



Enabling safe & secure modular UPdates, UPgrades and DynAmic Task reallocation and Execution for Software Defined Vehicles

D2.3 Communication Plane Prototype [initial] V1.1

Zettascale Technology (ZSC)

30.06.2026



Funded by
the European Union

The project, its research and all other activities leading to these results have received funding from the European Union and the granting authority CINEA under Grant Agreement No 101203060

PUBLIC

© Copyright 2025 UP2DATE4SDV Consortium Partners
101203060 — UP2DATE4SDV — HORIZON-CL5-2024-D6-01

Disclaimer:



**Funded by
the European Union**

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the granting authority (European Climate, Infrastructure and Environment Executive Agency - CINEA). Neither the European Union nor the granting authority can be held responsible for them.”

All rights reserved.

Deliverable Information		
Deliverable Number	D2.3	
Deliverable Name	Initial Communication Plane Prototype [initial]	
Nature	Report (R)	
Dissemination Level	Public	
Contractual Date of Delivery	30.06.2026	
Actual Submission Date	30.06.2026	
Lead Beneficiary	ZSCALE	
Lead Author	Ivan PAEZ	
E-Mail	ivan@zettascale.tech	
Contributing Beneficiaries	DLR, IKL, BSC, TTT, AVL, AVL TR, STM FR, SDM IT, NTR, KK, ITI, AUMOVIO	
Reviewing Beneficiaries	Organisation KK (WPL)	
	IKL	Ane Mijangos
	DLR	Patrick Uven
Work Package(s)	WP 2 Reconfigurable and resource composable SW deployment and execution layer r	
Related Task(s)	T2.1	Identification of SW deployment layer requirements and abstraction services
	T2.2	Identification of SW deployment layer requirements and abstraction services
	T2.3	Reconfigurable and resource controllable service-oriented communication plane implementation
Keywords	Software-Defined Vehicle, SDV, Horizon Europe, Automotive, CCAM, OTA, Update, Upgrade, DevOps, E/E-Architecture, Orchestration, Vehicle-to-cloud, Cloud-Edge Continuum, Reconfiguration, Eclipse, Zenoh, Pub-Sub	

Project Information	
Contract #	101 203 060
Fixed Project Start Date	01/07/2025
Project Duration	36 months
Model Grant Agreement based on	HE MGA — Multi & Mono - V1.2
Call	HORIZON-CL5-2024-D6-01
Topic	HORIZON-CL5-2024-D6-01-01
Type of Action	HORIZON-RIA
Service	CINEA/C/03
Project Website	https://up2date4sdv.eu/

Document History		
Revision	Date	Description (add and/or delete)
V0.1	01.06.2026	Initial version
V0.2	15.06.2026	Updated version with partners's inputs
V0.3	19.06.2026	Version for review (1 st complete draft)
V0.4	22.06.2026	Review comments included, version for acceptance
V0.5	26.06.2026	Review comments included, version for acceptance
V0.6	29.06.2026	Review comments included, version for acceptance
V0.7	29.06.2026	Review comments included, version for clearance
V1.0	15.09.2026	Review comments included, version for clearance
V1.1	28.09.2026	Final and reviewed Version

Table of contents

Table of contents.....	5
Executive Summary	7
Acronyms and Abbreviations	8
List of Figures.....	10
List of Tables.....	10
References and Background Material	11
1. Introduction	12
1.1. Project Outline	12
1.2. Technical Deliverable-Map	13
1.3. Work-Package Outline	14
1.3.1. Work-Package Objectives	14
1.3.2. Work-Package Description	14
1.3.3. Work-Package Status.....	15
1.4. Deliverable Outline	15
1.5. Purpose and Scope.....	15
1.6. Background and Related Work	16
1.6.1. Message-centric Communication	16
1.6.2. Data-centric-Communication	16
1.6.3. Limitations on the state-of-the-art.....	17
1.6.4. Expected requirements	18
2. Context within UP2DATE4SDV	19
2.1. UP2DATE4SDV's Architecture	19
2.2. Eclipse Zenoh's Architecture.....	20
3. Core Concept.....	22
3.1. Key Expression	22
3.2. Session Management.....	23
3.3. Publishers & Subscribers.....	23
3.4. Queryables & Replies.....	23
3.5. Storage Management	25
3.6. Other Programming Language Binding.....	26
4. Routing &Topology.....	27
4.1. Peer-to-peer mode	27

4.2.	Brokered and Routed mode.....	28
4.3.	Regions.....	29
5.	Security.....	30
5.1.	TLS/mTLS Configuration.....	30
5.2.	Access Control Lists.....	30
5.3.	Token-based authentication	31
6.	Embedded & bare-metal support	32
6.1.	Zenoh-pico	32
6.2.	Supported (RT)OSs and protocols.....	32
7.	Advanced & Ongoing topics	34
7.1.	Quality of Services (QoS).....	34
7.2.	Reliability & Congestion control	35
7.3.	Shared memory (zero copy).....	35
8.	Synergies within UP2DATE4SDV	37
8.1.	Resource monitoring between HAL and ORP	37
8.2.	Alignment and Support for Communication Monitoring through the HAL.....	38
9.	Contributions from the OS community.....	40
9.1.	STM32 ThreadX support	40
9.2.	ROS2 native support (RMW_Zenoh).....	41
9.3.	Eclipse Zenoh Transport Library for Eclipse uProtocol	42
10.	Summary and Discussion/Conclusion	44

Executive Summary

This document covers the Milestone 3 (MS3) deliverable D2.3 containing the initial communication plane, the specification of the software interfaces, the layered architecture, and the corresponding abstraction services provided by the communication middleware. The deliverable is a collection of the functionalities and the software components, and documents the synchronized view on those software components and what other partners can expect from them.

Acronyms and Abbreviations

Abbrev.	Full Name
AD	Autonomous-Driving
CA	Consortium Agreement
CAD	Connected and Automated Driving
CCAM	Connected Cooperative and Automated Mobility
CoCo	Coordinator Contact
COTS	Commercial Off-The-Shelf
CSA	Coordination and Support Action
D	Deliverable
Diss/Expl	Dissemination/Exploitation
DoA	Description of Action
DPC	Data Privacy Coordinator
EC	European Commission
ECU	Electronic Control Unit
EU	European Union
FSIGN	Financial Signatory
GA	Grant Agreement
HAL	Hardware Abstraction Layer
HW	Hardware
IP	Intellectual Property
JU	Joint Undertaking
LEAR	Legal Entity Appointed Representative
LSIGN	Legal Signatory (LSIGN)
M	Month
MBE	Model-Based Engineering
MBSE	Model-Based System Engineering
NDA	Non-Disclosure Agreement
NFA	National Funding Authorities
OAL	OS/MW Abstraction Layer
ODDs	Operational Design Domains
ORP	Orchestration and Reconfiguration Plane
PaCo	Participant Contact
PC	Project Coordinator
PCoCo	Primary Coordinator Contact
PGA	Project General Assembly
PPR	Periodic Project Report
PU	Public
R	Report
SASE	SAfety and SEcurity
SD-CCAM-V	Software-Defined CCAM vehicles
SDV	Software Defined Vehicle

SEN	Sensitive
SoM	System-on-Module
SOTA	Software-Over-The-Air
SW	Software
SyGMa	System Grant Management https://ec.europa.eu/research/participants/grants-app/reporting/DLV-101203060
TL	Task Leader
TSN	Time Sensitive Network
UCL	Use Case Leader
V&V	Verification and validation
V2C	Vehicle to Cloud
VIT	Virtual Integration Testing
VRUs	Vulnerable Road Users
WP	Work Package
WPL	Work Package Leader

List of Figures

Figure 1: Overview of the UP2DATE4SDV work-packages and their Interaction.	13
Figure 2: Overview of the UP2DATE4SDV technical deliverable map.	13
Figure 3: Object Management Group's Real-Time Data Distribution Service (DDS)	16
Figure 4: UP2DATE4SDV's centralised zonal E/E architecture.....	19
Figure 5: Proposed communication plane integration within the centralised E/E vehicle architecture.....	20
Figure 6: Eclipse Zenoh architecture.....	21
Figure 7: Session closing and cleanup.....	24
Figure 8: Pub/sub communication pattern	24
Figure 9: Request/response communication pattern.....	25
Figure 10: Storage pattern used to retrieve historical data	25
Figure 11: Eclipse Zenoh's API Programming Language bindings.....	26
Figure 12: P2P 1:N or N:1 communication.....	27
Figure 13: P2P N:N graph network	28
Figure 14: Eclipse Zenoh supported communications mechanisms.....	29
Figure 15: Zenoh-pico implementation and bindings.....	32
Figure 16: Zenoh Session's Default SHM Provider Workflow	36
Figure 17: Resource monitoring between HAL and ORP	37
Figure 18: Information collection via specific APIs	37
Figure 19: Adoption history of the Eclipse-Zenoh open source project	40
Figure 20: ROS2 Internal API architecture overview	42
Figure 21: uProtocol to Zenoh Mapping.....	43

List of Tables

Table 1: List of requirements from the ROS community	18
Table 2: Supported (RT)OSs and protocols	33
Table 3: Quality of Services (Qos) specification.....	34
Table 4: Types of priority	34

References and Background Material

Internal Background

- [1] UP2DATE4SDV - Project Proposal Project Part B: technical alignment (not public)
- [2] D2.1 Requirement Specification, January 2026. (public)
- [3] D4.1 Update Procedure and Architecture, June 2026. (public)

Internal Background

- [4] Eclipse Zenoh, Official GitHub repository. <https://github.com/eclipse-zenoh/zenoh>
- [5] Eclipse Zenoh-Pico GitHub repository: <https://github.com/eclipse-zenoh/zenoh-pico>
- [6] ROS2 RMW alternate. 2023-09. <https://discourse.ros.org/uploads/short-url/o9ihvSjCwB8LkzRklpKdeesRTDi.pdf> (visited on 05/06/2026).
- [7] ITU-T Technical Report. FGAI4AD-01 Automated driving safety data protocol – Specification. Sept 2022; https://www.itu.int/dms_pub/itu-t/opb/fg/T-FG-AI4AD-2022-PDF-E.pdf (visited on 18/06/2026).

Publications

- [8] A. Corsaro *et al.*, "Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller," *2023 26th Euromicro Conference on Digital System Design (DSD)*, Golem, Albania, 2023, pp. 422-428, doi: 10.1109/DSD60849.2023.00065
- [9] Baldoni, Gabriele, et al. "Data-centric service-based architecture for edge-native 6G network." *IEEE Communications Magazine* 62.4 (2023): 32-38, doi: 10.1109/MCOM.001.2300178

1. Introduction

The European research project UP2DATE4SDV (Enabling safe & secure modular UPdates, UPgrades and DynAmic Task reallocation and Execution for Software-Defined Vehicles), aims to develop a comprehensive ecosystem for seamless and efficient software updates, hardware upgrades and situation-dependent reconfigurations of software-defined vehicles.

1.1. Project Outline

The project focuses on the definition and development of three abstraction layers

- Hardware Abstraction Layer (HAL)
- OS/MW Abstraction Layer (OAL) for the operating system (OS) and middleware (MW)
- Exploration and prototyping of a secure orchestration and reconfiguration layer (ORP) between the vehicle and the cloud.

This will improve the reliability and performance of the CCAM system, extend vehicle life and reduce electronic waste. In addition, the project's new modularity concepts will accelerate the transition to safer, more sustainable automated driving and enable incremental steps towards Level 4 automation and longer vehicle lifecycles

The overall UP2DATE4SDV goal is to create a comprehensive ecosystem for updatable, upgradable, and reconfigurable software-defined connected and automated vehicles. This will be achieved by developing essential abstraction layers, advanced orchestrators, and an automated robust realization environment, all while adhering to the highest safety and cybersecurity standards. To achieve this vision, the project will target the following objectives:

- O1. Creation of a reconfigurable cloud connected zonal vehicle architecture blueprint based on heterogeneous COTS HW (exploiting and building on a unified HAL)**
- O2. Establishing a resource composable SW deployment, execution and communication reference layer, on top of existing open-source hypervisor and OS stacks (exploiting and building on a unified OAL)**
- O3. Realization of a dynamic update, orchestration & reconfiguration plane**
- O4. Provide harmonized and simplified development, integration and validation processes and framework for SDV applications**
- O5. Assess joint Safety and Security (SASE) of all elements from O1, O2 and O3**
- O6. Demonstration of UP2DATE4SDV solution on several CCAM relevant Use Cases**
- O7. Active participation around CCAM and SDV Initiative community and ecosystem**

To this goal, the demonstrators and use-cases (developed in WP5) define requirements (pink arrows) for the HW and the HAL (developed in WP1), the SW and the OAL (developed in WP2) as well as for the ORP (developed in WP3). Based on that, WP1, WP2 and WP3 provide a basic functional HW-SW-demonstrator for the use-cases – UC1- after Phase 1 (M6), UC2, UC3, and UC4 after Phase 2 (M24), and finally UC5 and UC6 after Phase 3 (M36).

