Tampere University

Alexander Visa

MODULE-LEVEL TEST AUTOMATION

Solution for distributed systems

Master's thesis
Faculty of Engineering and Natural Sciences
Examiner: Mikko Salmenperä
Examiner: Matti Vilkko
October 2023

TIIVISTELMÄ

Alexander Visa : Module-level test automation: Solution for distributed systems
Diplomityö
Tampereen yliopisto
Automaation tietotekniikka
Lokakuu 2023

Tämän työn tavoitteena oli selvittää, voidaanko kerroksellista arkkitehtuuria noudattavalle hajautetulle järjestelmälle toteuttaa testiautomaatioratkaisu moduulitasolla. Lisäksi työssä haettiin kriteeriä riittävän kattavalle testaamiselle sekä menetelmää kerätä luotettavia testituloksia. Työ oli konstruktiivinen tutkimus, jossa testattavaa kokonaisuutta tarkasteltiin mustana laatikkona. Tutkimus osoittaa yhden testiautomaatio ratkaisun toimivuuden.

Testiautomaatioratkaisulla on mahdollista testata hajautetun järjestelmän osakokonaisuus valitulla kattavuudella ja halutulla luotettavuustasolla ilman tietoa moduulin sisäisestä toteutuksesta. Testitulosten riittävä luotettavuus voidaan varmentaa toteuttamalla hypoteesitesti. Ainakin kerroksellista arkkitehtuuria noudattavat järjestelmät, jotka hyödyntävät OpenDDS-kehysympäristöä, on mahdollista testata moduulitasolla toiminnallisten ominaisuuksien osalta. Toteutetulla testiautomaatioratkaisulla voidaan testata kuormitus- ja tietoliikennehäiriöiden vaikutusta ohjelmistoratkaisuun. Testiautomaatio on tekniikka, jonka kehitystä kannattaa seurata.

Avainsanat: Automaation tietotekniikka, hajautetut järjestelmät, testiautomaatio, musta laatikkometodi

ABSTRACT

Alexander Visa : Module-level test automation: Solution for distributed systems
Master's Thesis
Tampere University
Information Systems in Automation
October 2023

This study aimed to determine whether a test automation solution could be developed for a distributed system adhering to a layered architecture. Additionally, the study sought criteria for comprehensive testing and a method for collecting reliable test results. This work was a constructive research endeavour, where the system under test was considered a black box. The research demonstrates the functionality of one test automation solution.

With the test automation solution, it is possible to test a subset of the distributed system with selected coverage and desired reliability level without knowledge of the internal module implementation. The sufficient reliability of test results can be ensured by conducting hypothesis testing. Distributed systems following a layered architecture and utilizing the OpenDDS framework can be tested at the module level in terms of functional features. The implemented test automation solution allows for testing the impact of load and communication disruptions on the solution. Test automation is a technique whose development is worth monitoring in the future.

Keywords: information systems in automation, distributed systems, test automation, black box method

PREFACE

I humbly thank both Mikko Salmenperä and Matti Vilkko for the support and guidance I received. Your knowledge and good questions gave me perspective and challenged me in a way that benefited this work greatly.

This thesis marks an end of an era in many ways. The world in which I started my studies is a very different from the one I now graduate to. One university fusion, a global pandemic and ever tightening political environment later, here we are.

For now, my academic years are left behind. During my time as a student, I learnt to adapt to changes, and I learnt the importance of challenging myself. I also learnt the joy of overcoming obstacles. All these things are of great value, as for now I leave the academic world and face new challenges in the professional world. Looking back, the time at Tampere University was well spent, although at times it was not easy.

Thank you for all my teachers throughout my student years. Thank you for the friends I made, and who made student life rememberable. I also want to thank my family and especially my fiancée for their support throughout my student years in Tampere.

In Tampere, Finland, on 10.10.2023

Alexander Visa

CONTENTS

1. INTRODUCTION 1
2. DISTRIBUTED SYSTEMS 4
2.1 Characteristics 4
2.2 Software architectures 5
2.3 Data communication 10
2.4 Comparison of functionalities 11
3 SOFTWARE TESTING 13
3.1 Types of errors 13
3.2 Quality in software context 15
3.3 Testing process as part of software development 15
3.4 Classification of testing 16
3.5 Test coverage 18
3.6 Test automation 20
3.7 Bernoulli trial 20
3.8 Binomial Test 21
4 DESCRIPTION OF THE SYSTEM UNDER TEST 22
4.1 Architecture 22
4.2 Test environment 24
4.3 Implementation 25
4.4 Constrains of implemented system 28
5 TEST STRATEGY AND PLAN FOR SYSTEM UNDER TEST 29
5.1 Black Box approach 29
5.2 Acceptance testing 30
5.3 Module-level testing 30
5.4 Automated tests 30
5.5 Bernoulli trial 31
5.6 Tests 31
5.7 Designing a hypothesis test 32
5.8 Acceptance Criteria 34
6 RESULTS AND DISCUSSION 35
6.1 Test results 35
6.2 Discussion 39
7 CONCLUSIONS 41
7.1 Limitations of automated tests 41
7.2 Alternative solutions and further development 42

REFERENCES..43

LIST OF FIGURES

Figure 1. *Example of layered architecture [27]* ***5***
Figure 2. *Example of object-oriented architecture [27]* ***6***
Figure 3. *Example of data-centric architecture [27]* ***7***
Figure 4. *Example of event-based architecture [27]* ***7***
Figure 5. *Client-server example [37]* ***8***
Figure 6. *Peer-to-Peer architecture [29]* ***8***
Figure 7. *Microservices architecture [28]* ***9***
Figure 8. *Diagram of error detection [1]* ***14***
Figure 9. *V-Model [5 p. 8] [26 p.27]* ***17***
Figure 10. *Testing strategies [19]* ***19***
Figure 11. *System description* ***23***
Figure 12. *Test report* ***36***
Figure 13. *Test log file* ***37***
Figure 14. *Test run error* ***38***

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application programming interface
ATDD	Acceptance test-driven development
ATT	Automated testing tool
DCS	Distributed Control System
DCOM	Distributed Component Object Model
DDS	Data Distribution service
IDE	Integrated development environment
IDL	Interface Definition Language
IoT	Internet of things
OMG	Object Management Group
RF	Robot Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDLC	Software Development life cycle
SUT	System under test
QA	Quality Assurance
QC	Quality Control

1. INTRODUCTION

Distributed systems and parallel computing have become topical in various applications, such as Edge Computing, The Internet of Things and Cloud Solutions. As a result of the mentioned application fields, distributed computing has also become more common as a research topic in computer science. In general, the most significant benefit of a distributed systems is the possibility to share resources independent from location and the possibility to make more comprehensive use of a large pool of resources [24 p. 1]. Resource sharing can mean, for example, the decentralisation of stored data. Location independence, on the other hand, means that the system is not dependent on the location of its components. Thanks to these features, distributed systems are suitable for various applications. However, distributed systems are often designed for a specific purpose and come in many forms. In this thesis, the term distributed system refers to a collection of independent computing units that exist on different devices and communicate with each other to achieve the desired goal.

Most large-scale software has bugs that become apparent later in the software lifecycle. This may be due an incomplete requirement specification at the design stage or be a result of adding a new feature. Detecting abnormalities or defects is challenging, even if it is a centralised system that is considered a simpler case. As a result of distributed architecture, finding anomalies is more laborious. For stakeholders utilising a software product, a hidden defect is a risk that can materialise regardless of the quality of the software. The probability of the risk materialising can be reduced to an acceptable level by testing the software. [5 pp. 145–154]

The importance of design and software quality is emphasised in distributed systems, as issues and problems easily multiply**.** In the worst case, a failure of the subsystem can cause an error that will lead to system instability or failure. Due to this risk and the problems mentioned in the previous paragraph, various methods have been developed to examine and evaluate the reliability of distributed systems. By system reliability, I refer to a definition according to which reliability refers to the ability of a distributed system to provide functional services to users regardless of threats, failures, errors, disruptions, and malicious attacks. Reliability, on the other hand, can be further divided according to characteristics such as availability, reliability, integrity, maintainability, and safety. How the reliability of a distributed system can be positively affected at different stages of the

software life cycle are failure avoidance, fault, and error detection, as well as diagnostics and error tolerance. [24 pp. 2–13]

High-quality software satisfies the needs of the stakeholders within the allocated budget and schedule. Quality is relative to this definition. Good design and implementation alone cannot build a complete distributed software. At its best, system reliability assessment helps to narrow down the areas affected by software failures. However, to avoid failures, as well as to detect errors, it is necessary to carry out software quality controls and verifications. Manufacturing quality software products is complex because software quality control covers the entire software development process. Quality is built on several things during the software development process. According to one model, the quality requirements can be further divided into three parts: quality requirements during use, quality requirements during the product life cycle and requirements during transitions. [9]

One of the key parts of software development to gain insight into the quality of systems is software testing. During testing process, the software product receives model inputs to generate a response, which then is analysed after a test run. Unlike many other industries, software product failures can occur and accumulate throughout the software's life cycle. [5 p. 37]

The testing of distributed systems differs from the testing of centralised systems for example in the terms of their scope. Testing can be used to collect information on the functioning of modules within a distributed system. In general, the test methodology can be divided into the white box and black box approaches 2008, pp. 47–49. p. 68].

In the white box approach, the structure and content of the object are known, and testing is planned by studying it. In the black box approach, detailed information about the structure of the object in not known and the test planning is based on statistical analysis of responses. In addition to these approaches, there are other possible methods such as functional testing, acceptance testing, performance testing, security testing and integration testing. Different kinds of approaches are used to collect information on the system's performance in different areas. [4]

In the test automation process, another program performs the manual steps or actions of testing which otherwise would be performed by the user. In automated testing, the validation of a software product is implemented by using another software as a tool. The testing software contains functionalities for performing and automating tests. Software specialized in automated testing performs the tests and guides their execution. In addition, the software reports the test results in a user-friendly format and compares them with previous results. [7 p. 3]

Software measurement collects metrics that can be utilised in software development and maintenance processes. Metrics can be collected from the software product itself or the software development process. The quantitative metrics collected from the product reflect the characteristics of said product. The quantitative metrics collected from the process reflect the software development process and relating features. These include, for example, the number of detected deviations or the number of defects found in the version. Based on the collected data, it is possible to monitor, how well the software product meets the requirements set for it. On the other hand, the information obtained from the process can be used to observe regulations set by laws or different practices. [10]

The research question in this case is formulated to examine **how the module-level functional properties of distributed systems can be automatically tested when the module can only be studied as a black box.**

Chapter two goes through what a decentralized system is in more detail. The third chapter discusses the theory related to testing. The fourth chapter describes the system to be tested and the fifth chapter contains the test plan. The sixth chapter presents the results and seventh chapter conclusions.

2. DISTRIBUTED SYSTEMS

In a distributed system, components work together across a network to achieve a common goal. According to this definition, a wide range of systems used in different sectors can be interpreted as distributed systems. Traditionally, distributed systems have been used in industrial automation, defence systems and data communications. [12]

Today, distributed systems can be found in multiple industries and contexts because of the world's interconnectedness. Using distributed systems in the banking sector, healthcare or by governments is becoming more and more common. [12] Most technical systems that are used in various industrial settings, for example the distributed control system, are distributed systems by their architecture. [15][8]

2.1 Characteristics

Distributed systems are not new. They have been used since the early days of computer science. However, the systems and their definitions have changed significantly over the years. As technology has evolved, parts of distributed systems have become increasingly sophisticated and complex. [23]

According to sources [12], [23–11] and [3]: distributed systems are those with the following characteristics:

1. The system consists of more than one process.
2. The Processes exchange information with each other over the network.
3. The Processes are in a different address space.
4. The Processes work together to achieve the desired goal.

