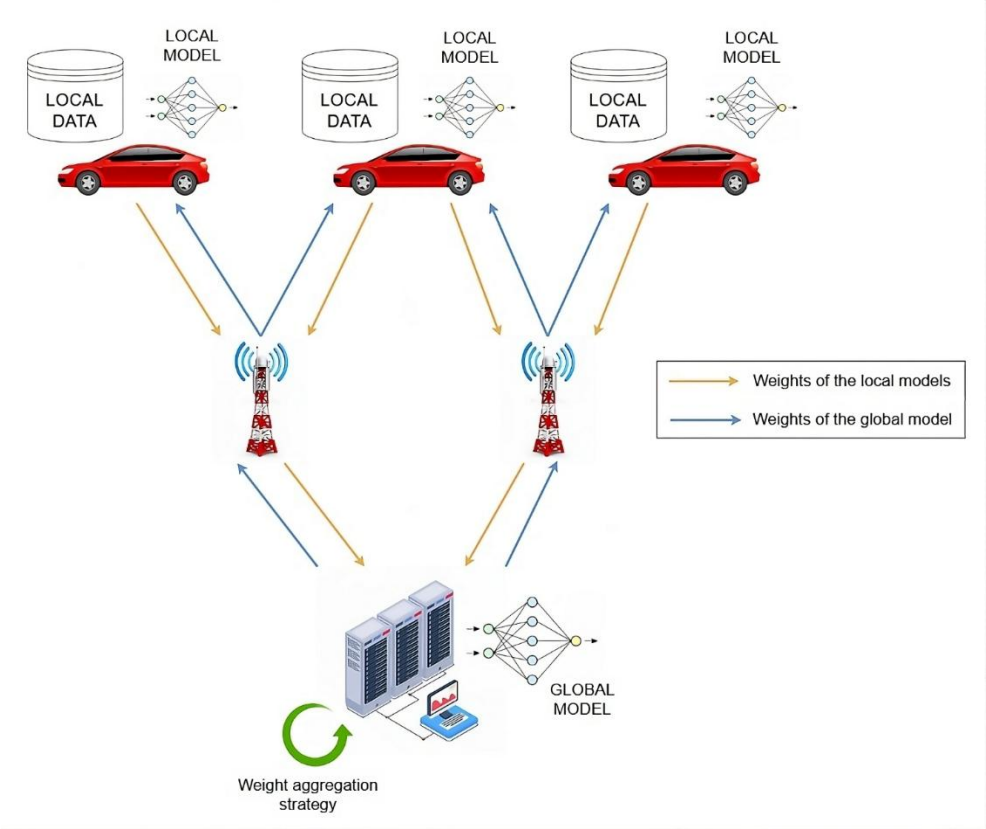

Figure 70: Operation of the Federated Learning-based IDS.

As an additional step to improve system security, differential privacy can be applied to the model weights when they are shared. This technique involves introducing controlled noise (usually Gaussian or Laplacian) to the model weights before each vehicle shares them back to the server. In this way, we ensure that small differences that may exist in the local data of each vehicle are not directly reflected in the weights being shared. However, while it is a way to increase the privacy and security of clients, the introduction of noise also results in a reduction in the accuracy and performance of the model, as the weights are not shared exactly as they are learned on the clients.

Considering all this general function, the proposed solution simulates an environment with multiple distributed vehicles for training an artificial intelligence model capable of detecting if a vehicle is benign, or if it is an attacker, in which case, five types of attacks are detected:

- Constant: the vehicle transmits a fixed position that does not change over time.
- Constant Offset: the vehicle transmits a position that is offset by a fixed amount from its actual position. This offset remains constant, so the reported position is always different from the real one by a fixed distance.
- Random: the vehicle transmits a random position within the simulation area, so the reported position can be very different from the real one and, therefore, highly unpredictable.
- Random Offset: the reported position is still random but constrained within a certain distance from the true position, making it more predictable than a complete random attack.

- Eventual Stop: the attacking vehicle behaves like a normal one for a while, sending real position data. However, after some time, the attacker changes and starts sending its current position repeatedly, which can lead to the impression that it has stopped.

The methodology involves applying machine learning techniques, including multilayer perceptrons and random forests, to evaluate their effectiveness for the problem. Furthermore, the use of differential privacy will be studied, and an attempt will be made to find a balance between the loss of precision and the increase in security it entails. If it is observed that it is impossible to find such a balance because differential privacy results in a significant reduction in the precision of the models, it will be discarded.

6 Design and simulation tools

This section describes the set of tools put in the hands of developers to simplify the experience and improve their workflow. The purpose of these tools is two-fold:

- **Simplifying Application Design and Creation**: Reducing the complexity of the creation of new applications by offering a set of tools to facilitate the design, configuration, deploy and reutilization of the application components and the links between them.
- **Optimizing Prototyping and Debugging:** Enabling faster prototyping, testing and debugging of the created applications and application components via the simulation different entities and replication of the behaviour of existing components

More specifically, the objectives of the tools presented in this section are the following:

- Simplify the design of modular, reusable application components.
- Streamline the creation of the deployments and the monitorization of their lifetime.
- Allow to monitor and replicate the behaviour of certain components to facilitate the debugging and the rapid prototyping of applications in the platform.

