Ravensburg Weingarten University

Bachelor Thesis

Developing a Concept for Automated Testing of IPC Middleware – Demonstrated with eCAL

Author: Emircan Tutar
Address: Heyderstr. 7
City: 88131 Lindau
Student ID: 35606
Program: Applied Computer Science

1st Supervisor:
Prof. Dr. Marius Hofmeister
Ravensburg Weingarten University

2nd Supervisor:
M.Sc. Kerstin Keller
Continental

July 14, 2025

Modern software development is characterized by the need for rapid adaptation to evolving requirements. Distributed systems, where components run on separate computing nodes and communicate via defined interfaces, are central to many domains, including automation, robotics, and embedded systems. To enable seamless communication between these components, inter-process communication (IPC) frameworks are widely used. These frameworks facilitate efficient and reliable data exchange, which is essential for the flexible implementation of new features and system updates.

As distributed systems become more complex, ensuring their reliability and functional correctness becomes increasingly important. Communication failures or unexpected interactions between components can have critical consequences, particularly in safety-relevant fields. Therefore, a central challenge emerges: how can the correctness and robustness of IPC-based systems be systematically validated through testing? This thesis addresses this challenge by developing a structured testing concept to the specific demands of IPC middleware environments.

To this end, established testing methodologies including integration, and system-level tests are analyzed and adapted for distributed IPC systems. The emphasis is placed on creating maintainable and reusable test structures that support long-term system evolution. The developed concepts will be demonstrated and evaluated using a practical implementation, ensuring their applicability to real-world middleware systems.

Contents

List of Figures

List of Listings

List of Abbreviations

API	Application Programming Interface
BDD	Behavior-Driven Development
CD	Continuous Delivery
CI	Continuous Integration
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
DDS	Data Distribution Service
eCAL	enhanced Communication Abstraction Layer
IPC	Inter-Process Communication
IoT	Internet of Things
N:1	Many-to-One Communication Pattern
1:N	One-to-Many Communication Pattern
N:N	Many-to-Many Communication Pattern
PID	Process Identifier
QoS	Quality of Service
RPC	Remote Procedure Call
ROS	Robot Operating System
SH	Shell Script
SHM	Shared Memory
TCP	Transmission Control Protocol
UAT	User Acceptance Testing
UDP	User Datagram Protocol

1 Introduction

This chapter introduces the overall topic and scope of the thesis. Section 1.1 outlines the motivation for developing a structured testing approach for IPC middleware. Section 1.2 defines the main objectives of the thesis, and Section 1.3 provides an overview of its structure.

1.1 Motivation

Inter-process communication (IPC) middleware plays a critical role in modern distributed systems. It enables software components (often running on different machines) to exchange data efficiently, coordinate actions, and maintain system coherence. This is especially important in domains such as robotics, automotive systems, and the Industrial Internet of Things (IIoT), where real-time requirements and fault tolerance are essential [1].

Despite the increasing reliance on IPC middleware, systematic and reusable testing concepts for these technologies are still lacking in many projects. While unit tests verify individual components in isolation, they are not sufficient for ensuring the correctness of interactions between distributed nodes, especially in failure-prone or time-sensitive environments.

Middleware frameworks such as ROS 2 and DDS have introduced their own tools and mechanisms to support system-level validation e.g., in ROS 2 `launch_testing` [2], or monitoring utilities in DDS implementations like eProsima Fast DDS and RTI Connext DDS [3], [4]. However, these approaches are often tightly coupled to the specifics of the respective middleware architectures and cannot be directly transferred to other systems.

The enhanced Communication Abstraction Layer (eCAL) is a IPC framework designed for fast, scalable communication across processes and hosts [5]. Although it provides a flexible communication model, eCAL currently lacks a standardized strategy for testing at the system level. While some components are covered by unit or functional tests, there is no unified infrastructure for evaluating end-to-end communication, message integrity under stress, or failure handling in realistic scenarios.

This thesis addresses that gap by developing a generic testing concept for IPC middleware and applying it concretely to the case of eCAL.

1.2 Objective

The goal of this thesis is to develop and evaluate a testing concept that enables structured and automated validation of inter-process communication (IPC) middleware. A particular focus lies on the design of a reusable system-level testing methodology that can be adapted to different middleware architectures. As a practical example, the concept will be implemented and demonstrated using the eCAL framework.

The core objectives are:

- Design a structured testing approach for conducting system tests of IPC middleware in distributed environments.
- Investigate suitable tools, frameworks, and technologies for implementing automated tests.
- Apply the developed testing concept to eCAL and validate its applicability through representative test scenarios.

1.3 Outline

The structure of this thesis is as follows:

- **Chapter 2: Theoretical Foundations**: Introduces testing principles and discusses core concepts of inter-process communication and middleware.
- **Chapter 3: Test Requirements for IPC Middleware**: Identifies key requirements for a testing strategy in IPC systems.
- **Chapter 4: Tool Evaluation and Selection**: Evaluates potential tools and frameworks for implementing the test system.
- **Chapter 5: Design and Automation of Tests**: Describes the test architecture and implementation of selected test cases.
- **Chapter 6: Conclusion and Outlook**: Summarizes the findings.

2 Theoretical Foundations

In this chapter, the theoretical background necessary to understand the development of a system testing framework for eCAL is presented. Section 2.1 introduces essential concepts in software testing, such as test levels, types, and techniques. Section 2.2 explains the concept of distributed systems, including their characteristics and common use cases. Section 2.3 provides the fundamentals of inter-process communication (IPC) and highlights different communication mechanisms. In Section 2.4, the role of middleware in distributed environments is discussed, with a focus on commonly used solutions such as ROS, DDS, and eCAL. Finally, Section 2.5 outlines specific challenges in testing middleware systems, presents current testing approaches, and analyzes the limitations of testing practices in the context of eCAL.

2.1 Software Testing Fundamentals

Software testing is a structured process aimed at verifying that software meets the defined requirements and performs reliably in its intended environment. This section introduces the core concepts of software testing, including test levels, test types, test techniques, and the test pyramid.

2.1.1 Test Levels

Software testing is organized into different levels to detect defects at specific stages of development. According to Ammann and Offutt [6], the main test levels are:

Unit Testing:

Unit testing focuses on verifying individual components or units of a system in isolation. These tests are typically written and executed by developers during the coding phase and aim to detect logic errors or incorrect function outputs early [7], [8].

Integration Testing:

Integration testing validates the interaction between different modules or components. This level ensures that data is correctly passed and interpreted between integrated parts of the application [9], [8].

System Testing:

System testing examines the entire integrated system as a whole. It verifies that the system behaves correctly under various conditions, including performance, security, and usability aspects [10], [8].

In distributed systems, however, the boundary between integration testing and system testing can become blurred. Since individual components often run on different machines or communicate asynchronously, validating their interaction frequently involves testing parts of the overall system behavior as well. As a result, what is formally considered integration testing may already include system-level characteristics such as network latency, synchronization, or fault tolerance.

Acceptance Testing:

Acceptance testing, also called User Acceptance Testing (UAT), determines whether the software fulfills the business requirements and is ready for deployment. It is usually performed by end users or stakeholders [11].

2.1.2 Test Types

There are two major types of tests commonly used in software quality assurance:

A) Functional Testing

Functional testing ensures that software features behave according to their specifications. It is typically performed through techniques like equivalence class partitioning and boundary value analysis [7].

B) Non-functional Testing

Non-functional testing addresses aspects such as performance, reliability, maintainability, and usability. These qualities are essential for a software product to function effectively in production environments [11].

2.1.3 Test Techniques

Various techniques are used to design efficient and targeted test cases:

Black-Box Testing

Black-box testing validates the system's functionality without considering its internal code structure. Testers use input-output analysis to confirm expected behavior [10], [8].

White-Box Testing

White-box testing is based on knowledge of the internal structure and logic of the system. Testers examine decision paths, loops, and control structures to ensure code coverage [9], [8].

Gray-Box Testing

Gray-box testing combines elements of black-box and white-box approaches. It requires partial knowledge of the internal workings to create more targeted and effective test cases [6], [8].

2.1.4 Test Pyramid

The test pyramid is a widely recognized concept used to structure testing strategies in a scalable and maintainable way. It was introduced by Mike Cohn [12] to help development teams allocate their testing efforts effectively across different levels of abstraction.

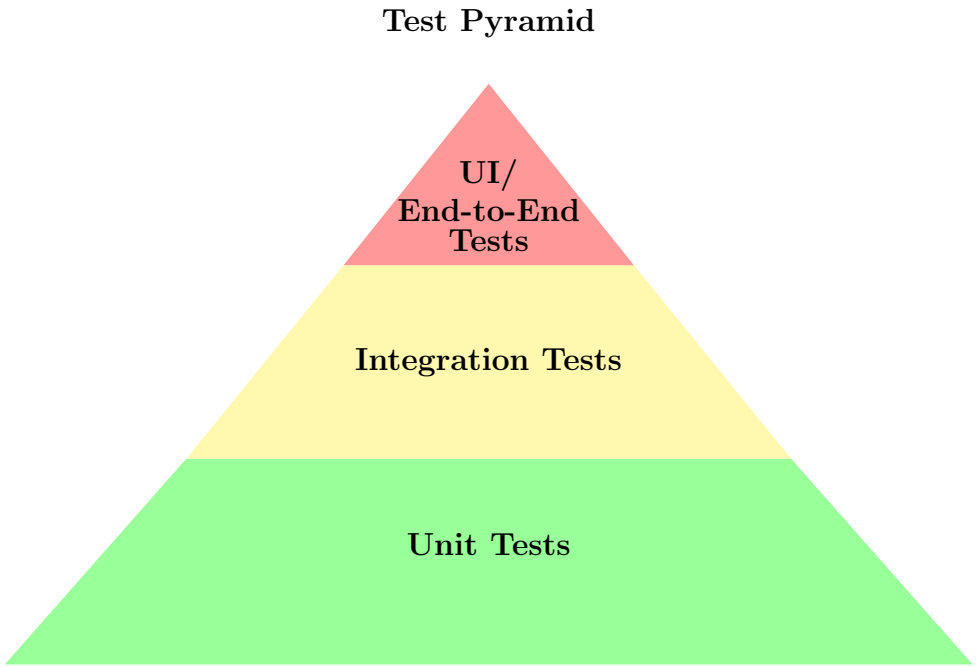


Figure 1: The Test Pyramid illustrating test distribution across levels (adapted from Cohn [12]).

At the base of the pyramid lies a large number of unit tests. These are fast, automated, and verify individual components or functions in isolation. Unit tests provide immediate feedback during development and help detect issues early, making them the foundation of efficient software testing.

The middle layer consists of fewer integration tests. These tests check how different modules or components interact with each other and help uncover interface-related defects that are not visible during unit testing.

At the top of the pyramid are the end-to-end or UI tests. These are high-level tests that simulate real user interactions across the entire system. Although essential for validating system behavior from a user perspective, they are typically slower, more fragile, and more expensive to maintain.

As visualized in Figure 1, the pyramid illustrates the principle that lower-level tests should be more numerous and faster, while higher-level tests should be fewer and more focused. This structure promotes reliable, maintainable, and cost-effective testing workflows.

2.2 Distributed Systems

2.2.1 Definition and Concepts

A distributed system is a network of independent computers that appears to users as a single coherent system. In distributed systems, multiple computing devices communicate and coordinate their activities by passing messages to achieve a common goal [13]. Such systems consist of independent components located on different networked computers, which interact with each other by exchanging messages.

The primary goals of distributed systems include improving performance, supporting scalability, and provide reliable and fault-tolerant operations. By coordinating tasks across multiple nodes, such systems can make more efficient use of available processing power, memory, and data storage [1].

Figure 2 illustrates a basic distributed system consisting of multiple independent nodes communicating over a network. Each node may serve a specific role, such as providing services, accessing shared resources, or coordinating tasks.

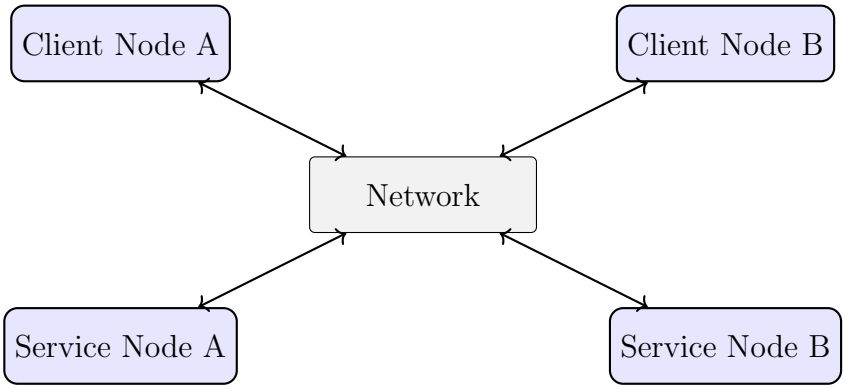


Figure 2: Basic architecture of a distributed system with multiple nodes communicating via a central network

2.2.2 Characteristics, Benefits, and Challenges

Distributed systems have several key characteristics, including concurrency, scalability, transparency, and fault tolerance. Concurrency refers to multiple processes executing simultaneously. Scalability describes the capability of the system to grow easily in size and workload. Transparency ensures the complexity of the system remains hidden from users, providing an impression of a single unified system. Fault tolerance describes the system's capability to continue operation even when individual components fail [13].