For example, the World Wide Web, social media, bank networks, peer-to-peer networks, real-time distributed systems, sensor networks, network computing and cloud computing are counted as distributed systems according to these criteria [22].

A distributed system is not comparable to decentralized computing. The latter refers to the allocation of resources concerning workstations or work centres. [16]

2.2 Software architectures

Although distributed systems are mostly designed for a specific purpose, there are also well-established structures for them. However, these software architectures are not mutually exclusive, as distributed systems can follow more than one software architecture simultaneously. The proposed architectures in Sources [27] and [36] contain a list of distributed systems software architectures such as:

1. Layered Architecture
2. Object-Oriented Architecture
3. Data Centered Architecture
4. Event-based Architecture
5. Client-Server Architecture
6. Peer-to-Peer Architecture
7. Microservice Architecture

A layered architecture consists of levels on which the components of a distributed system are arranged. Levels can be hierarchical or in free order.

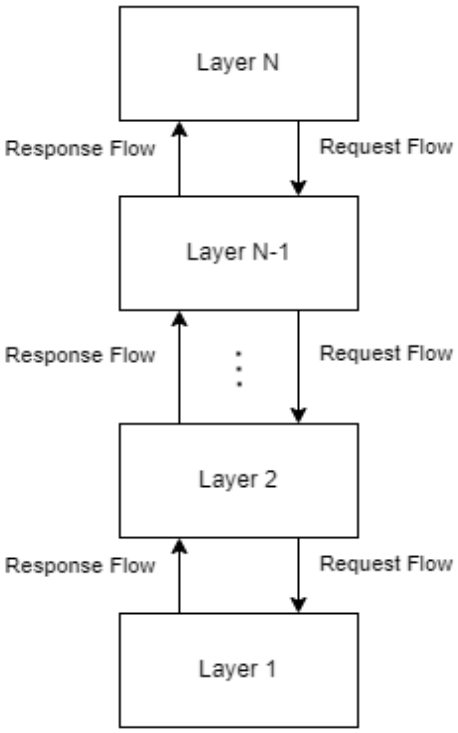


Figure 1. *Example of layered architecture [27]*

Their purpose is to divide the software into manageable sub-entities. In a layered architecture, communication between sub-entities is limited so that only modules located on neighbouring levels can exchange information with each other. Messages sent to levels more distant than neighbouring levels travel one level at a time until they arrive. However, layered architecture can in some cases be implemented in such a way that messages do not have to pass through each layer. This has the potential to improve system performance, but on the flip side, the arrangement requires that the level in question can communicate with levels other than neighbouring ones. [27][36]

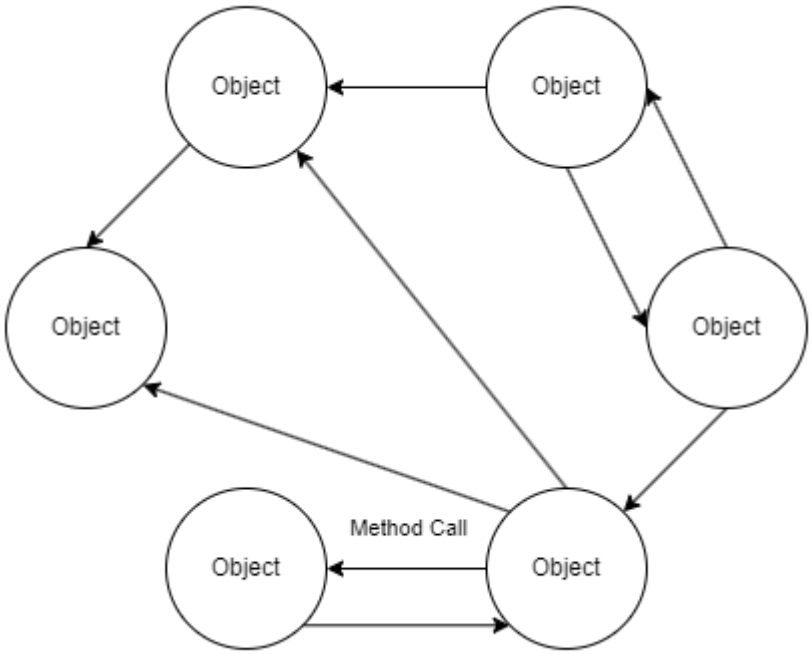


Figure 2. *Example of object-oriented architecture [27]*

In object-oriented architecture, a distributed system is built from objects capable of storing information and transmitting it. There can be an arbitrary number of objects. Objects form a loosely connected system, i.e., the objects mainly operate independently, but they can also form or break connections with each other. The interaction between objects takes place through method calls. The connection between objects is established using the principle of Remote Procedure Call (RPC) or Remote Method Invocation (RMI). [27][36]

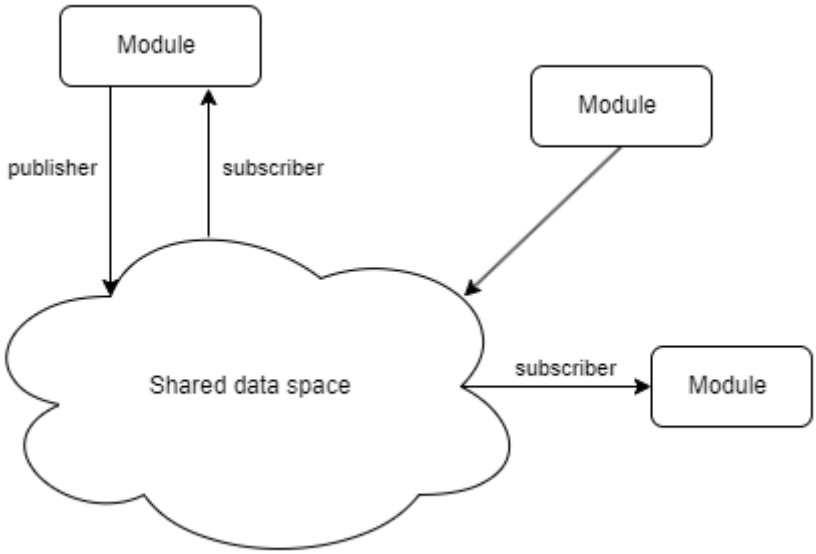


Figure 3. *Example of data-centric architecture [27]*

In a data-centric architecture data and its utilisation are central. In this architecture, the core of the system is formed by a centralized database to which all modules of the system are connected. Modules can export or import data to and from a database.[27][36]

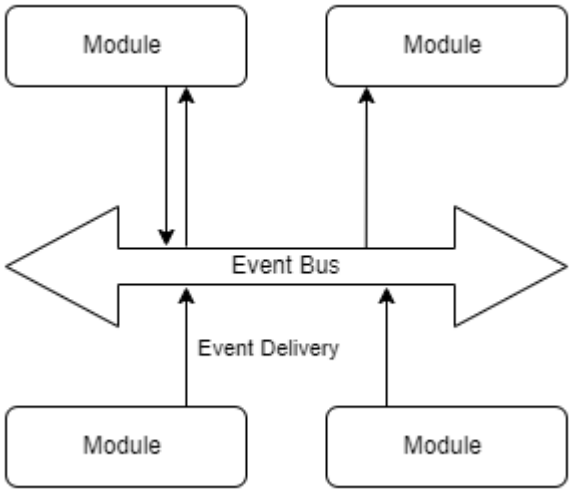


Figure 4. *Example of event-based architecture [27]*

Event-based architecture is very similar to data-centric architecture. Instead of data, events are at the heart of the system. Events are sent from the modules to the event bus, from where they reach their recipients. In this architecture, all communication between modules is implemented as events. Modules form a loosely connected system, like objects in object-oriented architecture. [27][36]

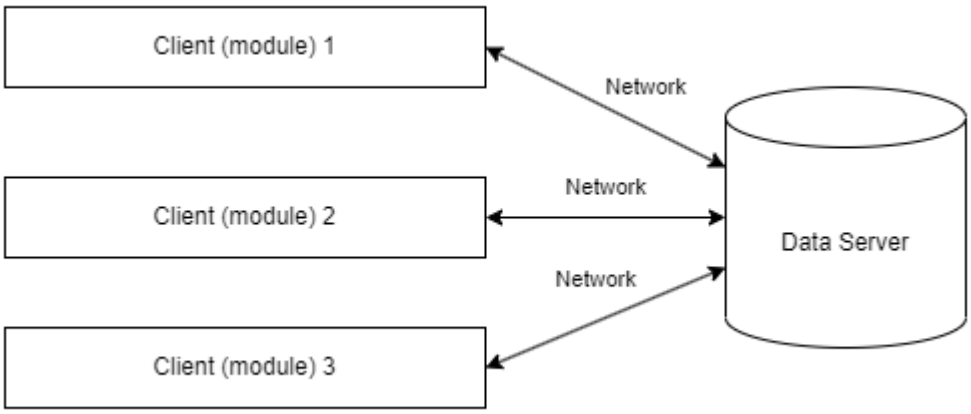


Figure 5. *Client-server example [37]*

In a client-server model the services offered by the server are modelled as a group. Here, the server refers to a process which receives requests and sends a response. A client refers to the process which sends requests to the server. [37]

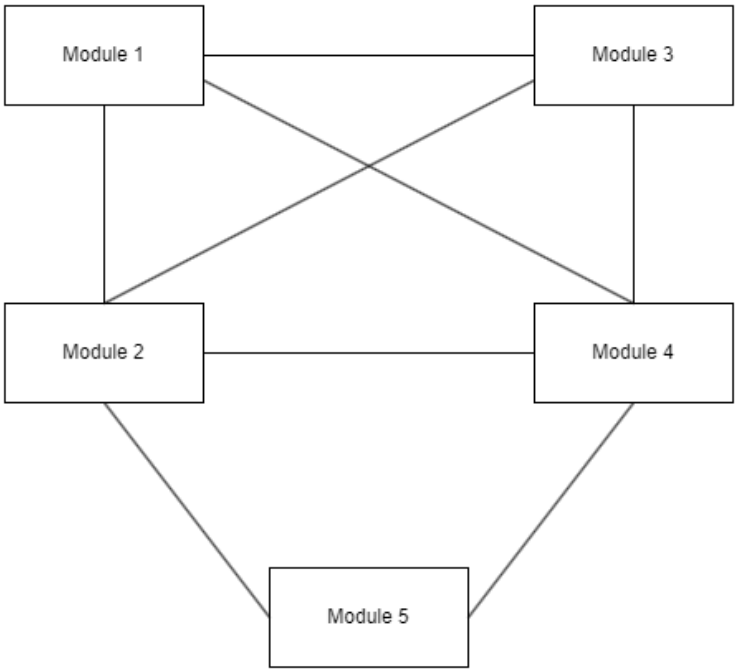


Figure 6. *Peer-to-Peer architecture [29]*

In peer-to-peer architecture, modules can have multiple functions. However, the system's modules are similar in characteristics. Thus, each component can provide or request resources as well as services when needed. [27][36]

