SAPIENZA
UNIVERSITÀ DI ROMA

Performance Evaluation of Software Architectures using Discrete Event Simulation

Faculty of Information Engineering, Computer Science and Statistics
Bachelor Degree in Computer Science

Radu Ionut Barbalata
ID number 2002688

Advisor
Prof. Enrico Tronci

Co-Advisor
Prof. Toni Mancini

Academic Year 2024/2025

Thesis defended on 24 October 2024
in front of a Board of Examiners composed by:

Prof. Tronci (chairman)
Prof. Arrigoni
Prof. Bottoni
Prof. Gorla
Prof. Maselli
Prof. Samory
Prof. Salvo

Performance Evaluation of Software Architectures using Discrete Event Simulation
Sapienza University of Rome

© 2024 Radu Ionut Barbalata. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: ionu.radu02@gmail.com

The destination becomes attainable only for those who embrace the journey,
traversing each step with purpose along the path.

Contents

1 Introduction **2**
1.1 Architecture Evaluation . . . 2
1.1.1 Challenges in Non-functional Requirement Testing . . . 2
1.2 Shadow Testing in Architecture Validation . . . 2
1.3 Accelerated Simulation for Performance Evaluation . . . 3
1.3.1 Simulating Client Behavior . . . 3
1.3.2 Architecture Simulation in Simulated Time . . . 4
1.4 Problem Statement . . . 4
1.5 Proposed Solution . . . 5
1.5.1 System Architecture . . . 5
1.6 Conclusion . . . 5

2 Preliminaries **6**
2.1 Concepts of Time . . . 6
2.1.1 Formal Definition of Simulation Time . . . 7
2.2 Modes of Execution . . . 7
2.3 Discrete-Event Simulation . . . 7
2.4 Discrete-Event Simulation Framework . . . 8

3 Design and Implementation **10**
3.1 System Architecture . . . 10
3.2 Component Design . . . 11
3.3 Process Interaction Primitives . . . 11
3.4 Managing Synchronization Issues . . . 13
3.5 Monitoring and Logging Mechanism . . . 14
3.5.1 Simulation Logger . . . 15
3.5.2 Performance Monitor . . . 15

4 Simulation Speed-up **16**
4.1 Key Variables . . . 16
4.2 Real-time Simulation Cost . . . 17
4.3 Simulated-time Simulation Cost . . . 18
4.4 Simulation Speed-up Factor . . . 19

5 Validation **20**
5.1 Problem Description . . . 20
5.2 Non-functional Requirements . . . 22
5.2.1 Performance Metrics . . . 22
5.2.2 Results and Observations . . . 23

6 Validation of Formula Results **25**
6.1 Methodology for Validation 25
6.1.1 Test Setup and Experimental Design 25
6.2 Error Analysis and Accuracy Metrics 25
6.2.1 Analysis of Results 26
6.2.2 Interpretation of Results 26

7 Conclusion and Future Work **28**
7.1 Summary of Results 28
7.2 Future Work 28
7.3 Conclusion 29

Bibliography **31**

Abstract

This thesis focuses on the validation of a newly designed e-commerce system, with particular emphasis on non-functional requirements (NFRs) related to performance. The proposed architecture consists of multiple replications of the system, with a load balancer that distributes client requests among different servers. Client behavior is modeled using a Markov chain, simulating typical customer interactions with the system. The primary goal of this validation is to determine how well different architectural designs meet performance requirements under various loads.

To achieve this, a series of experiments were conducted to measure critical performance metrics, such as latency, response time, and the overall speed-up achieved through simulation. We evaluated the system under different configurations, varying the number of clients and the volume of requests. The results revealed that simulated-time simulations tend to be more efficient with a smaller number of clients, where the overhead of real-time coordination is minimized. Conversely, as the number of clients increases, real-time simulations become more effective, distributing the workload more evenly and improving performance.

Additionally, we investigated the system's capacity to handle 100 million requests while maintaining a speed-up of at least 10x. The findings suggest that approximately 2.9 million clients are required to achieve this level of performance. This research provides valuable insights into the trade-offs between simulated-time and real-time testing, offering practical guidance for system architects on how to optimize performance testing in large-scale e-commerce environments.

Chapter 1

Introduction

Testing and validating software architectures often require robust methodologies to ensure that non-functional requirements (NFRs) such as performance, scalability, and efficiency are met. Traditional approaches, like live testing or simulation of user behaviors, can sometimes be time-consuming and resource-intensive. In this document, we delve into advanced techniques like shadow testing and accelerated simulation that streamline NFR validation while preserving test accuracy.

1.1 Architecture Evaluation

When evaluating an architecture's ability to meet NFRs, a comprehensive testing strategy is necessary. This chapter discusses various testing techniques and their impact on validating an architecture's readiness for deployment.

1.1.1 Challenges in Non-functional Requirement Testing

Traditional testing methods for NFRs often involve long-running tests that replicate real-world scenarios, requiring extensive resources. For instance, testing response times and scalability might involve simulating thousands of concurrent users over extended periods. Additionally, maintaining the accuracy of such tests, while avoiding interference with production environments, poses significant challenges.

To overcome these challenges, modern testing approaches like shadow testing and simulation-based time acceleration are employed. These methods allow for a more efficient way to measure and validate NFRs without the prohibitive costs associated with live testing.

1.2 Shadow Testing in Architecture Validation

Shadow testing, also known as shadow deployment or shadowing traffic, is a technique where production traffic is duplicated and sent to a new version of a system (V-Next) while the current version (V-Current) continues to serve actual users. This allows for a direct comparison between the two versions without affecting the end-user experience.

The Mechanics of Shadow Testing

Shadow testing works by splitting the incoming traffic at the network level or within the application itself, forwarding it simultaneously to both V-Current

and V-Next. Users interact with V-Current, and the responses are returned as usual. Meanwhile, V-Next processes the same requests in the background. The key advantage of this approach is that it allows testing under realistic load conditions, using real-world traffic patterns.

Example: Consider a web-based e-commerce platform that is planning to roll out a new search feature. Using shadow testing, the new search algorithm (V-Next) can be validated against the existing algorithm (V-Current) under actual user search patterns, ensuring it meets performance and scalability expectations before being fully rolled out.

Measuring Shadow Testing Outcomes

Shadow testing provides an opportunity to collect detailed performance metrics for the V-Next system without compromising the stability of the V-Current system. These metrics include:

- **Latency and Response Times:** Comparison of response times between V-Next and V-Current helps identify performance regressions.
- **Error Rates:** Observing errors in V-Next can reveal compatibility issues or bugs.
- **Resource Utilization:** CPU, memory, and network usage metrics can determine if the new version is more efficient.

By analyzing these metrics, architects and developers can make informed decisions about the readiness of V-Next for deployment.

1.3 Accelerated Simulation for Performance Evaluation

Live testing of NFRs can be cumbersome due to the time involved in simulating real-world conditions. To address this, accelerated simulation techniques are employed, allowing for the skipping of idle or sleep periods that occur during normal operation. This section explains how accelerated simulation can reduce testing time and improve the efficiency of validation processes.

1.3.1 Simulating Client Behavior

In a traditional simulation, client behavior is replicated by having the system wait or sleep during idle periods, such as waiting for user input or network responses. However, when testing NFRs like response time or throughput, these idle periods do not contribute valuable data. By skipping over these periods, we can focus solely on the operations that impact performance metrics.

The client behavior simulation in an accelerated environment is structured as follows:

Algorithm 1 Client Behavior Simulation with Time Skipping

```
Initialize the client parameters and states
while True do
    Simulate state transitions and client operations
    Skip idle periods to accelerate testing
    Send a request corresponding to the current state
    Record the request's send time
    Wait for the simulated server response
    Record the response's arrival time and calculate the latency
    Update state: state ← newState(state)
end while
```

By focusing on critical operations and skipping idle periods, this simulation technique enables quicker validation of NFRs.

