INFORMATICS INSTITUTE OF TECHNOLOGY

UNIVERSITY OF WESTMINSTER

5COSCO19C

Object Oriented Programming

Coursework report

Real-Time Event Ticketing System with Advanced Producer-Consumer Implementation

Module Leader: Mr. Guhanathan Poravi

Name: Janindu Amaraweera

Tutorial Group: SE-13

UoW Id: w2054022

IIT Id: 20230091

Acknowledgement

I wish to convey my heartfelt appreciation to all who assisted me in the completion of this project. I would like to express my gratitude to Mr. Guhanathan Pooravi, our module leader, for his unwavering assistance and direction during the project. His insightful feedback and recommendations really enhanced the project.

I would want to express my gratitude to Mr. Ammar Raneez for his encouragement and assistance throughout the tutorial sessions. His advice facilitated the project's compliance with industry norms.

I express my gratitude to all individuals who contributed to the project's success, whether directly or indirectly.

Table of Contents

Acknowledgement ii
Introduction 1
Diagrams 2
  Class Diagram 2
  Sequence Diagrams 3
Test Cases 4
Implementation 9
  Configuration Module 9
  Vendor and Customer Logic 10
  Logging and Error Handling 10
User Interface 11
Challenges and Solutions 11
  Concurrency Issues 11
  Deadlocks 11
Conclusion 12
  Future Enhancements 13

Introduction

The Real-Time Event Ticketing System aims to demonstrate the practical use of Object-Oriented Programming (OOP) principles, as well as multi-threading and synchronization concepts. This system replicates a dynamic environment in which event tickets are simultaneously released by vendors and acquired by customers. It seeks to address practical difficulties such as concurrency, data integrity, and resource management within a multi-threaded environment.

The project employs the Producer-Consumer pattern to optimize the management of ticket releases and purchases. Vendors function as producers by contributing tickets to a communal pool, whilst customers serve as consumers by extracting tickets from that pool. By integrating thread-safe operations, the system guarantees seamless functionality devoid of race situations or deadlocks.