However, developing and managing distributed systems can be challenging. Problems related to communication delays, synchronization between processes, security, and managing the complexity of the overall system must be addressed effectively to ensure reliable operations [1].

Despite these challenges, distributed systems offer several practical advantages that make them well suited for modern computing environments. One of the most important benefits is resource utilization: tasks can be divided among multiple machines, leading to better performance and efficiency. Additionally, scalability allows systems to grow by simply adding new nodes, without the need to redesign the entire architecture. Distributed systems also support high availability, as the failure of a single component does not necessarily affect the rest of the system. This improves fault tolerance and makes distributed architectures ideal for applications where continuous operation is critical. Finally, they enable geographic distribution, allowing systems to operate across locations and support global access to data and services [1], [13].

2.2.3 Application Areas

Distributed systems are widely used across many industries. In automotive systems, they support advanced driver-assistance systems (ADAS), autonomous driving, and vehicle communication systems. Robotics heavily relies on distributed systems for complex coordination tasks, such as collaborative robotics, autonomous navigation, and real-time control. The Internet of Things (IoT) is another prominent application area, where distributed systems enable efficient communication between countless smart devices and sensors, facilitating smart homes, smart cities, and industrial automation [1], [13].

2.3 Inter-Process Communication (IPC)

2.3.1 Overview and Importance

Inter-process communication (IPC) refers to mechanisms that allow processes to communicate and exchange data. IPC is a crucial component in distributed and concurrent systems, enabling different software processes running on one or multiple computers to coordinate and share information effectively [14]. Effective IPC mechanisms are essential for ensuring the smooth and reliable functioning of complex software systems, especially in critical applications such as automotive control systems, robotics, and industrial automation [15].

2.3.2 Common IPC Mechanisms

There are several commonly used IPC mechanisms, each suitable for different scenarios and requirements. The most frequently used methods include shared memory, message passing, and remote procedure calls [14], [15].

Figure 3 illustrates a simplified comparison of common IPC mechanisms. Shared memory allows processes to access a common memory region directly. Message passing transfers data through explicit send/receive actions. RPC abstracts communication by allowing a process to call functions in another process as if they were local.

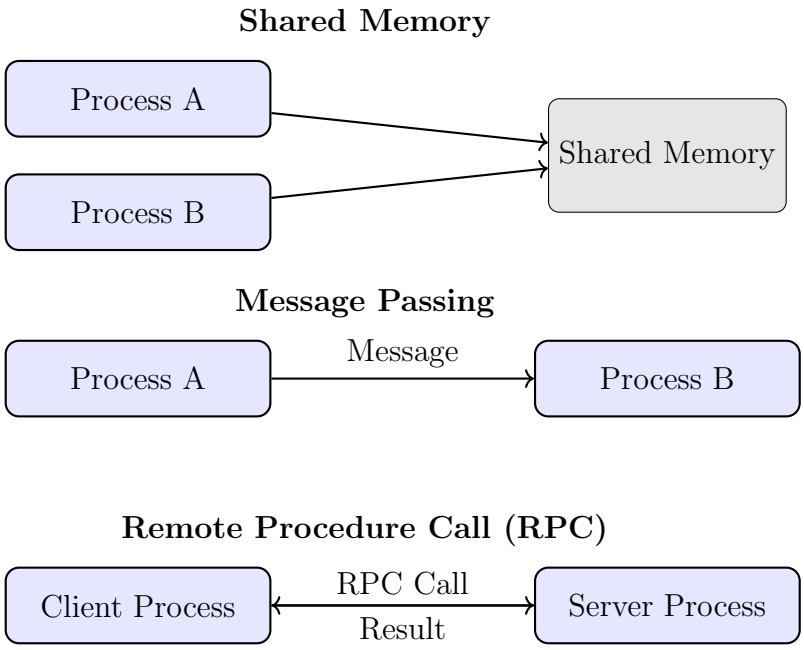


Figure 3: Comparison of common IPC mechanisms: Shared Memory, Message Passing, and Remote Procedure Call (RPC)

Shared Memory:
Shared memory is one of the fastest IPC methods, allowing processes to communicate directly by accessing a common area of memory. Processes write and read data directly into this shared space, avoiding the overhead of explicit communication. However, managing synchronization and ensuring data integrity can become challenging and requires careful implementation of synchronization methods, such as semaphores or mutexes [14], [16].

Message Passing:

Message passing involves processes communicating through messages. This approach ensures a clear separation between processes, providing safer communication. Messages are sent explicitly from one process to another using defined communication channels, such as pipes or sockets. While message passing typically has higher overhead compared to shared memory, it is easier to manage synchronization, and it allows better scalability in distributed environments [15].

Remote Procedure Calls (RPC):

Remote Procedure Calls enable processes to invoke procedures or functions on remote systems as if they were local calls. RPC abstracts network communication, making distributed system interactions easier to develop and maintain. RPC is widely used in distributed applications and middleware solutions due to its simplicity and clear programming model, despite some performance overhead from serialization and network transmission [1].

2.3.3 Zero-Copy Communication

Zero-copy communication is an advanced IPC technique designed to minimize unnecessary data copying between processes. Traditional communication methods often involve copying data multiple times, significantly reducing performance and increasing latency. Zero-copy mechanisms avoid this overhead by allowing direct data transfers between processes, usually through shared memory. By reducing the data copies, zero-copy enhances performance and efficiency in systems with high throughput and low latency requirements, such as high-performance computing or real-time systems [17].

2.4 Middleware in Distributed Systems

While IPC mechanisms define how processes communicate, middleware abstract these and provide a structured interface for distributed applications.

2.4.1 Definition and Role of Middleware

Middleware is software that sits between applications and the underlying operating system or network infrastructure, enabling easier communication and coordination within distributed systems. Its main role is to simplify development by abstracting complexities associated with networked and distributed computing, such as communication protocols, data exchange, and interoperability between diverse systems [18]. Middleware allows software developers to focus more on the application logic rather than the low-level communication and network details.

2.4.2 Common Middleware Solutions

Various middleware technologies are available today, each designed to address specific communication and coordination needs. Some widely used middleware solutions include the Robot Operating System (ROS) [19], the Data Distribution Service (DDS) [20], and the enhanced Communication Abstraction Layer (eCAL) [21].

Table 1 presents a layered architecture comparison between ROS, DDS, and eCAL. Each middleware abstracts the communication stack differently, but all provide core functionality for distributed system communication.

Feature	**ROS**	**DDS**	**eCAL**
Open Source	✓	✗	✓
Multi-Platform (Linux, Windows)	✓	✓	✓
Publish/Subscribe Model	✓	✓	✓
Request/Response (RPC)	✓	✓	✓
Real-Time Support	✗	✓	✗
Shared Memory Support	✗	✗	✓
TCP Support	✓	✓	✓
UDP Support	✓	✓	✓
Real-Time-Pub-Sub Protocol (RTPS)	✗	✓	✗
Integrated Dev Tools	✓	✗	✗
Built-in Visualization	✓	✗	✗

Table 1: Feature comparison of ROS, DDS, and eCAL.

The comparison in Table 1 illustrates that while all three middleware frameworks support basic communication patterns like publish-subscribe and RPC, their strengths vary depending on the intended application domain. ROS excels in developer support and tooling, DDS offers real-time capabilities and strict QoS control, and eCAL stands out with fast and efficient IPC and shared memory support.

Robot Operating System (ROS):

ROS is an open-source middleware widely used in robotics. It provides a structured communication framework that includes services like message passing, data visualization, hardware abstraction, and numerous tools for robotics development. ROS simplifies building complex robotic systems by offering standardized communication interfaces and extensive community-supported tools and libraries [19].

Data Distribution Service (DDS):

DDS is a standardized middleware designed primarily for real-time and high-performance distributed applications. It employs a publisher-subscriber communication model, where components communicate by exchanging messages without direct connections between producers and consumers. DDS provides reliable and scalable communication, making it suitable for critical applications such as automotive systems, aerospace, industrial automation, and healthcare [20].

Enhanced Communication Abstraction Layer (eCAL):

The enhanced Communication Abstraction Layer (eCAL) is an open-source middleware designed specifically for efficient inter-process communication (IPC) in distributed environments. It simplifies the exchange of data between processes running on the same device or across multiple networked computers. eCAL uses a decentralized publish-subscribe architecture, allowing multiple processes to communicate directly without relying on a central broker [5].

2.5 Testing in Middleware

2.5.1 Specific Challenges

Testing middleware within distributed systems presents several challenges due to the complexity of distributed architectures and the abstract nature of middleware functionality. Middleware often operates across heterogeneous platforms and coordinates communication between independent software components. As Tanenbaum and van Steen emphasize, ensuring interoperability and consistent behavior across diverse systems introduces technical and architectural complexity [13].

A core challenge is compatibility, as middleware must support various hardware, operating systems, network protocols, and data formats. According to Coulouris, this requires middleware to provide standard abstractions while hiding platform-specific details [1].

Performance and scalability also represent key testing concerns. Middleware must support low-latency communication and efficient resource use under variable loads. Stallings notes that poor synchronization or inefficient resource management at the middleware layer can lead to system-wide bottlenecks [14].

Finally, fault tolerance and security must be evaluated, particularly since middleware can be a single point of failure or a vector for attack in distributed architectures [23].

2.5.2 Existing Approaches and Best Practices

Testing middleware systems effectively requires an understanding of the system architecture and the communication patterns it supports. Burns and Wellings suggest that model-based testing is particularly suited for middleware due to its ability to describe behavior across abstraction layers [24].

Integration testing plays a vital role in evaluating message routing, service discovery, and state synchronization among components. Additionally, system-level testing validates functional correctness and non-functional requirements, such as real-time constraints and message reliability [25].

Best practices include the use of simulation environments for performance evaluation under controlled conditions, as well as leveraging monitoring and logging tools to capture middleware-level interactions for post-test analysis. Gorton and Liu also highlight the use of profiling and benchmarking frameworks to test middleware scalability across node clusters [25].

In addition, techniques such as fault injection or failure simulation are particularly valuable when testing distributed middleware. These methods deliberately introduce faults such as: process termination, artificial network loss, or resource exhaustion. These observe how the system responds under edge cases. For containerized systems, this can be achieved through controlled Docker container crashes or simulated network partitions. Such tests are essential to verify the middleware's robustness and fault tolerance under realistic failure scenarios.

2.5.3 Comparison of Middleware Testing

To evaluate the practical suitability of different middleware frameworks for testing, it is important to compare their support for common testing methods and tools. While all frameworks follow a publish-subscribe architecture and support integration testing in some form, their features differ significantly in terms of system-level support, real-time validation, simulation tools, and fault injection capabilities [24], [25].

The following list summarizes the most relevant testing aspects and how they are supported by ROS, DDS, and eCAL:

- **Integration Testing**
  - **ROS:** Supports structured integration tests, especially in ROS 2 using `launch_testing` to coordinate multi-node setups [2].
  - **DDS:** Enables integration testing via its standardized API; test setups are typically built manually per implementation [20].
  - **eCAL:** Currently, eCAL does not offer dedicated support for automated integration testing [21].
- **System-Level Testing**
  - **ROS:** ROS 2 includes tools like `launch_testing` for end-to-end validation of system behavior and topic exchange [2].
  - **DDS:** No native system test runner; system-level validation requires custom frameworks [4].
  - **eCAL:** No native system test runner available [21].
- **Fault Injection**
  - **ROS:** Faults can be simulated by terminating nodes or using simulation tools like Gazebo [19].
  - **DDS:** Supports disconnects, dropped messages, or QoS manipulation; commercial tools (e.g., RTI Connext) may offer advanced options [4].
  - **eCAL:** No built-in support for fault injection exists [21]. This thesis explores basic fault scenarios using Docker, such as process crashes or network interruptions.

- **Quality-of-Service (QoS) Testing**
  - **ROS:** ROS 2 allows limited QoS testing through its DDS layer; ROS 1 does not support QoS configuration [2].
  - **DDS:** Offers comprehensive QoS testing across multiple dimensions (e.g., reliability, deadline, liveliness) [20].
  - **eCAL:** Offers only basic pub/sub configuration; no standardized QoS model is provided [21].
- **Monitoring and Logging**
  - **ROS:** Tools like `rosbag`, `rosnode`, and `rostopic` provide strong monitoring and logging capabilities [19].
  - **DDS:** Vendor-specific tools such as RTI Admin Console and Fast DDS Monitor allow inspection and statistics collection [3], [4].
  - **eCAL:** Built-in tools such as eCAL Monitor and the logging API provide basic runtime analysis [21].
- **Simulation Support**
  - **ROS:** Integrates tightly with simulation environments like Gazebo and Ignition, allowing scenario-based testing [19].
  - **DDS:** No native simulation environment is provided, but integration with external simulators is possible [4].
  - **eCAL:** Provides recording and replay functionality for data exchange but no virtual simulation of sensor behavior [21].