These tools are developed to facilitate the development of the different DistriMuSe's pilots and the seamless integration of the pilots with the platform.

6.1 Development environment and design tools

To assist developers in creating new distributed applications for the DistriMuSe platform, the project should provide several key tools:

- A specialized language designed to facilitate the description of application components, their configuration, and data flow.
- A programming environment that supports the language, helping developers write code and possibly offering visual tools to simplify or abstract the language.
- A set of tools to deploy the developed programs across the distributed system.
- A collection of debugging tools to help troubleshoot and refine the application during development

This section provides and overview of the design tools for DistriMuSe and design decisions taken in the distributed platform to facilitate the creation of applications. Some specific pilot tools are also presented.

Furthermore, a state of the art in the field of development tools in distributed environments and especially in the domain of dataflow programming is presented below.

6.1.1 State of the art

There are not many equivalent works in the literature that can be adopted by the DistriMuSe distribute platform. Nevertheless, other authors also consider that the dataflow programming model offers advantages for environments such as the supported by DistriMuSe, and proposed frameworks for the developers. The dataflow programming model enhances developer productivity by abstracting underlying system complexities, such as hardware, protocols, and functionality, into nodes within a flow. This abstraction simplifies application design by focusing on node connections and data processing. This separation of roles between “data processing” (application components) and “node connections” (dataflow programs) streamlines the process, allowing each to focus on their respective areas of expertise. Unlike previous approaches, this model avoids complexity in interfacing with drivers or sacrificing flexibility (Giang, 2015).

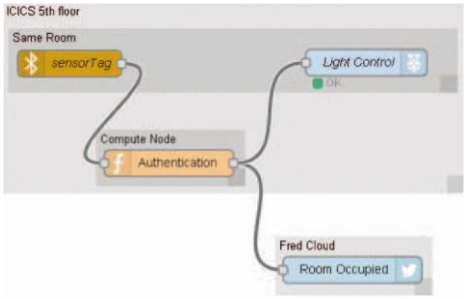

Figure 71: Development of distribute data flow applications with Node-RED (Giang, 2015)

In (Giang, 2015), the authors implemented the Distributed Data Flow programming model with a new dataflow application framework based on Node-RED, a visual dataflow programming language and runtime developed by IBM. Node-RED is built with JavaScript and Node.js technology. It provides a web-based visual editor with pre-built input, output, and processing nodes, allowing developers to create applications by connecting these nodes. The applications are executed on Node-RED's backend. While Node-RED was initially intended for single-device applications like Raspberry Pi, the authors extended it to support distributed dataflow applications across devices and the Cloud.

Their implementation involved Distributed Node-RED (D-NR) processes running across devices in local networks and servers in a Cloud infrastructure. A development server allowed developers to design the application flow using Node-RED's visual editor. All participating D-NR processes subscribed to an MQTT topic for flow updates. When an update was deployed, devices and Cloud servers pulled the latest version, parsed it, and determined which nodes should be deployed locally. Placeholder nodes were used to connect sub-flows across different devices.

In the paper, the authors address device heterogeneity. Each D-NR instance on a device describes its capabilities, including computational resources, network bandwidth, storage, and user-defined properties like location and device group. Constraints based on these capabilities allow developers to specify which devices a sub-flow should be deployed to. The design includes more complex constraints such as computational resources, network bandwidth, and storage, which balance flexibility and usability for device allocation based on time-sensitivity and communication delay.

There are other proposals providing at least a programming language. For example, Flask (Mainland, 2006) and ATaG (Bakshi, 2005) are data-driven programming models for Wireless Sensor Networks (WSNs) that abstract application logic into small processing units, forming dataflow graphs. However, they do not address vertical heterogeneity, i.e., leveraging device resources for application logic. Flask operates on homogeneous devices, while ATaG distinguishes devices based on services. The glue.thing (Kleinfeld, 2014) framework, also based on Node-RED, controls devices

within local networks or the Cloud but struggles with scalability due to its individual device management approach.

6.1.2 DistriMuSe development environment and design tools

To facilitate the development of distributed applications within the DistriMuSe platform, developers are encouraged to package their application components as containers. This approach ensures that each component is modular, scalable, and easily deployable across various environments. Alongside this, developers must provide a Domain-Specific Language (DSL) file, which outlines the components in use, the configuration settings, and the dataflow of the application.

The DistriMuSe platform supports a user-friendly graphical interface, enabling developers to upload their packaged components and the corresponding DSL file. This interface also simplifies the process of requesting deployment for the distributed application within the system. To further assist developers, the DistriMuSe project may offer tools that streamline the creation of DSL files without requiring in-depth knowledge of the DSL's low-level structure.