Besides the pure communication of results (green arrows) the proper integration of the results of WP1, WP2, WP3, and WP4 within the use-cases depends on three further cooperation aspects: the safety and security assessment (yellow arrows), the update-, upgrade-, and reconfiguration-processes and the involved component-interfaces (purple arrows), and the cloud-based MBSE-representation of

actual HW-/SW-components during the DevOps cycle establishing the real-world to cloud connection according to digital twins (blue arrows). Finally, while WP6 oversees the dissemination, communication, and exploitation of the project's results, WP7 coordinates all the WP and their communication, internally as well as externally to the SDV-Initiative and to other CCAM-partnerships. Figure 1 summarizes the described interaction and communication between the work-packages (WPs) of UP2DATE4SDV.

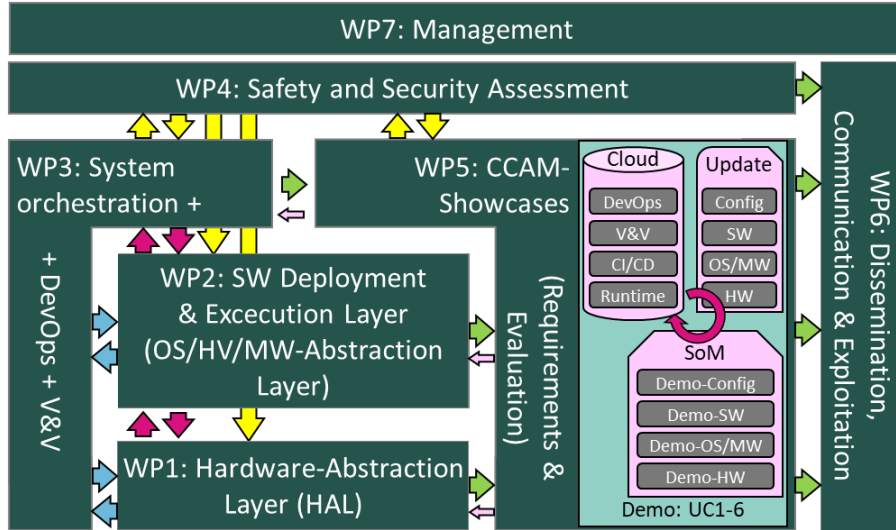


Figure 1: Overview of the UP2DATE4SDV work-packages and their Interaction.

1.2. Technical Deliverable-Map

UP2DATE4SDV has defined 38 deliverables of which 10 have an organizational nature and 28 a technical nature. To provide orientation to the reader, Figure 2 shows a technical deliverable map that gives a structural overview of all technical deliverables and their content-wise relation.

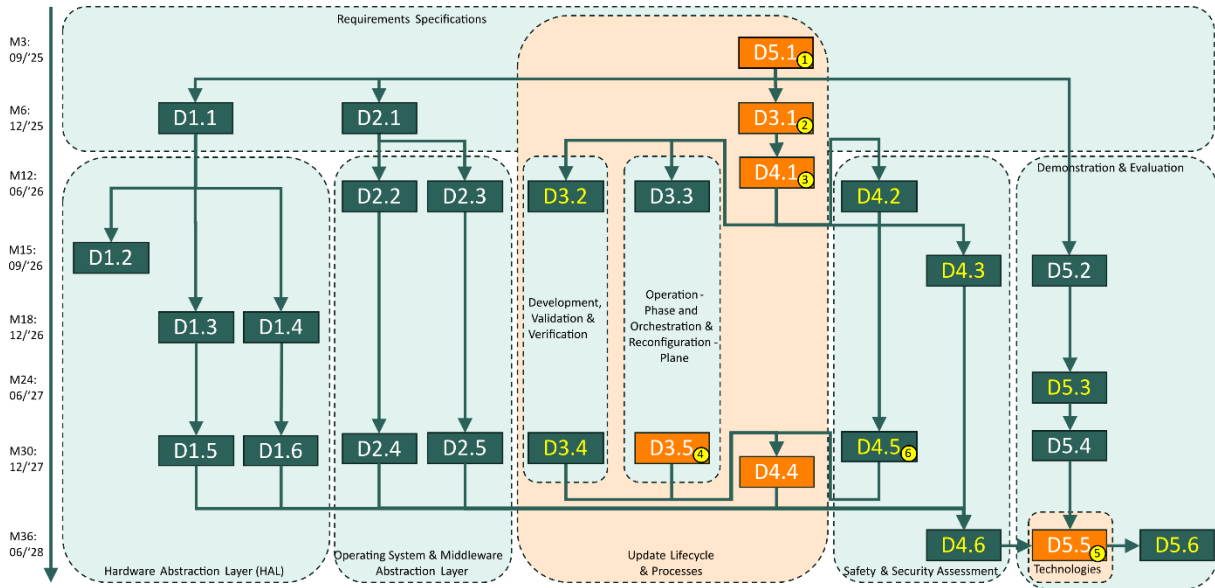


Figure 2: Overview of the UP2DATE4SDV technical deliverable map, showing the deliverables of the technical work-packages WP1-WP5, with restricted deliverables highlighted by the yellow text, a main-track highlighted by the orange nodes, and a recommended reading-order in the yellow circles. The arrows between deliverables are indicating the content development.

Deliverables are shown as rectangles, which are connected by arrows if the content of one deliverable is based on the contents of previous ones. The text in the rectangles shows the deliverable number and it is highlighted in yellow, in case of restricted deliverables, whereas public deliverables are labelled in white. The light-green and light-orange areas form technical content-categories, close but not equal to the work-packages. They show the hardware abstraction layer (HAL) in WP1, the Operating-System and Middleware Abstraction-Layer (OAL) in WP2, the Safety and Security Assessment in WP4, and the Demonstrators and the Evaluation in WP5, but they combine the Orchestration and Reconfiguration-Plane of WP3 with the safety- and security-analysis and corresponding measurements from WP4, forming the area of the Update Lifecycle and Processes, where Development, Validation and Verification, and Orchestration and Deployment of Updates are covered. Furthermore, the four initial deliverables of the work-packages requirements-definitions are also combined in a separate area at the very top. To get an overall impression of the full project, the main story-track of deliverables is highlighted in orange, and a suggested reading-order is indicated by numbers in yellow circles.

1.3. Work-Package Outline

As shown in Figure 1, WP2 is a central work package with interfaces to most other work packages in the project. In particular, its work items concern the hardware abstraction layer, the container middleware and the reconfigurable and resource controllable service-oriented communication plane. The developments in this work package are the basis of the CCAM-Showcases and the interfaces exposed by the components of WP2 are the knobs used by the orchestration layer from WP3.

1.3.1. Work-Package Objectives

This work package addresses challenges in a highly dynamic and reconfigurable SDV environment, ensuring safe and secure system operation during dynamic updates. The initial task focuses on collecting and analyzing requirements, abstractions, and interfaces for a flexible, individually updatable software and communications stack for future automotive HPC systems. Based on these findings, implementation begins on the container middleware and communication plane. The container middleware will enable container support on next-generation automotive hardware and extend the L4Re Hypervisor and OS framework with native container runtime capabilities. On the communications side, an Eclipse Zenoh-based networking- middleware will be developed, complemented by resource monitoring components integrated into the OAL to oversee communication interfaces.

1.3.2. Work-Package Description

T2.1: Identification of SW deployment layer requirements and abstraction services

The goal of T2.1 is to elicit the requirements, abstractions and interfaces of execution containers to control, supervise, and individually update parts of the software stack. This includes the definition of container formats and abstractions levels as well as of safety and security properties, the monitoring and controlling of time and resource consumption, the definition of suitable KPIs, and the definition of requirements for a service-oriented communications plane that can easily adapt to dynamic system reconfiguration. Another goal is the Analysis that will serve as the basis for a model-based security checking approach. For all these items it must be ensured that the proposed solutions are suitable for high-performance devices in general and on the SoCs used in the project in particular.

T2.2: Reconfigurable and resource controllable container middleware implementation

This task supports the implementation of reconfigurable and resource controllable container middleware. The Open Source L4Re Hypervisor and Operating System Framework shall be extended with resource and interface descriptions and a native container runtime format shall be proposed together with the corresponding infrastructure. Other focus points include resource monitoring and matching resource requirements to the SoC as well as executing containers on the STM next generation hardware for automotive to leverage STM32 MP2 and STELLAR capabilities, enhancing performance and scalability for a wide range of applications.

T2.3: Reconfigurable and resource controllable service-oriented communication plane implementation

This task focuses on the implementation of the middleware for the communication plane based on named-data networking for time and resource constrained devices, including the connection to Zenoh Flow and ROS2. Further resource monitoring and observing capabilities to monitor traffic generated and received by each node through the different communication interfaces (e.g. Ethernet, PCI, etc.) are developed. This includes the OS/Middleware Abstraction Layer (OAL)-Integration of the resource observer- and monitor-components. Alignment with the MBE work in WP3 and SASE mechanisms in WP4 shall be ensured.

1.3.3. Work-Package Status

At the time of writing, WP2 is in its initial design and prototyping phase for both currently running sub-tasks. Task 2.2 and Task 2.3 have consolidated their goals and requirements for container support and reconfigurable and resource controllable service-oriented communication, surveyed the relevant standards and comparable systems, and produced the architectural design presented here. The implementation of minimal prototypes is ongoing. The work-package is on schedule with respect to the planned effort for this phase.

1.4. Deliverable Outline

1.5. Purpose and Scope

This deliverable, D2.3, presents the initial UP2DATE4SDV communication plane prototype, which is based on the Eclipse Zenoh [4] open-source project. It focuses on the definition of the middleware communication plane and will serve as a future reference for the interfaces, APIs, and integration points with the different data sources, data consumers, and data computation nodes that will be implemented in the UP2DATE4SDV approach. As illustrated in Figure 1 its abstraction layers and is framed in the ongoing work of WP2.

D2.3's purpose is to present a guide and introduction to the selected open-source communication middleware within the UP2DATE4SDV project to carry out the data dispatching functions between the different components and subcomponents of the system.

Taking as reference the initial requirements specifications described in D2.1 [2], and to meet that end, there was a need for a communication protocol that was

- Efficient: a protocol that uses minimal wire overhead.
- Scalable-up: a protocol that can scale to internet-wide deployment
- Scale-down: a protocol that can run on bare-metal, small, embedded configurations, without the need to re-architect the system

- Flexible Topology: a protocol that can support, at the same time, peer-to-peer, brokered, and client-server configurations, mix them at the same time, without any constraint.
- Location Transparency: a protocol that can support geo-location transparency

The scope of this deliverable is to present the initial state of the proposed communication middleware at M12 of the project. During Y3 of the project, there will be a second iteration to present the progress, which will be reported in the final implementation of the communication data plane deliverable (M30).

1.6. Background and Related Work

1.6.1. Message-centric Communication

Message-centric infrastructure sends messages of unknown content. There are several types of message-centric middleware. These include direct point-to-point messaging, queuing, broker, and transactional systems. They support different interaction patterns such as request/reply, remote-method invocation, transactional exchange, queues, and publish/subscribe. Current implementations include RabbitMQ, IBM Websphere MQ, and Tibco Rendezvous. Popular standards include the Java Message Service (JMS) API, and the Advanced Message Queuing Protocol (AMQP) wire specification.

The important common property for this discussion is that these technologies consider the message itself to be the means of interaction. The infrastructure is unaware of the message contents. With a message-centric approach, developers write applications that send messages between participants.

1.6.2. Data-centric-Communication

Data centrality, by contrast, makes the data the means of interaction. A data-centric infrastructure must define the data it manages. Then, the infrastructure imposes rules that determine how data is structured, when data is changed, and how it is accessed. There is a notion of a "global data space", where data values, in known structures, are exchanged. The infrastructure is aware of the contents. With a data-centric approach, developers write applications that read and update entries in this data space, see Figure 3.

