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

April 4, 2025

Current software development demands quick solutions to fulfill constantly changing requirements. Distributed systems, where software processes run on separate computing nodes and communicate with each other, have become increasingly important. To enable effective communication within such systems, inter-process communication (IPC) frameworks like the enhanced Communication Abstraction Layer (eCAL) are commonly used. eCAL allows data to be exchanged rapidly and reliably, enabling faster implementation of new features and adjustments within complex software projects.

As these systems grow in complexity, ensuring software quality becomes more challenging but also more critical. Failures or errors in communication middleware can lead to significant problems, especially in areas like automotive or robotics, where safety and reliability are essential. Thus, the central question arises: how can the reliability and correctness of eCAL-based IPC systems be systematically ensured? The goal of this thesis is to develop and evaluate a structured system testing framework specifically tailored for eCAL-based communication, aiming to detect faults early and increase overall software quality.

To achieve this goal, various system testing approaches will be analyzed and adapted for eCAL. This includes evaluating unit tests, integration tests, and system-level testing methods. Additionally, automation and continuous testing techniques within CI/CD pipelines will be considered. A concrete example implementation of these test strategies will be demonstrated, evaluated, and compared to ensure practical applicability and effectiveness.

Contents

List of Figures

List of Listings

List of Abbreviations

IPC	Inter-Process Communication
eCAL	enhanced Communication Abstraction Layer
CI/CD	Continuous Integration / Continuous Delivery
RPC	Remote Procedure Call
IoT	Internet of Things
DDS	Data Distribution Service
ROS	Robot Operating System
UAT	User Acceptance Testing
QoS	Quality of Service
API	Application Programming Interface

1 Introduction

In this chapter, the topic of this thesis is introduced. Section 1.1 explains the motivation behind developing a systematic testing framework for eCAL-based inter-process communication. Then, section 1.2 describes the main objectives and goals of this thesis. Finally, section 1.3 provides an overview of the structure of the entire document.

1.1 Motivation

In modern software engineering, distributed systems have become very important due to their capability to handle large amounts of data efficiently and reliably. Middleware solutions, like the *enhanced Communication Abstraction Layer* (eCAL), play an important role because they enable different software processes to exchange data and communicate with each other across multiple computing nodes [1].

Reliability and correctness of these middleware solutions are especially important in areas such as automotive, robotics, and the Industrial Internet of Things (IIoT). Failures in communication could cause system breakdowns and significant safety risks, particularly in applications where real-time processing is essential [2]. Therefore, it is critical to develop comprehensive testing strategies to ensure middleware solutions such as eCAL operate correctly and safely.

Currently, eCAL does not have a standardized approach for system-wide testing. Although individual parts of eCAL are tested through unit tests, these do not fully cover complex distributed scenarios and real communication patterns [1].

Other middleware technologies, such as the Robot Operating System (ROS) and the Data Distribution Service (DDS), are architecturally different and offer dedicated testing mechanisms. For instance, ROS 2 includes the `launch_testing` framework, which supports integration testing of distributed nodes [3]. DDS implementations like eProsima Fast DDS and RTI Connext DDS also provide testing tools and built-in monitoring capabilities [4], [5]. However, these solutions are tightly coupled with their own architectures, making their testing strategies difficult to directly adopt for eCAL.

The creation of a test framework specifically for eCAL is therefore essential. Such a framework can significantly improve the quality and reliability of applications that depend on eCAL, especially for safety-critical use cases.

1.2 Objective

This thesis aims to develop and evaluate a dedicated system testing framework for eCAL-based IPC systems. The primary objectives are:

- Design a structured approach for conducting system tests specific to eCAL.
- Implement the testing framework and assess its effectiveness in real-world scenarios.
- Explore integration possibilities with continuous integration and continuous deployment (CI/CD) pipelines to facilitate automated testing and early fault detection.

1.3 Outline

The structure of this thesis is as follows:

- **Chapter 2: Theoretical Foundations** – Provides an overview of IPC principles, details the eCAL framework, and reviews existing testing methodologies.
- **Chapter 3: Framework Design** – Discusses the requirements and design considerations for the proposed testing framework.
- **Chapter 4: Implementation** – Details the development of test cases, simulation environments, and strategies for testing common failure scenarios.
- **Chapter 5: CI/CD Integration** – Explores the integration of the testing framework into CI/CD pipelines to enable automated testing.
- **Chapter 6: Evaluation** – Analyzes the framework's performance, including test coverage and execution efficiency.
- **Chapter 7: Conclusion and Future Work** – Summarizes the findings and suggests directions for future research.