Additionally, the DistriMuSe distributed platform includes monitoring and debugging tools that provide real-time insights into the application's performance. These tools will be also offered to developers to help them to ensure that the distributed application meets the specific requirements and objectives of the use case, helping to the validation and troubleshooting process during development.

6.1.3 Pilot specific tools

Recording and playing back messages

Aiming to facilitate the integration of the different demonstrators into the distributed platform and the debugging of such tools, One of the most convenient tools of a distributed platform is the one to record and recover the activity communicated through the platform. Similarly as it is done with *rosbag* tool for ROS, this tool allows the users to record all the messages that are sent through a set of communication channels. These messages are recorded in a file, together with their timestamps.

The same tool is also used to replay the sequence of recorded messages starting from a different time. Multiple use cases can be identified for this tool, as follows:

- Generation of datasets for offline analysis and training of learning systems. Thus, the same modules which are implemented to run in the control loop can be used to generate datasets with no reimplementation required.
- Testing of modules with (previously) recorded data. Since this tool allows the injection of previously recorded data into the platform, modules can be tested in the same control loop with no reimplementation.
- Offline analysis of recorded data. The access to these datasets facilitates the debugging and profiling of the modules and the platform itself. By using a library, the developer could create a script to recover and analyse the messages recorded from the network.

Generation of a database of video sequences for training and validation of AI algorithms

As part of the UC3, ValeriaLab at the University of Granada will create virtual reality environments of logistics tasks involving close interaction between collaborative robots and humans. To make the interaction safer, the simulations will be used to generate a database of video sequences and raw data that, in turn, will serve as the basis for training and validation of AI algorithms. These simulations will be in charge to generate realistic data of all the devices (sensors and actuators) that will interact in the real tasks, such as robotic arms, end-effectors, transport vehicles, RGB cameras and, even, the workers that might interact with the robots.

The simulations will be based on Unity 3D, a simulation engine that, in its origin, was mainly popular for videogame development, but more recently has become more frequent for industry simulation. By connecting the simulation framework with the distributed platform, all the sensors and video streams will be made available to other nodes which need this data to operate.

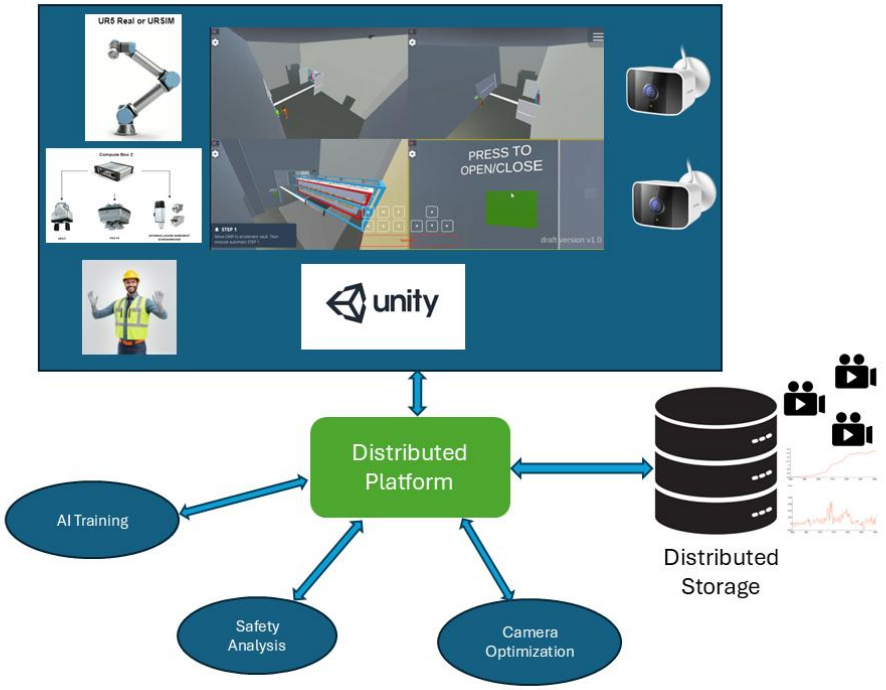

Figure 72: Different modules included in the database generation for UC3.

During the first cycle of development the virtual environments will generate synthetic data to train and to test the rest of the modules in the architecture (e.g. operator tracking systems, explainable AI systems, etc.). During the second cycle of development all the modules will work in closed-loop with the physical devices (e.g. the robotic arm, the human operators, the physical cameras, etc.), generating smart control commands for the robotic systems. To facilitate a soft transition from the first cycle (simulation) to the second cycle (real robots), the distributed platform must support the inclusion of simulated sensor sources in the distributed architecture, as well as other tools to record the communication in the network, producing a database that can be analysed off-line.

6.2 Testing environment and simulation tools

DistriMuSe aims to provide a set of tools to facilitate the testing of applications, alongside an additional set of tools for simulating different entities and components of an applications. These

tools allow to streamline the testing, rapid prototyping of applications and the promote the port of applications to the platform.