This comparison shows that ROS provides strong tooling for development and simulation-based testing, while DDS offers the most comprehensive support for Quality-of-Service validation. In contrast, eCAL does not natively support automated system-level testing or fault injection [21]. However, this thesis introduces a new testing approach later based on Robot Framework and containerized environments to address this gap [27].

3 Test Requirements for IPC Middleware

The previous chapter explained the main concepts related to software testing, distributed systems, inter-process communication (IPC), and middleware. It also presented common middleware frameworks and discussed their differences and challenges in testing.

To develop a structured and practical testing approach, it is important to clearly define what needs to be tested. The next chapter describes the specific requirements for testing IPC middleware. These requirements include functional aspects, such as correct message delivery, and non-functional aspects, such as timing and robustness. They form the foundation for designing a reusable and automated test system, which will be applied to eCAL in the later parts of this thesis.

3.1 Analysis of Test Requirements

Middleware-based systems pose unique testing requirements that differ from classical monolithic systems. Key challenges include the following:

- **Timing behavior:** IPC systems are sensitive to delays, jitter, and message timing. Integration tests must capture whether messages are delivered in time and in correct order.
- **Data consistency:** Published data must arrive at all intended subscribers with correct content. Corruption or loss during transmission must be detectable.
- **Component coordination:** Integration tests must ensure that multiple processes synchronize correctly. This includes proper startup/shutdown sequences and fault handling.
- **Fault Injection:** To evaluate the system's robustness, integration tests should include controlled fault scenarios. These may involve simulating message loss, disabling specific network routes, or forcing a publisher to stop sending data during transmission. Such tests help verify how well the middleware handles unexpected problems.

- **Transport abstraction:** Middleware supports different transport mechanisms (e.g. TCP, shared memory, UDP). Integration tests should verify that core functionality works across transports.

3.2 Functional and Non-Functional Test Objectives

Integration testing in IPC middleware not only verifies the correct exchange of data between distributed components but also examines how reliably and efficiently the system performs under realistic conditions. These test objectives are generally classified into two categories: *functional* and *non-functional* requirements [8], [13], [25].

Functional Objectives

Functional tests focus on whether the system behaves correctly in terms of its specified input-output behavior. In IPC middleware, this includes the delivery of messages, handling of subscriptions, and validation of message formats [8], [9]. Functional requirements describe *what* the system must do, independent of the conditions under which it is executed.

Examples of functional test objectives include:

- Ensuring that a subscriber receives only messages for topics it subscribed to.
- Verifying that malformed messages are rejected without crashing the system.
- Testing that RPC requests return valid responses or trigger appropriate timeout behavior.
- Validating transport abstraction by checking that core communication works across TCP, UDP, and shared memory.
- Verifying component coordination such as startup sequences and shutdown order across distributed nodes.

Non-Functional Objectives

Non-functional tests evaluate *how well* the system performs its tasks focusing on the system's efficiency, robustness, and quality attributes. These include timing, fault tolerance, scalability, and platform independence [8], [14].

Key examples of non-functional test objectives:

- **Timing Behavior:** Measure latency, jitter, and message ordering under various load and delay conditions.
- **Robustness and Recovery:** Evaluate how the system handles injected faults such as message loss, simulated crashes, or network disruptions.
- **Scalability and Stability:** Assess performance and resource usage as the number of processes and message rates increase.
- **Cross-Transport Behavior:** Ensure consistent system performance across different transport mechanisms.

According to *Basiswissen Softwaretest* [8], non-functional requirements often affect user acceptance. Although they are sometimes assumed implicitly (e.g. reliability, maintainability), they require explicit validation.

Note on Benchmarking

While certain test goals such as latency and throughput measurements are commonly associated with benchmarking, this thesis does not aim to conduct a benchmarking campaign. Instead, such metrics are used selectively within integration tests to validate non-functional characteristics such as timing behavior and robustness. The primary focus lies on correctness and communication stability under defined test conditions, rather than comparative performance evaluations across systems or configurations.

3.3 Communication Scenarios

To create meaningful test cases, representative communication scenarios must be modeled. The following scenarios form the foundation for test design:

- **One-to-one communication:** A single publisher sends messages to a single subscriber. Used to test baseline latency and correctness.
- **One-to-many communication:** A publisher sends to multiple subscribers. Tests consistency and fan-out behavior.
- **Many-to-one communication:** Multiple publishers send to a shared topic. Tests message interleaving and synchronization.

- **Multi-host communication:** Tests cross-host communication and transport compatibility.
- **Failure scenarios:** Introduce network failure, drop publishers/subscribers mid-test, or overload the system to test resilience.

3.4 Design of a Modular Test Strategy

To address these requirements, a modular test architecture is proposed, consisting of the following layers:

1. **Test scenarios:** Each scenario defines participating nodes, timing, and expected outcomes.
2. **Test orchestration:** A framework (e.g., Robot Framework), combined with supporting tools (e.g., Docker), manages the execution flow and the lifecycle of test components.
3. **Test Probes:** Lightweight probes are used to monitor the communication process during testing. They capture timestamps, logs, or message payloads for later verification.
4. **Result evaluation:** Automated scripts validate logs, output files, or metrics against expected results.

This layered design allows reuse of components across different test cases and enables automation for continuous testing.

3.5 Summary

This chapter defined the requirements for integration testing in IPC middleware environments and proposed a test architecture that is both reusable and automation-friendly. In the following chapters, this test architecture will be applied to an actual middleware using eCAL as a representative implementation platform.

4 Tool Evaluation and Selection

This chapter presents an evaluation of available tools and test frameworks that could be used to support the integration testing of IPC middleware. The goal is to identify solutions that align with the defined testing requirements and environmental constraints presented in the previous chapter. Based on a set of defined selection criteria, several tool candidates are reviewed and assessed in terms of their compatibility with eCAL and the specific demands of testing distributed communication systems.

4.1 eCAL as a Middleware Solution for IPC Testing

Overview and Architecture

eCAL (enhanced Communication Abstraction Layer) is a flexible middleware designed for efficient inter-process communication (IPC). It supports various transport layers, such as shared memory for communication on a single machine, and UDP or TCP for communication over a network. eCAL automatically selects the most suitable transport method based on system conditions [21].

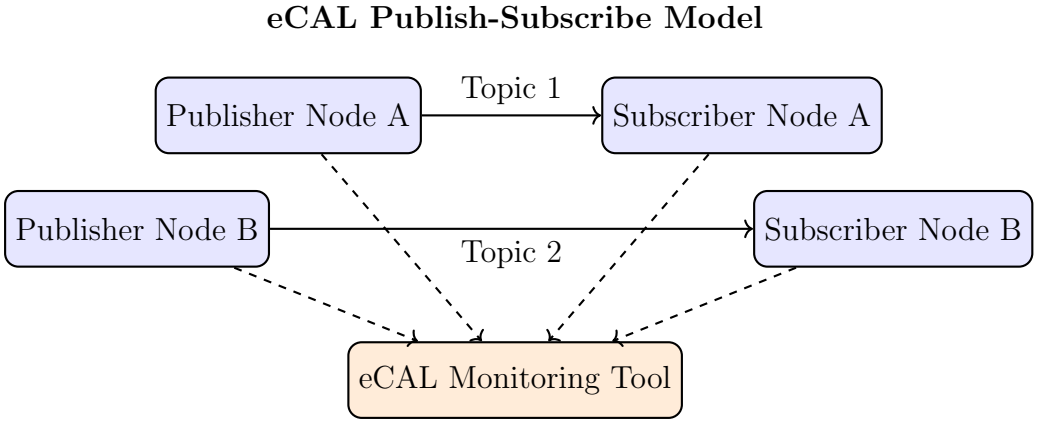


Figure 4: Simplified architecture of the eCAL framework with publisher, subscriber, and monitoring components

Figure 4 shows the basic structure of the eCAL system. It includes publisher and subscriber nodes that exchange messages over topics. A built-in monitoring tool observes the communication to support debugging and system analysis.

Main Features

eCAL offers several features that are useful for real-time and distributed systems:

- **High Performance:** Thanks to zero-copy in shared memory mode, eCAL can transmit data with very low latency and high throughput [5], [21].
- **Multi-Language Support:** It provides APIs for C++, C#, Python, Java, and Go, which makes it easy to use in different projects [21].
- **Cross-Platform Support:** eCAL works on Windows, Linux, and macOS, making it suitable for many system environments [21].
- **Integrated Tools:** eCAL includes tools for monitoring, recording, replaying, and analyzing communication, which are helpful during development and testing [5], [21].

Benefits and Challenges

Middleware like eCAL helps to manage communication between software modules in a structured way. It reduces the need to handle low-level networking code and allows systems to scale more easily. On the other hand, middleware adds complexity and can make debugging and performance tuning more difficult. Especially in real-time systems, the overhead from abstraction can become a limitation.

Limitations of Current Testing Approaches

Although eCAL includes helpful tools for monitoring and debugging, it does not provide a complete solution for full system testing. Most available tools are designed for developers and focus on individual components rather than entire communication workflows. To test distributed behavior, such as message loss over UDP, transmission delays over TCP, or network interruptions, additional tools and setups are required. In practice, this often means running publishers and subscribers in isolated Docker containers to simulate real network conditions. Without containerization, it is difficult to create realistic and repeatable test environments for distributed communication.

4.2 Test Tool Selection Criteria

To choose suitable testing tools and frameworks, several criteria were defined:

- **Language Support:** The framework should support the languages already in use in the development environment, particularly C++ and Python.
- **Automation Capabilities:** Support for test automation, headless execution, and integration into CI/CD systems is essential.
- **Multi-Process Orchestration:** The tool should be able to handle the orchestration of multi-process communication patterns, such as publish-subscribe and request-response.
- **Platform Compatibility:** The framework must work on Linux and Windows, as this is the primary target platform.
- **Observability and Reporting:** Tools should offer logging, result exporting, or structured test result visualization (e.g., HTML reports).

4.3 Evaluation of Tool Candidates

GoogleTest

GoogleTest is a widely used testing framework for C++ applications. It provides many useful macros and assertions that help developers write automated tests for individual functions, classes, and modules. In the eCAL project, GoogleTest is already used to verify internal logic, data handling, and utility functions. This makes it a good choice for unit testing, especially during early development stages.

However, GoogleTest is mainly designed for unit testing and does not support tests that involve multiple running processes. Integration testing of IPC middleware usually requires testing communication between different processes such as publishers and subscribers. To use GoogleTest in this context, additional scripts or wrapper functions are needed to start processes and simulate communication. While this is possible, it increases the complexity of test setups and reduces maintainability. Therefore, GoogleTest should be combined with other tools when performing full integration tests. More information is available in the official GoogleTest documentation [26].

Robot Framework

Robot Framework is a general-purpose, open-source test automation framework. It follows a keyword-driven approach, allowing users to write tests in a readable and structured way. Tests are usually written in plain text and can be run through the command line or integrated into automated systems.

One of the key strengths of Robot Framework is its flexibility and support for integration testing. With built-in libraries such as `Process`, `OperatingSystem`, and `BuiltIn`, it can start and stop applications, check logs, and validate output. These features are especially helpful for testing IPC middleware like eCAL, where several processes must run in parallel and communicate correctly. For example, Robot Framework can launch both publishers and subscribers, wait for data exchange, and verify the results.

Another important feature is automatic report generation. After each test run, Robot Framework produces HTML and XML reports that show test results, timing, and logs. This is very useful for continuous integration pipelines that run tests regularly. The official Robot Framework User Guide provides further details [27].

Gauge Test

Gauge is a modern test framework developed by ThoughtWorks. It lets users write test scenarios in Markdown format, which helps make tests easier to read and maintain. These scenarios are then connected to test code written in programming languages like Java, Python, or C#. Gauge also supports running tests in parallel and generating HTML reports.

Although Gauge has many useful features, it is mostly used for testing web applications, APIs, and user interfaces. It is not specifically designed for backend systems like IPC middleware. Also, its community is relatively small, and there are fewer plugins available compared to larger frameworks.

Using Gauge for testing IPC would require custom scripts and additional effort to manage process orchestration. Because of this, Gauge is not the best fit for testing systems like eCAL. Further documentation can be found on the official Gauge website [28].

Katalon and TestComplete

Katalon and TestComplete are test tools mainly used for testing web interfaces, APIs, and desktop applications. They often include graphical interfaces for designing and running tests, and they support features like recording user actions and parameterizing input values.