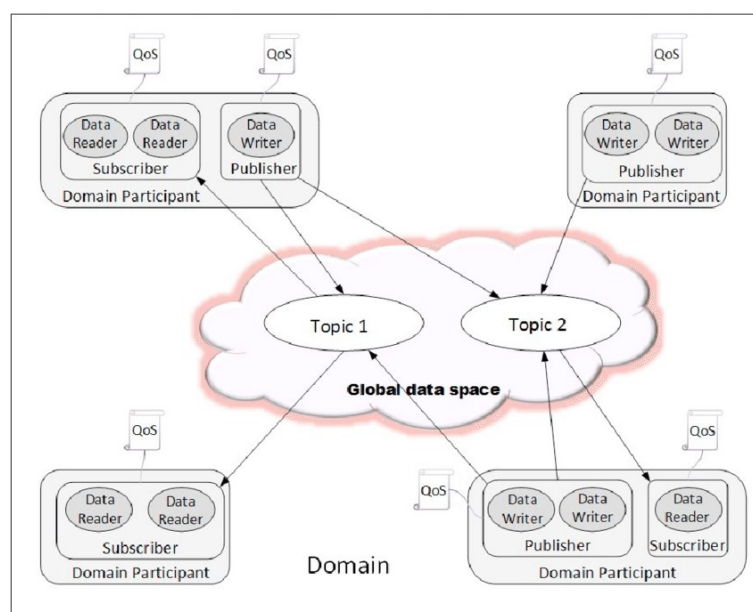


Figure 3: Object Management Group's Real-Time Data Distribution Service (DDS) provides a data-centric, publish/subscribe paradigm

The Object Management Group (OMG) standard for Real-Time Data Distribution Service (DDS) offers both API and wire specifications for data-centric middleware functionality

1.6.3. Limitations on the state-of-the-art

In September 2023, the ROS2 core team published a study titled "ROS 2 RMW Alternate" [6], a systematic evaluation of whether the existing protocols, like DDS, the middleware that has powered ROS 2 since its inception in 2015, should be supplemented with an alternative.

After surveying over 180 community members, deriving a formal requirements list, and comparing more than twenty middleware candidates against those requirements, the team reached a clear conclusion: Eclipse Zenoh was the best protocol that meets the requirements and was given the green light to be developed into the new non-DDS RMW alternative for ROS2.

The RMW (ROS Middleware) interface was designed as an abstraction layer that lets ROS 2 swap its underlying communication mechanism at compile time or runtime. All current Tier 1 implementations are DDS-based. DDS was a reasonable choice in 2015 — it had a long history in mission-critical deployments and addressed many of the same goals as ROS. But eight years of real-world use had accumulated a clear set of recurring problems.

- **Fully-Connected Graph:** DDS maintains a fully-connected graph: every participant, topic, and service in the network must be discovered by every other participant. This produces $O(n^2)$ discovery traffic and "packet storms" when new nodes join large networks. ROS programs are accustomed to creating many topics cheaply — an assumption that does not hold with DDS at scale.
- **UDP Multicast for Discovery:** By default, DDS relies on UDP multicast for peer discovery. Many institutional networks disable multicast for security or performance reasons, and large WiFi deployments routinely suppress it. The failure mode is silent: nodes simply do not find each other, leaving users to diagnose a network-level issue they may not even know exists.
- **Large Message Transfers:** DDS uses UDP as its default transport. While this gives fine-grained QoS control, UDP is far less optimized than TCP across the entire software stack — from OS kernels to network chipsets. Linux defaults to small UDP buffer sizes (~256 KB), which is routinely insufficient for the images and point clouds that are staples of robotics. Transferring large sensor data reliably requires manual kernel tuning that most users are not equipped to perform.
- **WiFi Reliability:** WiFi compounds the above problems. Spotty connectivity interacts badly with UDP-fragmented large messages, and disabled multicast breaks discovery. Since ROS 2 is routinely used on mobile robots and debugging laptops, poor out-of-the-box WiFi behaviour is a significant friction point.
- **Complex Tuning:** The above issues are, in principle, addressable through DDS configuration. In practice, the configuration surface is enormous, the parameters that work on one network often fail on another, and new users have no clear path to the right settings. This is complexity relocated, not eliminated.
- **Vendor-Specific Extensions:** Several DDS vendors have developed proprietary workarounds, custom discovery servers, non-standard transports, and closed-source tooling. These create vendor lock-in and cannot be freely used in an open-source framework. They also diverge from each other, multiplying what users must learn.

The study is careful to note that DDS works well when expertly tuned on a well-managed network — as evidenced by its use in mission-critical systems worldwide. The goal is not to replace DDS for

demanding applications, but to provide an alternative that "just works" for the broad majority of robotics use cases.

1.6.4. Expected requirements

The ROS team drew requirements from three sources: known ROS 2 use cases, targeted interviews with key stakeholders, and a public community survey. The following Table 2, summarizes the main requirements.

Table 1: List of requirements from the ROS community

Requirement	Definition
Pub/sub	Support for peer-to-peer communication is preferred for optimal performance
Security (Encryption, Authentication, and access control)	Support for secure and encrypted communication channels (using i.e. TLS, mTLS), support for certificate-based authentication, and support for identity validation and access control.
Session management	Support for graceful management of connection/disconnection critical for WiFi and ad hoc mobile robots' networks.
Resilience	Tolerance to bandwidth changes, WiFi signal can swing by orders of magnitude
Configurable network interface	Support for tailored configurable networking essential for virtual networks and routing
Large Payload Support	Support of megabyte messages i.e., image, point clouds at up to ~30Hz
Small message support	Support of small fast messages such as robot state at a ~1kHz, and < 1kB payload
Dynamic discovery mechanism	Support for restart discovery mechanism without restarting manually the nodes, this implies no single point of failure in safety critical systems
Cross-platform support	Support for multiple architectures including Ubuntu, Windows, MacOS, amd64/arm64
OSI-approved permissible license	No copyleft

This set of requirements, together with D2.1 Requirements specifications [2] sets the goal for the communication middleware that was selected for this project.

2. Context within UP2DATE4SDV

This section presents the role of the communication plane within the UP2DATE4SDV proposed architecture

2.1. UP2DATE4SDV's Architecture

As presented in D4.1 [3], UP2DATE4SDV adopts a centralized zonal electrical and electronic (E/E) architecture for the in-vehicle system, aligned with the dominant trend of software-defined vehicles (SDV), see Figure 4. In this architecture, the following components will be found:

- A high-performance Central ECU: concentrates the computing, the application logic, and the decision-making that used to be spread across several domain ECUs;
- A set of Zonal Gateways: group the physical interfaces, sensors, actuators, and edge ECUs by physical location in the vehicle rather than by functional domain, each gateway acting as a regional aggregator between the Central ECU and its zone;
- Edge Devices: are limited to time-critical and hardware-specific functions, while higher-level processing and orchestration are delegated to the zonal gateway and the central ECU

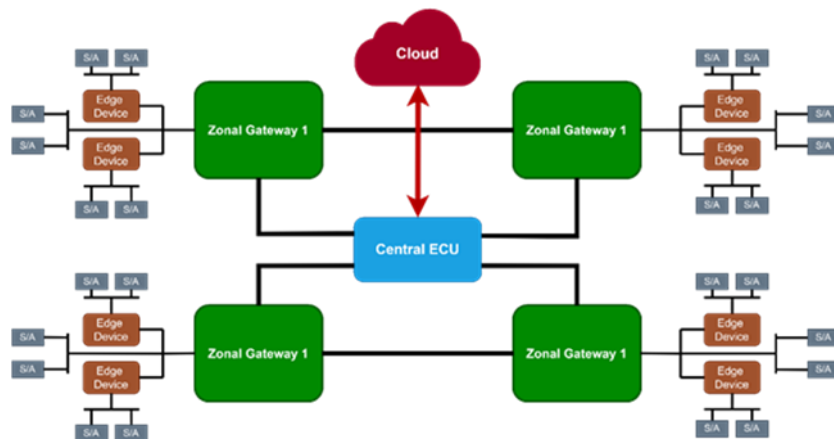


Figure 4: UP2DATE4SDV's centralised zonal E/E architecture

Within the context of the UP2DATE4SDV's architecture, we can consider a vehicle that employs a zonal architecture, logically dividing the vehicle into four zones. The main components in our setup could be instantiated as follows, see Figure 5:

- **Tire Sensors:** Each zone could include tire sensor services (sensor_ws1, sensor_ws2, sensor_ws3, and sensor_ws4), connected as clients to their respective zonal router. The sensors act as Zenoh publishers, publishing real-time measurements for tire pressure, temperature, and wear. Tire data are generated randomly and published to their respective key expressions at 1-second intervals.
- **Traction Control Systems (TCS):** Each zone has a dedicated Traction Control System (zone1_tcs, zone2_tcs, zone3_tcs, and zone4_tcs), responsible for reading measurement data from the tire sensors. Each TCS can subscribe to its zone's sensor, but it can also subscribe to data from other zones to allow traction-related decisions that require data from the entire vehicle.
- **Zonal Routers** - Each zone could have an Eclipse Zenoh router, which handles communication between the clients of the respective zone, and connects the zone with the central router to allow information exchanges between zones.

- **Central Router** - A central Eclipse Zenoh router (a.k.a. Zenoh gateway) manages cross-zone communication, allowing controlled data flow between vehicle zones—similar to a vehicle's central gateway. This router enables each zone's TCS to access data from other zones when needed, supporting a holistic view of the vehicle's tire status.

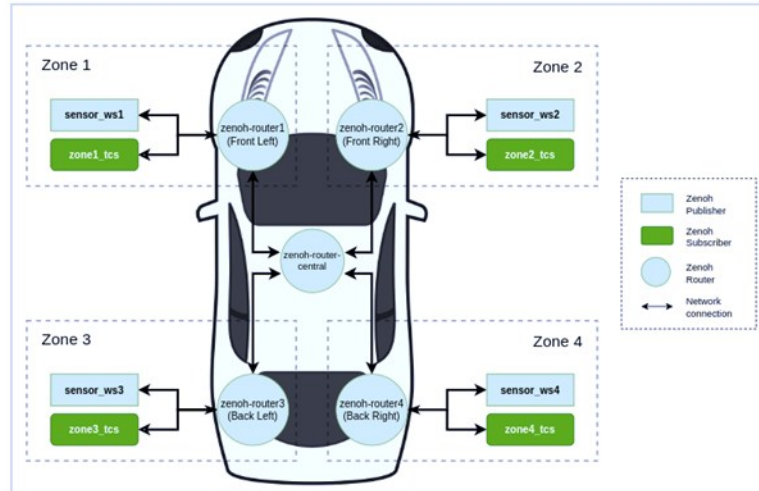


Figure 5: Proposed communication plane integration within the centralised E/E vehicle architecture

In the previous scenario description, we introduced technical concepts like publishers, subscribers, Zenoh-router, and key resources definition. The following subsection will explain Eclipse Zenoh's architecture, and its main concepts will be explained in Section 3.

2.2. Eclipse Zenoh's Architecture

Eclipse Zenoh architecture operates seamlessly across heterogeneous environments by abstracting over the underlying network topology. A Zenoh network is a graph of *nodes*, each of which plays one of three roles: router, peer and client. The Core of the communication protocol is built in Rust programming language.

The Figure 6 presents a characterization of Zenoh's layered architecture, and the possible configuration decisions to take into consideration when designing a decentralized and distributed data exchange scenario.

Eclipse Zenoh's high-level architecture is structured in distinct layers, from user-facing APIs down to network protocols. The architecture is layered to separate concerns. The User-Facing Layer provides the public APIs (Session, Publisher, Subscriber, Queryable, Querier) and command-line tools (zenohd). The Core Runtime layer manages session lifecycle and configuration. The Routing Layer handles message routing through a resource tree. The Transport Layer manages network connections and message transmission. The Plugin System Layer enables extensibility for REST APIs and storage backends.

In more detail, the Eclipse Zenoh's key architectural components are:

- The **user-facing layer** component is the main entry point for an application to interact with Eclipse Zenoh and its **zenohd** router daemon. The developer should select which communication pattern will be used: pub/sub, queryable, or both.

