# Real-Time Scheduling Simulator
# Project Report

Ahmet Enes Çiğdem 150220079

December 1, 2025

## Contents

## 1 Introduction

This project involves the design and development of a **Real-Time Scheduling Simulator**. The application is designed to simulate and visualize the execution of **Hard Real-Time Systems** containing both **Periodic** and **Aperiodic** tasks.

The primary objective is to analyze how different scheduling algorithms manage task execution under strict timing constraints and how specific Aperiodic Servers handle sporadic workloads without violating the deadlines of periodic tasks. The system provides a graphical interface (GUI) for configuration and a Gantt chart for visual verification.

## 2 System Architecture

The simulation is built using Python, utilizing **Tkinter** for the User Interface and **Matplotlib** for generating the Gantt charts. The core logic is divided into three main components:

### 2.1 Task Model

The system supports two types of tasks:

- **Periodic Tasks ($P_i$):** Characterized by Release Time ($r_i$), Execution Time ($C_i$), Period ($T_i$), and Relative Deadline ($D_i$).
- **Aperiodic Tasks ($A_i$):** Characterized by Arrival Time ($r_i$) and Execution Time ($C_i$). These tasks do not have hard deadlines but require a server mechanism to be executed alongside periodic tasks.

### 2.2 The Scheduler Engine

The scheduler operates on a **Time-Slicing** mechanism (Time Quantum $\Delta t = 0.01s$). This allows the system to:

- Handle non-integer timing (e.g., execution time of 1.5s).
- Detect "Deadline Misses" at the exact moment of failure.
- Support **Multi-Instance Concurrency**: If a periodic task's deadline is larger than its period ($D > T$), multiple instances of the same task can exist in the ready queue simultaneously without causing a false error.

### 2.3 Visualization

The output is a dynamic Gantt Chart that displays the task execution timeline, idle times, and visual error indicators. A red vertical line marks the exact time a deadline is missed, and the simulation aborts immediately to highlight the failure point.

## 3 Scheduling Algorithms Implementation

The simulator implements four standard real-time scheduling algorithms. Below are the logical descriptions and the corresponding Python implementation snippets used in the core engine.

## 3.1 Rate Monotonic (RM)

**Logic:** Priorities are assigned based on the **Period** ($T$). Shorter Period = Higher Priority.

**Implementation Code:**

```python
# Rate Monotonic: Sort by Period (T)
# 'candidates' is the list of ready tasks
if algo == 'Rate Monotonic (RM)':
    candidates.sort(key=lambda x: x.period)
    selected_task = candidates[0]
```

## 3.2 Deadline Monotonic (DM)

**Logic:** Priorities are assigned based on the **Relative Deadline** ($D$). Shorter Relative Deadline = Higher Priority.

**Implementation Code:**

```python
# Deadline Monotonic: Sort by Relative Deadline (D)
elif algo == 'Deadline Monotonic (DM)':
    candidates.sort(key=lambda x: x.deadline)
    selected_task = candidates[0]
```

## 3.3 Earliest Deadline First (EDF)

**Logic:** Priorities are driven by the **Absolute Deadline** ($d_{abs} = arrival + D$). The task closest to its deadline gets the CPU.

**Implementation Code:**

```python
# Earliest Deadline First: Sort by Absolute Deadline
elif algo == 'Earliest Deadline First (EDF)':
    candidates.sort(key=lambda x: x.current_abs_deadline)
    selected_task = candidates[0]
```

## 3.4 Least Laxity First (LLF)

**Logic:** Priorities are based on **Laxity** ($Laxity = d_{abs} - t_{current} - C_{remaining}$). A threshold mechanism is added to prevent thrashing (context switching too frequently).

**Implementation Code:**

```python
elif algo == 'Least Laxity First (LLF)':
    # Calculate Laxity
    def get_laxity(tsk):
        rem = tsk.current_job_rem
        return tsk.current_abs_deadline - current_time - rem

    candidates.sort(key=get_laxity)
    best_candidate = candidates[0]

    # Threshold Mechanism to prevent Thrashing
    # Only switch if the laxity difference is significant (> 0.1)
    if previous_selected_task in candidates:
        current_laxity = get_laxity(previous_selected_task)
        best_laxity = get_laxity(best_candidate)

        if (current_laxity - best_laxity) < self.llf_threshold:
            selected_task = previous_selected_task
        else:
            selected_task = best_candidate
    else:
        selected_task = best_candidate
```

## 4 Aperiodic Server Mechanisms

To handle aperiodic tasks without disrupting periodic tasks, the system implements three server types with a budget ($C_s$) and a period ($T_s$).

### 4.1 Background Server

- **Priority:** Lowest.
- **Logic:** Aperiodic tasks are executed **only** when the CPU is idle (no periodic tasks are ready).