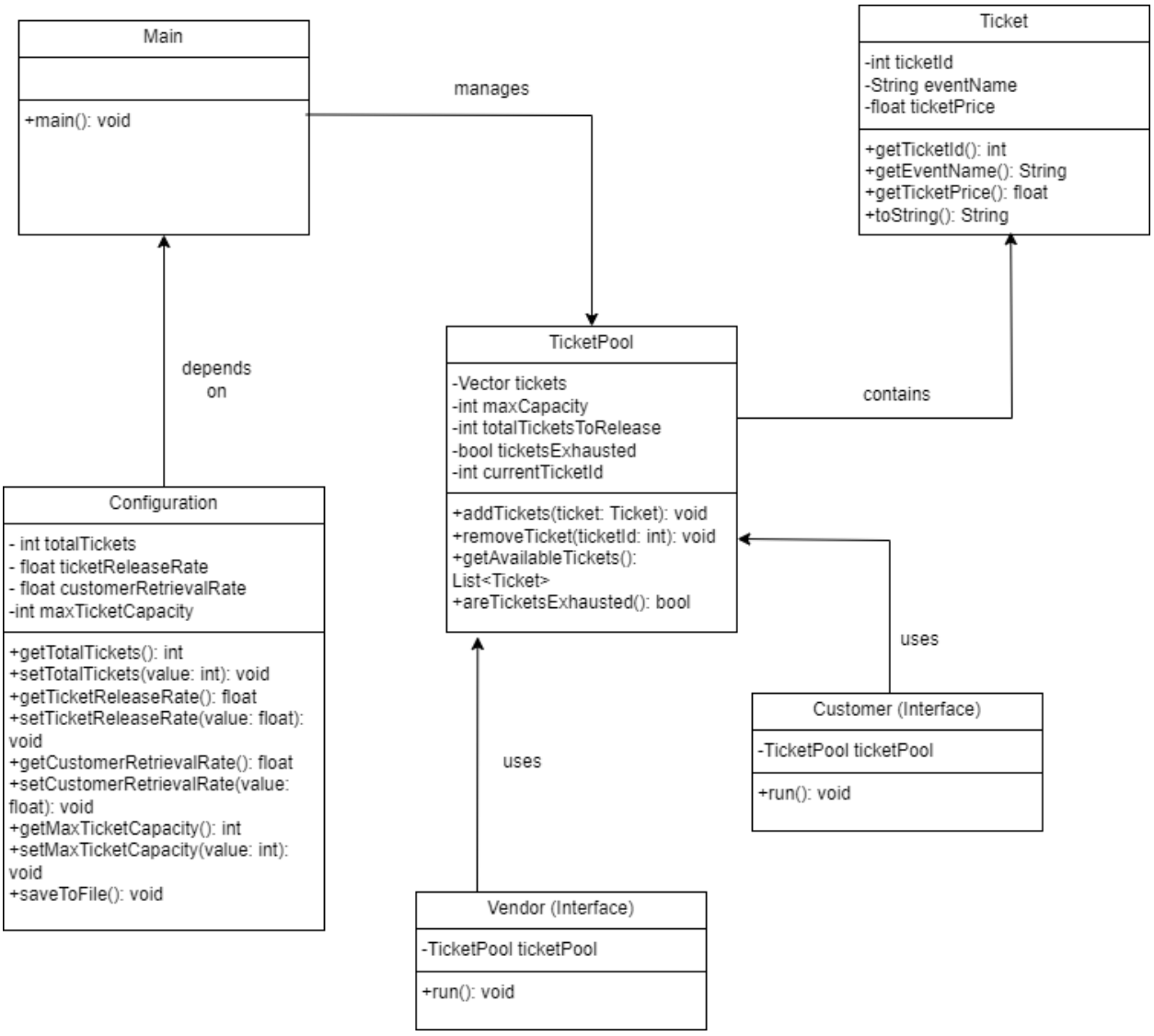
The system is designed with a React interface and a Spring Boot backend to improve usability and scalability. The React frontend offers a dynamic and engaging interface for configuration, system status oversight, and user engagement. The Spring Boot backend oversees the fundamental business logic, encompassing ticket management, multi-threading, and synchronization. This integration guarantees a fluid user experience while upholding strong and efficient backend operations.

This study examines the execution of fundamental elements including configuration management, synchronization, logging, and user interface design. The amalgamation of React with Spring Boot demonstrates the practical use of contemporary web development technologies to construct scalable and maintainable systems.