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 the concept of distributed systems, including their characteristics and common use cases. Section 2.2 explains the fundamentals of inter-process communication (IPC) and highlights different communication mechanisms. In Section 2.3, the role of middleware in distributed environments is discussed, with a focus on commonly used solutions such as ROS, DDS, and eCAL. Section 2.4 gives a detailed overview of the eCAL framework, including its architecture, features, and typical applications. Section 2.5 provides essential concepts in software testing, such as test levels, types, and techniques. Finally, Section 2.6 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 Distributed Systems

2.1.1 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 [6]. Such systems consist of independent components located on different networked computers, which interact with each other by exchanging messages.

The primary goal of distributed systems is to share resources, increase performance, and provide reliable and fault-tolerant operations. Resources such as processing power, memory, storage, and data can be shared between multiple nodes within the system, enhancing system efficiency and scalability [2].
Figure 1 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.

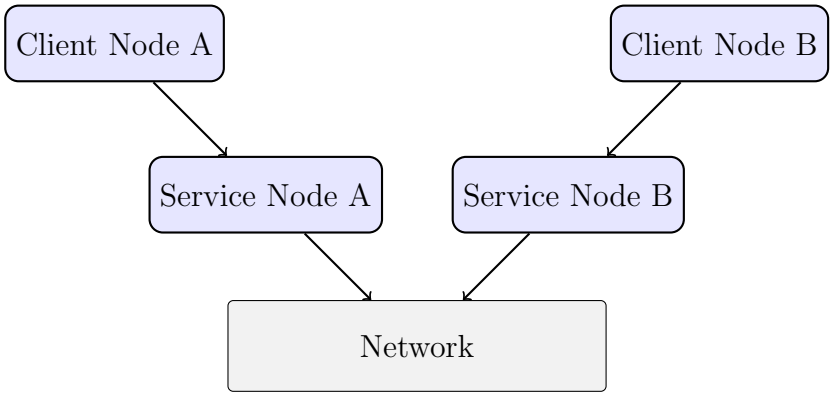

Figure 1: Basic architecture of a distributed system with multiple clients and services connected through a network

2.1.2 Characteristics 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 [6].

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 [2].

2.1.3 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 [2], [6].

2.2 Inter-Process Communication (IPC)

2.2.1 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 [7]. 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 [8].

2.2.2 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 (RPC) [7], [8].

Figure 2 illustrates a simplified comparison of common inter-process communication mechanisms. Shared memory allows processes to access a common memory region directly. Message passing transfers data through explicit send/receive actions. Remote Procedure Call (RPC) abstracts communication by allowing a process to call functions in another process as if they were local.

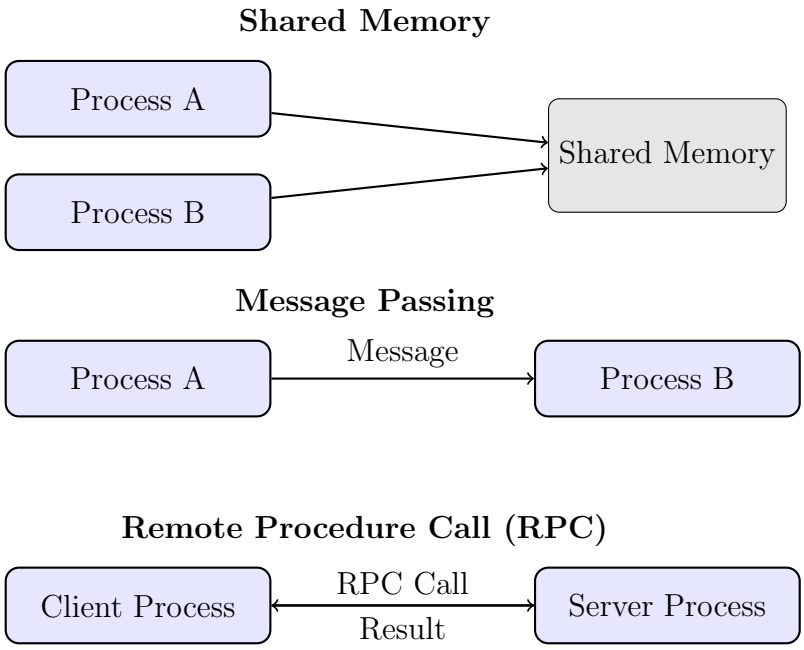

Figure 2: Comparison of common IPC mechanisms: Shared Memory, Message Passing, and Remote Procedure Call (RPC)

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 [7], [9].