Being more specific, the suite of tools offered by DistriMuSe in this regard can be categorized into two different groups:

- **Testing Tools**: These tools are designed for testing, debugging and validate the functionality of already developed components. Helping to resolve issues and ensure the proper functioning of the applications.

- **Simulation Tools**: These tools aim to provide the developer with the replicate the entities and provide the means of expose the application a controlled environment. These tools allow the system and applications to be exposed to specific scenarios and environments in a controlled manner. This improves the experience of creation and enhancement of applications.

This section includes a review of the state of the art and a description of some specific tools for different use cases and pilots. The state of the art, presented below, explores the use of simulators for testing and prototyping complex software systems.

6.2.1 State of the art

Simulation levels can vary from lightweight software-only simulations to more sophisticated Hardware-In-the-Loop (HIL) simulations. Software-In-the-Loop (SIL) is a basic simulation environment where everything is simulated in software without needing specific hardware, with hardware requirements emulated. SIL is useful for quick testing cycles, allowing developers to test software and hardware simultaneously in a low-cost setup. In contrast, HIL uses real hardware components of the vehicle, with software testing interacting with real systems. HIL is typically more expensive and complex but serves as a final testing phase before real-world testing (Ruusiala, 2022).

3D game engines like Unity and Unreal Engine, when combined with specialized simulators such as CARLA and AirSim, enable the creation of highly realistic and customizable simulations. These platforms can generate detailed sensor data, including outputs from complex devices like cameras and LiDARs, leveraging advanced technologies initially designed for video games to support diverse simulation needs.

DistriMuSe scenarios are diverse, so we can’t have a unique simulation environment. For the sake of simplicity, we analyse how CARLA (Dosovitskiy, 2017) can be integrated with a distributed environment system enabling different kind of simulations. This model can be used to develop other environments or to create tools to simulate part of the scenarios.

CARLA is an open-source driving simulator designed for developing autonomous driving systems, primarily in the automotive sector but adaptable for other applications like heavy machinery automation. It supports both Software-In-the-Loop (SIL) and Hardware-In-the-Loop (HIL) testing. Key features include realistic urban environment simulation, traffic and pedestrian modelling, and advanced sensor simulation.

CARLA employs a client-server architecture. The server, built on Unreal Engine 4 (UE4), handles the 3D world, physics simulation, and sensor data rendering. It updates the world state, computes interactions, and manages the physics of various simulation entities, referred to as actors. Actors include vehicles, pedestrians, sensors, and objects like traffic lights and signs. “Ego vehicles”, which are controlled externally by autonomous driving systems, are a special subset of vehicles. The server

communicates with clients using a plugin and TCP/IP protocol. Clients, which operate separately from the server, manage actor logic, adjust environmental conditions, and execute scenario controls. They can connect to the server using APIs available in Python or C++, or through the Robot Operating System (ROS/ROS2) bridge, which translates API interactions into ROS-compatible formats. This architecture allows distributed simulation setups, with the server and client running on different machines.

Actors are at the core of CARLA's simulation capabilities. These vehicles can function as “ego vehicles” or as part of autonomous traffic controlled within the simulation. Sensors, another critical component, provide data from the simulated environment. These include RGB cameras, LiDARs, radars, and collision detectors, all of which can attach to vehicles and send data to clients for further analysis. Traffic lights, pedestrians, and other road elements interact realistically with vehicles, contributing to a dynamic and immersive simulation environment.

The simulator provides a variety of pre-built maps designed for urban and highway scenarios. These maps, based on the OpenDRIVE standard, feature detailed road networks and infrastructure. Users can create custom maps using the Unreal Engine editor or by generating them from OpenDRIVE files. Environmental conditions such as weather, lighting, and atmospheric effects can be customized to simulate various scenarios, although some effects like snow are not yet supported.

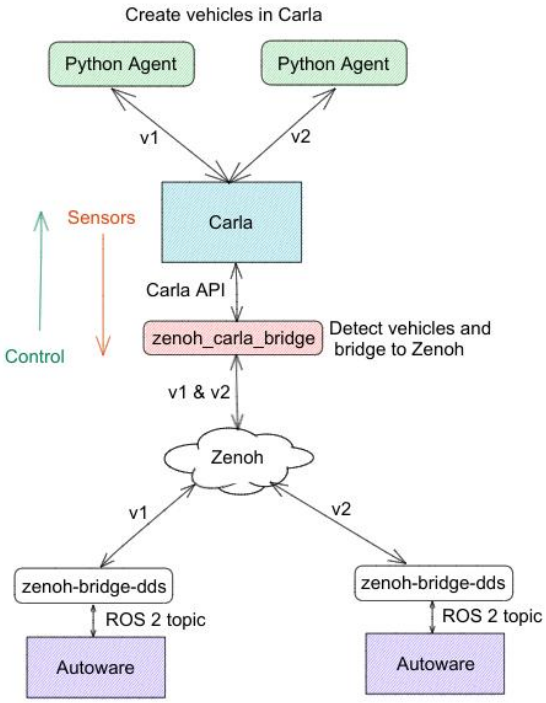

Figure 73: Diagram about the connection of CARLA with a Zenoh network (Kuo, 2023)

Traffic and scenario simulation tools further enhance the simulation. The Traffic Manager enables autonomous behavior for vehicles and pedestrians, managing routes, obeying traffic rules, and avoiding collisions. For more complex scenarios, the Scenario Runner tool allows developers to define and execute detailed driving scenarios using Python scripts or OpenSCENARIO files. Scenic, a probabilistic programming framework, and SUMO, a microscopic traffic simulation tool, provide additional options for creating and controlling traffic dynamics within the simulation.