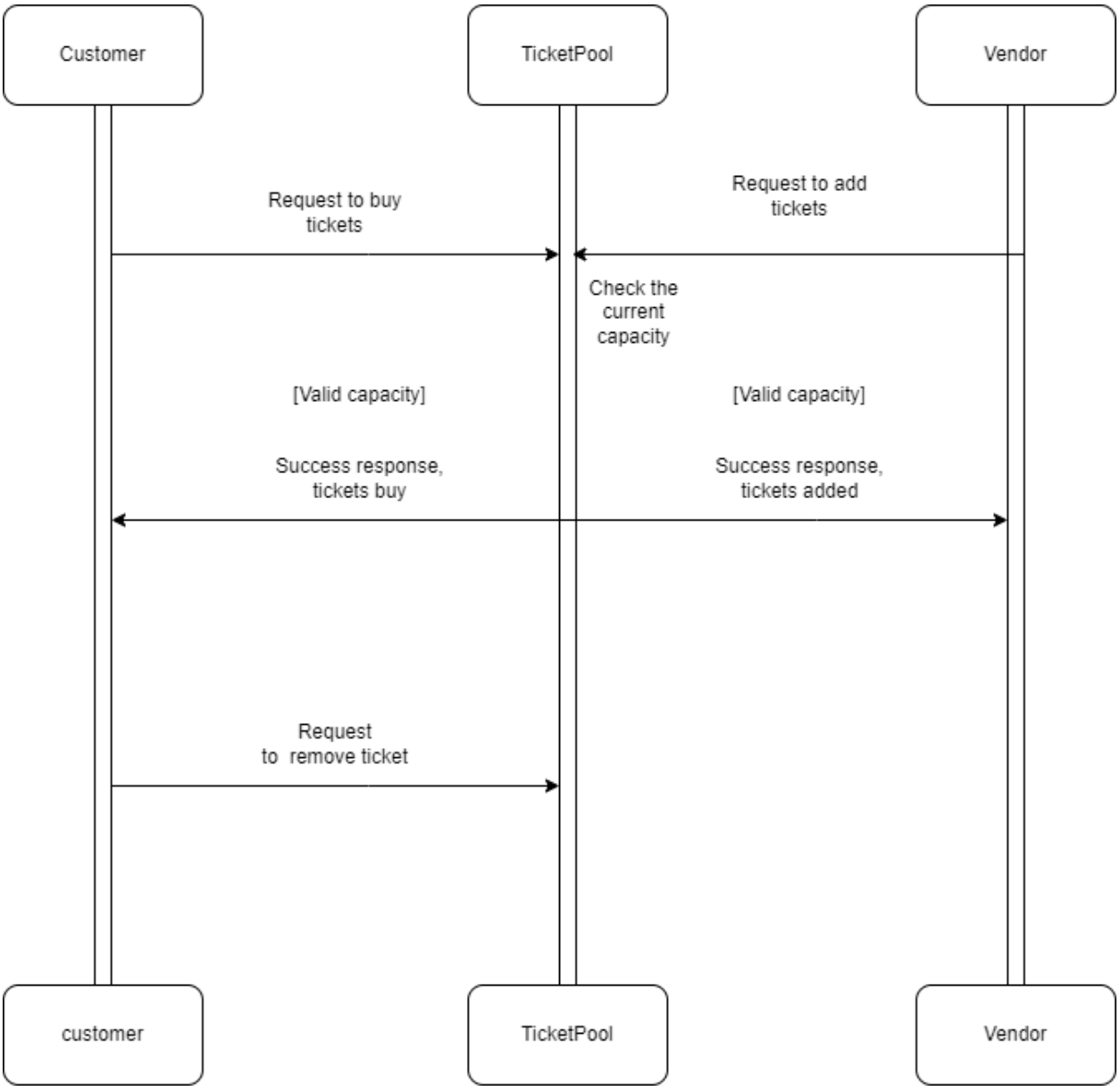
This study delineates the procedures used to build the system, the obstacles encountered during implementation, and the resolutions applied. It offers a detailed elucidation of the system's architecture, functionality, and testing, providing insights into the practical application of programming principles and web technologies in a real-world context.

Diagrams

Class Diagram



Sequence Diagrams



Test Cases

ID	Scenario	Expected Result	Pass/Fail	Issues Found	Resolution
CF1	Submit the form with valid values.	Form submits successfully, system initializes the pool, and a success message is displayed.	Pass	None	Works as expected.
CF2	Submit the form with missing/invalid values.	Form shows an error message, and submission is blocked until all fields are valid.	Pass	None	Validation logic is functional.
CF3	Start processes for vendors and customers	Vendors and customers start working with the ticket pool, and logs show activity.	Pass	None	Thread operations run smoothly.
CF4	Stop running processes.	Processes stop immediately, and a success message is displayed.	Pass	None	Graceful termination works correctly.
CF5	Reset the ticket pool.	The pool resets, logs confirm the action, and the UI updates.	N/A	Reset functionality not implemented	Add a reset method in TicketPool.
CF6	Save a new configuration.	Configuration is saved successfully, and a confirmation message is displayed.	Pass	None	No issues. Configuration saved successfully.