However, these tools are not intended for testing IPC systems. They do not support low-level message handling or process orchestration, which are important when testing middleware like eCAL. In addition, most of these tools are not open source, and they often rely on graphical interfaces, which makes them less suitable for headless environments or automation using Docker and CLI.

For these reasons, these tools are not recommended for integration testing of eCAL or similar IPC frameworks. More information can be found in their official documentation [29], [30].

behave and Behavior-Driven Development (BDD)

Another approach worth mentioning is Behavior-Driven Development (BDD), which combines principles from test-driven development (TDD) and domain-driven design (DDD). BDD focuses on describing software behavior in a language that both developers and non-developers can understand [31].

A popular Python framework for BDD is `behave` [32], which uses the Gherkin syntax (Given–When–Then) to define executable specifications. The actual test logic is implemented in Python, offering flexibility for system and integration testing scenarios. However, orchestration and reporting must be implemented separately using custom Python scripts or shell tools.

4.4 Tool Selection for Middleware Testing

Table 2 presents an overview of evaluated tools in this Chapter.

Framework	CLI	Multi-Process	Open Src	CI/CD	Reports
Google Test	✓	Partial	✓	✓	XML only
Robot Framework	✓	Yes	✓	✓	HTML, XML
Gauge Test	✓	Limited	✓	✓	HTML
behave (BDD)	✓	Partial	✓	✓	Text/ Custom
Katalon	✗	No	✗	✗	HTML, GUI
TestComplete	✗	No	✗	✗	HTML, GUI

Table 2: Comparison of test frameworks for IPC middleware testing with focus on multi-process orchestration

Following the evaluation of multiple testing frameworks, Robot Framework was identified as a suitable tool for integration testing in middleware-based systems. It fulfills key requirements such as support for keyword-driven test definitions, automation of command-line tools, and coordination of multiple parallel processes. These features are valuable when testing communication patterns like publish-subscribe or service-based exchanges.

An alternative to Robot Framework is `behave`, which is suitable for teams following a BDD approach. It enables human-readable scenario definitions and flexible test logic using Python. However, since `behave` lacks built-in features for process orchestration and structured reporting, it requires additional scripting and infrastructure to reach the same level of automation. For the use case of testing IPC middleware like eCAL, where containerized process management and test reproducibility are critical, Robot Framework remains the more practical and efficient choice.

For unit-level testing of individual components, GoogleTest remains the preferred choice. Its compatibility with C++ and strong support for isolated module testing make it a practical solution for verifying internal logic, data processing, and error handling at the code level. By combining Robot Framework for system-level validation with GoogleTest for component-level checks, a comprehensive and layered testing strategy can be achieved.

4.5 Summary

This chapter presented a comparative evaluation of tools and frameworks suitable for testing IPC middleware systems. After outlining the architecture and current testing limitations of the eCAL framework, a set of tool selection criteria was defined to guide the evaluation process. Multiple frameworks were assessed, including GoogleTest, Robot Framework, Gauge, behave, Katalon, and TestComplete.

The results show that while GoogleTest is well-suited for unit testing of C++ components, it lacks support for orchestrating multi-process communication. On the other hand, Robot Framework offers strong capabilities for integration testing, including process control, CLI automation, and report generation, making it a practical choice for testing middleware. Gauge and behave offer readable test syntax but require additional scripting for IPC use cases. GUI-based tools like Katalon and TestComplete are not suitable for headless, distributed middleware testing.

5 Design and Automation of Tests

This chapter presents the design and implementation of a test system focused on **integration testing** of the eCAL middleware. In integration tests, multiple components or subsystems are combined and tested as a group to verify that they interact correctly. This is especially relevant for communication middleware, where the interaction between distributed publishers, subscribers, servers, and clients must be validated under real-world conditions.

The primary goal is to evaluate eCAL's behavior in realistic scenarios involving inter-process communication across various transport mechanisms. To achieve this, a container-based test infrastructure was created that allows isolated, repeatable, and automated execution of test cases. The infrastructure combines Robot Framework for test orchestration with Docker for process isolation, fault simulation, and reproducibility.

The test environment targets publish-subscribe and RPC-based communication patterns and includes both normal operation and fault scenarios (e.g., process crashes or network failures). All tests are designed as integration tests and verify the end-to-end behavior of communicating components, rather than isolated units.

The structure of this chapter is as follows: Section 5.1 introduces the architecture of the test environment and its components. Section 5.2 details the infrastructure and core implementation elements. Section 5.3 explains how edge cases and failure scenarios are handled. Section 5.4 presents selected test cases along with their validation logic. Section 5.5 explains the automation of test execution and how it integrates with a CI/CD pipeline. Section 5.6 describes the visualization of test results via web-based reports. Finally, Section 5.7 provides a summary of the findings and key insights.

5.1 Architecture of the Test Environment

The integration tests for eCAL are executed in a containerized environment using Docker. This approach provides a clean, reproducible, and isolated setup, which is especially important when simulating complex communication scenarios or fault conditions. By separating each component into its own container, it becomes easier to control timing, simulate crashes, or observe

network behavior. Figure 5 provides an overview of the overall test setup and its components.

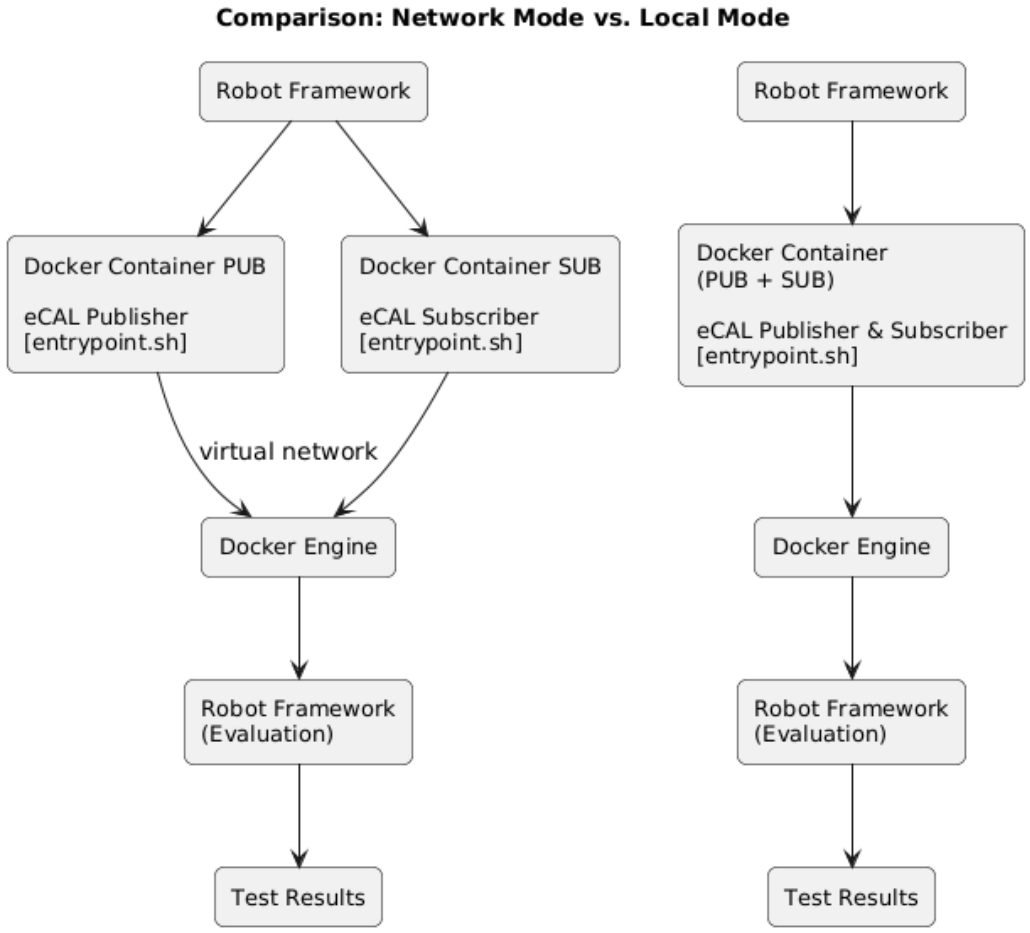


Figure 5: Overview of the test environment architecture

The core components of the test architecture are:

- **eCAL Nodes**: Each eCAL node is built as a separate C++ application and runs in its own Docker container. These processes communicate via eCAL using shared topics.
- **Docker Network**: All containers are connected to a common virtual Docker network (e.g., `ecal_test_net`), which enables communication using the selected transport layers.

- **Robot Framework**: The test logic is defined using Robot Framework. It is responsible for starting containers, injecting faults, checking logs, and validating test success criteria.
- **Entrypoint Scripts**: Each container uses an `entrypoint.sh` script that decides which executable to run (e.g., a publisher, subscriber, server, or client), based on input arguments.

The test environment supports all major eCAL communication modes:

- `local_shm`: Local communication using shared memory.
- `local_udp`: Local communication using UDP multicast.
- `local_tcp`: Local communication using TCP.
- `network_udp`: Network communication using UDP multicast.
- `network_tcp`: Network communication using TCP.

This flexible design allows for testing a wide range of scenarios, including those where communication modes must be explicitly selected or changed during runtime. The environment also allows easy simulation of errors, such as killing a process, disconnecting a network, or injecting delays, which are critical for the validation of robustness.

5.2 Implementation of the Test Infrastructure

The integration test infrastructure for eCAL was designed to be modular, reusable, and easy to extend. It allows new test cases to be added or removed quickly by following a consistent folder structure and execution pattern. This is particularly useful for one of the most common use cases in test-driven development: adding a new test scenario with minimal overhead.

Each component of the infrastructure follows a clear responsibility, which improves maintainability and simplifies debugging. The separation of test logic (e.g., message exchange and failure simulation) and infrastructure logic (e.g., container orchestration) ensures clean design and scalability.

1. DockerLibrary.py

This custom Python library for Robot Framework provides keywords to manage Docker containers. It simplifies the interaction with Docker by offering

reusable commands that can be used directly in test cases. The library hides complex details and enables clear and reliable test automation.

Some of the main features of the DockerLibrary are:

- **Starting and stopping containers:** Containers can be launched with custom names, arguments, or even fixed IP addresses if needed.
- **Log collection:** Output from running containers is collected and made available for test evaluation and debugging.
- **Exit code handling:** The library waits for the container to finish and reads its exit code to determine if the test passed or failed.
- **Network control:** Containers can be connected to or disconnected from Docker networks during the test, allowing the simulation of network failures.
- **Container tracking:** Containers are stored in memory using unique names, so they can be easily referenced and stopped later in the test.
- **Error handling:** If a container has already been removed or cannot be found, a warning is shown instead of causing a crash.

A common use case is stopping and cleaning up test containers after the test has finished. Listing 1 shows a simplified version of the `Stop Container` keyword in Python, while Listing 2 shows how this keyword is used in a Robot Framework test case.

```
@keyword
 def stop_container(self, name):
   if name in self.containers:
      try:
          self.containers[name].stop()
          self.containers[name].remove()
      except NotFound:
          BuiltIn().log_to_console(f"Container {name}
          already removed.")
      finally:
          self.containers.pop(name, None)
```

Listing 1: Example keyword implementation in `DockerLibrary.py`

```
*** Settings ***
 Library           lib/DockerLibrary.py

*** Test Cases ***
 Start Test
  Start Container    my_test_container

 ---Test Implementation here---

 Cleanup After Test
  Stop Container    my_test_container
```

Listing 2: Calling `Stop Container` in a Robot Framework test

2. GlobalPathsLibrary.py

This library handles dynamic path and tag resolution across all test modules. It defines the active test case, provides access to configuration scripts, and ensures that Docker image names, container labels, and result folders are consistently named and resolved.

3. ecal_config_helper.h / .cpp

This shared C++ utility is responsible for configuring communication parameters such as transport mode and layer activation. It includes two central helper functions:

- `wait_for_subscriber()` – ensures that messages are only published once a subscriber is available (see Listing 3).
- `setup_ecal_configuration()` – configures the desired communication mode (e.g., UDP, TCP, SHM) for each process (see Listing 4).

```
void wait_for_subscriber(std::string topic){
  while (!has_subscriber(topic)){
    sleep_for(std::chrono::milliseconds(100));
 }
}
```

Listing 3: Simplified wait_for_subscriber logic

```
1  if (mode == "local_udp"){
2   config.communication_mode =
3                        eCAL::eCommunicationMode::local;
4    if (is_publisher){
5       config.publisher.layer_priority_local =
6                         {eCAL::TransportLayer::udp_mc};
7    } else {
8        config.subscriber.layer.shm.enable = false;
9        config.subscriber.layer.udp.enable = true;
10       config.subscriber.layer.tcp.enable = false;
11    }
12  }
```

Listing 4: Partial example for UDP setup

5. build_images.sh

This shell script builds the Docker images required for each test. It invokes CMake to compile the C++ test binaries and packages them together with all dependencies. This ensures consistency between test environments and local development.