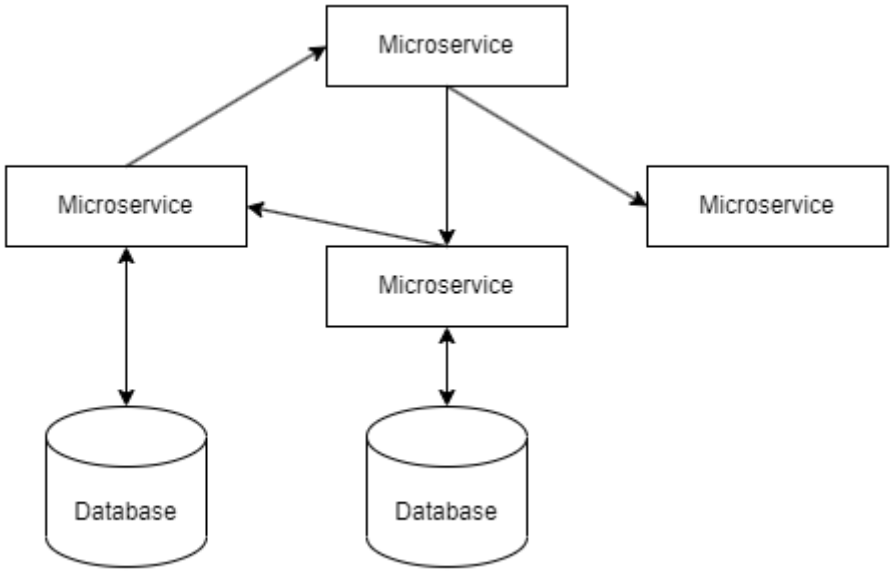


Figure 7. *Microservices architecture [28]*

In microservice architecture, a system consists of a collection of small, independently functioning services that solve problems together. Each service operates independently and implements a single functionality. [27][36][28]

2.3 Data communication

One of the key areas regarding distributed systems are the data communications solutions. Their functions as part of distributed systems are to enable the transfer of information between the different parts of the software. This is to ensure that the parts responsible for the functionality of the software can perform their functions. Designing and implementing a data communications solution suitable for different applications has not been straightforward. As technologies have developed, data communications solutions have changed by leaps and bounds. [11]

Table 1. Distributed systems paradigms and communication solutions [11]

Year	Technology and Paradigm	Communication
1960–1970	Inter-process Communication (IPC) Client-server Supercomputer ARPANET and early internet	Datagram transport (ATM, X.25)
1970–1980	ARPANET UNIX Initial conception of TCP/IP and UDP protocols Distributed operating systems	IP addressable hosts can communicate using datagrams
1980–1990	Home Computer Standardized TCP/IP and initial Internet Generalised OS	TCP/IP becomes the standard internet protocol of the internet
1990–2000	HTTP .HTML WWW P2P Mobile Computing	Remote objects and procedures, enabled, the development of early middleware. HTTP over TCP/IP popularise the internet. P2P protocols, group communication
2000–2010	Web Services Grid computing Community Computing Virtualized Commodity Clusters Cloud computing	Cluster middleware REST, WSDL, XML, JSON, MQTT, XMPP, XEN and KVM hypervisor
2010–2020	IoT Edge Computing Fog Computing	P4, OpenFlow Open SVN

Table 1 summarises the trajectory of distributed systems [11]. Table 1 above collects the technologies, paradigms and communication solutions used in the development of distributed systems. The table shows how, as technology has developed, data communication solutions for distributed systems have become increasingly sophisticated. On the

other hand, the number of different possible data communication solutions has also increased over time as different technologies require different features of data communications.

The architecture chosen in the system design phase, and the application of the system, largely determines the communication solution of the application. As the size of distributed systems has grown, the demands on data communications solutions have also increased. Developing and maintaining data communications solutions has also proven to be a complex and laborious task over time. This has eventually led to the outsourcing the data communications to middleware and various connection middleware. Ready-made standardized solutions help focus the resources on the development of the system itself.

2.4 Comparison of functionalities

Distributed systems have been a research area in the field of information technology for more than 60 years. Over the decades, the perception of distributed systems and the opportunities they provide has changed. During this time, distributed systems have, among other things, influenced the development of information technology and laid the foundation for the emergence of the Internet. In addition, because of technological breakthroughs, they have led to the emergence of new paradigms such as cloud computing, fog computing and the Internet of Things. [11]

As technology has developed, the number of functionalities offered by the systems and the requirements placed on them have multiplied. Several sources examine the benefits, but also the disadvantages, of distributed systems in general. Comparison of functionalities is mainly indicative, as practical solutions determine the actual characteristics of the system. Source [23] provides a table summarising the main differences between centralised systems and decentralised systems.

Table 2. Comparison of benefits and risks [3, p. 9]

Criteria	Centralized system	Distributed system
Economics	Low	High
Availability	Low	High
Complexity	Low	High
Consistency	Simple	difficult
Scalability	Poor	good
Technology	homogenous	heterogenous
Security	High	Low

Compared to centralized systems, distributed systems can achieve significant benefits, especially in the areas of cost-effective performance, scalability, and fault tolerance. On

the other hand, as the system grows, it easily becomes complex, difficult to maintain, and vulnerable to external hostile attacks. [12]

3 SOFTWARE TESTING

Testing is a key support function that is utilized at different stages of the software product's life cycle. It is essential for both the product development process and maintenance. As a result, testing can be found in several different software development models. The software development process typically proceeds in cycles that can be sequential, repetitive, or grow the product step-by-step. Because of this, and the considerable amount of work required for testing, which in software development models has been broken down into levels, each with its own purpose. [4 pp. 1–3]

3.1 Types of errors

Software often has different types of errors, and it is important to determine them to fix them and, on the other hand, to be more efficient. Different types of errors require different approaches to correcting them. There also is not any universally valid, comprehensive, and precise definition for an error. Defects can be classified, for example, based on the ways in which they occur. For example, an error is an anomaly that occurs during programming, while a fault or an exception is the occurrence of one or more errors during program execution. A failure occurs when code is executed with an exception and the consequences are multiplied which leads to eventually changing the response of the program. [1][26 pp. 21–22]

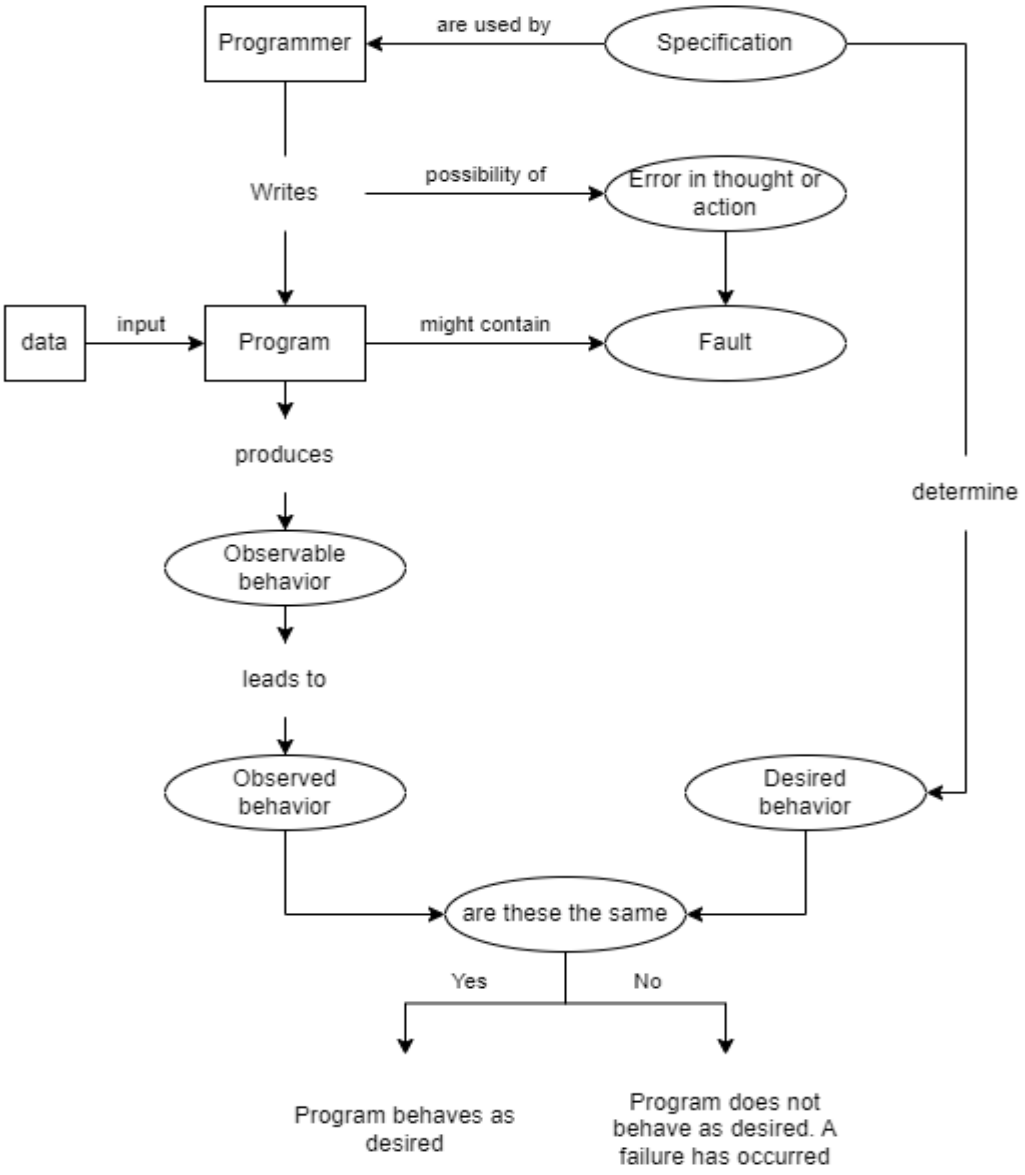


Figure 8. *Diagram of error detection [1]*

An error or other type of malfunctioning is therefore the result of one or more interruptions or exceptions. However, the terms error, mistake and defect are also used to denote the same thing. A bug is a type of anomaly that is result of errors specified above. However, a malfunction or error can also be more commonly referred to as a bug or defect. A fault is a malfunction that occurs during the definition phase. On the other hand, a fault is called a bug if it occurs in the source code. A failure is a consequence of performing a fault. When the output produced by the program does not match the expected result, it is also called a failure. [1][26 pp. 21–22]

For example, syntax errors that occur in the software context, are often clearer than semantic errors. Regardless of the origin of the error, it is often difficult to detect them. As a result, it is justified to study software empirically. Testing process is one way to determine whether there are errors in thinking, functioning, or implementing of a product. The procedure collects information on anomalities and similarities, i.e., how well the previously mentioned factors correspond to the desired state specified in the requirements. Testing can also determine how well the operations and the implemented product comply with the requirements set for them. Testing can also be used to verify that the concept or method has been sufficiently understood. [1]

Test deliverables can be used to collect large amounts of metrics about the software and its development process. Testing can also be used to measure, how the software has developed in between testing. This measurement can provide information on the development of test coverage, the number of deviations and the reliability of the software. Even when directly quantitatively measuring software features, the results must be analyzed separately. [4 p. 29] It is important to also keep in mind that rare errors can occur over a long period of time [26 p. 29].