CF7	Clear log data.	Logs are cleared, and a success message confirms the action.	Pass	None	Implement log clearing functionality.
CF8	Load an existing configuration file.	Configuration loads if the file exists, or an error message is displayed if it doesn't.	Pass	None	Loading functionality is effective.
CF9	Exceed the ticket pool capacity.	System pauses ticket addition until space becomes available.	Pass	None	Synchronization ensures pool capacity is respected.
CF10	Try to buy tickets when none are available.	Customers wait, and system resumes when tickets are added.	Pass	None	Customers handled correctly in wait scenarios.
CF11	Monitor ticket availability.	Number of tickets in the pool is displayed every 2 seconds.	Pass	None	Periodic monitoring works well.
CF12	Exit the system during execution.	All processes stop gracefully, and a success message is displayed.	Pass	None	System exits without leaving running threads.
CF13	Start the system with a large configuration.	System initializes correctly and handles high ticket volume without crashing.	Pass	None	No performance issues.
CF14	Enter invalid input for configuration.	System shows an error message and	Pass	None	Input validation works as intended.

		asks for valid input.			
CF15	Simultaneous vendors and customers.	Vendors and customers operate without blocking or errors.	Pass	None	Thread safety ensured by ReentrantLock.
CF16	Save configuration during execution.	Configuration is saved without interrupting running processes.	Pass	None	Configuration saving doesn't disrupt threads.
CF17	Check status before starting the system.	System informs the user that the system hasn't started yet.	Pass	None	Clear and helpful messaging.
CF18	Release tickets beyond the configured limit.	Vendors stop releasing tickets once the maximum is reached.	Pass	None	System respects ticket limit.
CF19	Customers buy tickets faster than vendors add them.	Customers wait until tickets are available, then resume automatically.	Pass	None	Synchronization ensures smooth operation.
CF20	View current configuration.	Configuration details are displayed in a human-readable format.	Pass	None	Display functionality is effective.
CF21	Handle thread interruptions.	Threads stop gracefully when interrupted.	Pass	None	No errors observed during thread interruption.
CF22	Start system without configuration file.	System prompts for new configuration and initializes successfully.	Pass	None	Works as expected.

CF23	Simultaneous "start" and "check" commands.	"Check" shows current status while "start" initializes processes.	Pass	None	Both commands handled in parallel.
CF24	Start system with two vendors.	Vendors release tickets concurrently, logs confirm activity.	Pass	None	Concurrent vendor processes work.
CF25	Start system with two customers.	Customers buy tickets concurrently, and logs confirm purchases	Pass	None	Concurrent customer processes work
CF26	Invalid configuration file path.	System shows an error message and prompts for new configuration.	Pass	None	Error handling works.
CF27	Save configuration with invalid values.	Configuration file is not saved, and error messages are shown.	Pass	None	Validation prevents invalid saves.
CF28	Overload customer thread count.	customers still wait and synchronize correctly.	Pass	None	No deadlocks observed.
CF29	Overload vendor thread count.	Vendors wait for available capacity without errors.	Pass	None	No capacity related issues.
CF30	View empty ticket pool.	System correctly reports 0 tickets.	Pass	None	Ticket pool displays 0 accurately.
CF31	Save configuration with special characters in file name.	File saves successfully.	Pass	None	Handles special characters.
CF32	Exceed maximum ticket	Customers cannot buy	Pass	None	Handles special characters.

	limit while buying.	more tickets than available			
CF33	Load corrupted configuration file.	System shows error and prompts for new configuration.	Pass	None	Error handled gracefully.
CF34	Vendor threads handle interruptions.	Vendors stop releasing tickets when interrupted.	Pass	None	Thread interruption managed correctly.
CF35	Customer threads handle interruptions.	Customers stop buying tickets when interrupted.	Pass	None	Thread interruption managed correctly.
CF36	Use invalid characters in numeric input.	System prompts for valid numeric input.	Pass	None	Input validation is effective.
CF37	Simulate slow ticket release.	system normally runs with slower ticket addition rates.	Pass	None	Adapts to slower release rates.
CF38	Simulate slow ticket purchase.	System runs normally with slower ticket buying rates.	Pass	None	Adapts to slower purchase rates.
CF39	Check synchronization under high load.	System remains synchronized without deadlocks or race conditions.	Pass	None	Synchronization mechanisms are effective
CF40	Vendor retries when ticket pool is full.	Vendor pauses and resumes once capacity becomes available.	Pass	None	Retrying works correctly.
CF41	Exit during high activity.	System stops gracefully without errors or thread leaks.	Pass	None	System shuts down correctly.