1.3.2 Architecture Simulation in Simulated Time

Architecture simulation in simulated time involves defining the system's components and their interactions, simulating events as they would occur in real-world conditions. However, the simulation time is managed dynamically, skipping over non-relevant periods and accelerating the overall test process.

The simulation framework is illustrated in the following algorithm:

Algorithm 2 Simulated-time Server Request Handling

```
Initialize the server parameters and states
while True do
    Wait for incoming client requests or internal events
    Simulate the request handling based on predefined response times
    Skip idle or processing times that do not impact performance metrics
    Send responses back to clients
end while
```

This approach allows testing of complex scenarios in a fraction of the time required by traditional methods.

1.4 Problem Statement

Software testing and validation traditionally involve time-consuming processes to measure system performance under various conditions. For instance, testing the scalability of a web application under heavy load can take hours or even days when done in real-time, as clients need to remain idle for long periods between requests to simulate realistic user behavior.

The problem becomes more acute when testing microservices or distributed systems, where each component might experience different load patterns and have unique idle times. In such scenarios, traditional real-time testing is not only inefficient but can also result in skewed performance metrics due to the excessive duration of the tests.

To address this issue, we propose an approach that accelerates the testing process by skipping over idle periods and focusing solely on the operations that impact system performance. By orchestrating time dynamically, we achieve more

accurate performance measurements in a fraction of the time required by conventional methods.

1.5 Proposed Solution

The proposed solution leverages simulated time and discrete event simulation (DES) to create a controlled testing environment where the passage of time is managed programmatically. This allows for skipping over idle periods, focusing only on significant events, and accelerating the overall testing process.

1.5.1 System Architecture

The system consists of the following components:

- **Client Simulator:** Mimics client behavior, generating requests based on a predefined state machine and skipping idle periods.
- **Server Simulator:** Handles client requests, processes them according to predefined rules, and skips idle or processing times that do not contribute to NFR validation.
- **Simulation Orchestrator:** Manages the flow of time, ensuring that idle periods are skipped and only critical events are processed.
- **Metrics Collector:** Records performance metrics such as response times, throughput, and resource utilization for analysis.

The orchestrator component is responsible for managing the time flow, ensuring that the simulation advances quickly while maintaining the accuracy of event timing.

1.6 Conclusion

The use of advanced testing techniques like shadow testing and accelerated simulation offers significant benefits in validating software architectures against non-functional requirements. By leveraging real-world traffic in a non-intrusive manner and accelerating time in simulations, these approaches reduce testing time and resource consumption while preserving the accuracy and relevance of test results. Future work will explore the integration of machine learning models to further optimize the testing process and predict system behavior under various load conditions.

Chapter 2

Preliminaries

Designing an effective Discrete Event Simulation (DES) system requires a solid understanding of the fundamental concepts underpinning parallel and distributed simulation systems [2].

Simulation, at its core, involves creating a model that captures the behavior of a real-world system over time. Such a system is characterized by a set of states that evolve dynamically in response to events. To accurately simulate a physical system, three essential criteria must be met:

1. **Representation of the System State**: A mechanism to define and maintain the system's state variables.
2. **Mechanism for State Evolution**: A way to modify the state to reflect changes over time, typically triggered by the occurrence of discrete events.
3. **Abstraction of Time**: A logical construct for time that allows the simulation to progress through events, preserving temporal relationships.

In computer simulations, the system state is represented using a set of state variables, which are updated according to the simulated progression of the system. These state variables are usually expressed in high-level programming languages such as C++ or Java. Time is modeled using an abstraction known as *simulation time*, which will be discussed in detail in the subsequent sections.

2.1 Concepts of Time

A robust understanding of the various notions of time is essential for creating effective simulation models. The primary time concepts relevant to simulations include:

- **Physical Time**: The actual time in the real-world system being simulated.
- **Simulation Time**: An abstraction used to model the physical time within the simulation environment.
- **Real Time**: The actual elapsed time during the execution of the simulation, typically measured using the hardware clock of the underlying operating system.

2.1.1 Formal Definition of Simulation Time

Simulation time is formally defined as a totally ordered set of values, where each value represents a specific temporal instant within the simulated system. For two simulation time values, T_1 and T_2, if $T_1 < T_2$, it signifies that the event corresponding to T_1 occurs before the event corresponding to T_2. This ordering of simulation time values ensures that temporal relationships among events are preserved within the simulation environment.

2.2 Modes of Execution

Simulations can be executed in different modes depending on how simulation time progresses relative to real time:

- **Real-Time Execution**: The simulation time advances at the same rate as real time, ensuring that simulated events occur synchronously with real-world events.
- **Scaled Real-Time Execution**: The progression of simulation time is governed by a constant scaling factor, allowing it to run faster or slower than real time. For example, a scaling factor of 2 means that every second of real time corresponds to two seconds of simulated time.
- **As-Fast-As-Possible Execution**: The simulation progresses as quickly as possible, unconstrained by real time. This mode is ideal for scenarios that require multiple iterations, as it allows the simulation to complete in the shortest possible duration.

2.3 Discrete-Event Simulation

Discrete-event simulation (DES) is characterized by updating the state of the system only when specific events occur. An event is defined as an instantaneous change in the system's state and is associated with a timestamp that indicates its occurrence in simulation time. The simulation advances by processing events in chronological order, updating state variables only when necessary.

Algorithm 3 Discrete-Event Simulation Based on Real-Time Execution

1: **Initialize** simulation parameters and state variables.
2: **while** simulation is running **do**
3: Wait until `W2S` (real time) $\geq$ simulation time.
4: Compute the system's state at the end of this interval.
5: Advance simulation time to the next event time.
6: **end while**

Unlike continuous simulations, where time progresses in fixed intervals, DES jumps from one event to the next, skipping periods where no state changes occur. This event-driven approach enhances efficiency by minimizing unnecessary state computations.

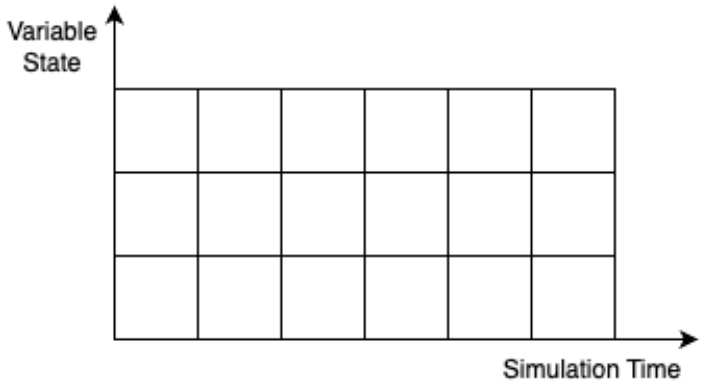

Figure 2.1. Space-Time Diagram for Time-Stepped Simulation

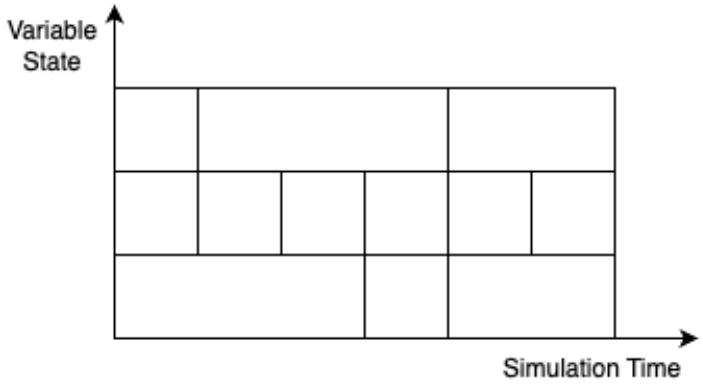

Figure 2.2. Space-Time Diagram for Event-Driven Simulation

In scenarios where real-time consistency is required, discrete-event simulations can be synchronized to ensure that simulation time progresses in lockstep with real time. This synchronization enables a more cohesive and accurate representation of real-world processes.