3.2 Quality in software context

There are many definitions of quality in the context of software. Software quality is a multidimensional, yet measurable feature. Typically, quality factors are divided into two parts: static factors, i.e., code and documentation relating to it, and dynamic factors, i.e., behavior during execution. [1]

Static quality factors are structure, maintainability, and testability of the code. Dynamic quality factors are determined based on several test runs. Such quality factors are correctness, reliability, readiness, integrity, usability, and performance. [1]

It is likely that a part of a system consisting of numerous modules will fail. A failure can occur on any layer of a distributed system. As a result of the layered architecture of a distributed system, it is often more sensible to divide the system into testable levels rather than try to implement testing with an end-to-end approach. [18]

3.3 Testing process as part of software development

Software testing can be carried out at different stages of software development using different methods. Usually, software is already tested in the development phase, but it can also be done after the product has been completed. Thus, software testing consists of verification and validation. Verification is defined as the evaluation process of a system

or component, in which it is determined whether a product implemented during the software development period meets the conditions set for it. Verification is carried out throughout the software development phase to ensure that the software being built corresponds to the original design. Validation refers to the process by which a system or component is evaluated at the final stage of its development against the requirements imposed on it. Validation is dynamic testing and requires running the program to determine failures and their causes. [26 p. 20]

Quality Assurance (QA) activities are designed to monitor standards and techniques used to improve the development process. In addition, QA activities are aimed at preventing known previous failures. QA activities aim to achieve a high-quality software product by investing in the quality of software development throughout the software life cycle. In addition, the activities aim to prevent shortcomings in software design and specification in the software development process. Quality Control (QC, is a method that aims to build a software system and test it comprehensively and extensively. This procedure is aimed at eliminating the cause of errors and failures. Quality Control focuses on products more holistically than QA. The monitoring and correction of software flaws is done after the software development ends at its various levels. [26 p. 23]

3.4 Classification of testing

Throughout time, numerous different ways of testing software have been developed. However, choosing the appropriate combination of approach, method and testing level is not always a simple task. Testing methods can be structured through various independent attributes related to testing activities. The most significant attributes are summarized in the Table 3 below.

Table 3. Independent attributes of a software testing activity [2, pp. 147-149]

The primary attributes	The secondary attributes
Scale	Oracle
Goal	Test life cycle
Property	Test assumptions
Method	Test completion
	Required artifacts
	Stakeholders
	Test environment
	Position in the life cycle

The table contains attributes that can be defined to identify a suitable way to test the software. The first column contains the primary attributes. These include, for example, the size of the software being tested, the purpose of testing, the features to be tested

and the methods by which the test data is produced. The second column contains secondary attributes used for classification. The first mentioned is the oracle, i.e., the agent, who checks that the outcome of the test performance is consistent with the expected outcome. Other attributes listed in the column include test lifecycle, assumptions, test completion criteria, required artifacts, involved stakeholders, test environment, and test position in the life cycle.

Software can be divided into smaller sub-entities to facilitate testing. The number and scope of levels may vary on a case-by-case basis. Depending on the software development model, it is not always possible to start testing right away. For example, when developing software according to the waterfall model, it may not be possible to test the system during the development phase [20]. One widely used software development model that divides the software into testable levels during the development phase is the V model. The V-model is one of the many lifecycle models of software development. It is also based on the waterfall model. There are different versions of the V-model, which differ in terms of limits and quantities of test levels. [26 pp. 26–27]

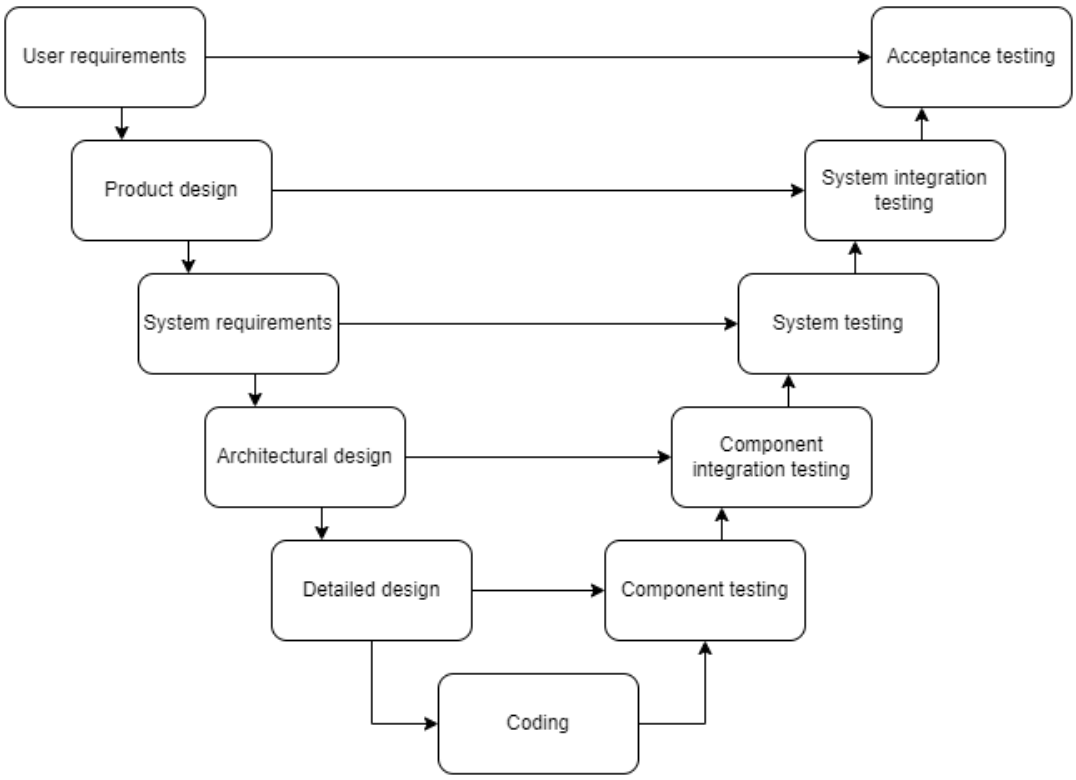


Figure 9. *V-Model [5 p. 8] [26 p.27]*

V-model divides software validation into test levels and defines an appropriate testing method for each level. The template includes four levels, which are in order from the top level to the lowest level:

1. Requirement analysis and specification
2. High level design
3. Detailed design
4. Implementation

The implementation of the requirement specification is verified by means of acceptance tests based on the requirements presented in the level plans. The correctness of high-level system design, i.e., architecture, is verified through system tests. The correctness of the architecture and module design are verified with both unit tests and integration tests. [26 pp. 26–27]

By utilizing the v-model in software development, testing can begin immediately after the software development has ended. By starting software verification at an early stage of software development, the probability of finding faults increases and fixing them can equally be started earlier. However, due to its rigidity, the V-model is not suitable for projects whose requirement specifications change. Failures that occur later at low levels of the software development process also cause problems. [26 pp. 26–27]

3.5 Test coverage

The term test case and the term test are used interchangeably. The test case is defined as consisting of input to the program and assumed output values. The feed may also contain preconditions and output postconditions depending on the intended use. In addition, the test cases must be identifiable, for example, according to the identifier assigned to them. In many cases, a description of the purpose of the test, the name of the test author and the date of design are added to the test case. A test case has been successfully completed if the output produced by the program matches the output expected from the program and any pre- and post-conditions are met. In general, a test case that reveals a malfunction or a failure is considered a good test case. [26 p. 21]

A collection of tests, i.e., a set of test cases, is called a test suite. The term may consist of various tests and their combinations. However, the test suite is not order-dependent. The test suites are comparable to other software documents, as they contain information that can be utilized in software maintenance. [26 p. 22]

Test coverage measures the quality and scope of testing. However, there is no single simple formula for deciding the appropriate coverage. [6].

The quality of the program under consideration cannot be assured unless the operation of the software has been examined with all possible inputs under all possible conditions. This of course is not realistically achievable due to limited resource, for example. In this case, it is important to examine what the aim of testing is. For example, testing the software can find and eliminate bugs or show that no bugs have been found. Other possible objectives of testing may be to estimate the frequency of errors or to ensure that errors are rare. Depending on the testing goal, different logical or stochastic requirements can be defined to ensure that the program is tested sufficiently comprehensively. [2 pp.163–182] [6] [19 p.27]

Software testing can be approached strategically from many perspectives. One way to break down testing is by their implementation techniques. The three most significant software testing techniques are functional testing, structural testing, and operational testing. [6][19]

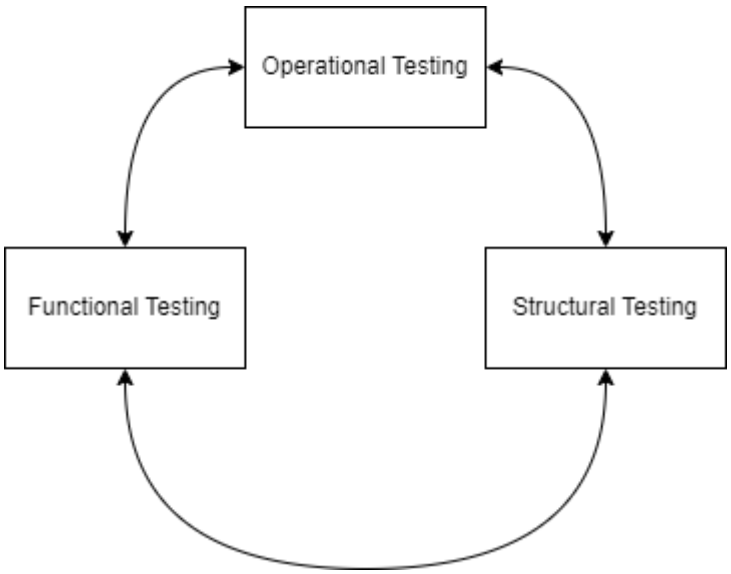


Figure 10. *Testing strategies [19]*

Test coverage is defined depending on which part of the software is being viewed. The software being tested (SUT) is broken down to observable entities. For example, it is possible to select a function or variable as an entity. After chopping the program code, it is possible to calculate a simple KPI for test coverage. [19 pp. 27–29]

$$Test\ coverage = \frac{amount\ of\ covered\ entities}{amount\ of\ available\ entities} \cdot 100\ \% \quad (1)$$

Test coverage is sufficient when all the entities have been tested. However, full test coverage does not guarantee that the system under test, i.e. (SUT), is flawless. In other words, a limited number of tests cannot ensure the complete operation of the system.