CARLA has been integrated with Zenoh to test an autonomous driving application before its deployment in a real-life scenario (Kuo, 2023).

Figure 73 shows how CARLA agents were connected to a Zenoh network, allowing the integration of external applications using Zenoh to be integrated in the simulated world.

In this scenario, the zenoh_carla_bridge interfaces with CARLA's API to retrieve and set simulation data while connecting to the Zenoh network. On the other end, the zenoh-bridge-dds processes Zenoh data originating from CARLA and translates it directly into ROS 2 messages for Autoware's corresponding topics. This ensures that simulation data flows smoothly into Autoware without manual topic management. Similarly, control commands and data generated by Autoware can be sent back to CARLA via the same bridge mechanism, completing the loop.

A critical advantage of using Zenoh is its "scope" feature. This allows identical ROS topics from multiple Autoware instances to remain isolated, preventing conflicts and ensuring proper communication. This capability is a key reason for adopting Zenoh, as it simplifies the management of multiple instances and streamlines the integration process.

6.2.2 Tools in DistriMuSe

DistriMuSe will offer a range of tools to help developers simplify the development of their algorithms. These tools primarily focus on emulating sensors or simulating complex environments. While they are not specifically designed to run the DistriMuSe distributed platform, they can be used to test and validate individual components of applications in a controlled setting.

UC2

In the frame of UC2, RE:LAB provides an industrial-grade driving simulator featuring AVSimulation SCANeR™ simulation software. The driving simulator incorporates a range of virtual sensor simulations to replicate real-world vehicle data collection. These include radar for object detection and distance measurement, GPS for location data, and LIDAR for detailed environmental mapping. Various camera types are simulated, including standard and infrared, to capture visual data under different conditions. Ultrasonic sensors provide short-range detection capabilities. The system also simulates light sensors for ambient light detection, road sensors for lane and road feature recognition, and an Inertial Measurement Unit (IMU) for acceleration and orientation data. Vehicle-to-everything (V2X) communication is replicated to simulate information exchange between the vehicle and other entities. Additionally, the simulator generates comprehensive vehicle telemetry data, including speed, acceleration, steering inputs, and other dynamic parameters.

The simulator is designed to recreate complex traffic scenarios involving pedestrians, cyclists, and motorcyclists, collectively known as Vulnerable Road Users (VRUs). It generates data on VRU positions, speeds, directions, and behaviours, such as road-crossing actions. The system also models various infrastructure elements, including traffic signals, road signs, intersections, and lane markings. It classifies road types (e.g., highway, urban, rural) and simulates different road conditions, such as surface types and weather-related factors. The simulator includes Vehicle-to-Infrastructure (V2I) communication simulation to replicate information exchange between vehicles and road infrastructure.

The simulator replicates a range of vehicle control systems. It models basic manual controls such as steering, acceleration, and braking, as well as more advanced features like cruise control, lane-keeping assistance, and collision avoidance systems. The simulation extends to autonomous driving functionalities, including path planning and rule-based traffic navigation. These systems interact with

the simulated sensor data, allowing for testing of how control systems respond to various environmental inputs. The simulator supports scenario-based testing, enabling the evaluation of vehicle control systems under diverse traffic and environmental conditions.

The Human-Machine Interface component of the simulator focuses on the interaction between the driver and the vehicle's information systems. It models interfaces such as dashboard displays, head-up displays, and audio alert systems. The simulated HMI presents information from the virtual sensors and control systems to the driver, replicating how warnings, notifications, and other critical data would be conveyed in a real vehicle. This simulation allows for the study of how different interface designs and information presentation methods affect driver awareness and decision-making.

The simulator's architecture integrates these various components, allowing data to flow between the simulated sensors, control systems, and HMI elements. This integration enables the study of complex interactions between different vehicle systems and the driver, providing a platform for analysing and refining automotive technologies in a controlled, virtual environment.

Macq also provides a City and Traffic Simulator. It delivers tools to create roads, crossroads, cameras, vehicles, and recognition scenarios. It includes a "tardis-mode" to accelerate time for simulations. Traffic situations such as trajectory controls, congestion, convoys, vehicles with special plates or various nationalities, traffic jams, and peak-hour conditions are generated to test Automatic Number Plate Recognition (ANPR) functionality.

The system simulates Vulnerable Road Users, essential for intersections, and includes potential equipment issues like network problems. In augmented reality mode, it integrates real systems with simulated scenarios, supporting real cameras and roadside equipment. Digital Twins and actor-based simulations allow testing of complex system dynamics.