2.4 Discrete-Event Simulation Framework

The focus of this work is on discrete-event simulations. A typical DES program relies on three fundamental data structures:

1. **State Variables**: These variables represent the current state of the system.
2. **Event List**: A priority queue containing events scheduled to occur at future simulation times, each with a corresponding timestamp.
3. **Global Clock Variable**: Represents the current simulation time. If the clock variable has a value T, it indicates that all activities up to time T have been simulated, while activities beyond T are yet to be processed.

An event in the DES framework typically consists of a data structure containing its timestamp (e.g., 9:16 AM), event type (e.g., airplane arrival), and various parameters specifying details of the event (e.g., Flight 396 arriving at LAX).

In a physical system, events such as airplane arrivals occur autonomously. In the simulation, however, nothing happens unless an event is explicitly created by the simulation computation. This process is referred to as *event scheduling*. For example, if the simulation reaches time 9:00 and an event indicates that Flight 200 has landed, the simulation can then schedule a new event for this plane's departure at 9:56. Event scheduling involves allocating memory for a new event, populating its fields (timestamp, event type, and parameters), and adding it to the event list.

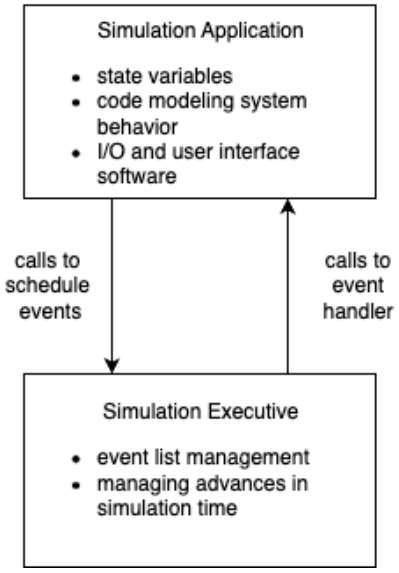

Figure 2.3. Separation of the Simulation Program into the Simulation Application and Executive Components

A discrete-event simulation program can be divided into two major components, as illustrated in Figure 2.3:

- **Simulation Executive**: This component maintains the event list and the global clock variable. It operates independently of the specific physical system being simulated, serving as the backbone of the simulation program. Often, it is provided as a generic software library adaptable to simulate various types of systems.
- **Simulation Application**: This component contains the state variables and software routines for modeling the physical system. It is closely tied to the specific characteristics and behaviors of the real-world system being simulated. In its simplest form, the simulation executive must only provide one primitive procedure to the simulation application: a method for scheduling events.

The core of the simulation executive is an event-processing loop that repeatedly removes the event with the smallest timestamp from the event list, simulates its occurrence, and advances the simulation clock.

Chapter 3

Design and Implementation

3.1 System Architecture

The proposed system is designed based on a Centralized Barrier synchronization model [2], where a central entity, referred to as the **Simulation Executive**, coordinates the synchronization and communication among all other processes in the simulation. Each process interacts with the Simulation Executive to communicate its state and synchronize its progress. When all processes have communicated their current state, the Simulation Executive determines the minimum simulation time that all processes can safely advance to, ensuring consistent progression.

This centralized approach reduces the overall communication overhead significantly. In a typical client-server simulation with m client requests, the number of synchronization messages can reach $O(nm)$, where n is the number of processes, and each request results in at least one 'syncSleep(T)' call. Our architecture optimizes this to achieve $O(m)$ synchronization messages by eliminating redundant broadcasts to all processes and instead notifying only the process with the lowest simulation time. The overall system architecture is depicted in Figure 3.1.

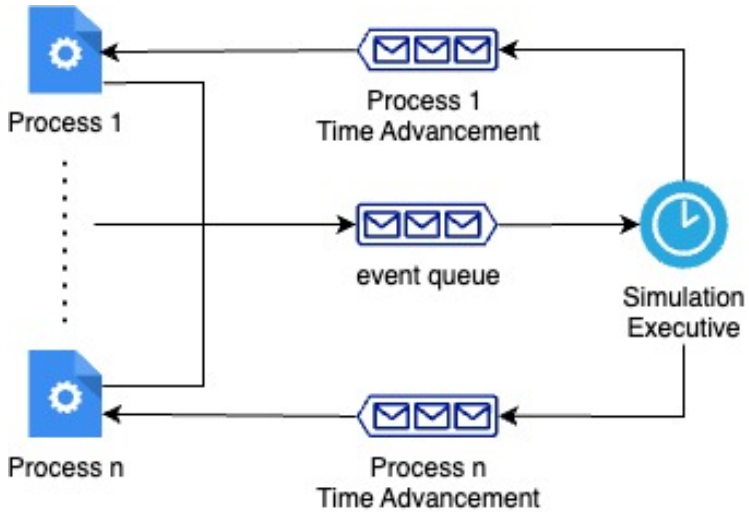

Figure 3.1. System Architecture

Although this approach introduces additional complexity in managing the state of each process, it significantly reduces the number of messages transmitted over

the network, leading to an overall performance improvement. At the start of the simulation, each process establishes a connection with the Simulation Executive, creating a dedicated messaging stream. The Simulation Executive handles all connections and disconnections dynamically, ensuring seamless communication and synchronization.

3.2 Component Design

The implementation of the Simulation Executive requires careful consideration of several key components and mechanisms:

- **Event List Management**: Efficient management of the event list is crucial to handle scheduling operations effectively. We employ a priority queue [3] to manage events, allowing insertion and removal of the minimum element in $O(\log n)$ time, where n is the number of events.
- **Managing Simulation Time Advances**: The Simulation Executive must correctly manage the states of all connected processes and handle various edge cases. The simulation time can advance if and only if:

$$\begin{gathered}\#\text{list\_management} \\ = \\ \text{n - \#disconnected\_processes - \#processes\_in\_blocking\_call}\end{gathered}$$

  This formula ensures that the simulation time advances only when all active processes are ready. Each process in the system can exist in one of the following states:

  - **Running**: The process is actively executing instructions.
  - **Sleeping**: The process is waiting in the event list.
  - **Blocking**: The process is waiting for a message or response.
  - **Disconnected**: The process has left the simulation and is no longer participating.

  To manage these states efficiently, we use an unordered map [5], which supports $O(1)$ operations for insertion, deletion, and element counting.

3.3 Process Interaction Primitives

The interaction between the processes and the Simulation Executive is facilitated through a set of primitive functions:

- **connect()**: Establishes a connection between a process and the Simulation Executive.
- **alertBlockingCall()**: Notifies the Simulation Executive that the process is waiting for a message or response.
- **exitBlockingCall()**: Notifies the Simulation Executive that the process has exited the blocking state.

- **syncSleep(long double T)**: Signals that the process will resume after a specified simulated time T.
- **mySleep(long double T)**: Emulates real-time sleep, allowing the process to pause for T time units.
- **disconnect()**: Disconnects the process from the Simulation Executive.