The test coverage of functional characteristics may be measured accordingly. The software under test (SUT) is divided into entities that fall into either the high, medium, or low groups based on their appearance frequency. The groups are then given emphasized coefficients according to their importance. In this case, the emphasized arithmetic mean can be calculated. The formula for calculating test coverage for functional properties can be used to determine test coverage. On the other hand, it can be used to decide on adequate test coverage. [19 p.32]

$$Operational\ coverage = \frac{\sum_{i=1}^{3} w_i \cdot x_i}{\sum_{i=1}^{3} w_i} \cdot 100\ \% \tag{2}$$

3.6 Test automation

Testing large or otherwise complex systems is not only laborious, but it also requires high accuracy. As the number of tests and the number of times they are performed increases, so does the risk of human error. To maintain the comparability of test results, the homogeneity of the test runs and for efficiency, it makes sense to automate the testing work. However, not all tests can be easily automated, and the entities should be examined on a case-by-case basis. [1]

3.7 Bernoulli trial

Bernoulli trial, also known as the repeat test, is a random experiment with exactly two outcomes. In addition, the probability of success remains constant with each repetition. The simplest example of a repeat experiment is a coin toss, where there are two outcomes, and both are equally likely. [21 p. 29]

The binomial distribution occurs when a series of independent experiments with two outcomes are carried out. Number of experiments *n*, probability of success *p*.

$$\Pr\{X = k|n,p\} = \binom{n}{k} p^k (1-p)^{n-k}, kun\ k = \{0,1,2,\dots,n\} \tag{3}$$

When the number of repeats in the repeat experiment is high, the expected binomial distribution is the probability *p* times the number of replicates *n*. By experimenting with different values for probability, the number of repetitions required can be calculated. [21 pp. 47–48] [17 p. 86]

3.8 Binomial Test

A Hypothesis test is called binomial test, when the observed data can be divided into two differentiable groups. The sample is divided to *n* amount on independent tests. Each separate test trial belongs to either group C_1 or C_2. Observations n_1 belong to group C_1 and the observations n_2 belong to the group C_2. accordingly. Each test trial bears the identical probability, p, which belongs to group C_1 and is alike with all the n amount of tests. The probability $q = 1 - p$ and its results belong to group C_2. A binomial test can be symmetrical depending on the hypothesis itself. [25 p. 47–48]

The following formulas 4-6 showcase a two-sided case and two one-sided cases:

$$H_0: p = p_0 \quad H_1: p \neq p_0 \tag{4}$$

$$H_0: p \leq p_0 \quad H_1: p > p_0 \tag{5}$$

$$H_0: p \geq p_0 \quad H_1: p > p_0 \tag{6}$$

In this work the binomial test is inspected as a one-sided test. When the amount of test trials is more than 25, the binomial distribution can be approximated with the normal distribution. The variable t is taken from the following binomial table:

$$t = n \cdot p_0 + z_\alpha \cdot \sqrt{n \cdot p_0 \cdot q_0} \tag{7}$$

Let's look at the error in decision-making.

Table 4. Test conclusion

Condition	Do not reject H_0	Reject H_0 and use instead H_1
H_0 True	Valid	Error
H_1 True	Error	Valid

When the null hypothesis is valid, the counterhypothesis is rejected. This applies also vice-versa, as demonstrated in table 4 above.

4 DESCRIPTION OF THE SYSTEM UNDER TEST

The simple system implemented in the work models the operation of a distributed system module. The purpose of the system is to demonstrate that it is possible to use automated testing to test the functionalities of a distributed system module that at minimum follows a layered architecture. The system's data communication has been implemented in such a way that it supports real-time data transfer. The system under test has been built using existing technologies.

4.1 Architecture

The computing units of distributed systems may not be completely identical in their characteristics. Examples include IoT systems, where component performance varies from case to case. On the other hand, the chosen data transfer solution is important for the reliability of operations.

Software architecture selection is a multi-step process that includes going through the properties of different structures to determine the most suitable solution for the purpose. There are numerous possible architectures, but the object-oriented architecture, data-centric architecture and layered architecture presented earlier appear to be most usable during preliminary examination, as a system that follows the architectures can be implemented agilely in parts. However, layered architecture was chosen as the architecture of the software implemented in the work, since it allows the system to be narrowed to be suitable for testing. In addition, its strengths are also expandability and clarity.

Table 5. System dependencies

Dependency	Version Number	Purpose
Ubuntu	22.4.2 LTS	operating system
OpenDDS	3.23.1	framework
pyopendds	0.2.0	framework
Make	4.3	build automation tool
cmake	3.22.1	build automation tool
python3	3.10.6	programming language
c++	11	programming language
Perl	5.34.0	programming language
Pip	22.0.2	package installer for python
Git	2.34.1	version control system
ninja	1.11.1	package
Jinja2	3.1.2	package
pybasic	0.0.0	package
pylocation	0.0.0	package
pygments	2.9	package
Sphinx	2.4	package
Robot Framework	6.1	automation framework

Table 5 lists the dependencies required by the system and their version numbers. As shown in the table, different types of packages and programming languages are needed in this test automation implementation and in distributed systems more generally.

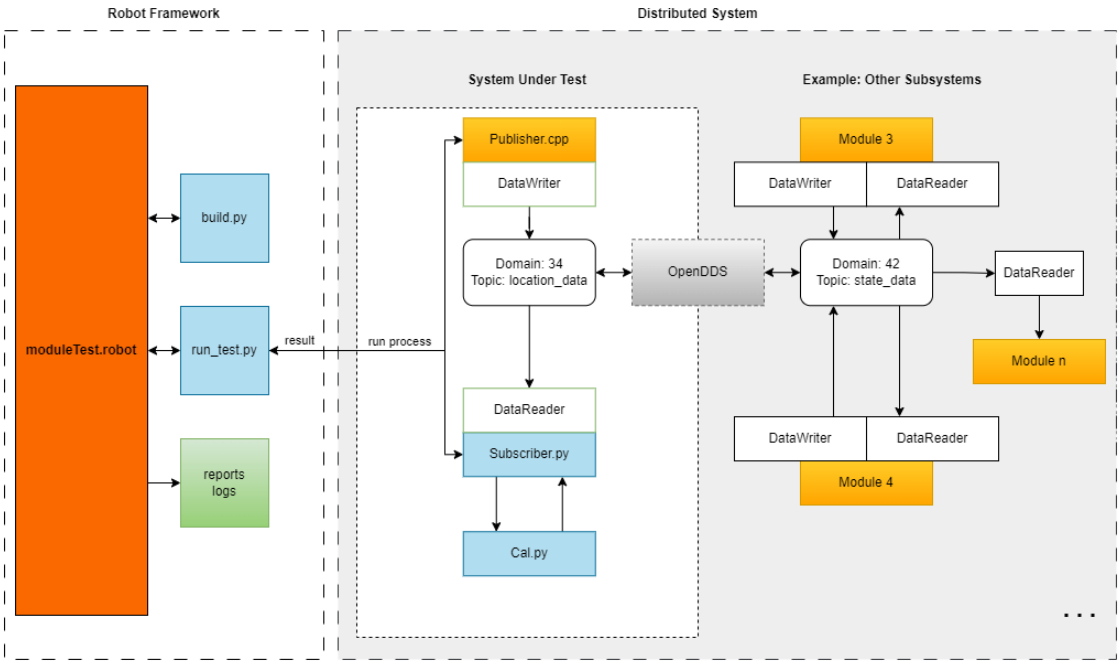


Figure 11. *System description*

The test automation solution and the system under test are described in Figure 11. The system consists of two parts, a robot framework implementation, and a distributed system from which a part has been selected for testing. The robot framework is responsible for building the program, performing the tests, and reporting the results. System Under Test consists of a publisher and subscriber module.

4.2 Test environment

The environment is built in such a way that it runs on a Linux-based Ubuntu operating system. Alternatively, the environment could be based on different versions of the Windows or MacOS operating system. The operating system selection criteria included the customizability of the environment and the requirement that versions of the software be available that work on that operating system. Visual Studio Code (VS code) is the free distribution version used as the programming environment (IDE). [38]

There are several options for system data communication solutions. Although the data communications solution is generally a key part of distributed systems, in terms of implementation, it is secondary. In this case, it makes sense to choose a ready-made middleware solution. These include MQTT (Message Queuing Telemetry Transport), OpenDDS (Open Data Distribution Service), AMQP (Advanced Message Queuing Protocol), CoAP (Constrained Application Protocol). OpenDDS is an open-source implementation of the Object Management Group (OMG) machine interface standard that enables reliable, efficient, real-time, and scalable collaboration between software. In addition, OpenDDS middleware is implemented on top of the ACE abstraction level, which in turn enables parallel communication. [32] [35]

Much of the OpenDDS middleware is implemented with C++ programming language. [31] Programming languages suitable for the implementation of the tested system include Java, NodeJS, Ruby and Python. Python was chosen as the main programming language for this work due to its version compatibility, large extension library, and straightforward syntax. [33] [39]

An open-source implementation called the Robot Framework was chosen as the test automation framework environment to be used in the work due to its versatility. In addition, it is possible to extend the functionalities of the framework environment with Python libraries. Other alternative test automation tools include Selenium, Cucumber, Cypress, TestNG, and Gauge. [34]

A key component of the test environment is the PyOpenDDS framework environment, which implements the DDS application programming interface (API). By utilizing the interface, the functionalities required for data communications can be implemented in the Python programming language. [30]

4.3 Implementation

The tests carried out in the work are built based on example tests provided by the Pyopendds Framework. The Robot Framework test automation system is responsible for the automatic execution of the tests. The automated tests are implemented with the Robot Framework test "script" as shown in Program 2.

moduleTest.robot

```
*** Settings ***
Documentation    Module test
Library          Process
Library          OperatingSystem
Library          String
Suite Setup      Prepare
Suite Teardown   Terminate All Processes  kill=True


*** Test Cases ***
Distance Between Warsaw and Kyiv
    [Tags]  happy_case
    # Warsaw: 52.226934793927875, 21.006939565272003
    # Kyiv: 50.447434195322955, 30.529324477208196
    # Distance ~ 693.30 Km
    ${result}=    run process    python3    run_test.py    1
    Should be equal as integers  ${result.rc}  0
    ${ans}=  Get Result  ${result.stdout}
    Should Be Equal    689.859    ${ans}

```

Program 1. *Test collection settings and test schema example*

The tests carried out follow a similar structure. The tests are marked with a happy_case or unhappy_case tag. At the beginning of the test, the coordinates of the location in question have been added as a comment. Test runs use figures scaled by a thousand to avoid errors caused by floating point numbers. The run process section utilizes the run process routine found in the Robot Framework's Process library, which is used to executing processes and capturing the response they produce. This starts the publisher and subscriber processes. After the processes are completed, the library routine returns

an object that contains information about the process. When the state of the return value in this field is zero, it is known that the process has been completed successfully. This is followed by the Get Result helper function, which reads the result of the calculation operation from the output stream of the process. If the assumed and obtained results are the same, it is stated that the system is working correctly, and the test performance is marked as successful.

```
111
112  *** Keywords ***
113  Prepare
114      Set Log Level    TRACE
115      ${result}=    run process    python3    build.py
116      Should be equal as integers  ${result.rc}  0
117      log    Build Successful
118      log    ${result.stdout}
119
120  Get Result
121      [Arguments]  ${str}
122      log  ${str}
123      ${r}=  Get Lines Containing String  ${str}  Subscriber << RESULT
124      ${ans}=  Fetch From Right  ${r}  =
125      RETURN  ${ans}
```

Program 2. *Helper functions used in tests*

Helper functions are required to perform tests. The upper helper function is performed only at the beginning of the test suite. Its purpose is to compile the source code to run the tests.

The module of a distributed system calculates the distance *d* between two points on the Earth's surface using the following form of the Haversine formula [14 pp. 163–164]

The Haversine formula is used in the program in symbolic form. Alternatively, the program could use the Vincenty formula, which ensures better accuracy. Haversine, on the other hand, is simpler to calculate.

$$d = 2 \cdot r \cdot \sin^{-1}\left(\sqrt{\sin^2\left(\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)\right)}\right), r = 6371\ \text{km} \qquad (8)$$

The program code for calculating the distance between two points is written based on the source [14 p. 164].

```python
from math import radians, sqrt, sin, cos, asin

def haversine(lat1,lon1,lat2,lon2):
    lat1,lon1,lat2,lon2=map(radians,[lat1,lon1,lat2,lon2])
    dlat=lat2-lat1
    dlon=lon2-lon1

    # Haversine formula
    R=6371
    a=pow(sin(dlat/2),2)+pow(sin(dlon/2),2)*cos(lat1)*cos(lat2)
    distance = 2*R*asin(sqrt(a))

    return distance
```

Program 3. *Haversine formula [14 p. 164]*

OpenDDS uses a publish-subscribe template to share data. The publisher sends data to a predefined topic, and the subscriber reads the data from the same topic. The topic is a specific type of data that can be shared between applications. Topic types are defined in the Interface Definition Language file.

```
module LocationData {

  @topic
  struct distance {
    string locationName1;
    string locationName2;
    long lon1;
    long lat1;
    long lon2;
    long lat2;
  };
};
```

Program 4. *The data structure used for data transfer*

‘The location.idl file specifies the distance content. The data structure includes two variables for naming coordinate points and four integer variables.