UC3

Simulations will be useful for UC3 because they allow for testing robotic applications within a controlled, distributed environment. By simulating robotic components and their interactions, developers can safely evaluate the performance and behaviour of the system without the need for physical robots. This helps in identifying potential issues, such as latency and communication delays, between the distributed platform and VR simulations. Simulations also enable the testing of different scenarios, helping to profile the system, measure performance, and pinpoint sources of delays.

The simulation in virtual reality of robotic environments represents a test-bench for the distributed platform according to the following aspects:

- A single node simulating a robotic environment will produce a large amount of data corresponding to many (tens or even hundreds) of virtual sensors. These sensors result from the simulation made by the simulation engine (Unity3D in demo 3.2) where considering a collaborative robotic arm with an end-effector, several autonomous mobile robots, a worker who is doing his/her job interacting with the rest of devices, … The simulation allows us to create several virtual cameras which produces video streams of a realistic reproduction of the tasks to be performed. Thus, the distributed platform can be tested with high load even with only one or very few hardware (physical) nodes and measuring the delay in communications between nodes.
- In virtual reality simulations, the environment can be generated faster or slower than real-time on demand. Faster simulations can be useful to accelerate the generation of databases, resulting in increased load of the communication network. On the contrary, slower simulations

can be generated if the computational resources are more restrictive. With this strategy, we can test the amount of data to be managed by the distributed platform.

- With simulated environments, virtual sensors can be arbitrarily distributed along one or multiple nodes, increasing the load of the platform as the number of nodes/workers is increased.

7 Conclusion

DistriMuSe proposes a robust distributed platform architecture to address the diverse use cases considered, managing sensors, varied information types, secure communication across heterogeneous networks, and diverse processing devices. By leveraging distributed algorithms, particularly AI-driven ones, the platform utilizes the computing power of devices across the mist, edge, and cloud, balancing latency, resource efficiency, data transmission, and privacy concerns.

The platform's abstraction capabilities simplify development by shielding developers from low-level specifics of sensors, communications, storage, and computing infrastructure. Developers package components in containers and utilize a Domain-Specific Language (DSL) to describe configurations, components, and dataflows. The development process is simplified with the help of integrated tools supporting debugging and performance evaluation.

The design process has emphasized stakeholder requirements, using an iterative refinement of system needs, and focusing on architectural innovation. This approach seeks that the platform will be able to support diverse use cases, reducing complexity, and allowing developers to focus on creating advanced, distributed services without the burden of extensive low-level customization.

References

Aluvalu, R. & V., U. (Eds.). (2024). Serverless Computing Concepts, Technology and Architecture. IGI Global. https://doi.org/10.4018/979-8-3693-1682-5

Baldini, G. (2023). In-Vehicle Network Intrusion Detection System Using Convolutional Neural Network and Multi-Scale Histograms. Information, 14(11), 605.

Baldoni, G., Loudet, J., Guimarães, C., Nair, S., & Corsaro, A. (2023, October). A Data Flow Programming Framework for 6G-Enabled Internet of Things Applications. In 2023 IEEE 9th World Forum on Internet of Things (WF-IoT) (pp. 1-8). IEEE.

Barzegaran M., and Pop, P. (2021). "Communication Scheduling for Control Performance in TSN-Based Fog Computing Platforms". IEEE Access, vol. 9, pp. 50782-50797, doi: 10.1109/ACCESS.2021.3069142.

Bakshi, A., Prasanna, V. K., Reich, J., & Larner, D. (2005, June). The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services (pp. 19-24).

Belapurkar, A., Chakrabarti, A., Ponnapalli, H., Varadarajan, N., Padmanabhuni, S., & Sundarrajan, S. (2009). Distributed systems security: issues, processes and solutions. John Wiley & Sons.

Bresch, M., & Salman, N. (2017). Design and implementation of an intrusion detection system (IDS) for in-vehicle networks.

Campos, E. M., Hernandez-Ramos, J. L., Vidal, A. G., Baldini, G., & Skarmeta, A. (2024). Misbehavior detection in intelligent transportation systems based on federated learning. Internet of Things, 25, 101127.

Dick, J., Hull, E., Jackson, K. (2017). Requirements Engineering. Springer Cham

Docker. (2024). Docker Swarm. Available at: https://docs.docker.com/engine/swarm/. Accessed: 04-12-2024

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017, October). CARLA: An open urban driving simulator. In *Conference on robot learning* (pp. 1-16). PMLR.

European Commission (2024). Cyber Resilience Act. Available at: https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act. Accessed: 04-12-2024

Ferrer, A. J. (2023). Beyond edge computing: swarm computing and ad-hoc edge clouds. Springer Nature.

Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2021). Software Architecture: The Hard Parts. " O'Reilly Media, Inc.".

Fortier, P., Mouël, F., & Ponge, J. (2021). Dyninka: a FaaS framework for distributed dataflow applications. Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems. https://doi.org/10.1145/3486605.3486789.