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 [8].

Remote Procedure Calls (RPC):
Remote Procedure Calls (RPC) enable processes to invoke procedures or functions on remote systems as if they were local calls. RPC abstracts net-

work 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 [2].

2.2.3 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 or specialized network interfaces. By reducing the number of data copies, zero-copy significantly enhances performance and efficiency in systems with high throughput and low latency requirements, such as high-performance computing or real-time systems [10].

2.3 Middleware in Distributed Systems

2.3.1 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 [11]. Middleware allows software developers to focus more on the application logic rather than the low-level communication and network details.

2.3.2 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) [12], the Data Distribution Service (DDS) [13], and the enhanced Communication Abstraction Layer (eCAL) [14].

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 [12].

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 [13].

enhanced Communication Abstraction Layer (eCAL):
eCAL is a middleware specifically designed for efficient inter-process communication (IPC) in distributed environments. It offers high-speed message exchange, remote procedure calls (RPC), and shared memory communication. eCAL's lightweight nature and focus on performance make it ideal for scenarios that require fast data transfer, such as automotive software systems, industrial automation, and complex IoT solutions [1].

Layer	**ROS**	**DDS**	**eCAL**
Application Layer	User Applications	User Applications	User Applications
Middleware API	rospy, roscpp	DDS API	eCAL API (C++, Python, etc.)
Core-Components	ROS-Master, Topics, Services	RTPS Protocol	Pub/Sub, RPC
Transport Layer	TCP-ROS, UDP-ROS	UDP/IP	Shared-Memory, UDP, TCP
Operating System	Linux, Windows, macOS	OS-dependent	Linux, Windows

Table 1: Architecture comparison between ROS, DDS, and eCAL middleware

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.

2.3.3 Benefits and Challenges of Middleware

Middleware plays a crucial role in distributed systems by abstracting the complexity of communication between software components. It enables interoperability between heterogeneous systems, promotes modular design, and supports scalability by decoupling application logic from low-level infrastructure details [2], [15]. Through standardized interfaces and reusable communication patterns, middleware simplifies the integration of new components and accelerates system development.

However, the use of middleware also introduces several challenges. The additional abstraction layers can lead to performance overhead, making it more difficult to meet strict real-time requirements. Furthermore, debugging and testing become more complex due to the increased system opacity introduced by middleware components [2], [15]. Therefore, when choosing middleware, it is important to consider both its advantages and the limitations it may introduce to the overall system architecture.

2.4 eCAL Framework

2.4.1 Overview and Architecture

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 [1].

eCAL has a modular and flexible architecture. It supports multiple transport layers, including shared memory for communication between processes on the same machine and TCP or UDP for communication over networks. This adaptive approach ensures optimal performance by automatically selecting the best available communication method based on the system's environment

and requirements [14].

Figure 3 illustrates the basic eCAL architecture. It shows how multiple publisher and subscriber nodes communicate over topics. Additionally, a monitoring component observes the activity of each node to support debugging and analysis.