4.4 Constrains of implemented system

In the work, the test automation solution implemented for the demo system is not so advanced that it would be able to capture the types of errors in the distributed system. It is possible to simulate a distributed system with a single processor, but in this case, not all types of errors will appear. One computer is used in the examination arrangement of this thesis to keep the thesis within realistic dimensions for achieving the goal of answering the main question of this research.

5 TEST STRATEGY AND PLAN FOR SYSTEM UNDER TEST

Before testing the software, a testing strategy that indicates the general requirements for the tests is planned. The strategy aims to briefly describe in general how the risks caused by possible defects in the product will be prevented. The strategy can be a formal document or a free-form output that develops during the software lifecycle. However, the review level of the strategy is limited to suit its purpose. Commonly known approaches include analytical, model-based, consultative, methodological, heuristic, standard-compliant and regression-averse. Strategies can also be combinations of the above strategies. [4 p.88–91]

The choice of testing strategy is influenced by various factors, such as available resources, requirements set for the project, the method used in software development and the life cycle of the software product. In this work, testing is approached from the perspective of quality control. As a result, module-level validation tests will be conducted using a hybrid strategy that is partly analytical, and model-based, but also heuristic.

It is possible to choose the type of tests to be carried out based on various factors. Significant attributes for decision-making include the goals set for testing, the software to be tested, the method of software development, financial constraints, as well as other costs.

5.1 Black Box approach

One method for software testing is the black box testing. The methodology focuses on examining the performance of a system or a component solely based on the results they produce with model input. The method is also called as functional testing [23 p. 119]. Functional testing focuses on designing tests that are likely to cause deviations. Valid or invalid values can both be used as a sample input. The procedure aims to cover all functional features of the software. Numerous tests are required to test functional features. [27 pp.37–38]

However, this work focuses on testing a subset of a distributed system in terms of functional properties without knowing the internal implementation. In this case, the tests shall be feasible without internal system documentation.

5.2 Acceptance testing

When a product is completed or its component is produced, they are presented to the customer. After this presentation, the customer carries out acceptance tests based on the requirement specification and the customer's own needs. Acceptance tests can either be carefully planned or they can be more informal. Testing takes from weeks to months, depending on the scope of the object being tested. The software provider will fix any defects found and implement the necessary changes. Finally, the working software is delivered to the customer. [26 p.22] Based on the chosen testing strategy and the above factors, acceptance tests are selected as the type of tests.

5.3 Module-level testing

The module level is the testing level after unit tests. Within this framework, freely selectable sub-entities can be tested. However, when testing at the module level, the aim is not to complete the program, but rather to focus on an important entity. Such entities include, for example, subprograms, procedures, and classes.

By testing the module level of the software, it is possible to test several modules simultaneously and in parallel. Module tests should be carefully planned to maximise their additional benefits. When building a module test, both the specifications defined for the module and the implemented source code must be considered. In the first phase, the White Box method can be used to test the logic of the module and the Black Box method can be used to examine the accuracy of the module's specifications.

5.4 Automated tests

The suitability of test automation for different testing levels depends on the used software development life cycle model. For example, distributed systems are modular, which means that the implementation of a test automation solution can be started at an earlier stage. The goal of testing also affects how well test automation is suitable for utilization.

The purpose of automated testing is to reduce the amount of manual work and to ensure the comparability of test deliverables. However, it does not make sense to carry out all testing automatically for varying reasons. Automated tests are often time-consuming and require a lot of resources to maintain. For this reason, this work focuses on testing the functionalities of the implemented distributed system component with a limited number of model inputs. (In most cases, automated testing can also be carried out manually)

5.5 Bernoulli trial

Data communication disruptions and load issues are stochastic in nature. Finding them by testing is usually very difficult. With a repeat test, it is possible to bracket the problem areas in question.

An acceptance test is a test performed on the system being tested with two outcomes. The test result is either accepted or rejected. When repeating a series of tests *n* times, the probability that the test sequence will pass flawlessly can be determined. To increase the level of confidence in the test results, acceptance testing may be viewed as a repeat test. In this case, a confidence interval can be calculated, i.e., it can be shown that the system works.

Acceptance tests should be treated as repeat tests when it is known that the system or application contains randomness or is sensitive to interference. The success of one test run may not provide a sufficiently reliable picture of the operation of the software. Data transfer over wireless transmission channels can be very prone to interference. A repeat test is suitable for examining acceptance testing when the system contains randomness, as a test can only have two outcomes and a pre-set number of test sequences are performed.

5.6 Tests

A testing plan is a detailed document that outlines the testing strategy, goals, schedule, estimates, compilation of results, and resources reserved for testing. The test plan aims to determine the necessary measures to verify quality. In the experimental part of this work, the functionalities of a simple distributed system module are tested. The testing of a module is approached by looking at it as a black box, in which case its internal implementation is unknown. Instead, the module review focuses on system response values.

Testing of the subset of the distributed system carried out in this work is done according to the quality management process. Software development follows the V-model. The tests carried out on the system are validation tests and thus they are placed on the right side of the V-model used in software development. The tests have been implemented at the module level of the system and they follow the black box testing method. Implemented automated tests are acceptance tests on the functional features of a distributed system.

There are different ways to determine test coverage. This thesis examines the module-level functional properties of a distributed system in a situation where internal implementation is unknown.

The goal of the test collection is to show that there are no systematic failures in the functional characteristics. In other words, the purpose is to show that test failure is rare. [2 p. 168]

The test strategy chosen earlier for this work's test implementation has been chosen to suit a smaller SUT. Normally, in the case of extensive software testing a coverage procedure showcased in chapter 3 page 19 would be used. In this thesis, formulas 1 and 2, see pages 19 and 20, are not used as they do not provide any additional value nor efficiency. However, testing the functional characteristics of the system is implemented in this work. At the module level, sufficient coverage of functional feature testing is achieved with a few correct use cases and a few borderline cases.

Some of the test data has been collected informally by extracting the coordinates of random cities. Another part of the test data is the extreme values of the input. The test data of the tests carried out in the work has been collected by extracting the coordinates of random cities from Google Maps. The first five tests are valid use cases. The data used by the latter tests are borderline cases.

Table 6. Test data converted to integers.

locationName1	lat1	lon1	locationName2	lat2	lon2
Warsaw	52227	21007	Kyiv	50447	30529
Tokyo	35620	139756	Sydney	-33828	151223
Berlin	52520	13411	Rome	41823	12746
London	51443	-0130	Amsterdam	52552	5012
Stockholm	59324	18020	Washington	39337	-76485
P1	0000	0000	P2	0000	0000
P1	45000	45000	P2	45000	45000
P1	0000	0000	P2	90000	0000
P1	0000	0000	P2	0000	180000
P1	0000	0000	P2	90000	180000

The table 6 has the test data scaled up by a thousand so that the data can be transferred as integers instead of floating-point numbers.

5.7 Designing a hypothesis test

The repeat test is used to estimate the error probability *p,* which in this case is assumed to be zero. Hypothesis testing is used to check the reliability of the estimate. To determine the parameters, test suites of different magnitudes and different confidence intervals were considered. The number of repetitions of the repeat experiment was estimated by counting a few results with assumed values.

Table 7. Determination of repeat test parameters

Number of repetitions	Confidence interval	Probability	Result
100	0.01	0	0
100	0.01	0.001	9.99*10^-296
100	0.01	0.01	9*10^-197
100	0.01	0.1	9*10^-98
100	0.05	0	0
100	0.05	0.001	7.49*10^-278
100	0.05	0.01	7.16*10^-183
100	0.05	0.1	4.45*10^-88
1000	0.01	0	0
1000	0.01	0.001	2.88*10^-159
1000	0.01	0.01	8.27*10^-64
1000	0.01	0.1	0.04
1000	0.05	0	0
1000	0.05	0.001	3.89*10^-151
1000	0.05	0.01	1.08*10^-59
1000	0.05	0.1	0.04
10000	0.01	0	0
10000	0.01	0.001	3.28*10^-62
10000	0.01	0.01	0.04
10000	0.01	0.1	5.94*10^-313
10000	0.05	0	0
10000	0.05	0.001	3.07*10^-58
10000	0.05	0.01	0.04
10000	0.05	0.1	3.66*10^-317

Table 7 shows that the results do not differ significantly between one hundred and a thousand repetitions when the confidence interval is 0.01 or 0.05. Based on this, the number of repetitions *n* can be selected as 100.

The results of the tests are interpreted as binary, i.e., the test performance either succeeds or is rejected. Based on the calculations made 100 repetitions were decided, see Table 7. The test suites and the tests they contain are independent of each other. Based on the result of the repeat test, the probability that the tests will fail is determined. A 99% confidence interval is required for the test result.

Error probabilities have been defined for the data communication channels and a target value has been designed for load capacity. In this work, the target values for the error probabilities mentioned are zero in both cases, i.e., neither error would occur. The purpose of the repeat test is to estimate these values, the correctness of which is checked by a hypothesis test. The repeat test estimates the probability of failure, and hypothesis testing indicates the reliability of the estimate at the set confidence level.

5.8 Acceptance Criteria

A subset of an implemented distributed system can be found to work in terms of functional properties if no significant deviations are detected in the results produced by the repeat test and the result of the repeat test remains within the confidence interval. For a single test, this means that the result must be similar to the model answer. If the test fails, the cause of the failure must be inspected and specified. The premise for testing is good, as the actual level of transmission errors and load is known. This will make it easier to fix possible defects.

The result of a test performance is accepted when it is aligned with a predefined outcome. This means that no errors must occur during the test performance for the test result to be accepted.

6 RESULTS AND DISCUSSION

In this work, the testing of distributed systems using test automation was studied. The automated tests carried out focused on the module-level functional features of the distributed system. The thesis answers the question, **how the module-level functional properties of distributed systems can be automatically tested when the module can only be studied as a black box.**

In the repeat test, a collection of acceptance tests was performed on the part of the distributed system implemented in the work. The tests were carried out using the Robot Framework test automation tool, which generated a report and a test log file for each repetition.

6.1 Test results

A repeat test was carried out in which a test suite of ten tests were repeated for 100 times. In total, tests were carried out a total of 1000 times. Half of the tests used valid test data and the other half contained limit values. During the execution of the tests, one anomaly was detected in the test log files. The test automation software produced a report for each repeat of the test series. The produced test documents show a lot of different information about the tests. Figure 12 is one of 100 module test reports. The report includes the time of the test run, the start and end time of the tests, the time taken for the test run, the tests according to their identifiers, and a table showing the total number of successful tests.