Using these primitives, we define the behavior of client, server, and service processes in the simulation. The pseudocode for each process is presented below:

Algorithm 4 Client Behavior Simulation

```
1: Initialize client parameters.
2: while True do
3:    mySleep(based on the current state)
4:    syncSleep(time to send a request)
5:    Send a request corresponding to the current state.
6:    alertBlockingCall()
7:    Wait for the server's response.
8:    exitBlockingCall()
9:    syncSleep(time to update the state)
10:   Update state: state ← newState(state)
11: end while
```

Algorithm 5 Server Request Handling

```
1: Initialize server parameters.
2: while True do
3:    alertBlockingCall()
4:    Wait for incoming requests from clients or services.
5:    exitBlockingCall()
6:    syncSleep(time to process the request)
7:    if the request is from a client then
8:       Forward the request to the corresponding service.
9:    else
10:      Send the response back to the client.
11:   end if
12: end while
```

Algorithm 6 Service Response Handling

```
1: Initialize service parameters.
2: while True do
3:    alertBlockingCall()
4:    Wait for requests from the server.
5:    exitBlockingCall()
6:    syncSleep(time to process the request)
7:    Send the response back to the server.
8: end while
```

3.4 Managing Synchronization Issues

One of the primary challenges in implementing a centralized synchronization model is handling delays caused by network latency. Figure 3.2 illustrates potential synchronization issues when processes transition between different states.

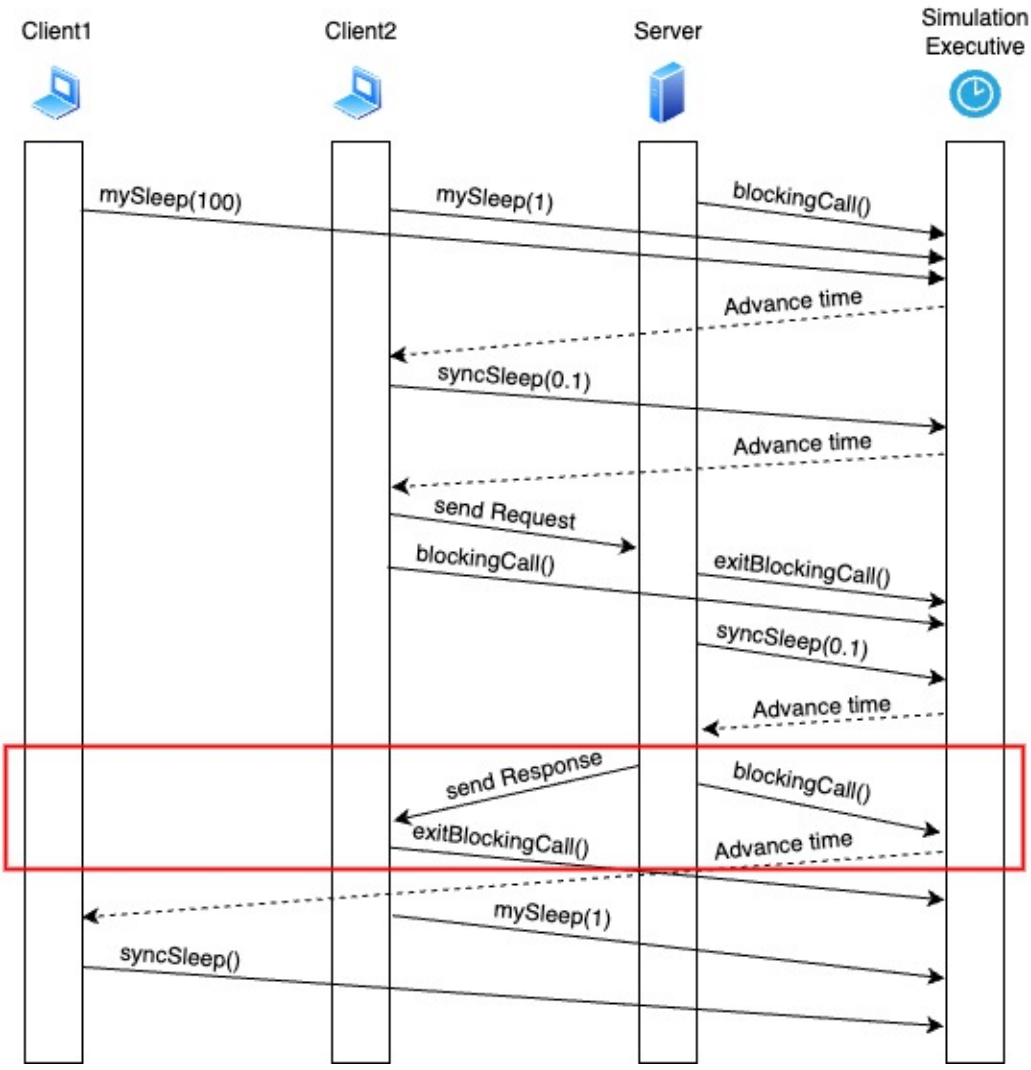

Figure 3.2. Space-Time Diagram of the Simulation

For instance, after sending a request, ‘client1‘ might enter a ‘Sleeping‘ state, while ‘client2‘ is in a ‘Blocking‘ state, waiting for a response from the server. To address these issues, we introduce an additional state:

- **Ready to Exit Blocking Call**: Represents processes that are in a blocking state but are expected to receive a response shortly.

To support this state, we define the following additional primitives:

- **sendId(std::string id)**: Associates each process with a unique identifier, facilitating easier tracking of communications.

- **sendingTo(std::string receiverId)**: Notifies the Simulation Executive that a process is sending a message to ‘receiverId‘, allowing it to anticipate a blocking exit.

The updated condition for simulation time advancement now becomes:

$$\#list\_management = n - \#disconnected\_processes - (\#processes\_in\_blocking\_call - \#processes\_exiting\_blocking\_call)$$

This refinement ensures that simulation time advances correctly even under varying network conditions and communication delays.

Simulation Executive

To start the Simulation Executive, specify the number of processes expected to connect. The executive will then wait for connections to be established with each process before initiating the simulation.

Simulation Application

The system includes default classes such as ‘Client‘, ‘Server‘, and ‘Service‘, which ensure correct interaction with the Simulation Executive. Custom processes can also be implemented using the described primitives to interact seamlessly with the Simulation Executive, allowing for flexible simulation scenarios, there are also a provided classes that simulate the client based on some inputs, server, and services.

3.5 Monitoring and Logging Mechanism

The system includes two critical components for tracking simulation activities and collecting performance metrics: the **Simulation Logger** and the **Performance Monitor**. These components are designed to provide real-time insights into the simulation, detect anomalies, and enable efficient debugging. Such a monitoring architecture follows principles similar to distributed logging systems, which enhance observability and fault detection by aggregating data from multiple sources, thereby offering a comprehensive view of system performance [10]. Figure 3.3 depicts an illustration of these monitoring mechanisms.

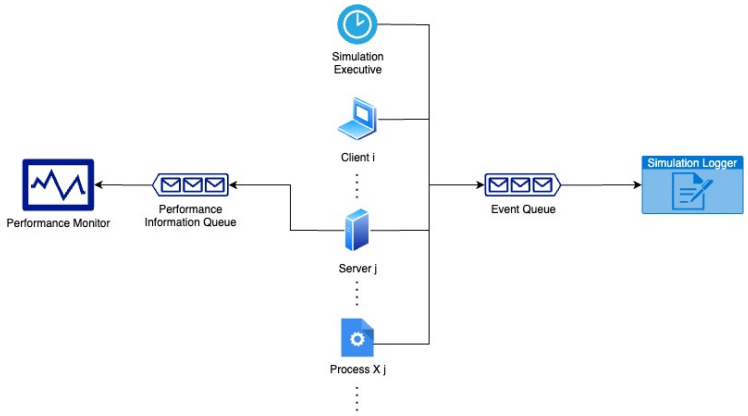

Figure 3.3. Monitoring and Logging Mechanism

3.5.1 Simulation Logger

The **Simulation Logger** records all significant events that occur during the simulation. Each event is logged with the following details:

- **Timestamp**: The timestamp when the event occurred.
- **Event Type**: The type of event (e.g., 'connection established', 'alertBlockingCall', etc.).
- **Stream Name**: The stream where the event originated.
- **Values**: Any associated data relevant to the event.