CF42	Buy tickets while pool is full.	Customers buy tickets without issues.	Pass	None	Purchases proceed normally.
CF43	Add tickets while pool is empty.	Vendor adds tickets successfully.	Pass	None	Vendor adds tickets successfully.
CF44	Save configuration with JSON syntax error.	System prevents save and shows error message.	Pass	None	Prevents saving corrupted files.
CF45	Monitor long-running processes.	Processes continue to function correctly over extended periods.	Pass	None	Stability confirmed for long runtime.

Implementation

Configuration Module

The Configuration Module enables users to establish essential characteristics of the ticketing system, including the ticket release rate, client retrieval rate, total ticket count, and maximum ticket pool capacity. This module is engineered for versatility, offering a graphical interface (via React) and a command-line interface (CLI).

The React front features a straightforward form that allows users to enter necessary parameters, with real-time validation guaranteeing the acceptance of only valid inputs (e.g., positive values). Users receive notifications of errors via explicit feedback messages. Upon validation of the inputs, the configuration is transmitted to the backend using RESTful API calls, where it is processed and stored.

The backend, developed with Spring Boot, oversees configuration through a specialized class responsible for storing and retrieving these parameters. Configurations can be preserved in JSON files, enabling users to restore prior settings, facilitating repetitive system configurations. This modular methodology guarantees a seamless interface between the front end and back end, rendering it user-friendly and dependable.

Vendor and Customer Logic

The ticketing system emulates real-time processes via vendors and customers as concurrent threads. Vendors function as producers, contributing tickets to the communal pool, whilst customers serve as consumers, extracting tickets from the pool.

The backend utilizes a thread pool to manage these threads, ensuring efficient resource use and seamless operation. Vendors function autonomously and allocate tickets to the pool at a rate determined during configuration. Likewise, clients endeavour to get tickets simultaneously, with their purchasing rates influenced by the configuration. Synchronization mechanisms guarantee that operations on the shared ticket pool are thread-safe, averting problems such as race situations or data corruption.

The system emphasizes effective thread management by requiring vendors to pause when the pool is at capacity and customers to wait when it lacks resources. This conduct preserves system stability and guarantees a balanced ticket flow. The ticketing system can manage large concurrency while maintaining data integrity by utilising multi-threading and synchronisation.

Logging and Error Handling

Logging and error management are essential for ensuring transparency and resilience within the system. All significant activities are recorded, including ticket releases, purchases, and thread disruptions. The logs furnish comprehensive data regarding system operations, facilitating the monitoring of performance and the diagnosis of difficulties.

The backend records activities in the console and a log file for enduring documentation. For example, occurrences such as "tickets added to the pool" or "tickets purchased by a customer" are recorded with timestamps to monitor the system's performance over time.

Error management guarantees the system's stability in the face of unforeseen occurrences. Problems such as erroneous user inputs, file access failures, or thread disruptions are managed adeptly. Users receive explicit error alerts as console outputs in the CLI or as pop-up notifications in the React front. This method reduces user misunderstanding and guarantees rapid system recovery from mistakes without failure.

The system achieves great dependability and user satisfaction through meticulous logging and effective error management, facilitating seamless operation under challenging situations

User Interface

Challenges and Solutions

Concurrency Issues

Concurrency was a fundamental issue in the Real-Time Event Ticketing System due to the simultaneous actions of vendors (producers) adding tickets and customers (consumers) retrieving tickets. Inadequate synchronization may result in race situations, causing data corruption or inconsistent states within the shared ticket pool.