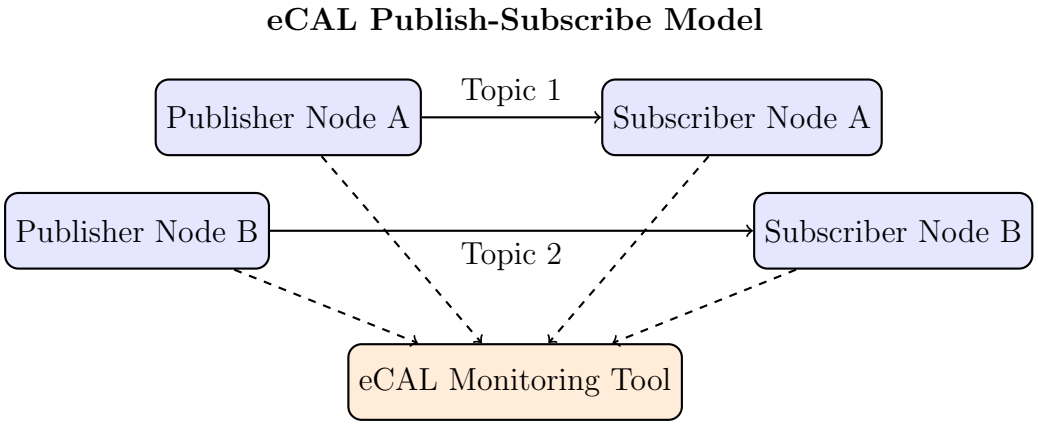

Figure 3: Simplified architecture of the eCAL framework with publisher, subscriber, and monitoring components

2.4.2 Core Features

eCAL provides several important features which make it suitable for a wide range of applications:

- **High Performance:** eCAL achieves very low latency and high throughput through optimized data transport methods, including zero-copy shared memory communication. This makes it ideal for real-time and performance-critical systems [1], [14].

- **Multi-Language Support:** eCAL offers interfaces for several programming languages such as C++, C#, Python, Java, and Go, enabling easy integration into various software projects [14].

- **Cross-Platform Compatibility:** eCAL supports multiple platforms, including Windows, Linux, and macOS. This compatibility allows easy deployment in diverse environments and distributed systems [14].

- **Built-in Tools and Monitoring:** eCAL includes tools for visualizing,

recording, replaying, and monitoring inter-process communication. These tools assist developers in debugging, performance analysis, and overall system reliability [1], [14].

2.4.3 Applications and Use Cases

Due to its efficiency and reliability, eCAL is widely used in industries that demand robust real-time communication. For example, in automotive applications, eCAL supports advanced driver-assistance systems (ADAS), autonomous vehicles, and vehicle-to-vehicle communication. Robotics applications benefit from eCAL's high performance in scenarios involving coordinated movements and sensor data processing. Additionally, industrial automation and IoT solutions utilize eCAL for efficient and reliable data exchange between numerous interconnected devices [1], [14].

2.4.4 Advantages and Limitations

One significant advantage of eCAL is its strong performance and scalability. Additionally, the framework's simplicity and diverse language support make it an attractive choice for a broad range of projects. Its open-source nature encourages community participation, ongoing improvements, and regular updates [1].

However, eCAL has some limitations. For instance, it lacks built-in Quality of Service (QoS) configurations, potentially restricting its suitability for applications requiring strict delivery guarantees or sophisticated communication controls. Furthermore, differences in maturity among various language bindings may impact ease of integration and implementation in certain programming environments [14].

2.5 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.

2.5.1 Test Levels

Software testing is organized into different levels to detect defects at specific stages of development. According to Ammann and Offutt [16], 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 [17].

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 [18].

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 [19].

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 [20].

2.5.2 Test Types

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

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 [17].

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 [20].

2.5.3 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 [19].

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 [18].

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 [16].

2.5.4 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 [21] to help development teams allocate their testing efforts effectively across different levels of abstraction.