This detailed logging allows for comprehensive post-simulation analysis, facilitating the identification of execution patterns, interactions between processes, and debugging of unexpected behavior.

3.5.2 Performance Monitor

The **Performance Monitor** collects and analyzes performance metrics during the simulation. It continuously receives data from connected servers and updates key statistics such as:

- **Average Response Time**: The mean response time for all processed requests.
- **Minimum Response Time**: The shortest observed response time.
- **Maximum Response Time**: The longest observed response time.
- **Variance**: The variance in response times, which indicates the consistency of the system's performance.

The Performance Monitor not only identifies and alerts users of potential bottlenecks but also allows them to intervene during the simulation. This proactive capability helps in addressing performance issues as they emerge, minimizing disruptions. All performance logs are stored for post-simulation analysis, enabling a comprehensive evaluation of the system's efficiency and facilitating more granular analysis, such as tracing specific inefficiencies or assessing particular scenarios that might have led to performance degradation.

Chapter 4

Simulation Speed-up

Understanding the performance gains achieved by utilizing simulated time in a Discrete Event Simulation (DES) model requires a detailed analysis of the cost factors in both real-time and simulated-time scenarios. This chapter explores the key variables influencing system performance and presents formulas to estimate costs and speed-up factors, providing a basis for comparing the efficiency of different simulation techniques.

4.1 Key Variables

To model the simulation's dynamics, the following key variables are defined:

- p: The total number of requests initiated by clients throughout the simulation.
- c: The number of clients participating in the simulation.
- n: The number of possible request types or states a client can have, such as connect, disconnect, or idle.
- P: The transition matrix for a Markov chain with dimensions $n \times n$, representing state transition probabilities that model clients' behavior.
- svp: Stationary distribution vector of the Markov chain, denoted as a $1 \times n$ vector, representing the long-term probability distribution of the system's states.
- cst: A vector of dimension $n \times 1$ representing the average sleep times of clients in each state.
- ast: A vector of dimension $n \times 1$ indicating the frequency of send operations performed by clients for each state.
- art: A vector of dimension $n \times 1$ representing the architecture's average response times for each state.
- $cmot$: A vector of dimension $n \times 1$ representing the machine operation times on the client side for each state.
- $amot$: A vector of dimension $n \times 1$ representing the architecture's machine operation times for each state.

- *cstoo*: The number of send operations that each client performs to the simulation executive for each request.
- *artoo*: The number of send operations that the architecture performs to the simulation executive for each request.
- τ: The network transmission time, representing the time it takes to send a request over the network.
- α: A constant used for scaling time or representing additional delays in specific scenarios.

These variables form the basis for calculating the costs in both real-time and simulated-time simulations, allowing for a comprehensive analysis of the speed-up achieved using the simulation framework.

4.2 Real-time Simulation Cost

In real-time simulations, the system operates in synchrony with actual time, meaning that clients follow their defined behavior and sleep periods without any acceleration. The total real-time simulation cost, denoted as T_{real}, can be decomposed into two main components: the time for client operations and the time for architecture operations.

$$T_{real} = TClient_{real} + TArchitecture_{real}$$

The time for client operations $TClient_{real}$ is given by:

$$TClient_{real} = \left\lceil \frac{p}{c} \right\rceil \cdot (svp \cdot cst) + \left\lceil \frac{p}{c} \right\rceil \cdot \tau + \left\lceil \frac{p}{c} \right\rceil \cdot (svp \cdot cmot)$$

Where:

- $\lceil \frac{p}{c} \rceil$: The expected number of requests per client.
- $svp \cdot cst$: The expected sleep times for a client, weighted by the stationary distribution vector *svp*.
- τ: The network transmission time for a client to send a request to the server.
- $svp \cdot cmot$: The expected time spent in machine operations by the client, weighted by the stationary distribution vector.

The time for architecture operations $TArchitecture_{real}$ is given by:

$$TArchitecture_{real} = \left\lceil \frac{p}{c} \right\rceil \cdot (svp \cdot art) + \left\lceil \frac{p}{c} \right\rceil \cdot (svp \cdot ast) \cdot \tau + \left\lceil \frac{p}{c} \right\rceil \cdot (svp \cdot amot)$$

Where:

- $svp \cdot art$: The expected server processing time for each request, weighted by the stationary distribution vector *svp*.
- $(svp \cdot ast) \cdot \tau$: The network time for sending responses to clients, where *ast* is the vector indicating the frequency of send operations based on the state.
- $svp \cdot amot$: The expected time spent in machine operations by the architecture, weighted by the stationary distribution vector.

4.3 Simulated-time Simulation Cost

In simulated-time simulations, the system operates based on simulated time, skipping idle periods and thereby accelerating the simulation process. The total simulated-time simulation cost, denoted as $T_{simulated}$, can be broken down into three main components: the time for client operations, the time for architecture operations, and the time for simulation executive operations.

$$T_{simulated} = TClient_{simulated} + TArchitecture_{simulated} + TSimExecutive_{simulated}$$

The time for client operations $TClient_{simulated}$ is given by:

$$TClient_{simulated} = (svp \cdot cstoo) \cdot p \cdot \tau + p \cdot \tau + p \cdot (svp \cdot cmot)$$

Where:

- $(svp \cdot cstoo) \cdot p \cdot \tau$: The total network time for messages from clients to the simulation executive, considering the number of requests p and the network transmission time τ.
- $p \cdot \tau$: The network time for sending requests to the server.
- $p \cdot (svp \cdot cmot)$: The time spent in machine operations by the clients.

The time for architecture operations $TArchitecture_{simulated}$ is given by:

$$TArchitecture_{simulated} = (svp \cdot artoo) \cdot p \cdot \tau + p \cdot (svp \cdot ast) \cdot \tau + p \cdot (svp \cdot amot)$$

Where:

- $(svp \cdot artoo) \cdot p \cdot \tau$: The total network time for messages from the architecture to the simulation executive.
- $p \cdot (svp \cdot ast) \cdot \tau$: The network time for sending responses to clients and internal messages within the architecture.
- $p \cdot (svp \cdot amot)$: The time spent in machine operations by the architecture.

The time for simulation executive operations $TSimExecutive_{simulated}$ is given by:

$$TSimExecutive_{simulated} = (svp \cdot (cstoo + artoo)) \cdot p \cdot \theta + p \cdot (\beta + \tau)$$

Where:

- $(svp \cdot (cstoo + artoo)) \cdot p \cdot \theta$: The processing time of a request by the simulation executive, considering the combined send operations of clients and the architecture.
- $p \cdot (\beta + \tau)$: The time to resolve synchronization and network delays in the system.

4.4 Simulation Speed-up Factor

The speed-up factor $SimulationSpeedUp$ achieved by using simulated time can be estimated as:

$$SimulationSpeedUp = \frac{T_{real}}{T_{simulated}}$$

This ratio quantifies the extent to which the simulation runs faster by utilizing simulated time, eliminating idle periods, and optimizing request handling through the simulation executive.

Chapter 5

Validation

5.1 Problem Description

We aim to design and implement a new e-commerce system in this scenario. Based on business studies, the expected number of clients and their behavior are defined. The goal is to evaluate different architectural designs against non-functional requirements, particularly focusing on performance. The architecture team is exploring how many replications of the architecture they need to know how many replications need to satisfy the performance requirements.