4. entrypoint.sh

This is the entry script executed inside each container. It uses the given arguments (e.g., role: publisher or subscriber) to launch the correct binary with matching configuration. It also supports scenarios where both publisher and subscriber or multiple binarys are launched together inside the same container.

Summary

The interaction and execution flow of the test system, from Robot Framework orchestration to container-level execution and evaluation, is illustrated in `Figure 6`. This infrastructure enables consistent and isolated execution across all test cases while remaining flexible enough to simulate failures, delays, or disconnects. New modes or roles can be added without modifying the core infrastructure.

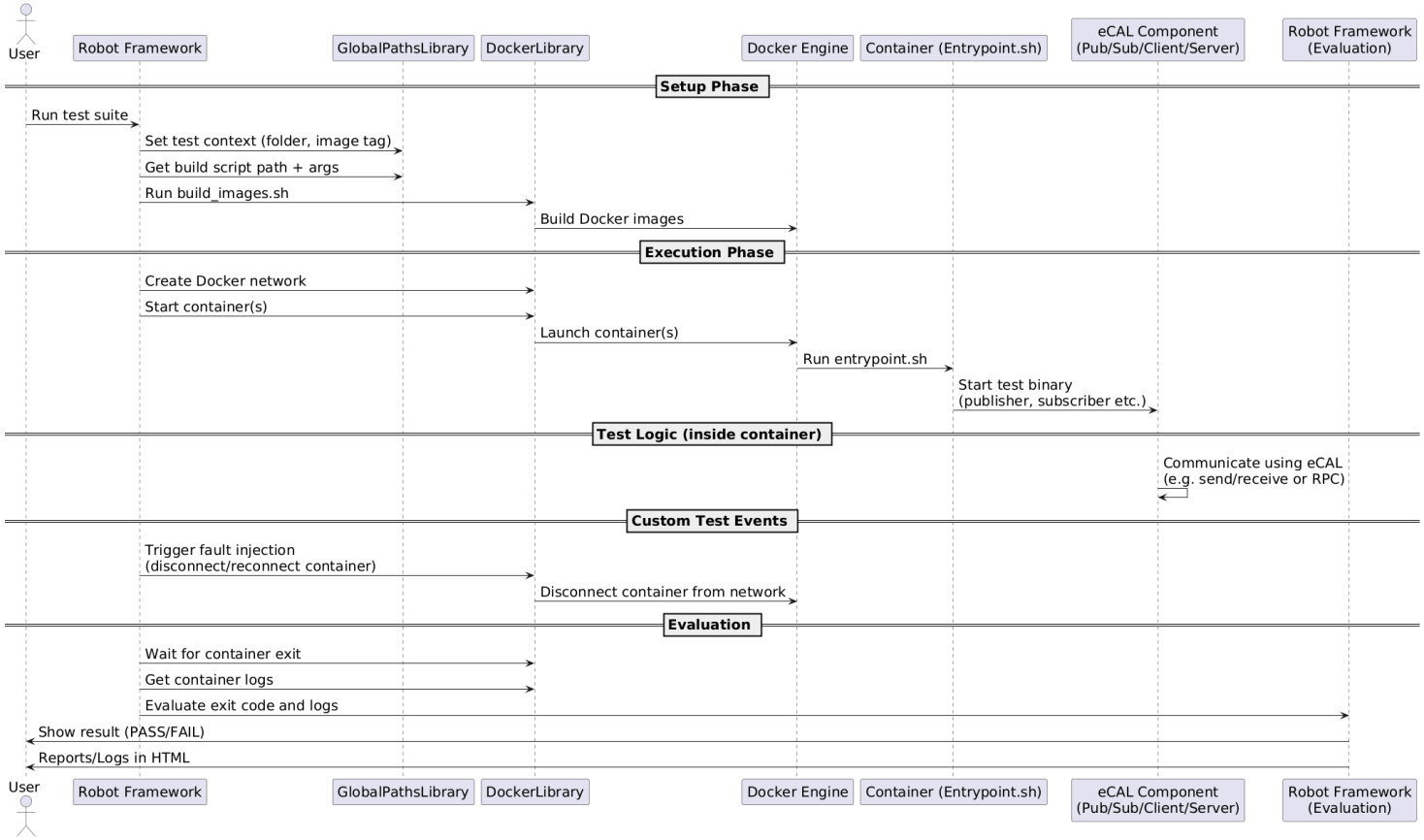


Figure 6: Test flow for eCAL-based integration testing

5.3 Handling of Edge Cases

A key requirement for testing communication middleware is the ability to validate system behavior under abnormal or unexpected conditions. Distributed systems are inherently prone to failures, such as delayed messages, process crashes, or temporary network interruptions. Therefore, in addition to verifying functional correctness, the test environment must support controlled simulation of such conditions.

This section introduces the strategies used to simulate and evaluate edge cases and fault scenarios within the automated test infrastructure. The goal is to uncover potential weaknesses in eCAL's communication logic and validate its resilience.

Fault Injection Capabilities

To enable realistic fault simulations, the Robot Framework infrastructure supports fault injection mechanisms:

- **Crash simulation:** Docker containers running eCAL nodes can be stopped manually during the test to simulate process crashes.
- **Network disconnection:** Containers can be detached from their virtual Docker network at runtime to mimic network cable unplug or wireless dropout.
- **Network reconnection:** Containers can be reconnected to the network after a delay, allowing tests to evaluate recovery behavior and reconnection logic.
- **Delays and timeouts:** Artificial delays are injected using sleep commands or slow message loops, which help simulate jitter or processing backpressure.

These mechanisms are reusable and can be invoked in any test scenario. For example, Listing 5 shows how a network disconnect and reconnect is triggered inside a Robot Framework test case:

```
1 Disconnect Container From Network   test_client   ecal_net
2 Sleep    6s
3 Reconnect Container   test_client   ecal_net  ip=172.28.0.10
```

Listing 5: Simulating a temporary network disconnect and reconnect

Observed Edge Case Categories

The following edge case categories were explicitly targeted in test scenarios:

- **Missing receivers:** Publishers start sending before any subscribers are active. These tests confirm that no blocking or crashes occur and that late subscribers still receive messages once available.
- **Message loss and recovery:** By crashing publishers or subscribers mid-test, message loss is deliberately introduced. The system's ability to continue communicating correctly with the remaining nodes is then evaluated.
- **Reconnect scenarios:** Clients or servers are disconnected from the network and reconnected after a delay. These tests assess whether communication can be re-established without restarting the processes.
- **Large message handling:** Tests involving very large payloads (e.g., 50 MB) are used to verify memory usage and crash behavior, especially under Zero-Copy Shared Memory configurations.

Validation Methods

To verify correct behavior during and after edge conditions, the following evaluation strategies are used:

- **Exit code checks:** Each component returns an exit code indicating success or failure. Robot Framework interprets these codes to determine whether the test passed.
- **Payload inspection:** Received messages are validated for correctness (e.g., expected values like 42, 43, 44) and assigned to the correct source based on payload content.
- **Log analysis:** All container logs are automatically collected and printed in the test report. This helps identify at which point failures occurred and what internal state was reached.

Conclusion and Role in Test Design

The ability to inject faults and simulate edge cases is a central part of the overall test strategy. It ensures that communication logic is not only functionally correct but also resilient under stress and partial failure. Many of the following test cases build directly on these capabilities, combining normal communication with injected disconnects or crash conditions. This approach helps create a realistic, production-like test environment that increases confidence in system stability and fault tolerance.

5.4 Test Case Design

This section presents selected integration test cases that were implemented to validate the behavior and robustness of the eCAL middleware. The tests are executed using Robot Framework and Docker-based containers, building upon the infrastructure described in Section 5.2. Each test targets a specific aspect of IPC such as message delivery, fault handling, or service requests and follows a consistent structure in terms of execution and evaluation.

To provide a clearer overview, the test cases are grouped into two categories:

- **Normal Operation Scenarios**: Tests that verify the expected behavior of the middleware under standard conditions.
- **Fault Injection Scenarios**: Tests that simulate failure situations such as process crashes or network disconnects.

Table 3 summarizes all implemented test cases, including their classification and main focus.

ID	Title	Type
TC 1	**Publisher to Subscriber**	Normal Operation
TC 2	**Multiple Publishers/Subscribers**	Normal Operation
TC 3	**RPC Ping Request-Response**	Normal Operation
TC 4	**RPC N:N Communication**	Normal Operation
TC 5	**Publisher Crash (in Transmission)**	Fault Injection
TC 6	**Subscriber Crash (in Reception)**	Fault Injection
TC 7	**Network Failure Simulation**	Fault Injection
TC 8	**RPC Reconnect (Network Failure)**	Fault Injection

Table 3: Overview of test cases and their classification

Each test case includes a description of its objective, execution flow, success criteria, configuration details, and an evaluation of the observed results. In addition, code examples are provided to illustrate relevant implementation details, including how publishers, subscribers, or RPC clients are configured and how data is processed. The next subsections present the individual test cases in detail.

5.4.1 Test Case 1: Publisher to Subscriber Communication

Objective:

Ensure that a message published by a single publisher is reliably received by a single subscriber using a specific transport layer (e.g., shared memory, TCP or UDP) (see Table 4).

Execution:

- In local modes (e.g., `local shm`, `local udp`), the publisher and subscriber run inside a single container.
- In network modes (e.g., `network udp`, `network tcp`), the publisher and subscriber run in separate containers connected via a Docker network.
- The publisher sends a small binary payload of value `42` (repeated) and logs each send event.
- The subscriber listens on the topic, logs received values, and exits successfully if the expected messages where received within the timeout window.

Success Criteria:

- Subscriber receives 100 % of all messages sent.
- No crashes or communication timeouts occur.
- The subscriber exits with return code `0`.

Component	Role	Transport Mode	Payload
basic_pub	Publisher	All Modes	`0x2A` (42)
basic_sub	Subscriber	All Modes	42

Table 4: Configuration for Basic Communication Test

Code Examples:

To illustrate key aspects of the basic communication test, the following code excerpts highlight how the publisher and subscriber are implemented.

The publisher creates a binary buffer of size 10 where each byte is set to `0x2A` (decimal 42). This value is used consistently across all test scenarios to simplify result verification (see Listing 6).

The subscriber callback reads the first byte of the received buffer and casts it to an integer. This ensures that the payload can be easily validated (see Listing 7).

```
std::vector<unsigned char> buffer(10, 42);
pub.Send(buffer.data(), buffer.size());
```

Listing 6: Binary buffer with value 42 used by the publisher

```
int value = static_cast<int>(
static_cast<const unsigned char*>(data_.buffer)[0]
);
```

Listing 7: Extracting first byte from the received message in subscriber

```
TCLAP::ValueArg<std::string> mode_arg(
"m", "mode", "Transport mode", true, "", "string");

TCLAP::ValueArg<std::string> topic_arg(
"t", "topic", "Topic name", false, "test_topic", "string");

TCLAP::ValueArg<std::string> name_arg(
"n", "name", "eCAL node name", false, "pub_test", "string");

TCLAP::ValueArg<int> count_arg(
"c", "count", "Number of messages", false, 3, "int");

TCLAP::ValueArg<int> delay_arg(
"d", "delay", "Delay between sends", false, 1000, "int");
```

Listing 8: Argument setup using TCLAP in both publisher and subscriber

The configuration with TLCAP in Listing 8 allows full flexibility for running the same binary with different roles and transport modes. The `--mode` parameter (e.g., `local_tcp`) enables switching between eCAL transport layers without modifying the code. The use of TCP is especially useful for tests that simulate network scenarios across Docker containers, where shared memory is not applicable.

Evaluation:

This basic test confirms that eCAL reliably delivers messages across all supported transport modes under ideal conditions. The results show that communication remains consistent in both local and networked setups, provided the configuration parameters are correctly applied. This test serves as the foundation for more advanced scenarios such as crash handling, message validation, or multi-topic communication.

5.4.2 Test Case 2: Multiple Publishers and Subscribers

Objective:

Verify that multiple publishers can send distinct payloads on the same topic and that multiple subscribers can receive both streams reliably.

Execution:

- Two publishers send different payloads (42 and 43) on the same topic.
- Two subscribers listen to the same topic and count how many messages they receive for each value.
- All four nodes run either in a single container (local mode) or separate containers connected via Docker network (network mode).

Success Criteria:

- Each subscriber receives 100 % of the messages from both publishers.
- No crash or message loss occurs during transmission.
- All containers exit with return code `0`.