### 4.2 Polling Server (Poller)

- **Replenishment:** At the beginning of every server period ($k \cdot T_s$), the budget is replenished to full capacity ($C_s$).
- **Critical Restriction:** If the server is scheduled to run but the **aperiodic queue is empty**, the server **immediately discards its budget** (Budget $= 0$) for the rest of that period.

### 4.3 Deferrable Server (DS)

- **Replenishment:** At the beginning of every server period ($k \cdot T_s$), the budget is replenished to full capacity ($C_s$).
- **Advantage:** Unlike the Poller, the Deferrable Server **preserves its budget**. If no aperiodic task is ready at the start of the period, the budget remains available. This significantly improves response time for sporadic arrivals.

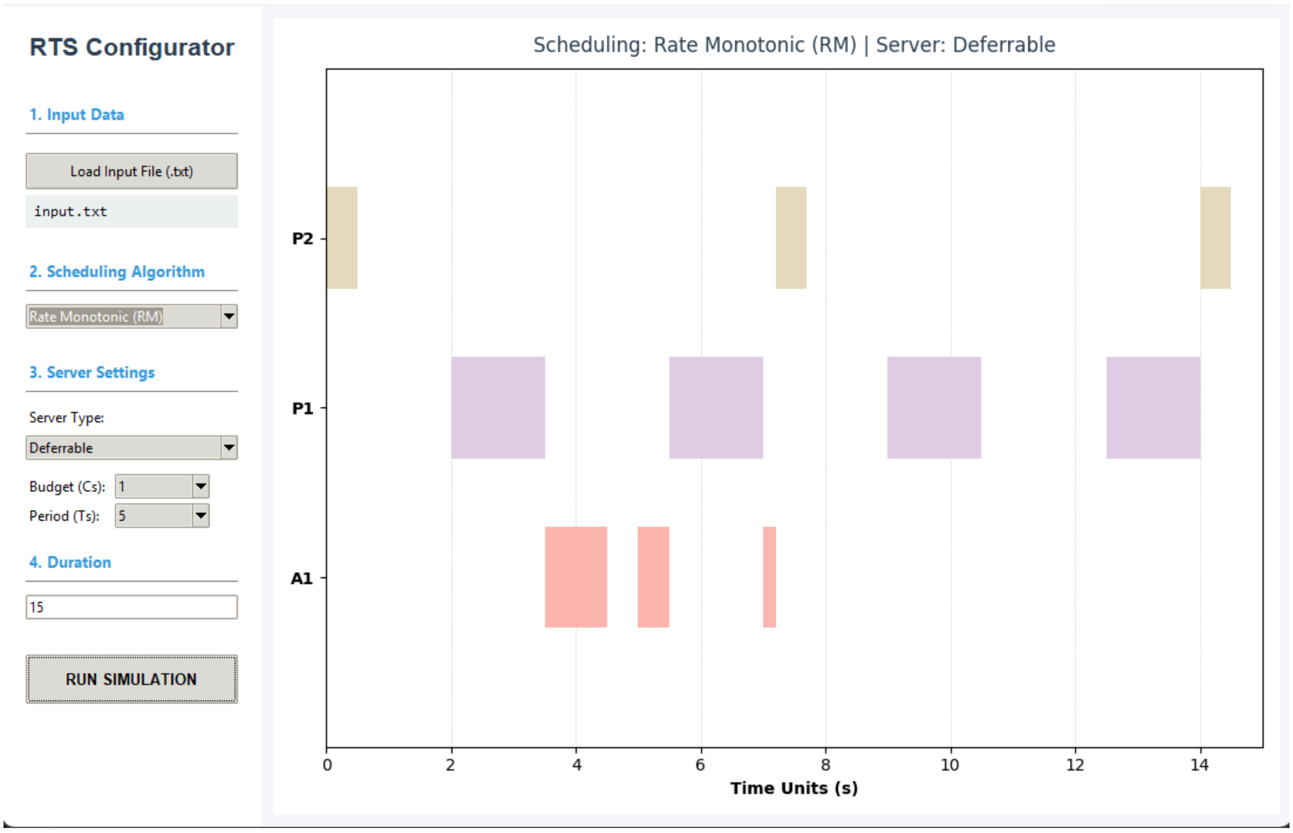
## 5 Experimental Results

In this section, we analyze specific scenarios derived from standard real-time scheduling literature. The screenshots below correspond to the simulation outputs for each case.