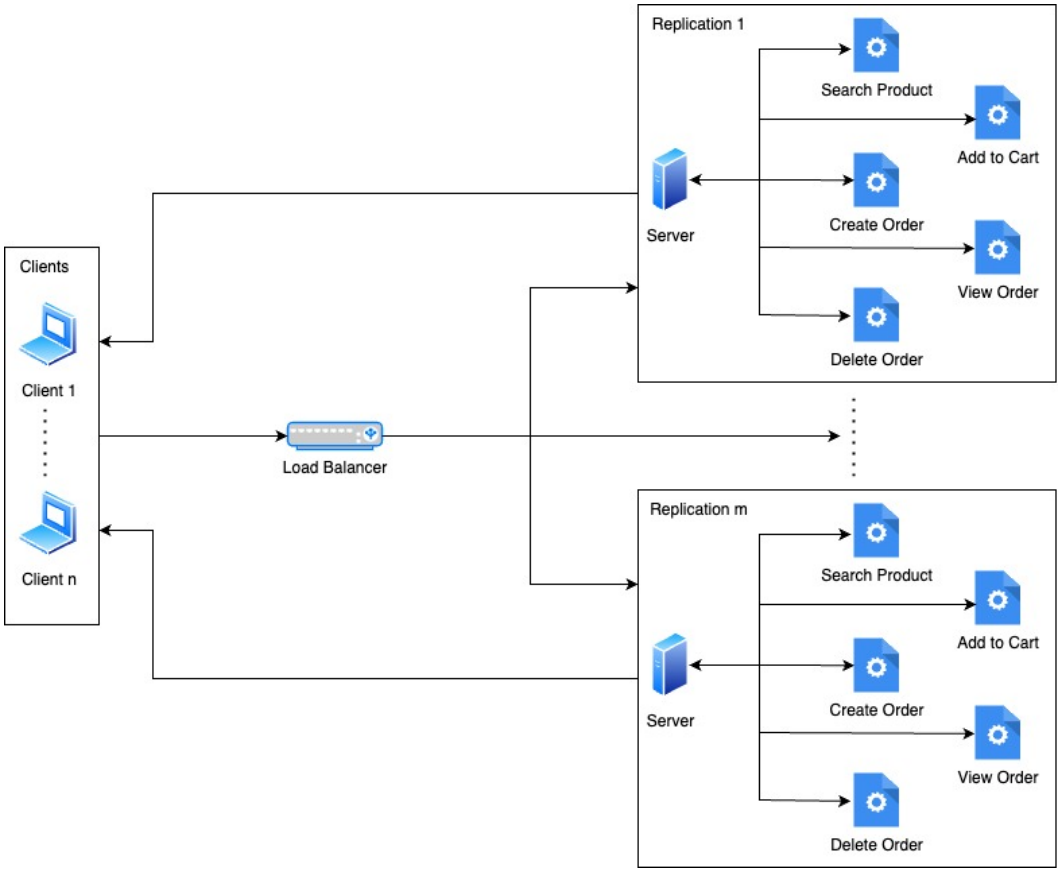

Figure 5.1. Eccomerce Software Design

Line you can see, the architecture 5.1 is divided in m replication, and there is also a load balance [7] that will just randomacaly make connect a client to the server of a replication, than the client can make some requests based on their behaviour implemented with a markov chain [8], also the load balancer can be considered as a DNS load Balancer.

And to define the behavior of the customer we can use a markov chain 5.2

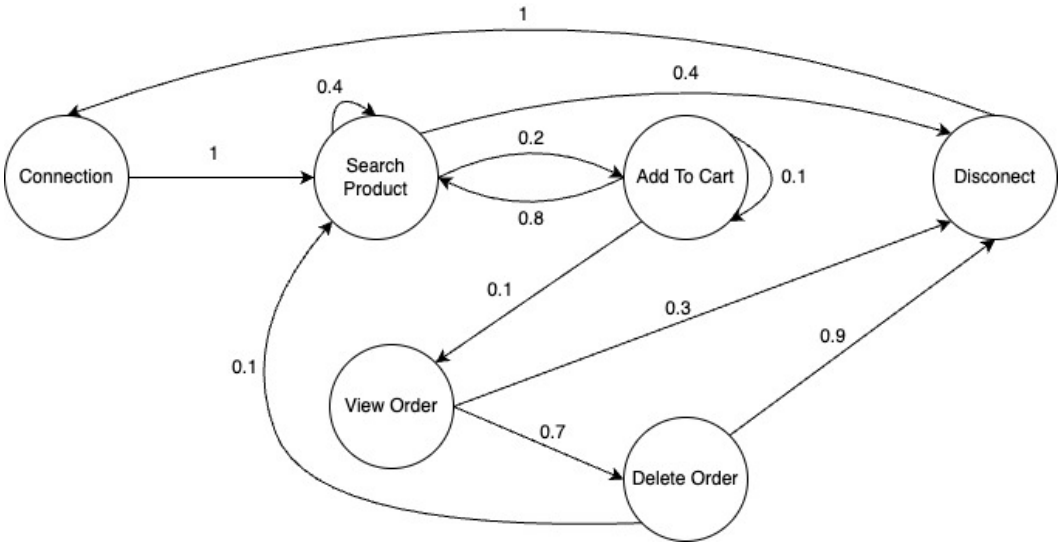

Figure 5.2. Customer behaviour though Markov chain

Now let's define the variables that will allow us to do the simulation and define the *SpeedUpSimulation*:

- $P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0.2 & 0 & 0 & 0.4 \\ 0 & 0.8 & 0.1 & 0.1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.7 & 0.3 \\ 0 & 0.1 & 0 & 0 & 0 & 0.9 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$
- $cst = \begin{bmatrix} 5 \times 86400 & 20 \times 60 & 16 \times 60 & 20 \times 60 & 1 \times 60 & 3600 \end{bmatrix}^T$
- $ast = \begin{bmatrix} 1 & 3 & 3 & 3 & 3 & 0 \end{bmatrix}^T$
- $art = \begin{bmatrix} 0.3 & 0.7 & 0.4 & 0.5 & 0.4 & 0.1 \end{bmatrix}^T$
- $cmot = \begin{bmatrix} 10^{-7} & 10^{-7} & 10^{-7} & 10^{-7} & 10^{-7} & 10^{-7} \end{bmatrix}^T$
- $amot = \begin{bmatrix} 2 \times 10^{-7} & 2 \times 10^{-7} & 2 \times 10^{-7} & 2 \times 10^{-7} & 2 \times 10^{-7} & 2 \times 10^{-7} \end{bmatrix}^T$
- $network\_cost = 0.00023$
- $cstoo = \begin{bmatrix} 4 & 3 & 3 & 3 & 3 & 1 \end{bmatrix}^T$
- $artoo = \begin{bmatrix} 3 & 9 & 9 & 9 & 9 & 2 \end{bmatrix}^T$

- $\theta = 5 \times 10^{-7}$
- $\beta = 1 \times 10^{-7}$
- $\alpha = 30$

5.2 Non-functional Requirements

Non-functional requirements (NFRs) define the quality attributes of a system, and they are critical for evaluating its performance, reliability, and maintainability. To

Here, we will outline the key NFRs for our system, focusing on how they can be measured and how our simulation tool helps ensure performance qualities.

Server ID	Request Type	Request Time	Read Request	Response Time
2100	connection	322.896095	322.896325	322.896555
2100	connection	734.715306	734.715536	734.715766
2100	searchProduct	966.871627	966.871857	967.572547
2100	searchProduct	1066.145295	1066.145525	1066.846215
2100	searchProduct	1432.340573	1432.340803	1433.041493

Table 5.1. Request times and responses.

Using the log structure (Table 5.1), we can calculate important metrics like minimum, maximum, mean, and variance for each request type, allowing us to assess the overall performance. These metrics give us insights into various NFRs, as explained below.

5.2.1 Performance Metrics

Performance is often the most critical non-functional requirement in systems handling large amounts of requests, as in our e-commerce scenario. We can calculate several performance-related metrics such as:

- **Latency**: Time taken to process a request, measured from the logs.
- **Min, Max, Mean, and Variance**: Statistical measures to track response times, helping to identify performance bottlenecks.