Component	Role	Transport Mode	Payload
multi_publisher	Publisher	All Modes	`0x2B` (43)
multi_publisher2	Publisher	All Modes	`0x2A` (42)
multi_subscriber	Subscriber	All Modes	Both
multi_subscriber2	Subscriber	All Modes	Both

Table 5: Configuration for Multi-Publisher and Multi-Subscriber Test

Code Examples:

Listing 9 shows how `multi_publisher` sends a binary payload containing value 43. Listing 10 illustrates the second publisher, which uses value 42. Both publishers send 15 messages with a 1000 ms delay between sends.

```
1 std::vector<unsigned char> buffer(10, 43);
2 pub.Send(buffer.data(), buffer.size());
```

Listing 9: Publisher 1 sends 0x2B (43)

```
1 std::vector<unsigned char> buffer(10, 42);
2 pub.Send(buffer.data(), buffer.size());
```

Listing 10: Publisher 2 sends 0x2A (42)

Each subscriber uses a callback function that increments counters depending on the first byte received. The values 42 and 43 are tracked separately (see Listing 11).

```
1 if (value == 42) ++count_42;
2 if (value == 43) ++count_43;
```

Listing 11: Subscriber callback counting 42 and 43

Evaluation:

This test confirms that eCAL supports N:N communication over a shared topic. Both subscribers successfully received messages from both publishers in all tested transport modes. This demonstrates the middleware's ability to handle concurrent sources and destinations. This is a critical feature for scenarios involving aggregation, monitoring, or distributed decision-making.

From a combinatorial perspective, a complete evaluation of the publish-subscribe model would require testing all communication patterns: one-to-one (1:1), one-to-many (1:N), many-to-one (N:1), and many-to-many (N:N). In practice, however, N:N scenarios inherently cover the functional aspects of both 1:N and N:1 communication patterns. This is because every N:N test includes multiple publishers and subscribers and therefore implicitly verifies the correctness of message delivery from one to many (1:N) and from many to one (N:1) within the same execution.

By implementing N:N tests across all five eCAL transport modes (`local_shm`, `local_udp`, `local_tcp`, `network_udp`, `network_tcp`), we effectively validate the core functionality and robustness of the middleware under realistic and complex conditions. This strategic reduction in test permutations allows for efficient validation without sacrificing coverage.

5.4.3 Test Case 3: RPC Ping Request-Response

Objective:

Verify that a client can successfully call a remote method ("Ping") on an eCAL RPC server and receive a valid response.

Execution:

- The `rpc_ping_server` registers a service method named `Ping`.
- The `rpc_ping_client` connects to the server and sends a request with content `"PING"`.
- The server responds with `"PONG"`.
- The client validates the response and exits with code `0`.

Success Criteria:

- The server receives and processes a request.
- The response string `"PONG"` is printed by the client.
- The client exits with return code `0`.

Component	Role	Mode	Method	Expected Response
rpc_ping_server	RPC Server	network_udp	Ping	PONG
rpc_ping_client	RPC Client	network_udp	Ping	PONG

Table 6: Configuration for RPC Ping Test

Code Examples:

The server registers a method named `Ping` that responds with `PONG` if it receives `PING` as request (Listing 12).

```
server.SetMethodCallback(method_info, [](const eCAL::
SServiceMethodInformation&, const std::string& request,
std::string& response) -> int {
  if (request == "PING"){
    response = "PONG";
    return 0;
  }
  response = "UNKNOWN";
  return 1;
});
```

Listing 12: RPC Server handler for method "Ping"

The client connects to the server, sends `"PING"`, waits for the reply, and prints the server's response (Listing 13).

```
1 std::string request = "PING";
2 std::vector<eCAL::SServiceResponse> responses;
3 bool success = client.CallWithResponse("Ping", request,
     responses, 1000);
4
5 for (const auto& response : responses){
6   std::cout << "[Client] Server response: " << response.
       response << std::endl;
7 }
```

Listing 13: RPC Client sends Ping request and prints response

Evaluation:

This test verifies the correct implementation of eCAL's RPC interface using a request–response pattern. It confirms that the client and server can communicate over UDP using named service methods. The use of response validation and exit codes allows automated checking of RPC behavior in CI pipelines. It also shows how clients can discover services and call remote methods in a distributed testing environment using Docker and Robot Framework.

5.4.4 Test Case 4: RPC N:N Communication

Objective:

Verify that multiple clients can simultaneously call the same RPC method on multiple eCAL servers and receive a valid response. This test evaluates the scalability and correctness of eCAL's service-based communication pattern under N:N conditions.

Execution:

- Two RPC servers register a service named `rpc_n_to_n_service` with the method `Ping`.
- Each of the two clients connects to all available servers and sends a `"PING"` request.
- Each server responds with `"PONG"`.
- Each client verifies the number and content of responses and exits with return code `0`.

Success Criteria:

- All clients receive two responses, each containing `"PONG"`.
- No timeout or connection errors occur.
- All clients exit with return code `0`.

Component	Role	Count	Mode	Expected Response
rpc_n_to_n_server	RPC Server	2	network_udp	PONG
rpc_n_to_n_client	RPC Client	2	network_udp	2×PONG

Table 7: Configuration for RPC N:N Communication Test

Code Examples:
The client sends a Ping request and validates that all received responses are equal to `"PONG"` (Listing 14).

```
1 std::vector<eCAL::SServiceResponse> responses;
2 client.CallWithResponse("Ping", "PING", responses, 2000);
3 int pong_count = 0;
4
5 for (const auto& r : responses){
6     if (r.response == "PONG") ++pong_count;
7 }
8
9 return pong_count == 2 ? 0 : 1;
```

Listing 14: RPC Client receives multiple responses

Evaluation:

This test confirms that eCAL supports concurrent N:N RPC communication and that multiple clients can address multiple servers via shared service names. It demonstrates that clients receive multiple valid responses for a single call and can handle those responses correctly in parallel scenarios. The test also confirms that the middleware's internal discovery and routing logic is robust across multiple service providers.

From a test design perspective, this N:N scenario verifies both 1:N and N:1 communication patterns. Each client must discover and contact multiple servers (1:N), and each server must respond correctly to multiple clients (N:1). By executing this test case, the core mechanisms of eCAL's RPC handling are validated across all relevant communication directions, while reducing the total number of required test permutations.

5.4.5 Test Case 5: Publisher Crash During Transmission

Objective:

Evaluate the system's resilience when one publisher crashes mid-transmission, and ensure that the subscriber still receives messages from the remaining active publisher.

Execution:

- Two publishers are started: one sends `42` and crashes after 10 messages, the other sends `43` continuously.
- One subscriber is launched to receive messages from both.
- The test runs in all communication modes (local and network).
- The subscriber counts messages from both publishers and exits after a fixed timeout.

Success Criteria:

- The subscriber receives at least 25 messages with value `43`.
- The number of `42` messages is below the crash threshold (between 5 and 11).
- The subscriber exits with return code `0`.

Component	Role	Transport Mode	Payload
crash_pub	Publisher	All Modes	`0x2A` (42)
test_pub	Publisher	All Modes	`0x2B` (43)
test_sub	Subscriber	All Modes	42 + 43

Table 8: Configuration for Crash Resilience Test

Code Examples:

The crashing publisher sends a few messages and then exits by calling `std::abort()`, simulating a runtime failure (Listing 15). Meanwhile, the test publisher continues normal operation (Listing 16).

```
for (int i = 0; i < total_arg.getValue(); ++i){
 pub.Send(buf.data(), buf.size());
 sleep_for(delay_ms);
 if (i == crash_at_arg.getValue()){
  std::abort();  // Simulate crash
 }
}
```

Listing 15: Crash publisher sends and exits after 10 messages

```
for (int i = 0; i < count && eCAL::Ok(); ++i){
 pub.Send(buffer.data(), buffer.size());
 sleep_for(delay_ms);
}
```

Listing 16: Resilient publisher continues sending messages

The subscriber counts received values and returns 0 only if enough `43` values are seen and no unexpected continuation from the crashed publisher occurs (Listing 17).

```
if (count_43 >= 25 && count_42 < 11 && count_42 > 4){
  return 0; // Success: resilient communication
}else{
  return 1; // Failure: not enough or unexpected data
}
```

Listing 17: Subscriber decision logic

Evaluation:
This test demonstrates that eCAL remains fully functional and continues to deliver messages even when one of the publishers crashes unexpectedly. The second publisher, which sends payload `43`, continues to operate without interruption. This confirms that the failure of one communication node does not affect the functionality of others. The test was executed in all supported eCAL transport modes, showing that this reliability is consistent regardless of the communication layer.

The use of a crash publisher simulates real-world process failures, and the test verifies that message delivery remains uninterrupted. By comparing the message counts, the system ensures that no "phantom" messages are received after the crash point. This approach can be used as a template for testing fault tolerance in distributed IPC systems.

5.4.6 Test Case 6: Subscriber Crash During Reception

Objective:
Verify that the communication remains stable even when one subscriber crashes while receiving large messages.

Execution:

- A `large_publisher` sends three large messages (each about 50 MB) on the same topic.
- One `crash_subscriber` is configured to crash after a short time during message reception.
- One `test_subscriber` continues running and should receive all messages correctly.
- The test is executed in all supported eCAL transport modes, except for UDP which cannot handle large messages.
- A special variant uses SHM with `zero_copy_mode = true` to verify robustness of shared memory access.

Success Criteria:

- The stable subscriber exits with return code `0` and logs successful message reception.
- The crashing subscriber terminates due to a simulated failure after a few seconds.
- The publisher completes message delivery without interruption or failure.
- In SHM mode with Zero-Copy, shared memory corruption or deadlocks must not occur.

Component	Role	Transport Mode	Payload
large_publisher	Publisher	All Modes	∼50 MB
crash_subscriber	Subscriber (crash)	All Modes	∼50 MB
test_subscriber	Subscriber (stable)	All Modes	∼50 MB

Table 9: Configuration for Crash During Reception Test

Code Example:

The crashing subscriber aborts its process intentionally after two seconds of runtime if it receives any message (see Listing 18).

```
void OnReceive(
    const eCAL::STopicId&, const eCAL::SDataTypeInformation&,
    const eCAL::SReceiveCallbackData& data_)
{
 std::cout << "[Crash_Sub] Received "
 << data_.buffer_size << " bytes\n";

 if (elapsedtime >= 2) {
  std::cerr << "[Crash_Sub] Simulating crash after 2 sec";
  std::abort();
 }
}
```

Listing 18: Crash condition inside subscriber receive callback

To simulate high-throughput conditions, the publisher sends three 50 MB messages and logs each transmission result (see Listing 19).

```
std::string buffer(50L * 1024L * 1024L, 'X');
for (int i = 0; i < count; ++i){
  bool sent = pub.Send(buffer.data(), buffer.size());
  std::cout << "[Publisher] Send result: "
            << (sent ? "pass" : "fail");
}
```

Listing 19: Large message publisher with send confirmation

Evaluation:

This test confirms that eCAL maintains stable communication even when a subscriber crashes during the reception of a large message. In all tested configurations except the Zero-Copy SHM variant, the system behaved as expected: the publisher completed its transmission successfully, and the stable subscriber received all messages without interruption. The crash was isolated to the failing subscriber, indicating proper error containment and robustness of the communication layer.

The variant using Zero-Copy in Shared Memory mode highlighted a specific weakness in fault isolation. In Zero-Copy mode, the publisher shares direct memory access with subscribers, which eliminates the need for memory copying and significantly improves performance in high-throughput scenarios. However, this optimization also introduces a stronger coupling between processes at the memory level.

If a subscriber crashes while holding a pointer to shared memory, the publisher (or other subscribers) may experience access violations, resource locks, or inconsistent memory states. This was observed during the test: although the publisher attempted to continue sending data, the shared memory segment could no longer be accessed safely once the crashing subscriber aborted during callback execution.

Therefore, while Zero-Copy SHM offers performance advantages, it requires careful handling in fault-tolerant systems. Mechanisms such as memory fencing, process monitoring, or fallback copy modes should be considered if system resilience is a priority.

In conclusion, this test underscores both the reliability of eCAL in standard SHM, TCP, and UDP scenarios and the current limitations when using Zero-Copy SHM without additional safeguards. Future testing or deployment scenarios should carefully consider the trade-off between performance and fault tolerance when using advanced features like Zero-Copy.

5.4.7 Test Case 7: Network Failure Simulation

Objective:

Verify that after a simulated network failure, a new local UDP publisher can be successfully launched and establish communication with a running local UDP subscriber. This simulates a scenario where a redundant or backup component becomes active only after failure detection.

Execution:

- **Phase 1:** A local UDP publisher (`local_publisher_1`) and a subscriber run in the same container. The publisher sends payloads with value 42.
- **Phase 2:** A network UDP publisher (`network_publisher`) runs in a separate container and sends payloads with value 43.
- **Phase 3:** After 7 seconds, the `network_publisher` container is disconnected from the Docker network to simulate a network cable unplug.
- **Phase 4:** A second local UDP publisher (`local_publisher_2`) is started within the original container after a short delay. It sends payloads with value 44.
- The subscriber logs received values and verifies all expected transitions.

Success Criteria:

- Subscriber receives:
  - All sent messages with value 42 (from local_publisher_1).
  - At least 2 messages with value 43 (before network disconnection).
  - All sent messages with value 44 (from local_publisher_2 after crash).
- No communication deadlocks or crashes occur.
- Subscriber process exits with code `0`.