Gackstatter, P., Frangoudis, P. A., & Dustdar, S. (2022). Pushing serverless to the edge with webassembly runtimes. In 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (pp. 140-149). IEEE.

Giallorenzo, S., Mauro, J., Poulsen, M. G., & Siroky, F. (2021). Virtualization costs: benchmarking containers and virtual machines against bare-metal. SN Computer Science, 2(5), 404.

Giang, N. K., Blackstock, M., Lea, R., & Leung, V. C. (2015, October). Developing iot applications in the fog: A distributed dataflow approach. In 2015 5th International Conference on the Internet of Things (IOT) (pp. 155-162). IEEE.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., ... & Bastien, J. F. (2017, June). Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 185-200).

HashiCorp. (2024). Nomad. Available at: https://developer.hashicorp.com/nomad. Accessed: 04-12-2024

Hintjens P. ZeroMQ. O'Reilly & Associates, Sebastopol, CA., 2013.

Hyperledger. (2024). Hyperledger Fabric documentation. Available at: https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html. Accessed: 04-12-2024

Honar Pajooh, H., Rashid, M., Alam, F., & Demidenko, S. (2021). Hyperledger fabric blockchain for securing the edge internet of things. Sensors, 21(2), 359.

IoT Analytics (2024). State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally. Accessed: 04-12-2024

Iyengar, A., & Pearson, J. (2024). Edge computing patterns for solution architects: learn methods and principles of resilient distributed application architecture from hybrid cloud to far edge.

JWT (2024). JSON Web Tokens. Available at: https://jwt.io/. Accessed: 04-12-2024

Kabilan, N., Ravi, V., & Sowmya, V. (2024). Unsupervised intrusion detection system for in-vehicle communication networks. Journal of Safety Science and Resilience, 5(2), 119-129.

Kakati, S., & Brorsson, M. (2023, June). Webassembly beyond the web: A review for the edge-cloud continuum. In 2023 3rd International Conference on Intelligent Technologies (CONIT) (pp. 1-8). IEEE.

Khandelwal, S., & Shreejith, S. (2023, September). Exploring Highly Quantised Neural Networks for Intrusion Detection in Automotive CAN. In 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL) (pp. 235-241). IEEE.

Kim, Huy, Kang., Lee, Hyun, Sung., Jeong, Seong, Hoon. (2017). Appratus and method for detecting vehicle intrusion.

Kleinfeld, R., Steglich, S., Radziwonowicz, L., & Doukas, C. (2014, October). glue. things: a Mashup Platform for wiring the Internet of Things with the Internet of Services. In Proceedings of the 5th International Workshop on Web of Things (pp. 16-21).

Kristianto, E., Lin, P. C., & Hwang, R. H. (2024). Sustainable and lightweight domain-based intrusion detection system for in-vehicle network. Sustainable Computing: Informatics and Systems, 41, 100936.

KubeEdge. (2024a). A Kubernetes native Edge Computing framework. https://kubeedge.io/.

KubeEdge. (2024b). KubeEdge Documentation. https://kubeedge.io/docs

Kubernetes. (2024). Kubernetes Official Website. https://kubernetes.io/

Kuberos. (2024). Kuberos concept. https://kuberos.io/docs/concept

Kuo, ChenYing and Liang, William. Running Multiple Autoware Vehicles in CARLA using Zenoh. 2023. https://autoware.org/running-multiple-autoware-powered-vehicles-in-carla-using-zenoh/

Kutluca, H. (December, 2020). Robot Operating System 2 (ROS 2) Architecture. *Medium*. https://medium.com/software-architecture-foundations/robot-operating-system-2-ros-2-architecture-731ef1867776

Len Bass, P. C. a. R. K., 2012. Software Architectures in Practice. 3rd ed. s.l.:Addison-Wesley Professional.

Lux, Z. A., Thatmann, D., Zickau, S., & Beierle, F. (2020, September). Distributed-ledger-based authentication with decentralized identifiers and verifiable credentials. In 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS) (pp. 71-78). IEEE.

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics, 7. https://doi.org/10.1126/scirobotics.abm6074.

Mainland, G., Welsh, M., & Morrisett, G. (2006). Flask: A language for data-driven sensor network programs. Harvard Univ., Cambridge, MA, Tech. Rep. TR-13-06.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux j, 239(2), 2.

Morabito, R., Cozzolino, V., Ding, A. Y., Beijar, N., & Ott, J. (2018). Consolidate IoT edge computing with lightweight virtualization. IEEE network, 32(1), 102-111.

Navaratnam, S. (1987). Reliable group communication in distributed systems (T). University of British Columbia. Retrieved from https://open.library.ubc.ca/collections/ubctheses/831/items/1.0051921

OAuth (2024). OAuth 2.0. Available at: https://oauth.net/2/. Accessed: 04-12-2024

OpenGroup. (2024b). Integrating Risk and Security within a TOGAF® Enterprise Architecture. The OpenGroup Library. https://pubs.opengroup.org/togaf-standard/integrating-risk-and-security/. Accessed: 25-11-2024

OpenGroup. (January, 2015). Modeling Enterprise Risk Management and Security with the ArchiMate® Language, White Paper (W150). The Open Group Library. https://www.opengroup.org/library/w150.