Request Type	Min Request	Max Request	Mean Request	Var Request	Count Requests
addToCart	0.40092	0.630819	0.400996	1.745494e-05	3028
connection	0.00046	0.000460	0.000460	1.970162e-21	6539
createOrder	0.50092	0.500920	0.500920	2.117974e-21	300
searchProduct	0.70092	1.482193	0.702789	8.150574e-04	14197
viewOrder	0.40092	0.400920	0.400920	4.094827e-21	207

Table 5.2. Non-functional requirements results: Performance Analysis

5.2.2 Results and Observations

We conducted tests with a range of values for p (number of client requests) and different values for c (number of clients), observing the simulation speed-up, *SimulationSpeedup*. The goal is to determine when it is more convenient to use simulated-time simulations and when real-time simulations might be more efficient.

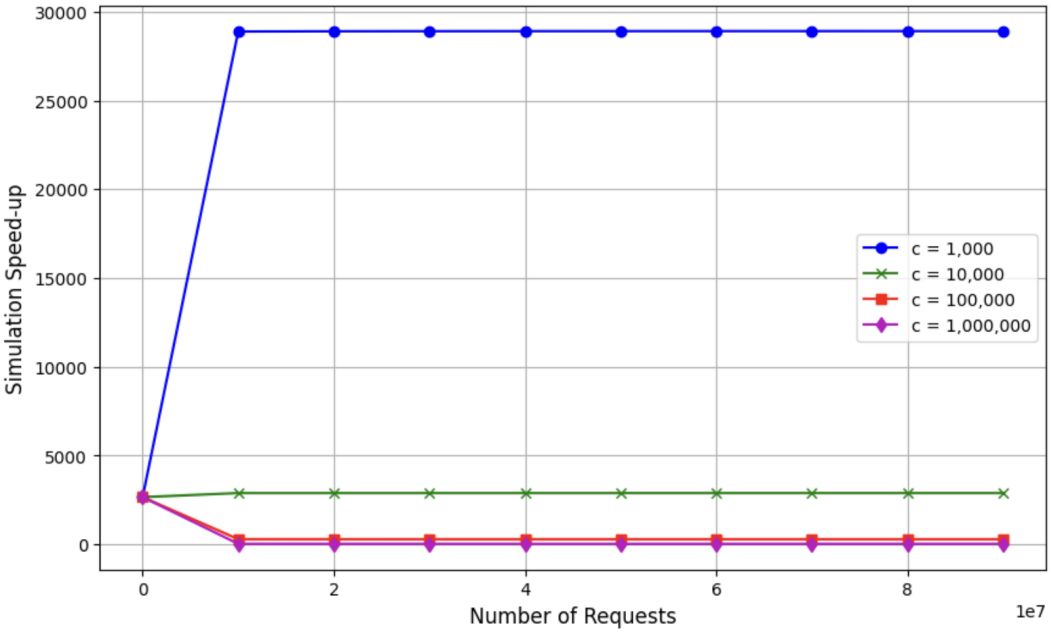

Figure 5.3. Performance ratio

Considering different values of c (number of clients), we observe that the simulation speed-up is significantly higher when the number of clients (c), is lower. This behavior is due to the parallel processing advantage of real-time simulations being less pronounced when there are fewer processes. With a lower number of clients, the overhead of coordinating real-time operations diminishes, making simulated time more efficient.

As the number of clients, c, increases, real-time simulations become more efficient, and the total simulated time, $T_{simulated}$, remains relatively unchanged. This is because real-time simulations can better distribute the workload across multiple machines.

Now lets' do see better if it's better than having a 10 times faster then real time5.4

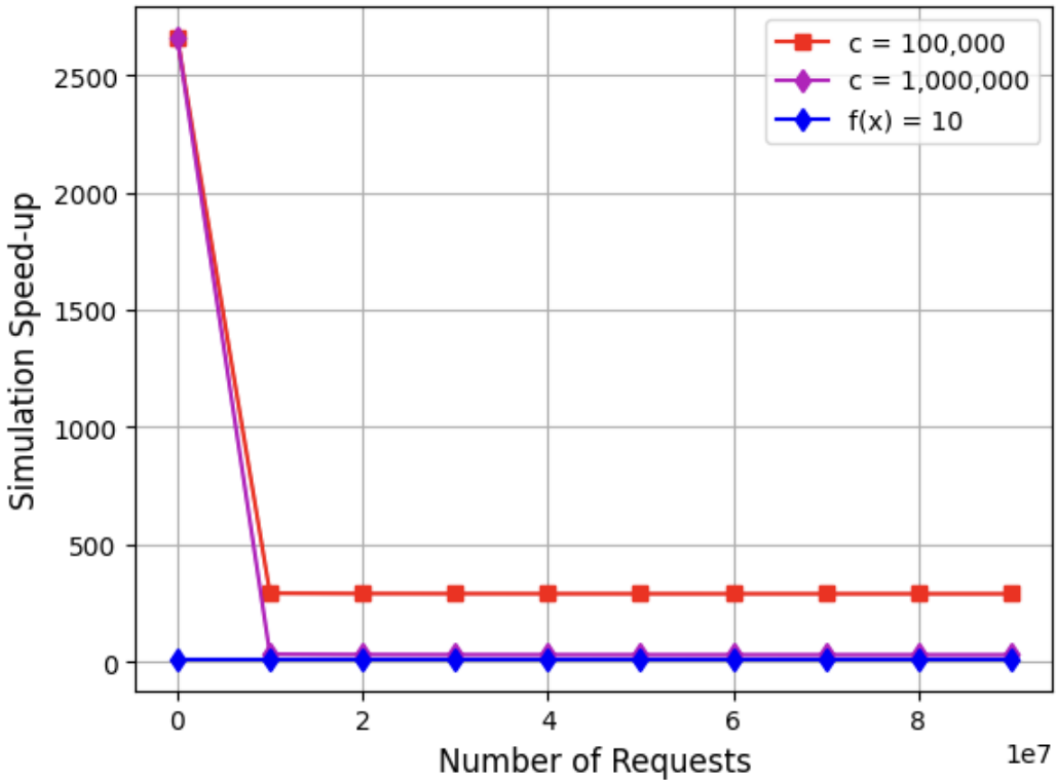

Figure 5.4. Zoom on function with lower *SimulationSpeedUp*

We also sought to determine the maximum number of clients c that can handle at least 100,000,000 requests (p) while achieving a $SimulationSpeedup \geq 10$. Our findings indicate that $c = 2,941,176$ is required to achieve a speed-up of 10x in this scenario.

Chapter 6

Validation of Formula Results

6.1 Methodology for Validation

Validating the proposed formulas for computing simulation time ($T_{simulated}$) is a critical step to ensure their reliability and applicability. While the formula for real-time execution (T_{real}) does not require further validation—since it directly reflects the time elapsed during real-world execution with actual delays—the simulated time formulas need rigorous testing to verify their accuracy.

To validate $T_{simulated}$, a series of experiments were conducted using various configurations of p (number of requests) and c (number of clients). The simulation results obtained from the tool were then compared against the theoretical values predicted by our formulas. This comparison was performed by measuring the actual time taken to execute the simulated test cases and calculating the percentage error between the observed and predicted values.

6.1.1 Test Setup and Experimental Design

The experiments were designed to evaluate the formulas across a range of scenarios with varying numbers of clients and requests. Each configuration was executed multiple times to account for variability in results due to system performance or environmental factors. The following test configurations were used:

- **Number of Requests (p)**: 10,000, 20,000, and 30,000.
- **Number of Clients (c)**: 100, 500, and 1,000.

The system under test was executed on a single machine, with the simulation tool configured to use the same settings for all runs. The predicted time values were computed based on the derived formulas, and the actual simulation times were recorded for each configuration.

