

|  |  |  |
| --- | --- | --- |
| **Module Code** | **:** | CT024-3-3-DCOMS – Distributed Computer Systems |
| **Intake Code** | **:** | APD3F2411CS |
| **Lecturer Name** | **:** | DR. KURUVIKULAM CHANDRASEKARAN ARUN |
| **Due Date** | **:** | Week 10 |
| **Group No.** | **:** |  |

|  |  |
| --- | --- |
| **Student ID** | **Student Name** |
| TP069502 | ALI AHMED ABOUELSEOUD MOUSTAFA TAHA |
|  |  |
|  |  |
|  |  |

**Table of Contents**

[1.0 Background 3](#_Toc188404213)

[1.1 Abstract 3](#_Toc188404214)

[1.2 Introduction 3](#_Toc188404215)

[1.3 Problem Statement 3](#_Toc188404216)

[1.4 Requirements 3](#_Toc188404217)

[2.0 Gantt Chart 4](#_Toc188404218)

[3.0 Use Case Diagram 5](#_Toc188404219)

[4.0 Research and Evaluation 6](#_Toc188404220)

[4.1 Remote Method Invocation (RMI) 6](#_Toc188404221)

[4.2 gRPC 6](#_Toc188404222)

[4.3 Simple Object Access Protocol (SOAP) 6](#_Toc188404223)

[4.4 Common Object Request Broker Architecture (CORBA) 6](#_Toc188404224)

[5.0 Role of Serialization, Object-Oriented Programming (OOP), and multi-threading 7](#_Toc188404225)

[5.1 Serialization 7](#_Toc188404226)

[5.2 Object-Oriented Programming (OOP) 10](#_Toc188404227)

[5.3 Multi-threading 10](#_Toc188404228)

[6.0 Implementation of Distributed Application 11](#_Toc188404229)

[7.0 Protocols 12](#_Toc188404230)

[8.0 Testing 13](#_Toc188404231)

[9.0 Conclusion 14](#_Toc188404232)

[10.0 Future Enhancements 15](#_Toc188404233)

[References 16](#_Toc188404234)

[Appendix 17](#_Toc188404235)

# Background

## 1.1 Abstract

## 1.2 Introduction

## 1.3 Problem Statement

## 1.4 Requirements

# 2.0 Gantt Chart

# 3.0 Use Case Diagram

# 4.0 Research and Evaluation

## 4.1 Remote Method Invocation (RMI)

## 4.2 gRPC

## 4.3 Simple Object Access Protocol (SOAP)

## 4.4 Common Object Request Broker Architecture (CORBA)

# 5.0 Role of Serialization, Object-Oriented Programming (OOP), and multi-threading

## 5.1 Serialization

The process of serialization transforms an object's state into a stream of bytes. This makes sure that data is unreadable to the human eye. The opposite procedure, known as deserialization, uses the byte stream to replicate the real Java object in memory. The object is persistent by this approach (*Serialization and Deserialization in Java with Example*, 2025). For example, we write and Object that contains a Username and Password into a serialization method, which will then be converted into a byte stream that will be saved to a file with the .ser extension. Upon opening the file, the data will be unreadable since its stored in binary. To read the contents inside, we have to perform deserialization. This will convert the byte stream back to an object which can then be read and printed for the users to read the data. Serialization is important when you want to hide sensitive data from being accessed directly through the file.

A screen shot of a computer program

Description automatically generated

*Figure 5.1.1: Employee Login Class*

The EmployeeLogin class represents the employee login credentials. It implements the Serializable interface, allowing its objects to be serialized. The class contains two private fields, employeeIC and password. These fields are the data that we save and later retrieve.

A screenshot of a computer program

Description automatically generated

*Figure 5.1.2: Serialization Class*

This class is responsible for saving and serializing EmployeeLogin objects to a file. It checks if the file already exists and whether it contains data. If the file is new, it creates it and writes the first object with a proper stream header. If the file already exists and has data, it appends the new object without writing an additional header to prevent corruption.

A computer screen shot of a program

Description automatically generated

*Figure 5.1.3: Deserialization Class*

To read the serialized data, we need to convert it back from byte stream to the original object, then we can read the data. This class handles loading the saved EmployeeLogin objects from the file. It reads each object until the end of the file. The objects are added to a list of EmployeeLogin Objects, which is then returned.

A screen shot of a computer program

Description automatically generated

*Figure 5.1.4: Custom Appending Class to handle header issues*

ObjectOutputStream writes a header whenever it's created by default. This causes issues when appending to a file because multiple headers corrupt the stream. To fix this, this class overrides the writeStreamHeader method to skip writing the header when appending. It ensures that only the initial header exists in the file. Meaning, when we are first an object, if the file is empty, we will insert a header. Next time we insert an object, we skip writing a header because there is data that already exists, avoiding corrupted headers exception.

A screen shot of a computer

Description automatically generated

*Figure 5.1.5: Ser File Contents*

As shown in figure 5.1.3, The data is not readable to the human eye, and it is in binary form.

## 5.2 Object-Oriented Programming (OOP)

## 5.3 Multi-threading

# 6.0 Implementation of Distributed Application

# 7.0 Protocols

# 8.0 Testing

# 9.0 Conclusion

# 10.0 Future Enhancements

# References

*Serialization and Deserialization in Java with Example*. (2025, January 4). GeeksForGeeks. <https://www.geeksforgeeks.org/serialization-in-java/>

# Appendix