Component	Role	Transport Mode	Payload
local_publisher_1	Initial Local Publisher	local_udp	`0x2A` (42)
network_publisher	Network Publisher	network_udp	`0x2B` (43)
local_publisher_2	Delayed Local Publisher	local_udp	`0x2C` (44)
subscriber	Receiver	local_udp	42, 43, 44

Table 10: Configuration for extended Network Failure Simulation

Code Examples:

All publishers follow the same sending logic (Listing 20) but use different payload values.

```
1 std::vector<unsigned char> buffer(10, payload);
2 // e.g. 42, 43, or 44
3  for (int i = 0; i < count; ++i){
4   pub.Send(buffer.data(), buffer.size());
5   sleep_for(std::chrono::milliseconds(delay));
6  }
```

Listing 20: General publisher loop

The new publisher `local_udp_pub_2` uses payload `44` and is launched inside the same container as the subscriber (Listing 21).

```
1 ./local_udp_pub --topic "$TOPIC" --name pub1_${NAME} &
2
3 ./network_crash_sub --topic "$TOPIC" --name sub_${NAME} &
4 SUB_PID=$!
5
6 sleep 8
7 ./local_udp_pub_2 --topic "$TOPIC" --name pub2_${NAME} &
8
```

```
wait $SUB_PID
EXIT_CODE=$?
exit $EXIT_CODE
```

Listing 21: Entrypoint launches second local publisher after delay

The subscriber logs all payloads and reports a summary (Listing 22).

```
int count_42 = 0, count_43 = 0, count_44 = 0;

void OnReceive(...) {
  int val = data_[0];
  if (val == 42) ++count_42;
  if (val == 43) ++count_43;
  if (val == 44) ++count_44;
}

// checks if the amount of messages are correct
if (expected_42 && expected_43 && expected_44)
  return 0;
else
  return 1;
```

Listing 22: Subscriber validation logic for all payloads

Evaluation:

This test simulates a common failure mode in networked IPC systems: a remote communication node becomes unavailable due to a crash or disconnection. The successful delivery of messages from the locally started fallback publisher demonstrates that eCAL can recover from partial system failures and maintain local communication paths without interruption.

The use of different payloads (42, 43 and 44) allows for precise identification of message origin. By verifying that all types were received in sufficient quantity, the test ensures that the system responded correctly to the simulated fault: it handled remote messages, then did not crash with local messages.

From an architecture perspective, this test validates the middleware's robustness and flexibility across dynamic transport topology changes. It demonstrates that eCAL can switch between local and network transport without requiring restarts or reinitialization of the subscriber process.

This scenario is especially relevant for distributed systems that combine in-vehicle communication (local) with cloud-connected components (network). It proves that local subsystems can continue operating even in case of external failure.

5.4.8 Test Case 8: RPC Reconnect After Network Failure

Objective:

Verify that an eCAL RPC client can recover from a temporary network disconnection and successfully reconnect to the server. The test simulates a failure and reconnection of the client's network interface during runtime.

Execution:

- To ensure stable addressing during the disconnect and reconnect phases, both containers are launched with predefined IP addresses. This guarantees that after a reconnect, the RPC discovery mechanism can still resolve and target the original address.
- The `rpc_ping_server` container is started first and registers a method `Ping` via eCAL RPC.
- The client container connects to the server and performs an initial `"PING"` request.
- Using Dockers network control, the client is then deliberately disconnected from the Docker network for several seconds. This simulates a temporary connection loss, such as would occur during wireless dropouts or hardware interruptions.
- After a delay, the client is reconnected to the same network and reattached with the original IP. Once eCAL restores communication, the client issues a second `"PING"` request.
- The test succeeds only if both calls are acknowledged by the server with `"PONG"` and the client exits with code `0`.

Success Criteria:

- The server receives and responds to two valid `"PING"` requests.
- The client prints two valid `"PONG"` responses.
- The client exits with return code `0`.

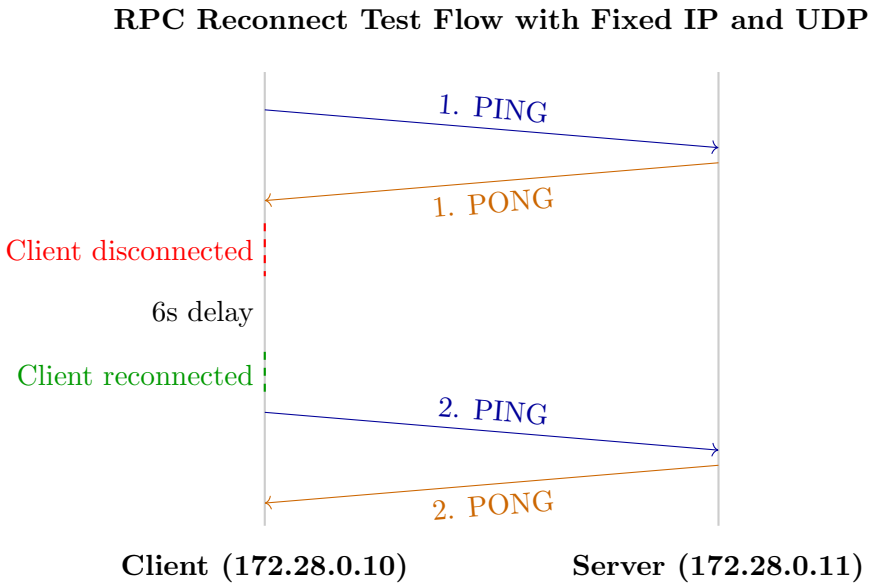


Figure 7: View of the reconnect test: the client disconnects and reconnects

Component	Role	IP Address	Mode	Response
rpc_ping_server	RPC Server	172.28.0.11	network_udp	PONG
rpc_ping_client	RPC Client	172.28.0.10	network_udp	2×PONG

Table 11: Setup for RPC reconnect test

Code Examples:

Fixed IP assignment in Robot Framework:

The following Robot Framework keyword starts the client and server containers with fixed IP addresses. This ensures consistent addressing across disconnect and reconnect steps.

```
1 Start Container With IP   rpc_server   ${IMAGE}   server
2 ...    network=${NETWORK}    ip=${SERVER_IP}
3
4 Start Container With IP   rpc_client   ${IMAGE}   client
5 ...    network=${NETWORK}    ip=${CLIENT_IP}
```

Listing 23: Starting containers with fixed IPs

Simulating disconnect and reconnect:

This part disconnects the client from the network and reconnects it later with the same IP. It simulates a temporary connection failure during runtime.

```
Disconnect Container From Network   rpc_client   ${NETWORK}
Sleep    7s
Reconnect Container To Network    rpc_client    ${NETWORK}
                                  ${CLIENT_IP}
```

Listing 24: Simulating network disconnect and reconnect

The client performs two RPC calls: one before and one after simulated network loss (Listing 25).

```
int pong_counter = 0;
int error = call_rpc(client, "Initial", pong_counter);
std::this_thread::sleep_for(std::chrono::seconds(8));
error += call_rpc(client, "Reconnect", pong_counter);
return (pong_counter >= 2 && error == 0) ? 0 : 1;
```

Listing 25: Client performs reconnect and second RPC call

Evaluation:

This test checks how well the eCAL RPC system can handle changing network conditions. By disconnecting the client from the Docker network for a short time, it simulates a temporary network failure. After reconnecting, the client tries to call the server again using the same IP address.

eCAL uses service discovery to manage RPC connections. In the case of UDP, this discovery is usually based on multicast messages that are sent across the network. For this to work correctly after a reconnect, the system needs to keep using the same network interface and IP address. The fixed IPs in this test make sure that the server still recognizes the client after reconnecting.

This setup is similar to real use cases, for example in embedded systems where network connections can drop and come back. The test shows that eCAL can recover from such a disconnection. Both calls (before and after the network drop) are successful, which means that the client could reconnect and the server was still available. This gives confidence that eCAL RPC over UDP multicast can be used in systems where short network problems may happen.

5.5 Automation and Repeatability

A key objective of the integration test design is to ensure that all tests can be executed in a fully automated and repeatable manner. Automation is important for enabling reliable regression testing, especially in the context of continuous integration and deployment (CI/CD).

In this project, the test cases are implemented using the Robot Framework and can be executed with a single command:

```
robot --outputdir results network_crash.robot
```

This command runs the specified test and stores all result files in the `results` directory. The following artifacts are automatically generated after each run:

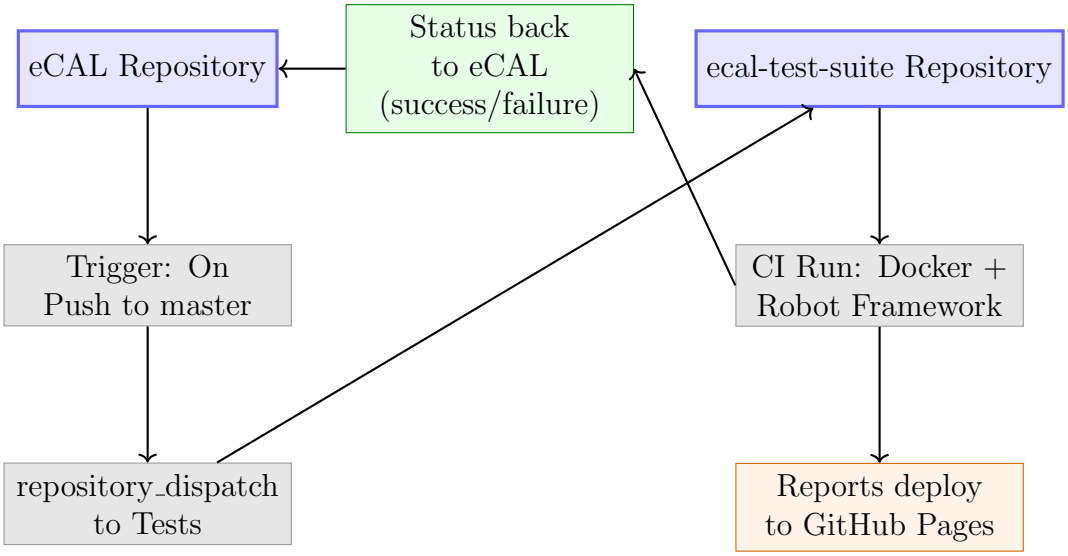
- **log.html** – A detailed execution log, including all console output
- **report.html** – A summary report presenting the test results in a structured format
- **output.xml** – A machine-readable file with raw test data for further processing or CI integration

To ensure automated and consistent test execution, the tests are fully integrated into GitHub Actions. This platform offers native support for defining CI workflows directly in the repository using YAML configuration files. The integration allows tests to be triggered automatically in the following scenarios:

- After each push to the main repository (continuous integration)
- Manually via the GitHub Actions web interface

Test results are stored and deployed via GitHub Pages, enabling direct access to all test reports through a structured web interface. Each test run is timestamped and, if available, linked to the triggering commit, ensuring traceability across test history.

The overall interaction between the main repository and the ecal-test-suite is illustrated in the following diagram:



This architecture ensures that integration tests are decoupled from the main development repository, yet fully synchronized through GitHub's event-based automation mechanisms. Status feedback, report visibility, and reproducibility are guaranteed at every stage of the workflow.

In addition, the use of Docker for test isolation ensures high portability and reproducibility. Tests can be executed in local development environments, on cloud-based runners, or within virtualized test infrastructure—independent of the host system configuration. This improves consistency across different environments and reduces the likelihood of platform-specific errors.

5.6 Visualization of Test Results

The successful automation of integration tests is only effective if test results can be accessed, interpreted, and compared efficiently. Therefore, this project includes a visual reporting system that makes all results available via the browser.

After each test execution, GitHub Actions deploys the generated reports to the **GitHub Pages** section of the test repository. The following figure shows the structured web interface, where all test runs are listed by timestamp. If available, the commit hash that triggered the test run is also displayed for traceability.