For clarity, we define T_{real} as the time measured for the actual implementation of the system, and $T_{simulated}$ as the value computed using our formulas.

6.2 Error Analysis and Accuracy Metrics

To quantify the accuracy of the simulation results compared to the predicted values, the following formula was used to calculate the percentage error:

$$\text{Error} = \frac{|T_{predicted} - T_{simulated}|}{T_{predicted}} \times 100$$

This error metric provides a measure of the deviation between the theoretical and experimental results, helping identify the scenarios where our formulas closely approximate the actual simulation time and where deviations occur. A lower error percentage indicates that the formulas are highly accurate, while a higher percentage suggests potential inefficiencies or inaccuracies in either the formula or the simulation model.

6.2.1 Analysis of Results

The results show that the formula for $T_{simulated}$ provides an accurate approximation of the actual simulation time in most cases. However, as the number of clients (c) increases, the error tends to rise slightly. This is likely due to the increased computational overhead of managing a larger number of processes in a single-machine environment. The kernel's scheduling and context-switching operations introduce additional delays not captured by the formulas, which assume an idealized processing scenario.

The experimental results are summarized in Table 6.1, which presents the predicted and actual simulation times, along with the calculated error percentages for each configuration.

Requests	**Clients**	$T_{predicted}$ **(sec)**	$T_{simulation}$ **(sec)**	**Error (%)**
10,000	100	60.40	58	3.97%
10,000	500	60.40	61	0.99%
10,000	1,000	60.40	62	2.64%
20,000	100	90.80	87	4.18%
20,000	500	90.80	90	0.88%
20,000	1,000	90.80	91	0.22%
30,000	100	121.20	114	5.94%
30,000	500	121.20	118	2.64%
30,000	1,000	121.20	122	0.66%

Table 6.1. Comparison of predicted and actual simulation times for varying numbers of requests and clients

6.2.2 Interpretation of Results

The results in Table 6.1 demonstrate that the formulas for $T_{simulated}$ perform well under most conditions, with an average error of less than 3% across all configurations. The highest error was observed for 30,000 requests and 100 clients, where the percentage error reached 5.94%. This suggests that the formula's accuracy may decrease slightly under higher loads when the number of clients is relatively small compared to the number of requests.

One possible explanation for this discrepancy is the impact of process management overhead, which is more pronounced when a small number of clients handle a large volume of requests. In such cases, the overhead introduced by context switching and I/O operations can lead to increased execution time, which is not accounted for in the simplified formulas used to derive $T_{simulated}$.

The validation of the formulas for $T_{simulated}$ confirms that they provide an accurate approximation of actual simulation times under a variety of configurations. While the formulas maintain high accuracy in most cases, further refinement may be needed to account for system-specific factors such as process management overhead and resource contention. Future work will focus on enhancing the formulas to incorporate these factors, improving the precision of simulation time predictions in complex scenarios.

Overall, the results demonstrate that the proposed approach is effective in estimating simulation times, enabling developers and architects to perform quick, reliable validations of system performance without the need to run exhaustive real-time tests.

Chapter 7

Conclusion and Future Work

In this thesis, we successfully designed and validated a new e-commerce system architecture focused on meeting essential non-functional requirements, particularly performance. Through rigorous experimentation and simulation, we established a framework to evaluate various architectural designs against expected client behavior, ultimately enhancing our understanding of how system performance can be optimized.

7.1 Summary of Results

The results demonstrated that our proposed architecture, characterized by multiple server replications and a DNS-based load balancer, effectively managed client requests. By employing a Markov chain to simulate customer interactions, we were able to capture the dynamic nature of user behavior and its impact on system performance. Key performance metrics, including latency, response times, and overall simulation speed-up, were analyzed under various configurations of clients and requests.

Notably, we observed that simulated-time simulations yielded significant speed-ups when the number of clients was limited, while real-time simulations became more efficient as client numbers increased. This insight allows architects and developers to strategically select testing methodologies based on system load, ultimately enhancing testing efficiency and reliability.

Additionally, our research established that achieving a target of 100 million requests with a speed-up of at least 10x necessitates approximately 2.9 million clients. This finding underscores the scalability potential of our architecture and emphasizes the importance of load management in high-demand scenarios.

7.2 Future Work

Despite the significant contributions of this research, several areas warrant further exploration to enhance the robustness and applicability of our findings:

- **Advanced Load Balancing Techniques:** Future work could investigate alternative load-balancing algorithms that adapt dynamically to changing workloads and client behaviors. This would improve resource allocation and further optimize response times.
- **Incorporation of Additional Non-Functional Requirements:** While this thesis focused primarily on performance, extending the analysis to include

other non-functional requirements such as reliability, scalability, and security could provide a more comprehensive evaluation of the system's capabilities.

- **Longitudinal Studies:** Conducting long-term performance assessments in real-world environments would help validate our simulation results and uncover any unforeseen issues that arise under sustained loads.
- **Multi-Cloud Deployments:** As cloud computing continues to evolve, examining the effects of deploying this architecture across multiple cloud platforms could provide insights into optimizing costs and performance while maintaining high availability.

7.3 Conclusion

In conclusion, this research successfully validated a novel e-commerce system architecture that meets crucial performance requirements through simulation and analysis. The insights gained from our experiments serve as a foundation for further developments in e-commerce systems, paving the way for more efficient architectures capable of handling increasing demand. By addressing the outlined future work, we can enhance the adaptability and robustness of our system, ensuring its relevance in the fast-paced landscape of digital commerce.

Acknowledgments

What started as an unexpected detour has now led me to this final chapter of my bachelor's degree. It's been a challenging journey, full of surprises, but I'm finally here.

First and foremost, I would like to express my deepest gratitude to my advisor, who always made himself available, and to my co-advisor for introducing me to this project.

I am profoundly grateful to my parents for their unwavering support, belief in my potential, and their sacrifices throughout this path, allowing me to pursue my own way and follow my aspirations.

To my university classmates, with whom I spent most of these years, thank you for the countless hours we shared, helping me grow not just technically but, more importantly, as a person.

To my closest friends, who have been there since kindergarten and middle school, thank you for being a source of comfort and for helping me take breaks from university life. Your support has meant the world to me.

Finally, I want to thank everyone who has helped me grow. Your support has made all the difference.

Bibliography

[1] Microsoft. Shadow Testing in Automated Testing. Available online: https://microsoft.github.io/code-with-engineering-playbook/automated-testing/shadow-testing/.

[2] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley-Interscience, 2000.

[3] cppreference.com contributors, *std::priority_queue*. Available: https://en.cppreference.com/w/cpp/container/priority_queue.

[4] Wikipedia contributors, "Big O notation", *Wikipedia, The Free Encyclopedia*, Available at: https://en.wikipedia.org/wiki/Big_O_notation

[5] cppreference.com contributors, *std::unordered_map*. Available: https://en.cppreference.com/w/cpp/container/unordered_map.

[6] Wikipedia contributors, "Non-functional requirement", *Wikipedia, The Free Encyclopedia*, Available at: https://en.wikipedia.org/wiki/Non-functional_requirement

[7] Amazon Web Services, Inc. *What is Load Balancing?*. Available at: https://aws.amazon.com/it/what-is/load-balancing/.

[8] J. D. Herniter and J. F. Magee, "Customer Behavior as a Markov Process," *Operations Research*, vol. 9, no. 1, pp. 105-122, Jan. - Feb., 1961.

[9] "MathWorks, "Markov Chain Analysis and Stationary Distribution," [Online]. Available: https://it.mathworks.com/help/symbolic/markov-chain-analysis-and-stationary-distribution.html.

[10] Vaishnavi Abirami, *Mastering Microservices Logging - Best Practices Guide*, August 29, 2024. Available: https://signoz.io/blog/microservices-logging/