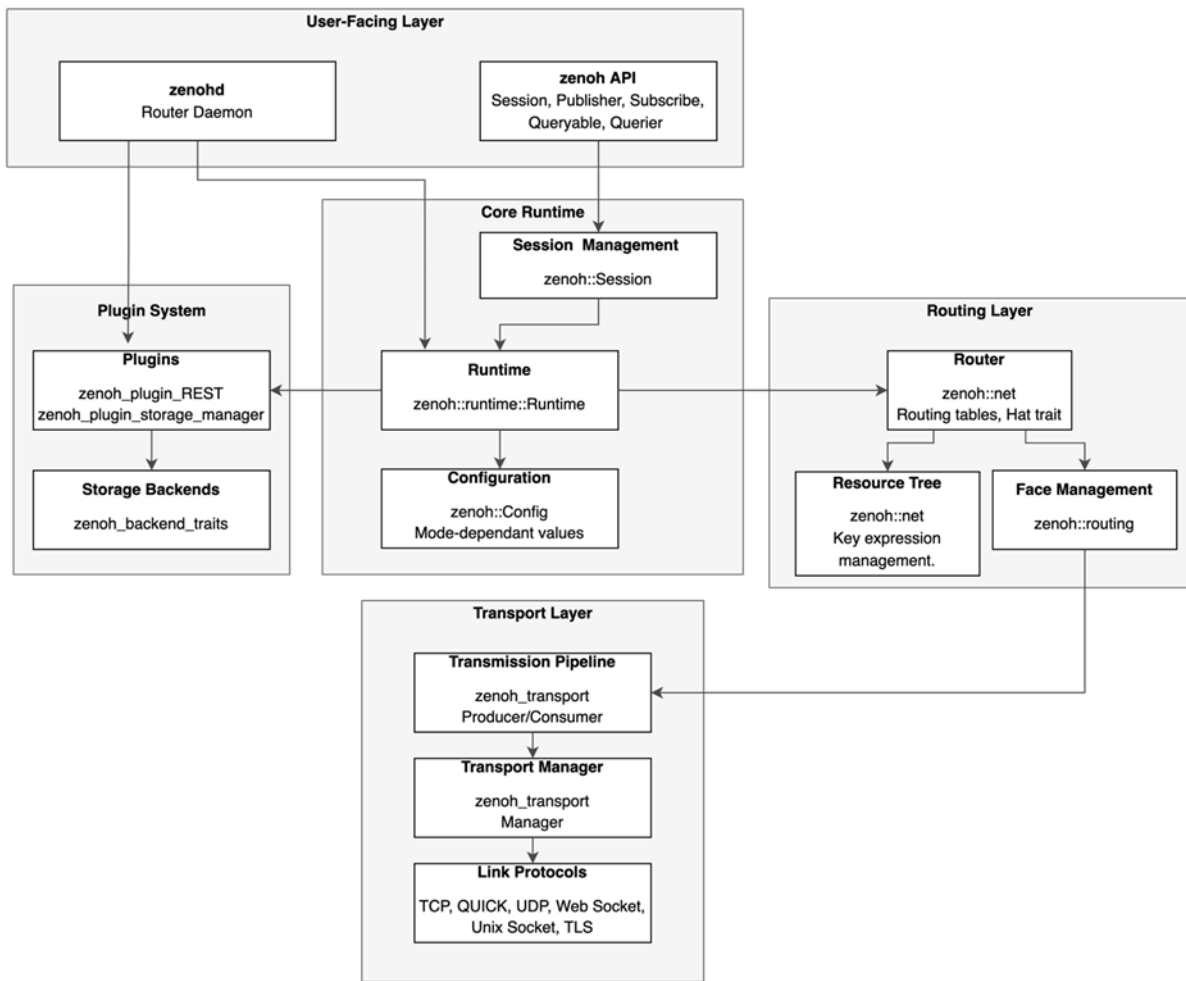


Figure 6: Eclipse Zenoh architecture

- The **Core runtime layer** is in charge of the session establishment process, supports dynamic discovery, and is automatic between two Zenoh instances/processes running in the same network. The liveness component is for monitoring system components (i.e., stay alive signal). It also contains the configuration component. The application developer can select to use either brokered, routed, or peer-to-peer communication patterns. Also, there is an admin space where the user can provide system monitoring and management capabilities.
- The **Plugin System** provides a mechanism for extending the **zenohd** router daemon with additional functionality through loadable modules. Plugins can be either dynamically loaded from shared libraries at runtime or statically linked at compile time. This system enables features like REST API access, persistent storage backends, and custom protocol handlers without modifying the core router code.
- The **Transport layer** handles the establishment and management of transport connections, it also manages the system's resources, and implements various transport protocols (TCP, TLS, UDP, QUIC, WebSocket, etc.).
- The **Routing component** is the central message dispatching mechanism that determines how messages flow between nodes. The routing layer sits between the Session API and the Transport layer, managing a resource tree for key expression matching, computing routes based on subscriptions and queryables, and applying topology-specific routing strategies through the HAT (Hierarchical Adaptive Topology) abstraction.

3. Core Concept

This section explains the fundamental abstractions that make the Eclipse Zenoh protocol different from the others. Understanding these concepts will help you design systems that are:

- Location transparent — producers and consumers are fully decoupled
- Topology independent — your application logic doesn't change when you move from a single machine to a cloud-to-edge deployment
- Efficient — zero unnecessary copies, minimal wire overhead

3.1. Key Expression

This section explains the naming system for all the protocol resources. A **Key Expression (KE)** is a UTF-8 string that identifies one or more *resources* in a Zenoh system. Key expressions follow a path-like structure and support structured wildcards for pattern matching.

```
key-expr    = chunk * ("/" chunk)
chunk       = 1*(unreserved / pct-encoded) / wild1 / wildN
wild1       = "*"
wildN       = "**"
unreserved  = ALPHA / DIGIT / "-" / "_" / "." / "~" / "@" / "$"
pct-encoded = "%" HEXDIG HEXDIG
```

A key expression is composed of *chunks* separated by `/`.

A well-formed key expression satisfies all of the following:

- No trailing `/`.
- No empty chunks (i.e., `//` is invalid).
- **MUST NOT be adjacent to another `*` or chunk** (e.g., `a//` is invalid).
- The total length **MUST** be at least 1 character.

Implementations **MUST** validate key expressions received from applications and **SHOULD** validate key expressions received on the wire. The two available wildcard options are:

* (wild1)

Matches exactly one chunk (any non-empty sequence of characters that does not contain `/`).

Example: `a*/c` matches `a/b/c` and `a/xyz/c`, but not `a/b/d/c` or `a/c`.

** (wildN)

Matches zero or more complete chunks (the empty sequence, or any sequence of `/`-separated chunks).

Examples: `a**/c` matches `a/c`, `a/b/c`, and `a/b/d/c`. `*` alone matches any key expression, including the empty string.

Intersection and Inclusion

- Two key expressions *intersect* if there exists at least one concrete key string that matches both.
- Key expression *A includes* key expression *B* if every string matching *B* also matches *A*.
- These relations are used by routers to determine routing paths: a PUSH addressed to key *A* is forwarded to a subscriber declared on key *B* if and only if *A* and *B* intersect.

Wire representation

On the wire, a key expression is transmitted as a `WireExpr` — a compact representation using a scope (`ExprId`) and an optional suffix:

Full string (scope = 0)

The global scope `ExprId 0` plus the full key expression string as the suffix.

ExprId reference (scope != 0)

A numeric alias for a previously declared key expression (via `D_KEYEXPR`), optionally with an additional suffix appended. This dramatically reduces wire overhead when the same key expression (or a prefix of it) is used repeatedly.

3.2. Session Management

The Session is the main entry point for all Zenoh API operations and coordinates the lifecycle of publishers, subscribers, queryables, and other entities. It holds the Zenoh runtime object, which maintains the state of the connection of the node to the Zenoh network.

Everything starts with the session lifecycle, creation via `zenoh::open()`, session state management, and the relationship between Zenoh nodes in the network.

The session is implemented in a way to provide cheap cloning and shared ownership. Sessions automatically manage resource cleanup when dropped, but can be explicitly closed for better error handling. A Zenoh session is instantiated using `zenoh::open` with parameters specified in the `Config` object.

Sessions can be explicitly closed using the `close()` method, which returns a close completed message, as illustrated in Figure 7.

3.3. Publishers & Subscribers

This section explains the publish and subscribe pattern to manage data-in-motion. The basic data exchange mechanism is the pub/sub communication pattern for data-at-rest or dataflows, meaning continuous streams of data. Publishers send data samples to subscribers that have declared matching key expressions. This is decoupled and many-to-many.

The following Figure 8 illustrates the interaction diagram of a pub/sub communication pattern, this allows distributed applications to publish data (i.e., payload) on a given key expression and subscribers to receive data matching those key expressions.

3.4. Queryables & Replies

The query/reply communication pattern happens when the queriers send queries that are answered by queryables. This enables request/response patterns and access to stored/computed data which is an appropriate mechanism for data-at-rest. The request is expressed in key-expression semantic, and can contain one or double wildcards (`*`, `**`).

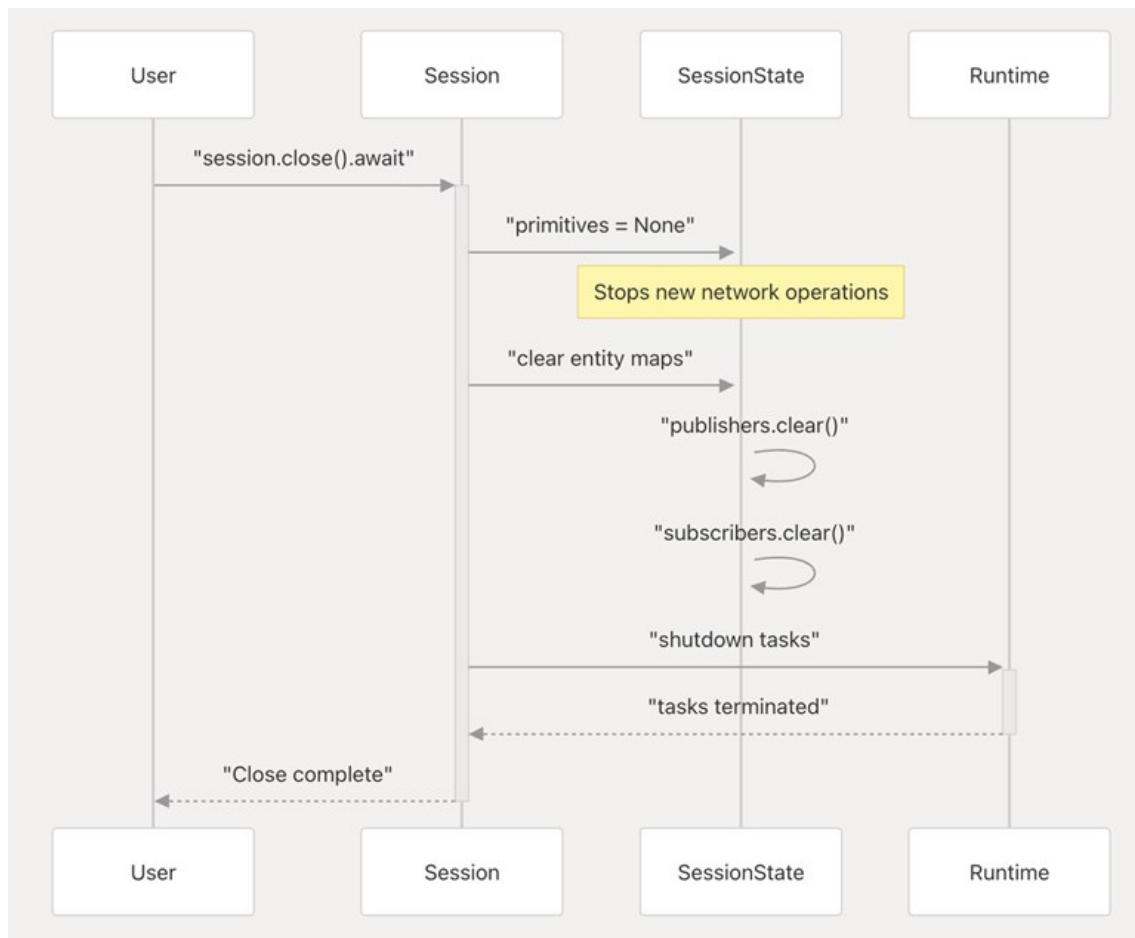


Figure 7: Session closing and cleanup

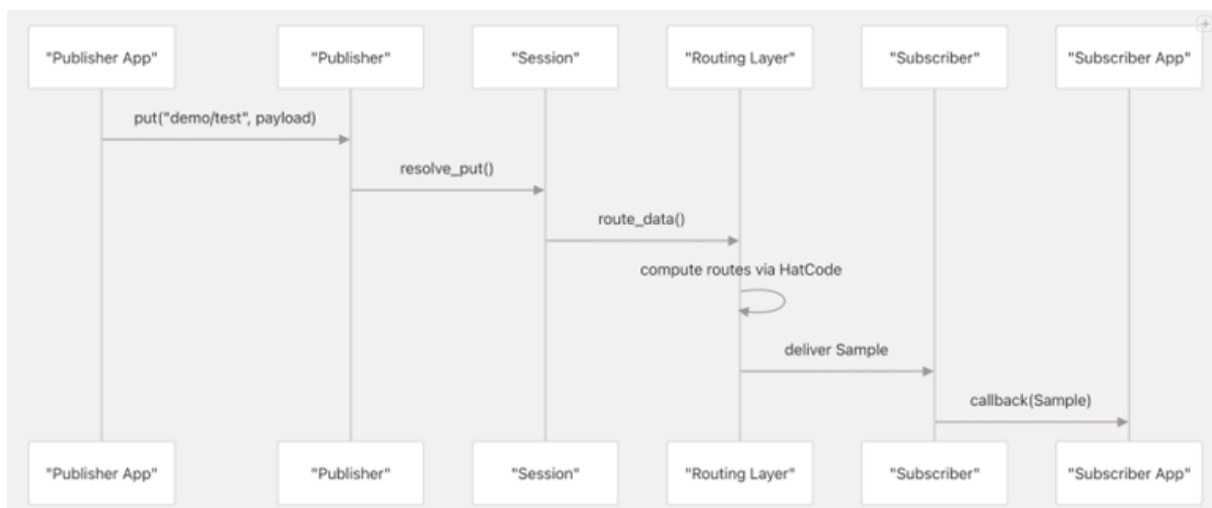


Figure 8: Pub/sub communication pattern

The following Figure 9 illustrates the request/response communication pattern. This pattern supports both point-to-point and broadcast queries across the Zenoh network, and enables distributed applications to request data from remote entities and provide responses.

