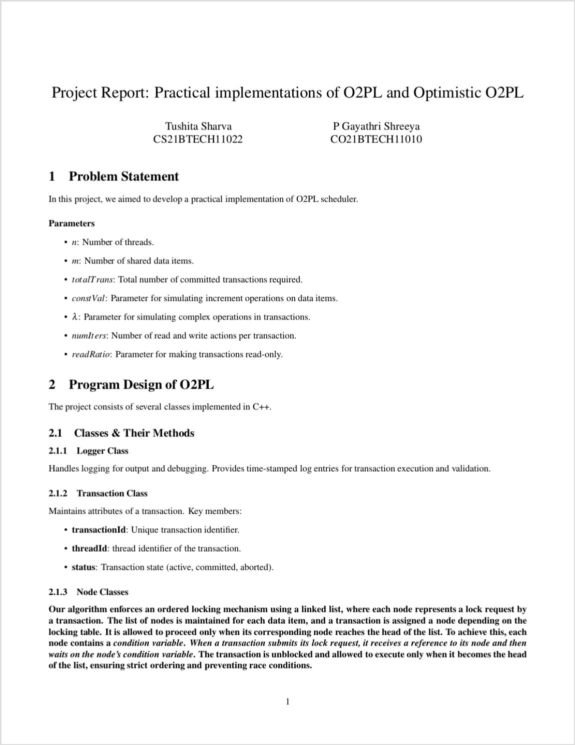
# Digital Receipt

This receipt acknowledges that Turnitin received your paper. Below you will find the receipt information regarding your submission.

The first page of your submissions is displayed below.

| | |
|---|---|
| Submission author: | Tushita Sharva |
| Assignment title: | CCTS Final Project Submission |
| Submission title: | ProjectReport-CS21BTECH11022.pdf |
| File name: | ProjectReport-CS21BTECH11022.pdf |
| File size: | 651.59K |
| Page count: | 9 |
| Word count: | 1,985 |
| Character count: | 10,632 |
| Submission date: | 06-May-2025 01:28PM (UTC+0530) |
| Submission ID: | 2667957140 |



Project Report: Practical implementations of O2PL and Optimistic O2PL

Tushita Sharva
CS21BTECH11022

P Gayathri Shreeya
CO21BTECH11010

## 1 Problem Statement

In this project, we aimed to develop a practical implementation of O2PL scheduler.

**Parameters**

- $n$: Number of threads.
- $m$: Number of shared data items.
- $totalTrans$: Total number of committed transactions required.
- $constVal$: Parameter for simulating increment operations on data items.
- $\lambda$: Parameter for simulating complex operations in transactions.
- $numIters$: Number of read and write actions per transaction.
- $readRatio$: Parameter for making transactions read-only.

## 2 Program Design of O2PL

The project consists of several classes implemented in C++.

### 2.1 Classes & Their Methods

#### 2.1.1 Logger Class

Handles logging for output and debugging. Provides time-stamped log entries for transaction execution and validation.

#### 2.1.2 Transaction Class

Maintains attributes of a transaction. Key members:

- **transactionId**: Unique transaction identifier.
- **threadId**: thread identifier of the transaction.
- **status**: Transaction state (active, committed, aborted).

#### 2.1.3 Node Classes

**Our algorithm enforces an ordered locking mechanism using a linked list, where each node represents a lock request by a transaction. The list of nodes is maintained for each data item, and a transaction is assigned a node depending on the locking table. It is allowed to proceed only when its corresponding node reaches the head of the list. To achieve this, each node contains a *condition variable. When a transaction submits its lock request, it receives a reference to its node and then waits on the node's condition variable*. The transaction is unblocked and allowed to execute only when it becomes the head of the list, ensuring strict ordering and preventing race conditions.**