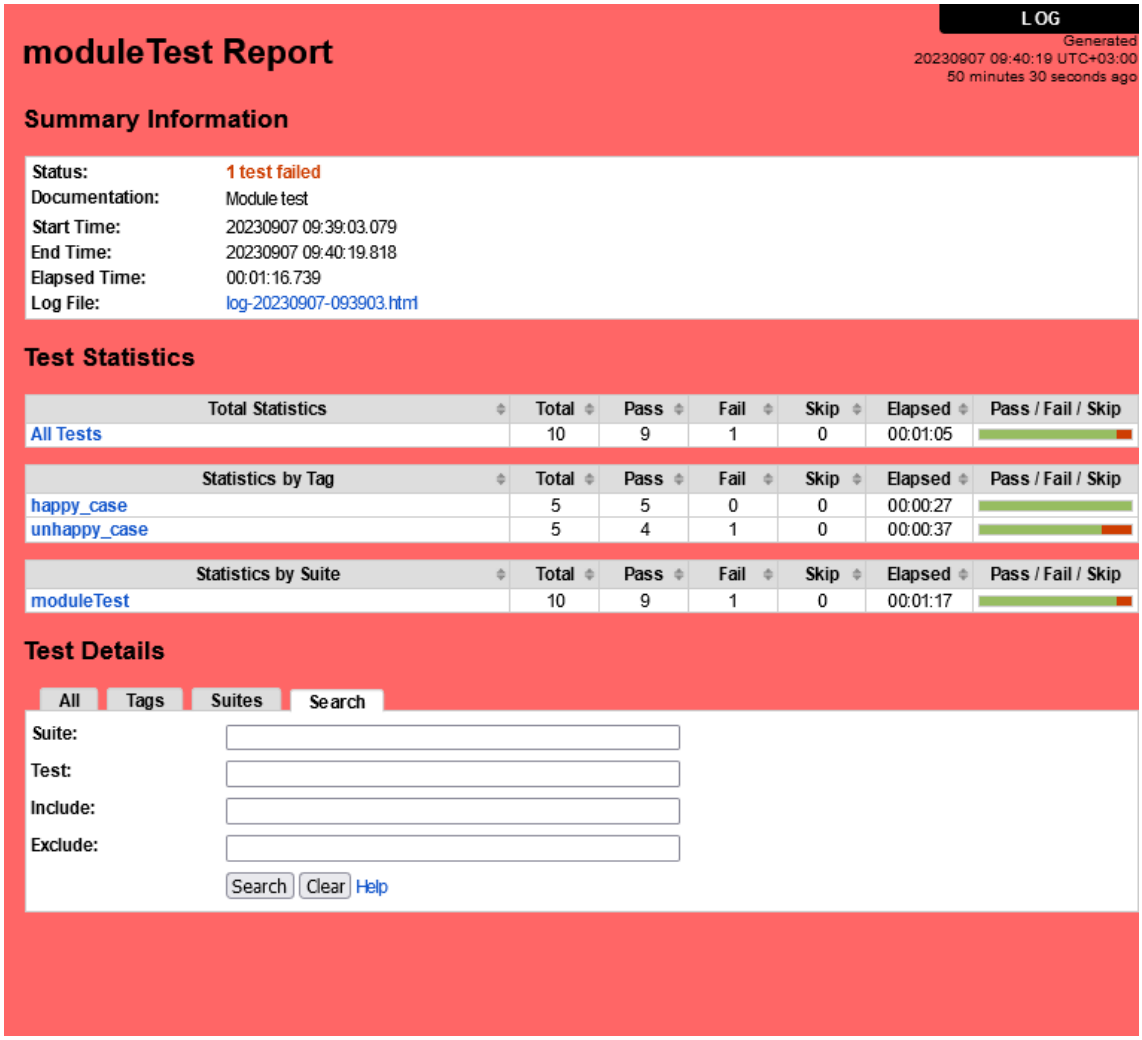
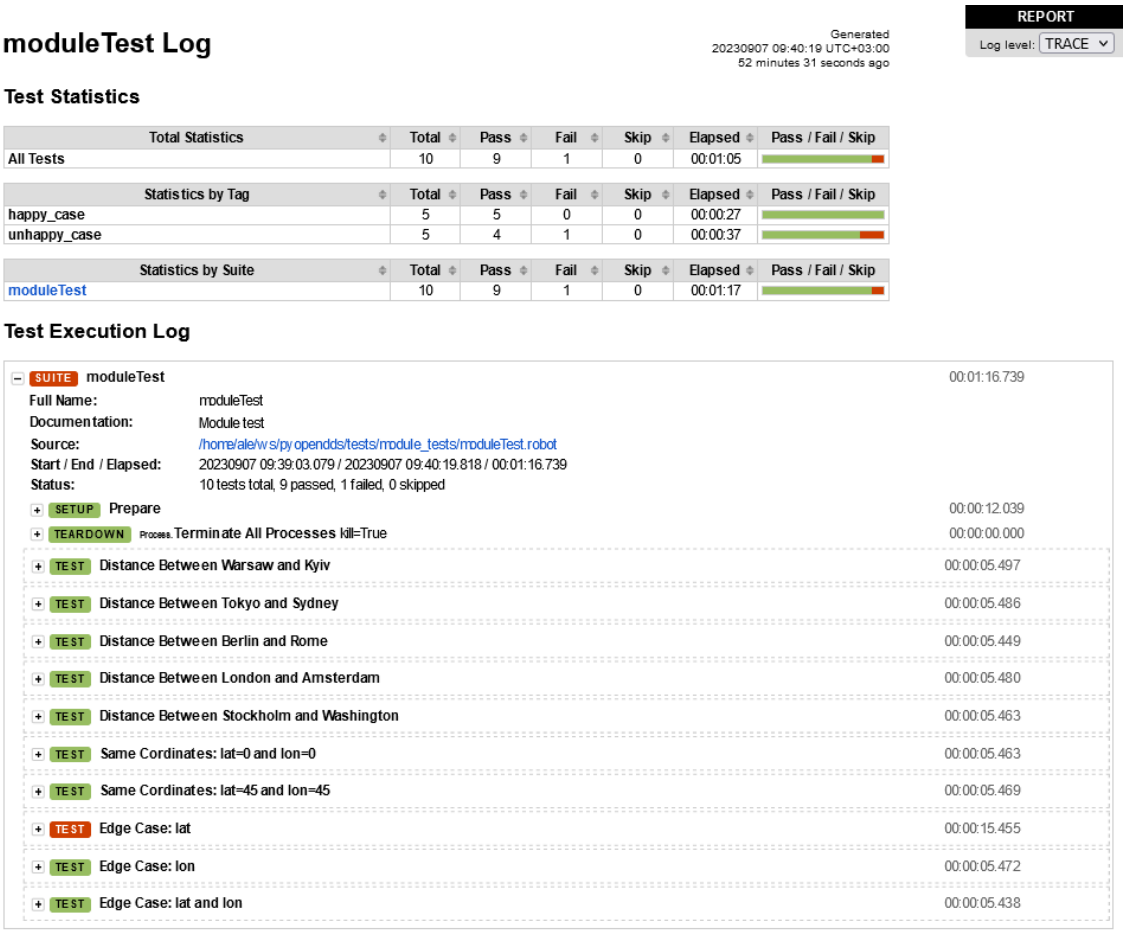


Figure 12. *Test report*

Picture 12 shows a test report showing that nine out of ten tests passed in this session. You can also see in Picture 12 that it took about a minute to complete one collection of tests. Based on the test results, the quality of the test data has no bearing on the test execution times. Tests using a valid test feed, i.e., tests tagged happy_case, performed flawlessly. Tests that use limit values as input, i.e., tests marked with unhapy_case tag, work except in one case.

In addition to the module-level test report, the test automation tool Robot Framework also generates a test log file containing the intermediate steps performed during the test run and the data used. Picture 13 shows the series of tests performed and the time taken to complete each test.



moduleTest Log

Generated
20230907 09:40:19 UTC+03:00
52 minutes 31 seconds ago

REPORT
Log level: TRACE

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	10	9	1	0	00:01:05	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
happy_case	5	5	0	0	00:00:27	
unhappy_case	5	4	1	0	00:00:37	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
moduleTest	10	9	1	0	00:01:17	

Test Execution Log

SUITE moduleTest 00:01:16.739
Full Name: moduleTest
Documentation: Module test
Source: /home/ale/ws/pyopendds/tests/module_tests/moduleTest.robot
Start / End / Elapsed: 20230907 09:39:03.079 / 20230907 09:40:19.818 / 00:01:16.739
Status: 10 tests total, 9 passed, 1 failed, 0 skipped

SETUP Prepare 00:00:12.039
TEARDOWN Process.Terminate All Processes kill=True 00:00:00.000
TEST Distance Between Warsaw and Kyiv 00:00:05.497
TEST Distance Between Tokyo and Sydney 00:00:05.486
TEST Distance Between Berlin and Rome 00:00:05.449
TEST Distance Between London and Amsterdam 00:00:05.480
TEST Distance Between Stockholm and Washington 00:00:05.463
TEST Same Cordinates: lat=0 and lon=0 00:00:05.463
TEST Same Cordinates: lat=45 and lon=45 00:00:05.469
TEST Edge Case: lat 00:00:15.455
TEST Edge Case: lon 00:00:05.472
TEST Edge Case: lat and lon 00:00:05.438

Figure 13. *Test log file*

Picture 13 shows the test log, which contains information about the tests performed, the outcome of the tests and the duration of the tests. The execution times for individual tests remain the same.

Figure 14. *Test run error*

Picture 14 shows an error that occurred during the execution of one of the tests. Based on the log, the process performed by the robot framework has returned a non-zero number. In addition, a failed test performance has lasted three times as long as other test cases.

The test suite is considered failed, when at least one of the tests performed fails. Based on the test results, it is established that the probability of such test suite with an error is p_0. Hypothesis testing is carried out when one of the tests is a test suite has failed. Counterhypothesis is a situation in which at least one test suite fails.

$$\begin{aligned} &H_0: All\ test\ series\ are\ successful \\ &H_A: At\ least\ one\ test\ series\ fails \end{aligned} \tag{9}$$

The null hypothesis is formed in a scenario where this is a one-sided case as shown on page 21.

$$\begin{aligned} &H_0:\ p \leq \frac{1}{100} \\ &H_1: p > \frac{1}{100} \end{aligned} \tag{10}$$

The parameters of the repeat test were: $p_0 = \frac{1}{100},\ n = 100,\ q_0 = 1 - \frac{1}{100} = \frac{99}{100}$

Let's look at the situation at different levels of confidence, determined using formula 11 below. Choose a 95% or 99% percent confidence level.

$$\alpha = \frac{100 - 95}{100} = 0.05, \qquad \alpha = \frac{100 - 99}{100} = 0.01 \tag{11}$$

The formula (7) consists of the sum of the expected value and the uncertainty due to the dispersion. Compare the number of errors detected with *n* with the calculated value *t*. The null hypothesis is refuted when the number of errors detected is greater than the calculated value *t*.

Let's explore in both cases, using confidence level 95% in formula 12 and confidence level 99% in formula 13.

$$t = 100 \cdot \frac{1}{100} + 1.64 \cdot \sqrt{100 \cdot \frac{1}{100} \cdot \frac{99}{100}} = 2.63 \tag{12}$$

$$t = 100 \cdot \frac{1}{100} + 2.33 \cdot \sqrt{100 \cdot \frac{1}{100} \cdot \frac{99}{100}} = 3.32 \tag{13}$$

$$1 > 2.63 \tag{14}$$

In the formula 14, the anomalities discovered in the test suite are compared to the calculated value *t.* When *n1* is less than the result *t,* as in the formula 12, the null hypothesis is retained. Even when using the confidence level 95% for *t*, we can see that 1 > 2.63, which retains the null hypothesis.