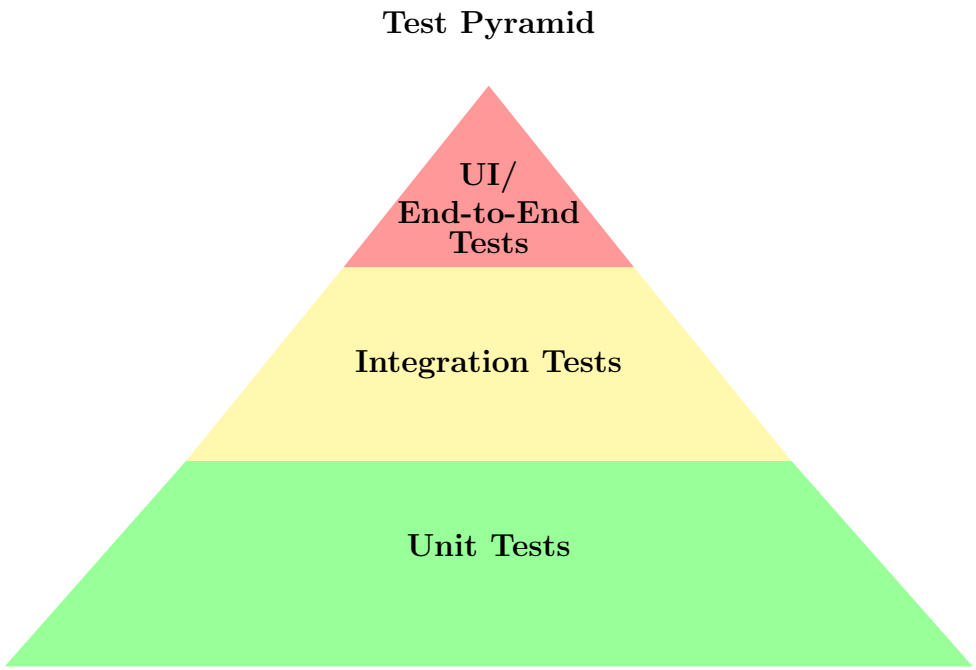

Figure 4: The Test Pyramid illustrating test distribution across levels

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 4, 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.

2.6 Testing in Middleware and Distributed Systems

2.6.1 Specific Challenges in Testing Middleware

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 [6].

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 [2].

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 [7].

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 [22].

2.6.2 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 [23].

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 [24].

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 [24].

2.6.3 Limitations of Current Testing Approaches for eCAL

The enhanced Communication Abstraction Layer (eCAL) is a high-performance middleware designed for fast inter-process communication. While eCAL provides tools such as monitoring and recording utilities, a comprehensive, standardized system testing framework is not currently available.

Existing validation tools focus primarily on **developer-centric debugging** rather than structured system-level testing. As a result, evaluating functional correctness under distributed conditions—such as message loss, timing jitter, or node failures—requires additional tooling or custom scripts.

Furthermore, eCAL's support for different transport protocols (shared memory, UDP, TCP) makes performance testing across deployment contexts more difficult. Without a formal benchmarking framework, assessing eCAL's performance under various configurations is challenging. Finally, interoperability testing with other middleware remains largely undocumented in the official literature, highlighting a gap in cross-platform validation methods.

3 Requirements and Design of Integration Testing for IPC Middleware

This chapter defines the requirements for integration testing in the context of IPC middleware and proposes a structured test strategy. The aim is to address communication-specific challenges such as message loss, timing behavior, and interaction between distributed components. Based on these requirements, a modular and reusable test architecture is designed, serving as the foundation for the implementation and evaluation in later chapters.

3.1 Analysis of Middleware 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, injection of delays, 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 and maintains stable communication.
- **Transport abstraction:** Middleware like eCAL 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

In the context of IPC middleware, integration testing does not only focus on testing individual components but also on validating how they interact with each other. These interactions include not just message correctness, but also performance, timing, and fault handling. Therefore, integration test objectives are typically divided into two categories: *functional* and *non-functional* requirements [6], [24].

3.2.1 Functional Objectives

Functional objectives ensure that the middleware behaves as expected during normal communication. One important goal is to verify that messages are delivered correctly from publishers to the intended subscribers. This also includes checking topic-based filtering, so that only relevant data is received by each component [15].

Tests should also cover how the system handles unexpected or incorrect messages. For example, messages with invalid formats, wrong types, or missing fields should not cause the system to crash or behave unpredictably [18].

In addition, if the middleware supports service communication such as Remote Procedure Calls (RPC), tests must confirm that each request gets a correct and timely response. This also includes timeout handling in case a service is not available.

3.2.2 Non-Functional Objectives

Non-functional objectives evaluate how well the system performs in different situations. One key metric is *latency*, which measures the time it takes for a message to travel from sender to receiver. Another is *throughput*, which shows how many messages can be transmitted in a given time [7].

Other important aspects include *fault tolerance* and *recovery*. The middleware should be able to handle failures, such as lost network connections or crashed processes, and continue working correctly after a restart [23].

Tests in this category often simulate high message loads, connection losses, or system restarts to see how the middleware reacts and whether it maintains stable communication without data loss.