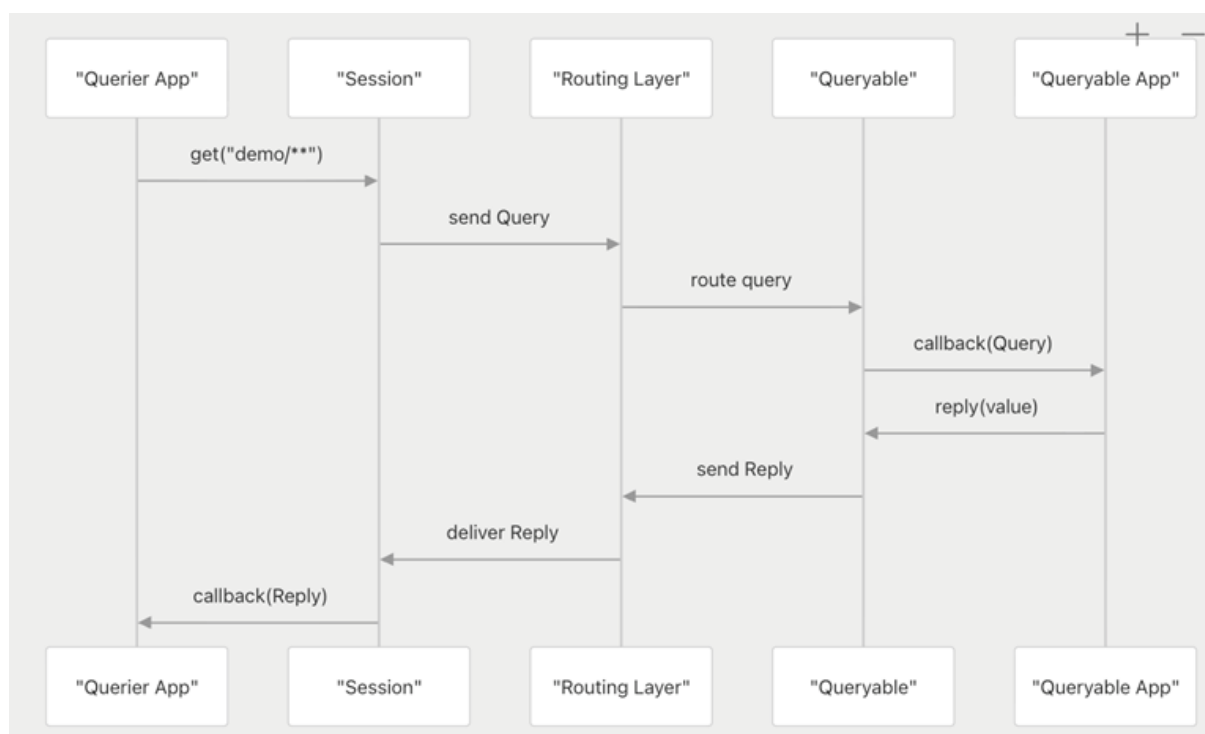


Figure 9: Request/response communication pattern

3.5. Storage Management

This section explains the built-in persistence mechanisms and the supported storage technologies that enable the location transparency property. The storage communication pattern combines subscribers (to store incoming data) and queryables (to answer queries).

Figure 10 presents the storage communication pattern, which effectively links the previously mentioned pub/sub and query/reply patterns, to enable the retrieval of historical data.

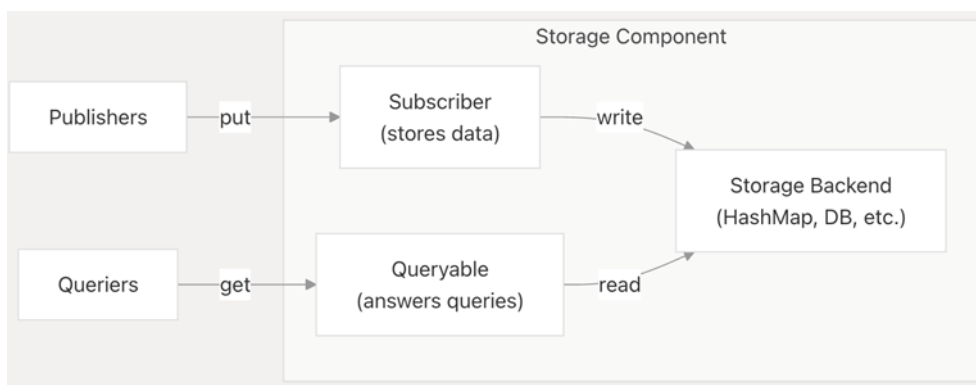


Figure 10: Storage pattern used to retrieve historical data

The storage manager plug-in allows connecting a Zenoh runtime (a.k.a. zenohd) to different storages (e.g., databases). This plugin is a plugin manager itself that loads its own plugins backends specific for the external storage API.

At the moment of writing this deliverable, Eclipse Zenoh supports the following type of storage backends:

- **File System backend:** this backend relies on the host's file system to implement the storage.
- **AWS S3 backend:** This backend relies on Amazon S3 to implement the storage. It is also compatible with MinIO object storage.
- **Rocksdb backend:** This backend relies on RocksDB to implement the storages.
- **Influxdb backend:** This backend relies on InfluxDB server to implement the storages.

The storage is configured in the storage manager plugin configuration in the plugins section of the config.json. Besides that, there are two concepts in the storage manager: "volumes" and "storages". The "volume" is a specific database backend plugin (memory, s3, RocksDB, etc.) plus a concrete backend configuration (database name, username, password, etc.). The "storage" is a mapping from a glob key expression to a volume, defining where data for each path is stored.

3.6. Other Programming Language Binding

This section explains the existing bindings in other programming languages, different from Rust, which is the Core Programming Language. A Rust core is the source of truth and implement Zenoh core functionalities. Zenoh-pico (pure C, embedded) and zenoh-c wrap it, zenoh-cpp wraps both, and zenoh-go wraps Zenoh-C, the Python, Java, and Kotlin implementation wrap Rust directly, while zenoh-ts connects over a WebSocket bridge, see Figure 11.

- Rust – main core repository
- C - There are two implementations with the same API (Zenoh-pico, and Zenoh-C)
- zenoh-c - Rust library binding
- C++ - zenoh-cpp - C++ wrapper over C libraries
- Python - zenoh-python binding for Zenoh
- Kotlin - zenoh-kotlin wraps Rust implementation
- Java - zenoh-java binding for zenoh wraps Rust implementation
- TypeScript - zenoh-ts - WebSocket client for the plugin in zenohd, connects over Websocket bridge
- Go - the Go bindings for Zenoh wraps zenoh-C

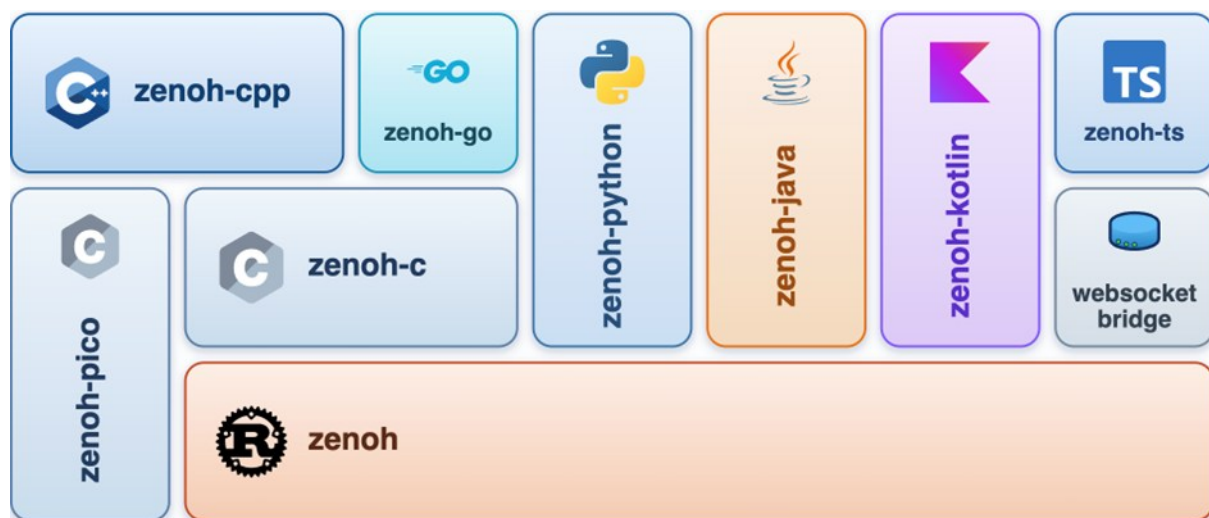


Figure 11: Eclipse Zenoh's API Programming Language bindings

4. Routing & Topology

This section explains one of Eclipse Zenoh's defining features, its topology freedom. The protocol imposes no constraints on how nodes are arranged. Eclipse Zenoh can route the data across arbitrary topologies.

The routing system is the central message dispatching mechanism that determines how messages flow between nodes. The routing layer sits between the Session API and the Transport layer, managing a resource tree for key expression matching, computing routes based on subscriptions and queryables, and applying topology-specific routing strategies through the HAT (Hierarchical Adaptive Topology) abstraction.

4.1. Peer-to-peer mode

By running Eclipse Zenoh in a peer-to-peer (P2P) decentralized system, all the interconnected nodes, called peers, share resources directly with each other without relying on a central server. Each peer can act as both a publisher and a subscriber of data exchange, enabling efficient file sharing and communication.

Figure 12 illustrates an example showing how to implement a 1:N (or N:1) communication graph.

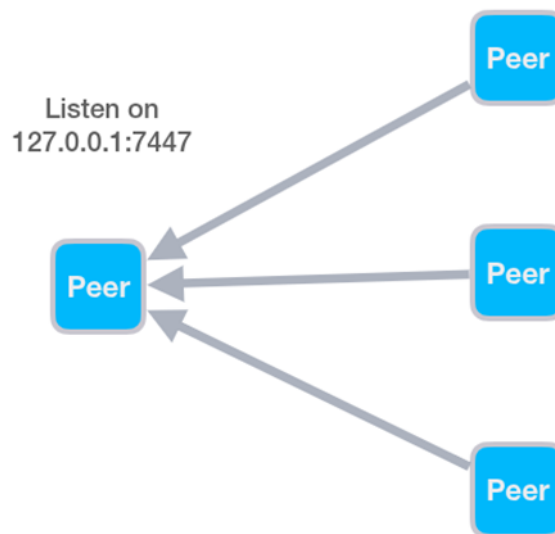


Figure 12: P2P 1:N or N:1 communication

To execute the scenario presented in Figure 12, a single publisher connected to 3 subscribers, here's how we could do it:

```
# we create one publisher
./build/example/z_pub -l tcp/127.0.0.1:7447

# we create 3 subscribers
./build/example/z_sub -e tcp/127.0.0.1:7447
./build/example/z_sub -e tcp/127.0.0.1:7447
./build/example/z_sub -e tcp/127.0.0.1:7447
```

Figure 13 presents a N:N graph, with all the nodes interconnected with each other.