The value of probability *p* is usually small, for example, 0.0001. For comparison, formula 15:

$$t = 100 \cdot \frac{1}{10000} + 1.64 \cdot \sqrt{100 \cdot \frac{1}{10000} \cdot \left(1 - \frac{1}{10000}\right)} = 0.17 \tag{15}$$

When the confidence level is increased to 99% and the sample is grown to 10 000, more anomalities can be allowed while the hypothesis test remains true.

$$t = 100000 \cdot \frac{1}{10000} + 2.33 \cdot \sqrt{100000 \cdot \frac{1}{10000} \cdot \left(1 - \frac{1}{10000}\right)} = 17.37 \tag{16}$$

6.2 Discussion

In this work, a type of test automation solution was developed that can validate the module-level functional properties of a distributed system that follows a layered architecture.

In addition, test coverage was determined for functional characteristics. The reliability of the test results was verified by performing a replicate hypothesis test on a series of tests performed on the system.

The null hypothesis set for the repeat test is not refuted because the result of the repeat test remains within the confidence interval. A high confidence requirement of 99% or 95% per cent is strict. It requires that there are no more than three or two errors in the execution of the test series with the number of repetitions used. As the null hypothesis is valid, it can be stated that the SUT is functioning properly within this context.

In the repetition test, a strict confidence interval was set to show that functional errors in the distributed system are at least not common. The most significant factor limiting further development is the incompleteness of the Pyopendds Framework. With the method implemented in the work, it is possible to test a distributed system that follows a layered architecture.

Based on the results, it can be stated that it is possible to solve the research problem with the solution implemented in the work. In other words, it is possible to build a test automation solution that can be used to examine the frequency of deviations and show the rarity of deviations. Based on the results of the repeat test, it can be concluded that there were few abnormalities in the set test coverage.

Test automation is one tool for achieving the goal set for testing. With the implemented test automation solution, it is possible to verify the functionality of the distributed system reliably, quickly, and accurately, which improves the quality of testing. In addition, the solution is expandable and can be used to test the common functionality of several modules.

This work has been a constructive research work. The existing theory has been studied. A demo system has been implemented and tested. The research problem has been answered and the test automation has been detected to be a useful tool.

7 CONCLUSIONS

With the test automation solution implemented in this work, it is possible to validate the module level functional properties in a distributed system that follows a layered architecture. Another aim of this work was to determine the test coverage of functional characteristics and try to ensure the reliability of the results. As the test subject's system is simple, it makes sense to focus on the reliability of the test results. The level of reliability of the test results was determined by conducting a hypothesis test.

Testing of functional properties automatically improves the reliability of the test results because of traceability, repeatability, and the number of repetitions. The solution succeeded in detecting a rarely repeated error that would otherwise have remained a hidden defect.

The test automation solution implemented works and it can be utilised as a method for validating the functional properties of distributed systems. Although the method was used in the work to demonstrate the rarity of defects, the solution can also be used to search for defects that are difficult to detect or rare.

7.1 Limitations of automated tests

With the developed method, it is possible to verify the properties of a distributed system to a limited extent. A limited number of tests cannot test everything. The method can be used to examine data transfer and load problems that are otherwise difficult to test.

The results obtained in this work may not be universally applicable to all possible distributed systems, even if they follow a layered architecture. As a test automation solution, the Robot Framework is highly expandable, but the test automation tool may limit testing possibilities in some cases. The currently implemented test automation solution does not classify defects in test execution according to their types. The development of test automation consumes time and resources. Information systems are assumed to be deterministic in principle, and therefore the method developed in the work is not useful in all types of cases. The usefulness of test automation largely depends on the quality requirements set for the application, but also on the goals set for testing.

Building test automation systems for module-level testing may not be easy or even possible, as the software product is not designed for this type of use. For example, cases where there is no actual clear interface. Automated tests can be implemented for different systems in the same way as manual tests. When designing tests, the influence of

various factors should be determined. These include, for example, the system being tested, the type of tests and the desired test coverage, available test automation tools, human resources and other factors related to organizations or other stakeholders. As a rule, the implementation of automated tests requires knowledge of the system being tested at least at a fundamental level and a significant number of resources, depending on the desired test coverage.

7.2 Alternative solutions and further development

The benefits of the system implemented in the work include the scalability and flexibility of its development work. As mentioned earlier, testing similar to automated tests can also be performed manually. On the other hand, the module level is not necessarily tested, but it can be stated that sufficient test coverage can be achieved, for example, through unit tests and system tests.

There are also many ways to implement test automation solutions. Some implementations may be more generalist and other ones more designed to fit a certain purpose. The test automation frameworks can also either be open-source or closed-source software.

In this work, an open-source software was used in test framework and in connection framework OpenDDS due to the extensibility and plasticity. Typically, open-source software used is documented minimally and requires some manual configurations. This also applied to the software used in the test solution here. Buying a closed-source software could have provided better documentation which could have expedited this project. Using both open- and closed-source software could have also provided an interesting comparison point to this works results.

Within the framework of this work, a simple test automation solution for testing module-level components of a distributed system was implemented. As a result of the scope of work, the functional testing of a simple demo system was refrained from. The test automation solution could be used to test a larger or more complex distributed system. On the other hand, different testing methods could be tried, such as performance testing.

REFERENCES

[1] A. Mathur, Foundations of software testing, Pearson, 2nd edition, 2013.

[2] A. Mili, Software Testing, John Wiley & Sons, Incorporated, 2015

[3] A. Puder, KR Distributed Systems Architecture: A Middleware Approach, San Francisco: Elsevier Science, 2011.

[4] AMJ. Hass, Guide to advanced software testing, Boston: Artech House, 2008.

[5] AS. Mahfuz, Software Quality Assurance: Integrating Testing, Security, and Audit. Boca Raton, Auerbach Publications, 2016.

[6] B. Hambling, P. Morgan, A. Samaroo, G. Thompson, P. Williams, Software Testing - An ISTQB-BCS Certified Tester Foundation Guide, Swindon: BCS The Chartered Institute for IT, 3rd edition, 2015.

[7] B. Jose, Test Automation. BCS, The Chartered Institute for IT, 2021.

[8] BR. Mehta, YJ. Reddy, Industrial Process Automation Systems - Design and Implementation, San Diego: Elsevier, 2014.

[9] CY. Laporte, A. April, Software Quality Assurance, Piscataway: Wiley, 1st edition, 2017.

[10] D. Galin. Software quality: concepts and practice, Hoboken, New Jersey: John Wiley &; Sons, 1st edition, 2017.

[11] D. Lindsay, SS. Gill, D. Smirnova, P. Garraghan, The evolution of distributed computing systems: from fundamental to new frontiers, Computing, Vol. 103 (8), 2021, pp.1859–1878.

[12] F. Buschmann, K. Henney, DC. Schmidt, Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Newark Wiley,1. Aufl. Vol. 4, 2007.

[13] J. Gao, H. Tsai, Y. Wu, Testing and Quality Assurance for Component-Based Software. Norwood: Artech House, 2003.

[14] J. Lawhead, Learning Geospatial Analysis with Python – An Effective Guide to Geographic Information System and Remote Sensing Analysis Using Python 3, Packt Publishing, 2nd edition, 2015.

[15] JH. van Schuppen, O. Boutin, PL. Kempker, J. Komenda, T. Masopust, N. Pambakian, et al., Control of Distributed Systems: Tutorial and Overview, European journal of control, Vol.17(5-6), 2011, pp. 579–602.

[16] KD. Pandl, S. Thiebes, M. Schmidt-Kraepelin, A. Sunyaev, On the Convergence of Artificial Intelligence and Distributed Ledger Technology: A Scoping Review and Future Research Agenda, IEEE Access, Vol. 8, 2020, pp. 57075–57095.

[17] KR. Koch, Parameter Estimation and Hypothesis Testing in Linear Models, Springer Berlin Heidelberg, 2nd edition, 1999.

[18] M. Macero, Testing the Distributed System, In: Learn Microservices with Spring Boot. Berkeley, CA: Apress, 2017. pp. 267–314.

[19] MJ. Escalona, F. Domínguez Mayo, TA. Majchrzak, V. Monfort, Hybrid Is Better: Why and How Test Coverage and Software Reliability Can Benefit Each Other, In: Lecture Notes in Business Information Processing, Switzerland, Springer International Publishing AG, 2020, pp. 25–38.

[20] PC. Jorgensen, Epilogue: Software Testing Excellence. In: Software Testing, CRC Press; 4th edition, 2014 pp. 433–438.

[21] PI. Good, Introduction to Statistics Through Resampling Methods and R, New York: Wiley, 2. Aufl., 2013, pp. 25-41.

[22] S. Ghos, Distributed systems: an algorithmic approach, Boca Raton, FL: Chapman and Hall/CRC, an imprint of Taylor and Francis, 2nd edition, 2014.

[23] TL. Thai, Learning DCOM, Sebastopol: O'Reilly Media Incorporated, 1999.

[24] W. TZhao, Building Dependable Distributed Systems, Somerset: Wiley, 1st edition, 2014.

[25] Y. Dodge, Binomial Test. In: The Concise Encyclopedia of Statistics, New York, NY: Springer New York, 2008, pp. 47–49.

[26] Y. Singh, Software Testing, Cambridge, Cambridge University Press, 2011.

[27] Architecture Styles in Distributed Systems, Geeks for geeks, 2023 [cited 2023 March 8], Available from: https://www.geeksforgeeks.org/architecture-styles-in-distributed-systems/

[28] Monolithic vs Microservices architecture, Geeks for geeks, 2022 [cited 2023 July 14], Available from: https://www.geeksforgeeks.org/monolithic-vs-microservices-architecture/

[29] What is P2P (Peer-to-Peer Process)?, Geeks for geeks, 2023 [cited 2023 July 14], Available from:https://www.geeksforgeeks.org/what-is-p2p-peer-to-peer-process/

[30] OpenDDS/Pyopendds, Github, 2022 [cited 2023 March 10], Available from: https://www.github.com/OpenDDS/pyopendds

[31] Goals and Purpose, Isocpp, [cited 2023 June 4], Available from: https://www.isocpp.org/about

[32] OpenDDS, OpenDDS, 2023 [cited 2023 June 4], Available from: https://www.opendds.org/about/

[33] Python – About Python, Python, 2023 [cited 2023 June 4], Available from: https://www.python.org/

[34] Robot Framewok – Introduction, Robot Framework, [cited 2023 June 4], Available from: https://www.robotframework.org/

[35] ACE Product Information, Object computing, [cited 2023 June 4], Available from: https://www.theaceorb.com/about/aboutace.html

[36] Architecture in Distributed Systems, S. Kumar, Tutorialspoint, 2023 [cited 2023 March 8], Available from: https://www.tutorialspoint.com/architecture-styles-in-distributed-systems

[37] Distributed Architecture, Tutorialspoint, [cited 2023 July 14], Available from: https://www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm

[38] The Story of Ubuntu, Ubuntu, 2023 [cited 2023 June 10], Available from: https://www.ubuntu.com/about

[39] Beginners guide, Python, 2023 [cited 2023 June 10], Available from: https://www.wiki.python.org/moin/BeginnersGuide/Overview