3.2.3 Visualization

To give a clear overview of these requirements, Figure 5 shows a mind map that summarizes the functional and non-functional requirements. This diagram is useful for understanding the full scope of integration testing and serves as a reference for designing test cases.

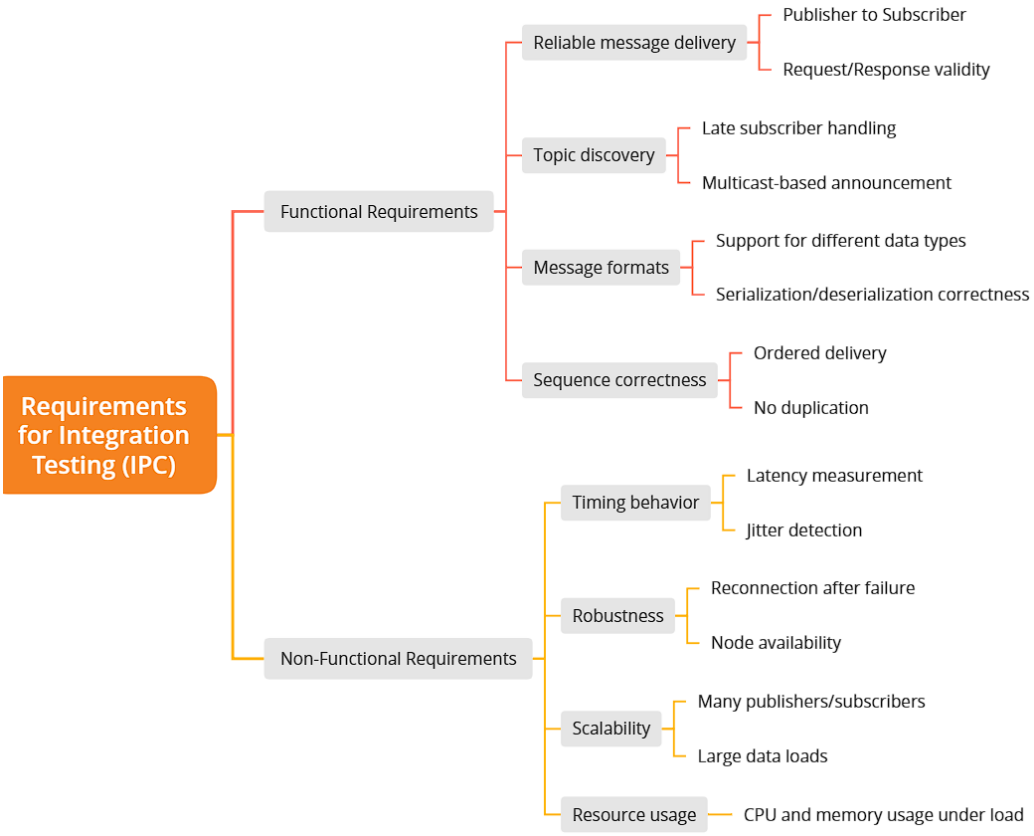

Figure 5: Functional, non-functional, and environmental requirements for integration testing of IPC middleware

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, IP multicast behavior, and transport compatibility.
- **Failure scenarios:** Introduce delays, drop publishers/subscribers mid-test, or overload the system to test resilience.

3.4 Design of a Modular and Reusable 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. These concepts will be implemented and validated using the eCAL framework in the following chapter.

4 Tool Evaluation and Framework 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 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++, Python, and Bash.
- **Automation Capabilities:** Support for test automation, headless execution, and integration into CI/CD systems is essential.
- **Support for IPC Mechanisms:** Tools 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 (especially Ubuntu 22.04), 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).

4.2 Evaluation of Tool Candidates

Figure 6 presents an overview of evaluated tools and orchestration options in the form of a mind map.