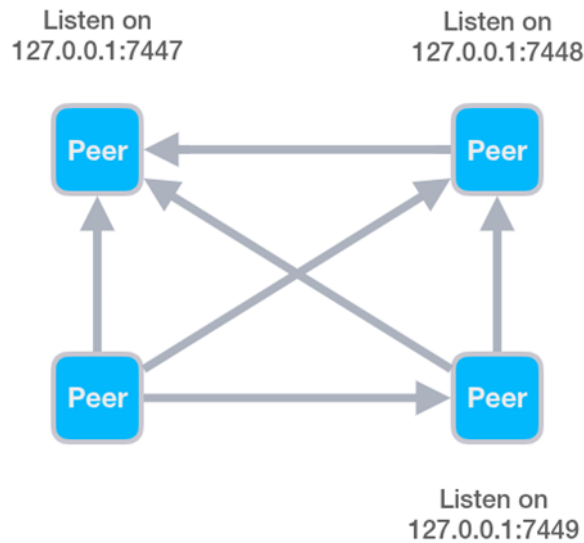


Figure 13: P2P N:N graph network

To execute the N:N clique scenario, in Figure 13, we can execute the following:

```

# Create one publisher
./build/example/z_pub -l tcp/127.0.0.1:7447

# Create 3 subscribers connected to each other
./build/example/z_sub -l tcp/127.0.0.1:7448 -e tcp/127.0.0.1:7447
./build/example/z_sub -l tcp/127.0.0.1:7449 -e tcp/127.0.0.1:7447 -e
tcp/127.0.0.1:7448
./build/example/z_sub -e tcp/127.0.0.1:7447 -e tcp/127.0.0.1:7448 -e
tcp/127.0.0.1:7449

```

Zenoh applications in peer mode run both multicast and gossip scouting to discover other applications or Zenoh routers and connect them. Zenoh applications in peer mode join multicast group 224.0.0.224 on UDP port 7446 and send scout messages on this address to discover local applications and routers. They automatically connect to all accessible peer mode applications and routers they discover. The scouting address and behavior can be configured.

4.2. Brokered and Routed mode

Zenoh **brokered** communication happens when different clients (pub/sub) communicate through a broker node. Communicating peer-to-peer implies establishing multiple sessions with multiple other peers and maintaining a state for those sessions. Maintaining such states can be undesirable for scalability reasons or because the application might run on a constrained device. In this case, the Zenoh application can be configured to operate in client mode. In this mode, the application will maintain, at any given time, a single session with another process (typically a Zenoh daemon: zenohd) that will grant it connectivity with the rest of the system.

Zenoh applications can also be deployed in **routed mode**, with the goal of routing data on behalf of other applications of the subsystem. This allows deploying applications in a mesh topology. By default, router nodes never try to connect to each other automatically and must be configured with the endpoints of the other router nodes they are supposed to connect to.

The **gossip scouting** algorithm in brokered or router mode forwards all local applications and routers that they already discovered to newly scouted applications. This is useful when multicast communications are not available. But applications need to connect first to an entry point to discover the rest of the system. This entry point is typically one or several Zenoh routers but can also be one or several other peers. Those entry points are configured through the connect section of the configuration.

4.3. Regions

A Zenoh system can be subdivided into **regions**. Each region can operate with a different communication model (Peer-to-Peer, Brokered or Routed).

A Zenoh gateway is a node that can serve one or several subregions called south regions. It will act as a representative of the south region(s) in the main region called the north region. The mode of the Zenoh gateway node defines what communication model is used in the north region and thus how the node should act in the north region. A Zenoh gateway can accept any communication model in each of its south regions. Thus, the configuration of the South regions' modes is not necessary. Figure 14 illustrates the combination of the different communications mechanisms across several regions, in light blue.

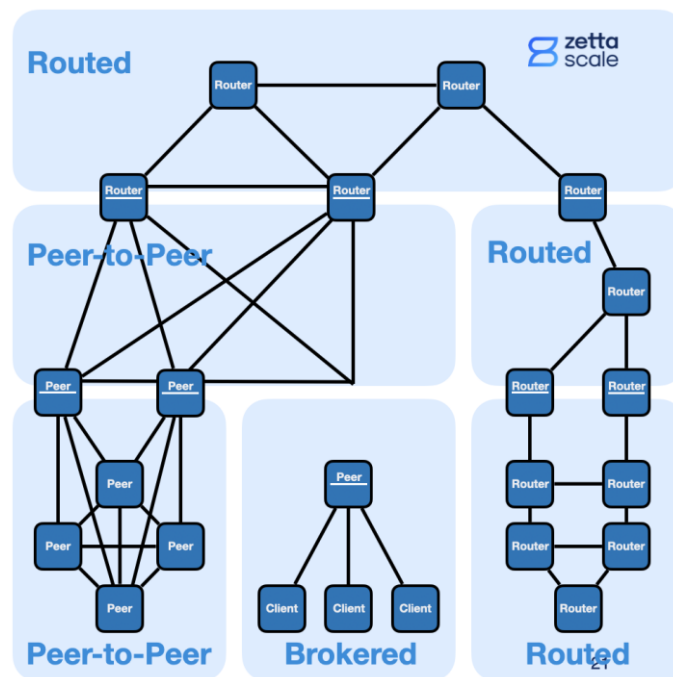


Figure 14: Eclipse Zenoh supported communications mechanisms

5. Security

Security is a critical aspect of any communication protocol, especially one designed to operate across diverse, mixed-criticality environments that may include untrusted components. Eclipse Zenoh's security model aims to minimize the attack surface through robust authentication, encryption, and access control mechanisms. The following features underpin Zenoh's security guarantees

5.1. TLS/mTLS Configuration

Zenoh supports TLS as a transport protocol. TLS can be configured in two ways:

- Server-side authentication: clients validate the server TLS certificate, but not the other way around, that is, the same way of operating on the web where the web browsers validate the identity of the server via means of the TLS certificate.
- mutual authentication (mTLS): where both server-side and client-side authentication is required.

The configuration of TLS certificates is done via a configuration file.

Client configuration: The field `root_ca_certificate` is used to specify the path to the certificate used to authenticate the TLS server. It's important to note that if the field is not specified then the default behaviour is to load the root certificates provided by Mozilla's CA for use with `webpki`. However, if we manage our own certificates, we need to specify the root certificate. Suppose we generated the certificate using MiniCA as explained below, then the configuration file for a client would be:

```
{
  /// The node's mode (router, peer or client)
  "mode": "client",
  "connect": {
    "endpoints": ["tls/localhost:7447"]
  },
  "transport": {
    "link": {
      "tls": {
        "root_ca_certificate": "/home/user/tls/minica.pem"
      }
    }
  }
}
```

When using such configuration, the client will use the provided `minica.pem` certificate to authenticate the TLS server certificate. Let's assume the above configuration is then saved with the name `client.json5`

5.2. Access Control Lists

Eclipse Zenoh routers support Access Control Lists (ACL) rules that restrict which subjects can perform which operations on which key expressions. The following script shows the configuration file for ACL rules, enforcing fine-grained access control, and restricting unauthorized data access

```
{
  access_control: {
    enabled: true,
    default_permission: "deny",
    rules: [
      {
        id: "allow-sensors-pub",
        messages: ["put", "delete"],
        flows: ["ingress"],
        permission: "allow",
        key_exprs: ["temperature/**"],
        subjects: [{ cert_common_name: "sensor-node" }],
      },
      {
        id: "allow-dashboard-sub",
        messages: ["declare_subscriber"],
        flows: ["egress"],
        permission: "allow",
        key_exprs: ["humidity/**"],
        subjects: [{ cert_common_name: "dashboard" }],
      },
    ],
  },
}
```

5.3. Token-based authentication

In addition to mTLS, Zenoh supports username/password authentication at the transport level.

```
{
  transport: {
    unicast: {
      auth: {
        usrpwd: {
          user: "zenoh-user",
          password: "p@ssw0rd**",
          dictionary_file: "/etc/zenoh/credentials.json5",
        },
      },
    },
  },
}
```

6. Embedded & bare-metal support

6.1. Zenoh-pico

Zenoh-pico is the Eclipse zenoh implementation that targets constrained devices, offering a native C API. It is fully compatible with its main Rust Zenoh implementation, providing a lightweight implementation of most functionalities, as illustrated in Figure 15. It requires as little as 2 KB of RAM and runs on virtually any MCU with a C compiler. The main features are:

- Implements the full Zenoh protocol (pub/sub, queryables)
- MISRA-C compliant, limits potential for memory errors
- Static analysis in CI
- Supports FreeRTOS, Zephyr, Arduino, and bare-metal

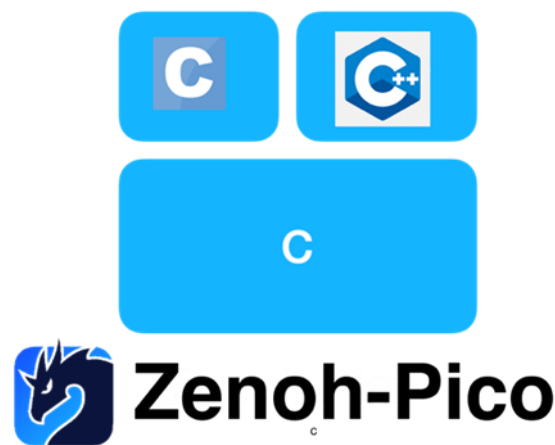


Figure 15: Zenoh-pico implementation and bindings

For more information, see the Zenoh-Pico official GitHub repository for platform-specific guides (ESP32, STM32, Arduino, Zephyr) [5].

6.2. Supported (RT)OSs and protocols

Eclipse Zenoh-pico is a C-native implementation that targets constrained devices, offering an API. It is fully compatible with the main Rust Zenoh core, providing a lightweight implementation of most functionalities.

At the moment of writing this deliverable, zenoh-pico provides support for the following (RT)OSs and protocols:

Table 2: Supported (RT)OSs and protocols

(RT)OS	Transport Layer	Network layer	Data Link Layer
Unix	UDP (unicast and multicast), TCP	IPv4, IPv6, 6LoWPAN	WiFi, Ethernet, Thread
Windows	UDP (unicast and multicast), TCP	IPv4, IPv6	WiFi, Ethernet
Zephyr	UDP (unicast and multicast), TCP	IPv4, IPv6, 6LoWPAN	WiFi, Ethernet, Thread, Serial
Arduino	UDP (unicast and multicast), TCP	IPv4, IPv6	WiFi, Ethernet, Bluetooth (Serial profile), Serial
ESP-IDF	UDP (unicast and multicast), TCP	IPv4, IPv6	WiFi, Ethernet, Serial
MbedOS	UDP (unicast and multicast), TCP	IPv4, IPv6	WiFi, Ethernet, Serial
OpenCR	UDP (unicast and multicast), TCP	IPv4	WiFi
Emscripten	Websocket	IPv4, IPv6	WiFi, Ethernet
FreeRTOS-Plus-TCP	UDP (unicast), TCP	IPv4	Ethernet
Raspberry Pi Pico	UDP (unicast and multicast), TCP	IPv4	WiFi (for “w” version), Serial, USB (CDC)

7. Advanced & Ongoing topics

7.1. Quality of Services (QoS)

Eclipse Zenoh exposes several QoS knobs that let applications trade off latency, throughput, reliability, and resource consumption on a per-message basis. QoS metadata is carried in the `ext_qos` extension block attached to PUSH, REQUEST, and RESPONSE network messages.

QoS Extension Wire Format: the QoS extension is a Z64 (ID=0x1, M=false) whose value encodes the following flags in a single byte, the description for each of the fields is summarized in Table 4.

```

 7 6 5 4 3 2 1 0
 +---+---+---+---+
 |0|r|F|E|D|prio|  low byte of the z64 value; upper bytes are zero
 +---+---+---+---+

```

Table 3: Quality of Services (Qos) specification

Field	Bits	Meaning
Prio	2:0	Priority class (0–7; see table below)
D	3	Don't-drop — use Block congestion control (sender blocks on queue full)
E	4	Express — bypass batching; transmit immediately
F	5	Don't-drop-first (BlockFirst) — first message blocks; rest drop; Patch ≥ 1 required
r	6	Reserved (zero)
	7	Reserved (zero)

Priority: Priority determines the order in which messages compete for transmission resources. Higher-priority messages are scheduled before lower-priority messages when the link is congested. Each priority level has an independent sequence-number space (when QoS is negotiated during INIT).