OpenGroup. (2024a). ArchiMate 3.2 Specification. The OpenGroup Library. https://publications.opengroup.org/c226. Accessed: 25-11-2024

OpenRobotics. (2024). Robotics Operating System v2. Open Robotics. https://www.ros.org

OpenYurt. (2024a). An open platform that extends upstream Kubernetes to Edge. https://openyurt.io

OpenYurt. (2024b). OpenYurt architecture. https://openyurt.io/docs/core-concepts/architecture

OpenWhisk. (2024). Apache OpenWhisk documentation. https://openwhisk.apache.org/documentation.html

OSTree (2024). OSTree. Available at: https://ostreedev.github.io/ostree/. Accessed: 04-12-2024

Ostrowski, A., & Gaczkowski, P. (2021). Software Architecture with C++: Design modern systems using effective architecture concepts, design patterns, and techniques with C++ 20. Packt Publishing Ltd.

Peiris, I. (November 21, 2023). Distributed System Architecture. Medium. https://medium.com/@t.i.tpeeriya/distributed-system-architecture-26b3bc03df4d

Puder, A., Römer, K., & Pilhofer, F. (2006). Distributed systems architecture: a middleware approach. Elsevier

RAUC. (2024). Robust Auto-Update Controller. Available at: https://github.com/rauc/rauc. Accessed: 04-12-2024

Ray, P. P. (2023). An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions. Future Internet, 15(8), 275.

Reed, D., et al. (2020). Decentralized identifiers (DIDs) v1.0. Draft Community Group Report.

Ruusiala, A. (2022). *Vehicle automation software development using software-only simulation* (Master's thesis).

Sabella, D. (2021). Multi-access edge computing: Software development at the network edge (pp. 3-379). Springer.

SUSE. (2024). Rancher Desktop. Available at: https://rancherdesktop.io/. Accessed: 04-12-2024

SWUpdate. (2024). Software Update for Embedded Linux Devices. Available at: https://github.com/sbabic/swupdate. Accessed: 04-12-2024

Tanaka, D., Yamada, M., Kashima, H., Kishikawa, T., Haga, T., & Sasaki, T. (2019). In-vehicle network intrusion detection and explanation using density ratio estimation. In 2019 IEEE Intelligent Transportation Systems Conference (ITSC) (pp. 2238-2243). IEEE.

Tanenbaum, A., & Van Steen, M. (2001). Distributed systems: Principles and Paradigms. Prentice Hall.

Tao, Z., Xia, Q., Hao, Z., Li, C., Ma, L., Yi, S., & Li, Q. (2019). A survey of virtual machine management in edge computing. Proceedings of the IEEE, 107(8), 1482-1499.

W3C. (2024). Decentralized Identifiers (DIDs). Available at: https://www.w3.org/TR/did-core/. Accessed: 04-12-2024.

WasmCloud. (2024a). WebAssembly components. https://wasmcloud.com/docs/concepts/components

WasmCloud. (2024b). WebAssembly concepts. https://wasmcloud.com/docs/concepts

WasmCloud. (2024c). WebAssembly providers. https://wasmcloud.com/docs/concepts/providers

Wasmcloud. (2024d). WebAssembly Lattice. https://wasmcloud.com/docs/concepts/lattice

Witanto, E. N., Oktian, Y. E., Lee, S. G., & Lee, J. H. (2020). A blockchain-based OCF firmware update for IoT devices. Applied Sciences, 10(19), 6744.

Xu, H., Berres, A., Yoginath, S. B., Sorensen, H., Nugent, P. J., Severino, J., ... & Sanyal, J. (2023). Smart mobility in the cloud: Enabling real-time situational awareness and cyber-physical control through a digital twin for traffic. IEEE Transactions on Intelligent Transportation Systems, 24(3), 3145-3156.

Xun, Y., Zhao, Y., & Liu, J. (2021). VehicleEIDS: A novel external intrusion detection system based on vehicle voltage signals. IEEE Internet of Things Journal, 9(3), 2124-2133

Yun, H., & Jun, M. B. (2022). Immersive and interactive cyber-physical system (I2CPS) and virtual reality interface for human involved robotic manufacturing. Journal of Manufacturing Systems, 62, 234-248.

Zhang, L., Afanasyev, A., Burke, J., Van Jacobson, kc claffy, Crowley, P., Papadopoulos, C., Wang, L. and Zhang, B. (July, 2014). Named data networking. SIGCOMM Comput. Commun. Rev. 44, 3, pp., 66–73. https://doi.org/10.1145/2656877.2656887

Zhang, L., Yan, X., & Ma, D. (2024). Efficient and Effective In-Vehicle Intrusion Detection System using Binarized Convolutional Neural Network. In IEEE INFOCOM 2024-IEEE Conference on Computer Communications (pp. 2299-2307). IEEE.