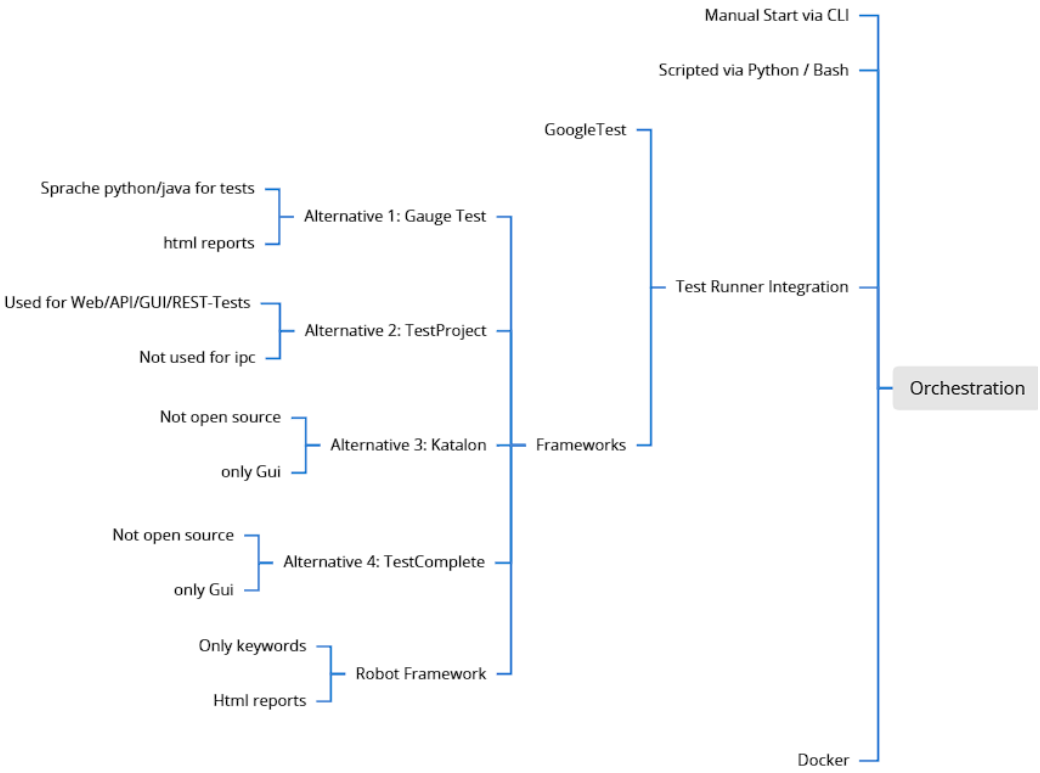

Figure 6: Evaluation of orchestration and framework options for IPC testing

GoogleTest

GoogleTest is a widely used unit testing framework for C++. It is already integrated into the eCAL source tree for verifying internal logic and module-level behavior. However, GoogleTest is primarily suited for unit testing and does not natively support multi-process orchestration. While it can be extended for IPC tests using wrappers or mocks, it lacks built-in support for orchestrating multiple processes across systems.

Robot Framework

Robot Framework is a generic test automation framework that supports keyword-driven testing. It can be used with Python and Bash and is well-suited for automating high-level integration tests, including orchestration and output validation. With extensions like the `Process` and `OperatingSystem` libraries, Robot Framework can start and stop eCAL processes, validate log output, and perform assertions on results. HTML reports and logs are automatically generated, making it ideal for integration into CI pipelines.

Gauge Test

Gauge is a test framework that supports writing specifications in Markdown and connecting them to test code written in Python, Java, or other languages. While it supports HTML reporting and automation, its community and ecosystem are smaller. It is also less commonly used for IPC scenarios.

TestProject, Katalon, and TestComplete

These tools are designed for GUI, API, and web testing and are not suitable for low-level IPC testing. Most of them are not open source and depend on graphical interfaces, which makes them unsuitable for CLI-based test automation in eCAL.

4.3 Feasibility for eCAL Integration

Based on the evaluation, Robot Framework was selected as the most suitable tool for testing eCAL-based IPC systems. It fulfills the necessary automation and scripting requirements, supports orchestration of multiple processes, and provides structured reporting. Furthermore, it can be easily integrated with Docker for simulating multi-host scenarios and is compatible with Ubuntu-based environments. GoogleTest will continue to be used for unit-level testing inside the eCAL source code, while Robot Framework will handle higher-level integration tests.

5 Implementation of Integration Tests for eCAL

6 CI/CD Integration

7 Evaluation and Discussion

8 Conclusion and Outlook

References

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

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

[3] 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`.

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

[5] 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`.

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

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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