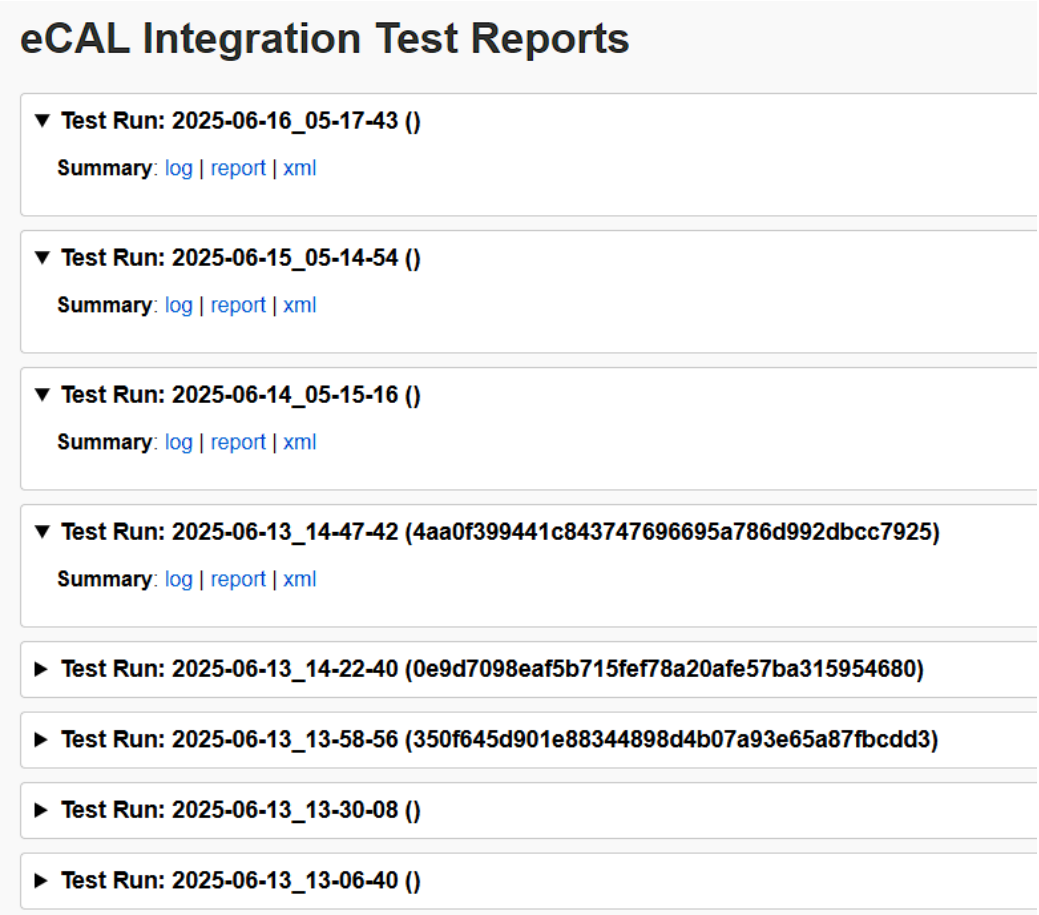


Figure 8: Test report overview hosted on GitHub Pages

Clicking on a specific test entry opens the detailed Robot Framework report.

This report contains structured sections with statistics, test case results, execution logs, and console output. The clear layout enables quick identification of failed tests and debugging of communication problems.

Basic Pub Sub & Message Validation & Multi Pub Sub & Network Crash & Pub Crash & Sub Crash & Sub Send Crash Report

Generated
20250616 07:41:00 UTC+02:00
6 hours 13 minutes ago

Summary Information

Status:	5 tests failed
Start Time:	20250616 07:28:45.641
End Time:	20250616 07:41:00.271
Elapsed Time:	00:12:14.630
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	26	21	5	0	00:10:54	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
basic_pub_sub_local_shm	1	1	0	0	00:00:16	
basic_pub_sub_local_tcp	1	1	0	0	00:00:16	
basic_pub_sub_local_udp	1	1	0	0	00:00:16	
basic_pub_sub_network_tcp	1	1	0	0	00:00:16	
basic_pub_sub_network_udp	1	1	0	0	00:00:16	
crash_receive_local_shm	1	1	0	0	00:00:35	
crash_receive_local_tcp	1	1	0	0	00:00:35	
crash_receive_network_tcp	1	1	0	0	00:00:31	
crash_receive_network_udp	1	1	0	0	00:00:31	
message_validation_malformed	1	0	1	0	00:00:00	
message_validation_valid	1	0	1	0	00:00:00	
multi_pub_sub_local_shm	1	0	1	0	00:00:18	
multi_pub_sub_local_tcp	1	1	0	0	00:00:19	
multi_pub_sub_local_udp	1	1	0	0	00:00:19	
multi_pub_sub_network_tcp	1	1	0	0	00:00:19	
multi_pub_sub_network_udp	1	1	0	0	00:00:19	
network_crash_test	1	1	0	0	00:00:30	
pub_crash_local_shm	1	0	1	0	00:00:35	
pub_crash_network_tcp	2	2	0	0	00:01:11	
pub_crash_network_udp	1	1	0	0	00:00:36	
sub_crash_local_shm	1	1	0	0	00:00:35	
sub_crash_local_tcp	1	1	0	0	00:00:34	
sub_crash_local_udp	1	1	0	0	00:00:34	
sub_crash_network_udp	1	1	0	0	00:00:34	
zero_copy_crash_receive_local_shm	1	0	1	0	00:00:35	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Basic Pub Sub & Message Validation & Multi Pub Sub & Network Crash & Pub Crash & Sub Crash & Sub Send Crash	26	21	5	0	00:12:15	

Figure 9: Detailed Robot Framework report for a test run

This visualization not only improves transparency and accessibility, but also enables historical comparisons of test outcomes over time. Developers and reviewers can quickly identify regressions or improvements in communication stability and test coverage.

5.7 Summary

This chapter presented the design, implementation, and execution of a modular integration test system for the eCAL middleware. By combining Docker-based process isolation with the flexible automation capabilities of Robot Framework, it was possible to validate key properties of publish-subscribe communication in a controlled and reproducible environment.

The selected test cases focused on core functionalities such as reliable message delivery, multi-node communication, and fault tolerance under crash or failure conditions. Most scenarios were tested across all five supported eCAL transport modes, these are shared memory (SHM), UDP, and TCP in both local and network configurations. This ensured broad coverage of both common and edge-case communication patterns.

Code examples demonstrated how publishers, subscribers, and test logic are implemented and configured, while structured success criteria and exit codes provided clear validation of expected behavior. The test infrastructure also proved effective in simulating failure scenarios such as process crashes and message loss, confirming the robustness and resilience of the eCAL middleware.

A notable insight from these tests is the performance–resilience trade-off introduced by advanced features like Zero-Copy SHM. While this mode improves throughput by eliminating memory copies, it also increases inter-process coupling and may compromise fault isolation if a subscriber crashes during message access. Future deployments should carefully assess whether the performance benefits of Zero-Copy justify its potential stability risks.

The framework is already equipped to be extended with further test categories. For example, future work could include RPC-based communication, service availability tests, and metrics such as round-trip time. Moreover, integration into CI/CD pipelines would enable automatic regression testing during development and deployment.

In conclusion, the developed test system successfully demonstrates how integration tests can be used not only for functional verification but also to explore system behavior under realistic and adverse conditions. It provides a solid foundation for continuous validation of middleware functionality in research, development, and production contexts.

6 Conclusion and Outlook

Summary of Results

The goal of this thesis was to develop a reusable and automated integration testing concept for inter-process communication (IPC) middleware, demonstrated using the eCAL framework. To achieve this, a modular and containerized test system was designed and implemented using Robot Framework and Docker.

The resulting solution enables the execution of isolated, reproducible, and scalable integration tests across a variety of communication modes. The key contribution of the work is the creation of a generic test infrastructure that abstracts away platform dependencies while supporting fault simulation, orchestration, and flexible configuration.

In total, eight integration test cases were designed and executed. These are divided into two categories: standard operation tests and fault injection tests. The scenarios cover both publish-subscribe and RPC patterns.

The test coverage includes all core transport modes supported by eCAL, such as shared memory (SHM), TCP, and UDP, both in local and network-based configurations. The test logic checks not only the correct functioning of eCAL communication under normal conditions but also how the system behaves in edge cases like component crashes or temporary network failures.

Evaluation of the Approach

The developed test system proved to be effective in validating both standard and failure scenarios. For the specific case of eCAL, the results showed that the middleware works reliably under ideal conditions and remains stable even during crash situations or network failures.

More generally, the solution demonstrates how integration testing can be applied to IPC systems with minimal manual effort. New scenarios, such as adding publishers or simulating network interruptions, could be added easily through configuration without changing the test logic.

Thanks to the modular design and use of standard tools like Docker and Robot Framework, the system supports full automation and integration into CI/CD pipelines. The test execution is reproducible, and all steps from building containers to validating logs are controlled by reusable Python and shell scripts.

Limitations and Outlook

While the current test system provides a solid foundation for automated IPC testing, it also has limitations. All tests are based on predefined containers and fixed communication patterns. Dynamic test generation, large-scale parallel setups, or real-time performance measurements were not within the scope of this work.

In addition, the tests are currently focused on the functional level. Topics such as security, data consistency under load, or long-term stability have not yet been addressed and could be explored in future work.

The general concept, however, can be extended beyond eCAL. The modular test infrastructure and test execution logic are applicable to other IPC systems, as long as the communication logic is containerizable and script-controlled. This makes the presented approach a reusable blueprint for testing in other distributed environments.

Final Remarks

This thesis has shown how integration testing of IPC middleware can be made more systematic, scalable, and reusable. The developed concept not only supports testing of eCAL, but also lays the foundation for future testing strategies in distributed systems.

From a personal perspective, working on this topic has deepened the understanding of software testing, containerization, and communication frameworks. It also highlighted the importance of test automation in modern development workflows and showed how infrastructure and tooling choices can strongly influence the maintainability of complex systems.

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th. Addison-Wesley, 2012.

[2] O. Robotics, *Writing basic integration tests with launch_testing*, Accessed: 2025-03-03, 2025. [Online]. Available: `https://docs.ros.org/en/rolling/Tutorials/Intermediate/Testing/Integration.html`.

[3] eProsima, *Eprosima fast dds*, Accessed: 2025-03-10, 2025. [Online]. Available: `https://github.com/eProsima/Fast-DDS`.

[4] RTI, *Comparing open source dds to rti connext dds*, Accessed: 2025-03-11, 2025. [Online]. Available: `https://www.rti.com/blog/picking-the-right-dds-solution`.

[5] E. Foundation, *Ecal - enhanced communication abstraction layer*, Accessed: 2025-03-01. [Online]. Available: `%5Curl%7Bhttps://github.com/eclipse-ecal/ecal%7D`.

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.

[7] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 8th. McGraw-Hill Education, 2014.

[8] A. Spillner and T. Linz, *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester*, 6th. dpunkt. verlag, 2019.

[9] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach*. Springer, 2003.

[10] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd. Wiley, 2011.

[11] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd. Wiley, 1999.

[12] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, 2009.

[13] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 3rd. Pearson Education, 2017.

[14] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th. Pearson Education, 2018.

[15] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th. Pearson Education, 2015.

[16] H. Dinari, "Inter-process communication (ipc) in distributed environments: An investigation and performance analysis of some middleware technologies", *International Journal of Modern Education and Computer Science*, vol. 12, no. 2, 2020.

[17] C. Raiciu, F. Huici, M. Handley, and D. S. Rosenblum, "Enabling efficient zero-copy data exchange in networked systems", in *Proceedings of the ACM SIGCOMM Workshop on Kernel-Bypass Networks*, New York, NY, USA: ACM, 2017.

[18] P. A. Bernstein, "Middleware: A model for distributed system services", *Communications of the ACM*, vol. 39, no. 2, 1996.

[19] M. Quigley et al., "Ros: An open-source robot operating system", in *ICRA Workshop on Open Source Software*, 2009.

[20] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview", in *23rd International Conference on Distributed Computing Systems Workshops*, IEEE, 2003.

[21] E. Foundation, *Ecal documentation*, Accessed: 2025-03-03. [Online]. Available: `%5Curl%7Bhttps://eclipse-ecal.github.io/ecal/stable/index.html%7D`.

[22] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, 2007.

[23] J. Liu and M. Parashar, "Enabling self-management of component-based high-performance scientific applications", *Future Generation Computer Systems*, vol. 25, no. 4, 2009.

[24] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, 4th. Addison-Wesley, 2009.

[25] I. Gorton and Y. Liu, "Software performance testing for distributed systems: A practical approach", *IEEE Software*, vol. 23, no. 3, 2006.

[26] Google, *Googletest user's guide*, Accessed: 2025-04-04, Google. [Online]. Available: `https://google.github.io/googletest/`.

[27] R. F. Foundation, *Robot framework user guide*, Accessed: 2025-04-04, Robot Framework Foundation. [Online]. Available: `https://robotframework.org/robotframework/`.

[28] ThoughtWorks, *Gauge documentation*, Accessed: 2025-04-04, ThoughtWorks. [Online]. Available: `https://docs.gauge.org/`.

[29] Katalon, *Katalon documentation*, Accessed: 2025-04-04, Katalon. [Online]. Available: `https://docs.katalon.com/`.

[30] S. Software, *Testcomplete documentation*, Accessed: 2025-04-04, SmartBear Software. [Online]. Available: `https://support.smartbear.com/testcomplete/docs/`.

[31] D. North, *Introducing bdd*, `https://dannorth.net/introducing-bdd`, Accessed: 2025-04-04, 2006.

[32] B. R. et al., *Behave – behavior-driven development, python style*, `https://github.com/behave/behave`, Accessed: 2025-04-04, 2025.