Table 4: Types of priority

Value	Name	Typical Use
0	Control	Protocol control messages; highest priority
1	RealTime	Hard real-time sensor readings; jitter-sensitive
2	InteractiveHigh	Human-in-the-loop: UI updates, teleoperation
3	InteractiveLow	Low-latency but not hard real-time
4	DataHigh	High-importance application data
5	Data	Default for all user messages
6	Background	Background application data
7	DataLow	Bulk transfers; lowest priority

The default priority is 5 (Data) when no QoS extension is present.

Priority Lanes (QoS-enabled sessions)

When both endpoints include the QoS extension during INIT, the session operates with 8 independent priority queues. Each FRAME carries a QoS extension identifying the priority of its payload; the transport layer can route FRAMES of different priorities over independent channels to prevent head-of-line blocking. In JOIN (multicast), the QoS SN extension carries per-priority sequence numbers.

When the E flag is set, the message bypasses the batching layer and is transmitted immediately in its own FRAME, even if other messages are waiting. This minimizes latency at the cost of per-message network overhead. Express should be used sparingly — typically only for the highest-priority control or real-time messages.

7.2. Reliability & Congestion control

Reliability is negotiated at the channel level (FRAME/FRAGMENT R flag), not per message:

- **Reliable (R=1):** Messages are delivered in order without loss. The transport layer handles acknowledgement and retransmission. Suitable for control messages, declarations, and application data where loss is unacceptable.
- **BestEffort (R=0):** Messages may be dropped or arrive out of order. Suitable for high-frequency sensor data where occasional loss is acceptable, and retransmission would cause staleness.

Congestion control governs the sender's behaviour when the transmit queue is full:

- **Drop (D=0, F=0):** The message is silently discarded if there is no queue space. This prevents a slow consumer from blocking a fast producer. **Default for PUSH.**
- **Block (D=1, F=0):** The sender blocks until queue space becomes available. Suitable when data loss is unacceptable and back-pressure is tolerable. **Default for REQUEST, DECLARE, INTEREST.**
- **BlockFirst (D=0, F=1):** Only the first message in a burst blocks; subsequent messages are dropped. Requires Patch ≥ 1 negotiated during INIT.

7.3. Shared memory (zero copy)

Eclipse Zenoh offers shared memory (SHM) support, which enables zero-copy message transfer between processes on the same host. The system allows large payloads to be transmitted by reference rather than by value, significantly reducing memory copies and improving throughput for high-bandwidth data flows (i.e., video, point-clouds, large images).

Eclipse Zenoh's shared memory support operates at two levels:

1. **Explicit SHM:** Applications allocate buffers from a ShmProvider and publish them directly. The transport layer serializes only the metadata (ShmBufInfo) and sends it to subscribers who can map the same physical memory.
2. **Implicit SHM:** The transport layer automatically wraps large regular buffers in shared memory during transmission if the payload exceeds a configured threshold. This provides transparent optimization without application changes.

The Session API includes SHM-specific types like SharedShmProvider and the ZShmMut (Zenoh shared memory mutex). The most common workflow pattern is obtaining the SHM provider from an existing session and using it to allocate shared memory buffers for publishing.

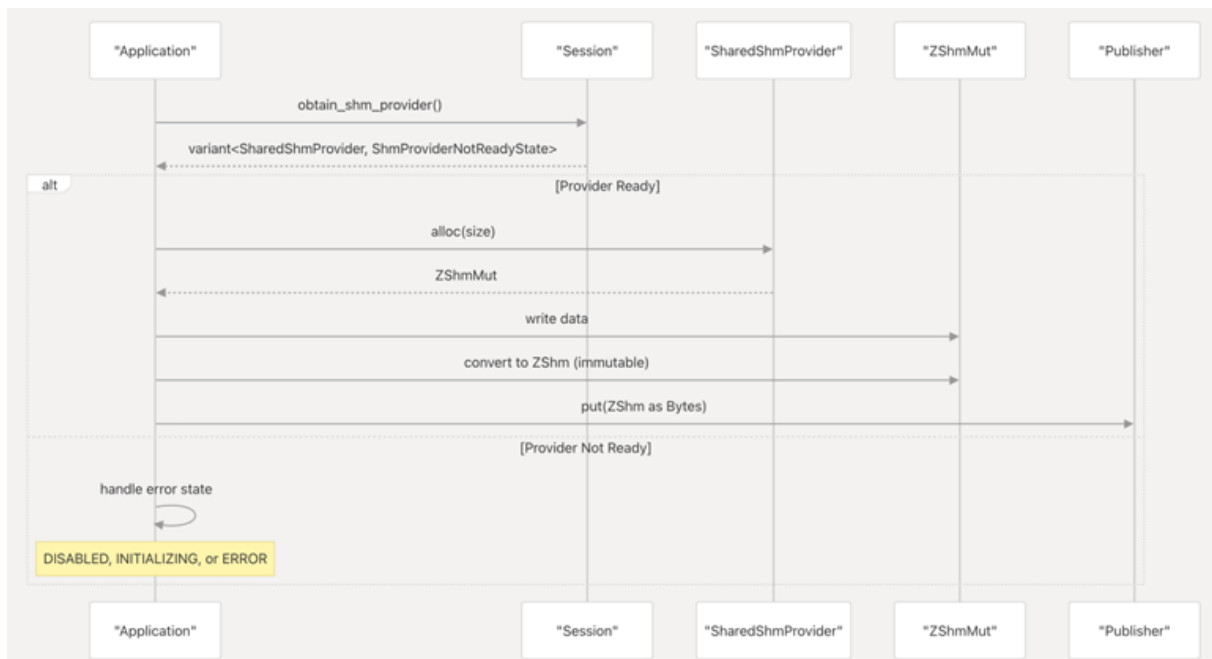


Figure 16: Zenoh Session's Default SHM Provider Workflow

Figure 16 illustrates the Zenoh session's default SHM provider workflow. It starts with the application obtaining the SharedShmProvider. This provider can have different states, once it is initialized and ready, the application starts the allocation process, which returns a mutex (ZshmMut), to block the required memory space. Then the application starts to write the data, followed by a put method to the Publisher.

8. Synergies within UP2DATE4SDV

8.1. Resource monitoring between HAL and ORP

The DLR resource monitoring creates a strong bond between the resource observation developed in WP1 and the update deployment in WP3. During update, upgrade or reconfiguration, the update middleware has to make decisions based on the past and current state of the system, observe the system during the process and continue to do so after the change to confirm the update was successful.

For resource monitoring, we consider different components that are part of the overall monitoring approach. The source of information is called an observer. Observers are running close to the observed unit, i.e. the module consuming, allocating and providing resources in our case, to collect information and provide this data to monitors.

In Figure 17, we show the logical placement of observers in a module (see D3.2 and later WP1 deliverables for more information about modules, observables and observer implementation), gathering resource use allocation before and after allocation in the module. For the implementation there will be very likely just a single observer component instance for each module, collecting all data, as available.

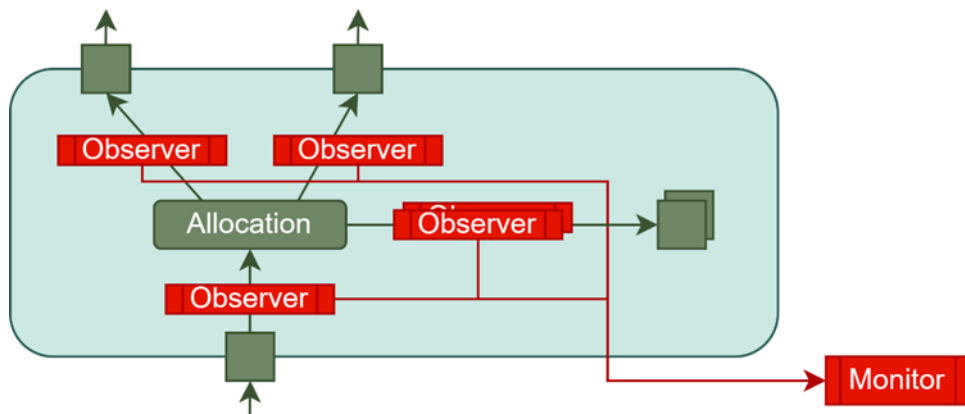


Figure 17: Resource monitoring between HAL and ORP

Observers can collect information of different types (e.g. events, continuous data, discrete data) directly via module-specific APIs, see Figure 18. This information then gets pre-processed by the observers into the required data packets expected by the monitors. This could again be event-based information or periodic measurements.

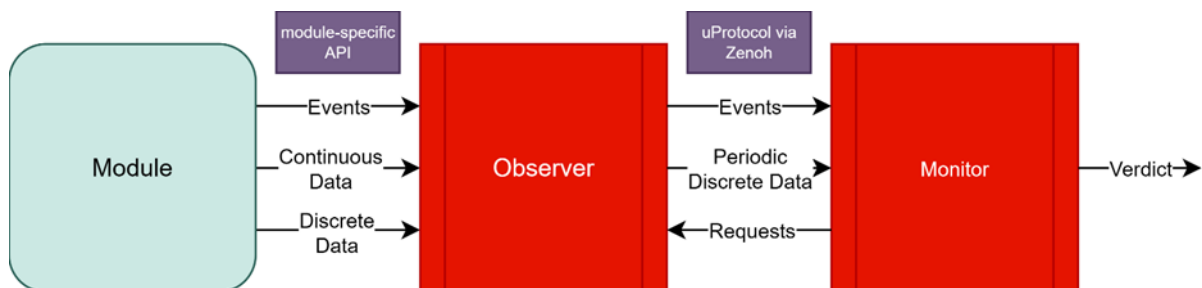


Figure 18: Information collection via specific APIs

The observer then provides this information to the monitors. The relation between observer and monitor is not 1 to 1. Typically, one monitors receives information from multiple observers, while observers can also provide the information to multiple monitors. The location of the monitors within the system provides multiple options as well. A monitor could be placed inside a module, connected to each of the modules observers, to provide a module-internal monitoring.

Monitors can also be placed within a specific module (e.g. a hypervisor partition), collection observer information from all over the system. Each monitor then generates a verdict based on the observer's input. The verdict is based on the formal description of each module. If the modules (or a composition of modules) resource consumption and provisioning is within the specified ranges, the module (or composition of modules) is as specified, and the previously verified isolation properties are given. If a monitor does not give a positive verdict, the system has to react accordingly. In UP2DATE4SDV, this monitoring information will mainly be used during the update process to validate that changes happened correctly after each step, but it can also be used indefinitely to detect misbehavior due to aging or rare software behavior.

The communication between observers and monitors is using uProtocol as communication protocol with Eclipse Zenoh as communication middleware backend. Monitors are currently developed to run inside the Linux OS and are written in Rust, together with the Linux OS module observers they use up-transport-zenoh-rust for the implementation of communication. The implementation for the L4Re hypervisor observers is still to be defined.

8.2. Alignment and Support for Communication Monitoring through the HAL

Although the focus of this deliverable is the early definition of the communication layer and protocols, the communication plane is also falling in the scope of system-level observability requirements. In this respect, bandwidth requirements and actual usage are relevant information both for orchestration and re-configuration decisions and SASE mechanisms.

The communication layer is expected to provide its own monitoring capabilities, exposing information related to the operation of the protocol and the communication services it implements. As BSC is responsible for gathering SoC-level monitoring information and making it available to upper layers in the UP2DATE4SDV stack, we do not aim to replace, duplicate, or prescribe the native monitoring mechanisms in Zenoh/Zenoh-pico. Instead, the objective is to consider how communication-layer monitoring can remain consistent with the broader SoC-level monitoring view developed through the hardware abstraction layer.

In this context, the hardware abstraction layer provides a natural reference point for collecting and exposing low-level resource and platform information. This may include, depending on the target platform and available support, observations related to traffic generated and received by each node through different communication interfaces. These observations are not intended to define the semantics of the communication protocol itself, but to contribute to a coherent view of resource usage and communication activity at the SoC level.

The relationship between the platform-level and communication-level monitoring is therefore one of alignment rather than direct dependency. The communication layer may expose protocol-level metrics, while the SoC-level monitoring infrastructure may expose hardware- and resource-level observations. As part of the integration activities, consistency between these views will be considered, for example in terms of metric naming, scope, granularity, timing assumptions, and the association of monitored traffic with nodes, interfaces, or communication endpoints. The goal is to avoid fragmented

or conflicting monitoring views across the software stack, while leaving the concrete implementation choices to the respective components.