For instance, if numerous threads concurrently attempted to add or remove tickets from the pool without synchronization, the system could execute overlapping operations, leading to erroneous ticket counts. Synchronization procedures were instituted to resolve this issue. The communal ticket pool was safeguarded utilizing thread-safe mechanisms, including locks and conditions. These mechanisms guaranteed that only a single thread could execute crucial activities, such as adding or removing tickets, at any moment. The solution mitigated race problems by restricting access to the pool during certain operations.

Furthermore, careful design minimized thread blocking to preserve system responsiveness. Operations were structured to expedite lock releases, optimizing thread management while maintaining data integrity.

Deadlocks

Deadlocks, in which threads are perpetually waiting for resources, posed a possible risk due to the system's multi-threaded architecture. A vendor may delay adding tickets when the pool is at capacity, whereas a consumer may concurrently await the availability of tickets in an empty pool. Improper management of this interdependence may lead to a stalemate.

To avert deadlocks, the system employed a variety of strategies:

Lock Ordering:
A consistent sequence for acquiring locks was implemented to ensure that threads obtain resources in a predetermined order, hence mitigating the potential for circular waits.

Timeouts:
Threads awaiting a condition, such as pool capacity or ticket availability, were assigned timeouts. Should the condition remain unmet within the designated timeframe, the thread may pursue alternative actions or attempt a retry subsequently, therefore disrupting the deadlock cycle.

Equity Policies:
Locks with fairness configurations were employed to guarantee equitable resource distribution. This mitigated thread starvation by guaranteeing that threads were addressed in a first-come, first-served fashion.
Through the implementation of these solutions, the system circumvented deadlocks and ensured seamless operation even under high concurrency. These precautions guaranteed that all threads could ultimately fulfill their responsibilities without becoming perpetually obstructed.

Conclusion

The Real-Time Event Ticketing System effectively showcases the application of Object-Oriented Programming (OOP) principles, multi-threading, and synchronization to address real-world issues in concurrent settings. The Producer-Consumer pattern facilitates efficient ticket release and purchase operations, allowing vendors and customers to operate concurrently.

The project features a React-based frontend that provides an intuitive and responsive user interface, enabling users to configure the system and monitor its performance. The backend, powered by Spring Boot, oversees essential functions such as ticket management, logging, and thread synchronization, ensuring robust and reliable performance. The system ensures transparency and effectively addresses potential issues through comprehensive logging and error management.

Key achievements include:

An uninterrupted interaction between the frontend and backend, facilitating dynamic configuration and real-time status updates.

Thread-safe procedures for administering the shared ticket pool, guaranteeing data integrity amidst high concurrency.

Thorough logging and error management to improve system stability and user experience.

This project effectively achieves its goals by delivering a scalable and modular ticketing solution that exemplifies the practical application of programming concepts acquired during the coursework.

Future Enhancements

While the system is fully functional, there are opportunities for future improvements and additional features to extend its capabilities and usability:

Mobile App Integration: Develop a mobile application using frameworks like React Native or Flutter to provide users on-the-go ticketing system access.

Priority Customer Handling: Introduce a VIP customer feature, where customers with higher priority can access tickets before others. This could involve a priority queue or additional synchronization logic.

Dynamic Thread Management: Enable users to add or remove vendor and customer threads in real-time through the interface, offering greater flexibility and control.

Real-Time Analytics Dashboard: Incorporate a visual dashboard with charts and graphs to display ticket sales trends, ticket availability, and thread activity in real time.

Database Integration: Persist ticketing data, user configurations, and transaction logs in a database, enabling data retrieval and reporting over time.

Multi-Language Support: Add localization features to support multiple languages, making the system more accessible to a global audience.

WebSocket Integration: Replace periodic polling with WebSocket-based communication for real-time ticket availability and system status updates.

Implementing these enhancements could enable the system to develop into a comprehensive ticketing solution adept at managing larger scales and more intricate scenarios, while maintaining a user-friendly experience.