Along this line, we will consider proper means for integration of communication-layer metrics in the Observability API designed and developed in WP1. The goal is to offer a more coherent observability model across the platform, where protocol-level monitoring and SoC-level resource monitoring can coexist and be interpreted consistently. The integration is also expected to favor the definition and deployment of Observed modules, implemented on top of the Observability API, to provide a software-level abstraction for collecting and exposing monitoring information. The integration activity will consider how these different monitoring views (low-level SoC monitoring, communication layer monitoring and Observers) can remain consistent and coherently interpreted.

9. Contributions from the OS community

The Eclipse zenoh project benefits from a strong contribution from the open source community. During the last couple of years, its adoption has increased considerably. The following Figure 19 represents the adoption history of this communication middleware in the last couple of years. It can clearly be stated that the community has continued to use its adoption in a positive trend, which can be measured in the number of stars and followers of the official GitHub repository [4]

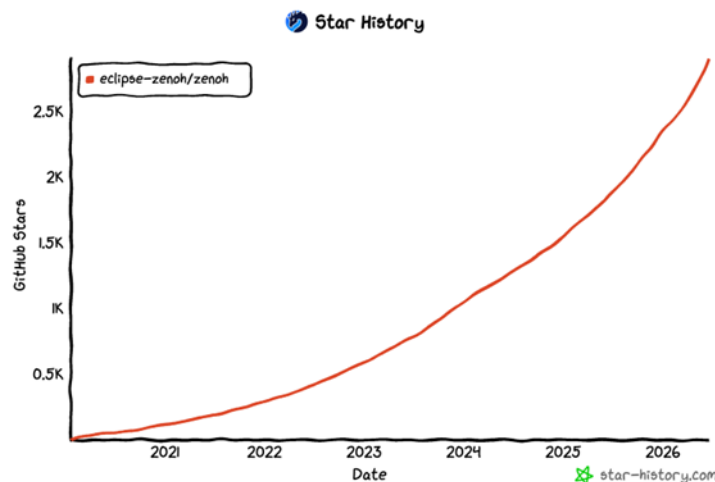


Figure 19: Adoption history of the Eclipse-Zenoh open source project

The following subsection mentions some of the ongoing community efforts that are relevant to the UP2DATE4SDV project

9.1. STM32 ThreadX support

Ubiquity robotics contributed to the Zenoh-pico code base a pull request that adds a new STM32 with ThreadX system implementation. It supports serial transport. Implementation is with circular DMA for data reception.

The steps for building and deploying zenoh-pico into the STM32 platform are described as follows:

- 1.) Create a new project using STMCubeIDE for your target MCU.
- 2.) In CubeMX project configuration tool:
 - Add and enable AZURE-RTOS(threadx) Middleware.
 - Enable UART periphery, set up RX DMA in circular mode and enable UART global interrupt.
 - Move HAL_Tick to TIM peripheral for threadx to work with HAL. (SYS - Timebase source configuration)
- 3.) Generate initialization code with CubeMX.
- 4.) Clone zenoh-pico repository to your project folder (or submodule, copy files)
- 5.) Add zenoh-pico/src to project source folders. Exclude any folders in platforms/* except common and threadx/stm32.

- 6.) Add zenoh-pico/include to project include paths.
- 7.) Edit zenoh-pico/include/config.h to exclude unsupported features or add compile flags
- 8.) Add "hal.h" file to your project. This file should include your target hal headers (ex. #include "stm32f4xx_hal.h")
- 9.) Add defines to project:
 - ZENOH_THREADX_STM32
 - ZENOH_HUART=huartx (huart1 / huart2, etc...)
 - ZENOH_THREADX_STM32_GEN_IRQ=0 (only if you want to write your own interrupt handler)
- 10.) Exclude from build <project_root>/Core/Src/app_threadx.c and replace with one of the z_.c files from zenoh-pico/examples/threadx_stm32/
- 11.) Set TICK_INT_PRIORITY in stm32xx_hal_conf.h to higher value than SysTick. (0 works ok for testing).
- 12.) Set static byt pool size bigger than 25kB.
- 13.) On host compile zenohd with serial support and run with:

```
zenohd -l serial//dev/ttyACM0#baudrate=115200
```

9.2. ROS2 native support (RMW_Zenoh)

ROS 2 is a set of software libraries and tools for building robotics applications. It was designed to be independent of its underlying communication middleware, which is a powerful architectural feature, but one that introduces complexity and overhead. In May 2025, ROS Kilted Kaiju¹ was the first ROS release to support Eclipse Zenoh as a Tier 1 middleware.

Tier 1 communication middleware means that it supports all of the listed Tier 1 operating systems for that particular ROS release, and we verify that support with a full battery of tests that run daily. Moreover, each Tier 1 middleware supports a full SROS2 security suite that includes access control, authentication, and encryption tools. It is also required that all Tier 1 middlewares support all the Tier 1 code quality requirements defined in REP-2004 and be included in the core ROS packages as listed in REP-2005. The following Figure 20 presents the ROS2 internal API architecture overview².

In the more recent release in May 2026, ROS2 Lyrical Luth³, Eclipse Zenoh continue to be one of the supported alternative communication middleware. This effort is carried out in collaboration with Intrinsic and Zettascale. For additional information and roadmap development, please visit the official Rmw_zenoh⁴ code source repository.

¹ <https://www.openrobotics.org/blog/2025/5/23/ros-2-kilted-kaiju-released>

² <https://docs.ros.org/en/rolling/Concepts/Advanced/About-Internal-Interfaces.html#internal-api-architecture-overview>

³ <https://docs.ros.org/en/lyrical/Releases/lyrical/supported-platforms.html>

⁴ https://github.com/ros2/rmw_zenoh

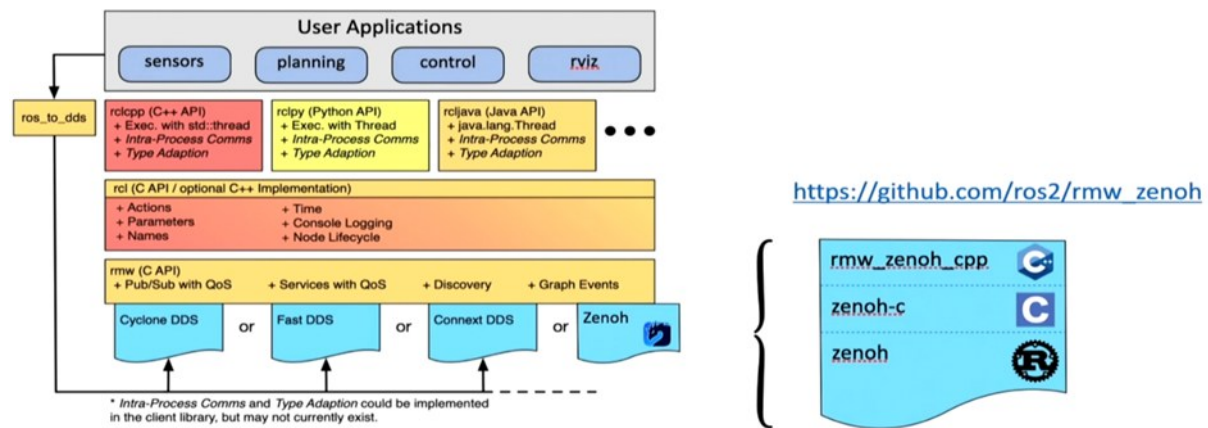


Figure 20: ROS2 Internal API architecture overview

9.3. Eclipse Zenoh Transport Library for Eclipse uProtocol

The uProtocol⁵ provides a small number of communication patterns, exposed via a consistent set of programming APIs that are available across the whole vehicle eco-system (in-vehicle ECUs, cloud, and mobile). This approach enables seamless communication between applications and services, regardless of where they are deployed. Using uProtocol, application developers can focus on implementing differentiating functionality, rather than the low-level plumbing necessary to access the relevant services they require.

The **up-transport-zenoh-rust**⁶ library provides a Rust implementation of the Eclipse uProtocol transport layer (L1) using Eclipse Zenoh as the underlying communication protocol. It enables uProtocol entities to exchange messages across distributed systems using a variety of communication patterns, including Publish/Subscribe, Notifications, and Request/Response (RPC).

The main key capabilities of this integration are:

- **Push-based delivery:** Implements the uProtocol push delivery model, where messages are delivered to registered listeners via callbacks
- **Flexible Configuration:** Supports Eclipse Zenoh's peer, client, and router modes through a builder pattern or external configuration files
- **Priority Mapping:** Automatically maps uProtocol UPriority levels to Zenoh's internal priority classes
- **Resource Management:** Provides a configurable limit on the number of active listeners to prevent resource exhaustion
- **Security Integration:** Leverages Eclipse Zenoh's Access Control Lists (ACL) and authentication mechanisms for address spoofing prevention and unauthorized access control

System mapping from natural language to code entities happens when the Eclipse uProtocol L1 abstractions are implemented on top of the Zenoh protocol, the internal structural components, and the data flow for message delivery. The following Figure 21 illustrates how high-level uProtocol concepts are mapped to specific structures and traits within the codebase.

⁵ <https://uprotocol.org/>

⁶ <https://github.com/eclipse-uprotocol/up-transport-zenoh-rust>

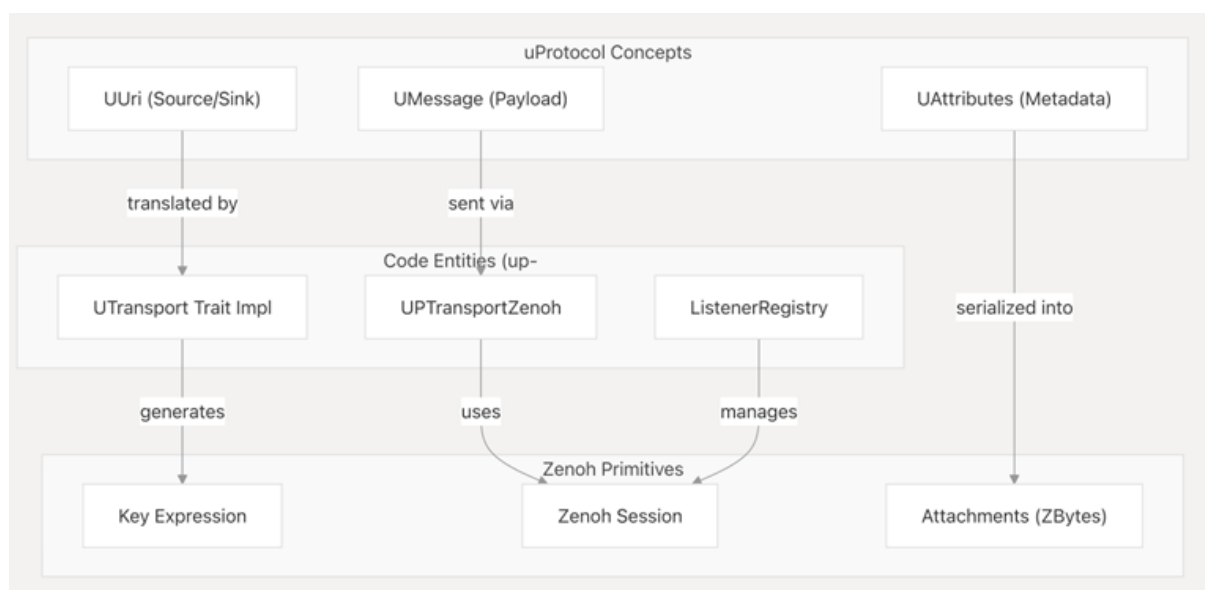


Figure 21: uProtocol to Zenoh Mapping

For additional information, visit the official GitHub repository of the `up-transport-zenoh-rust`⁷ library.

⁷ <https://github.com/eclipse-uprotocol/up-transport-zenoh-rust>

10. Summary and Discussion/Conclusion

D2.3 successfully introduced the initial communication plane for the UP2DATE4SDV project, which is the main goal of Task T2.3. This deliverable presented the reasoning behind the selection of the Eclipse Zenoh communication middleware as the data plane layer for this project.

The UP2DATE4SDV reference architecture was briefly presented; more on this in D4.1, and the intended mapping of such architecture with the flexible, decentralized, and distributed communication framework. This deliverable also serves as guidelines for the coming integration of the mentioned communication plane within the other technical work packages WP3, WP4, and their tasks.

Core concepts, routing topology options, and security support were also presented in this deliverable. Support for embedded and bare-metal systems is also included, as well as the ongoing research and implementation topics. Finally, the ongoing integrations and contributions from the open-source community were mentioned. This deliverable is expected to evolve during the execution of the project, and a second iteration is expected at M30.