### 5.1 Case 1: Deferrable Server Effectiveness

**Scenario:** (Slide 3, Page 197)

- $P_1$: $C = 2, T = 3.5$
- $P_2$: $C = 0.5, T = 6.5$
- $A_1$: Arrival=2.8, $C = 1.7$
- **Server:** Deferrable Server $(Budget = 1, Period = 5)$


RTS Configurator

1. Input Data

Load Input File (.txt)

input.txt

2. Scheduling Algorithm

Rate Monotonic (RM)

3. Server Settings

Server Type: Deferrable

Budget (Cs): 1

Period (Ts): 5

4. Duration

15

RUN SIMULATION

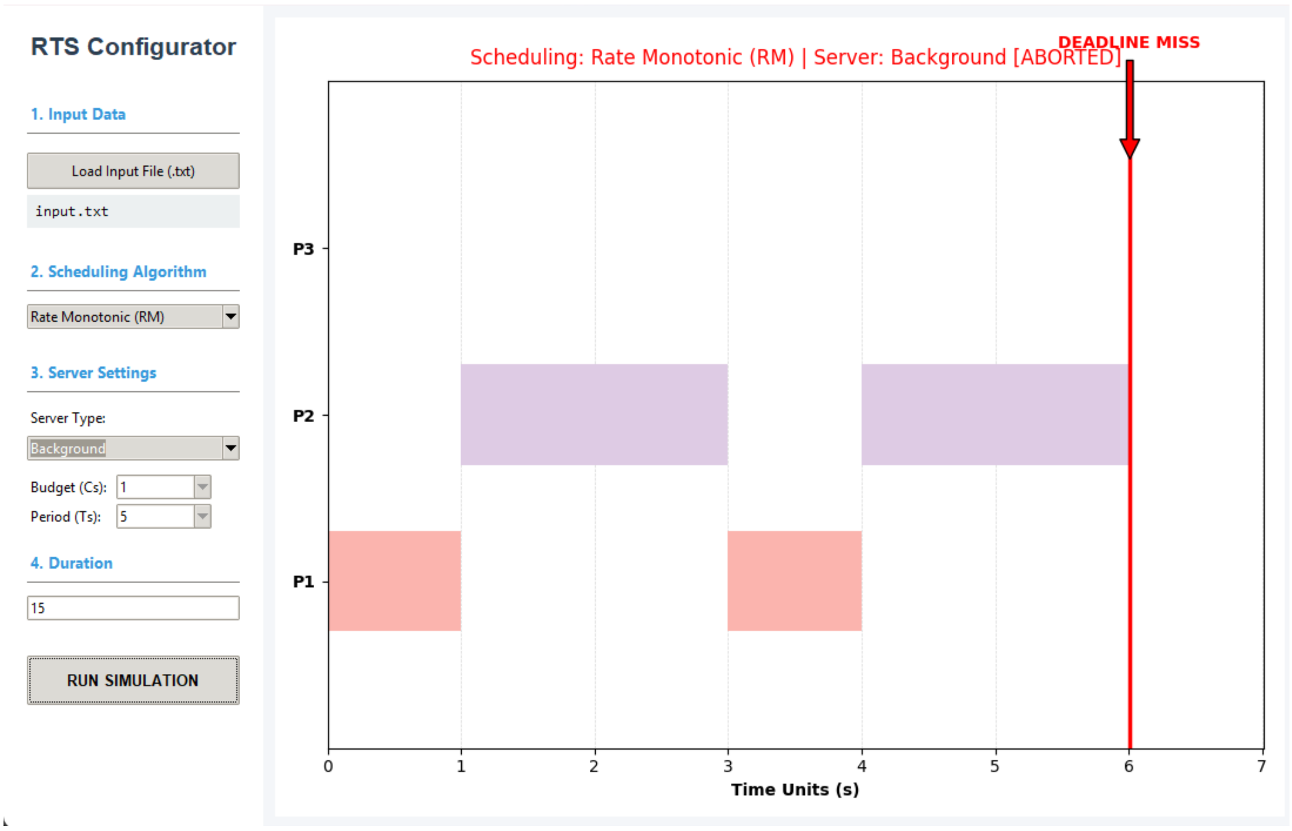
Figure 1: Rate Monotonic with Deferrable Server.

**Analysis:** This experiment demonstrates the advantage of the **Deferrable Server**. The Aperiodic task ($A_1$) arrives at $t = 2.8$. Since the Deferrable Server preserves its budget even if no task is ready at the start of the period ($t = 0$), it is able to service $A_1$ immediately upon arrival. This provides a significantly better response time compared to a Polling Server, which would have discarded its budget at $t = 0$.

## 5.2 Case 2: Rate Monotonic Deadline Miss

**Scenario:** (Slide 2, Page 24)

- $P_1$: $C = 1, T = 3$
- $P_2$: $C = 2, T = 4$
- $P_3$: $C = 1, T = 6$



RTS Configurator

1. Input Data

input.txt

2. Scheduling Algorithm

Rate Monotonic (RM)

3. Server Settings

Server Type: Background

Budget (Cs): 1

Period (Ts): 5

4. Duration

15

Scheduling: Rate Monotonic (RM) | Server: Background [ABORTED]

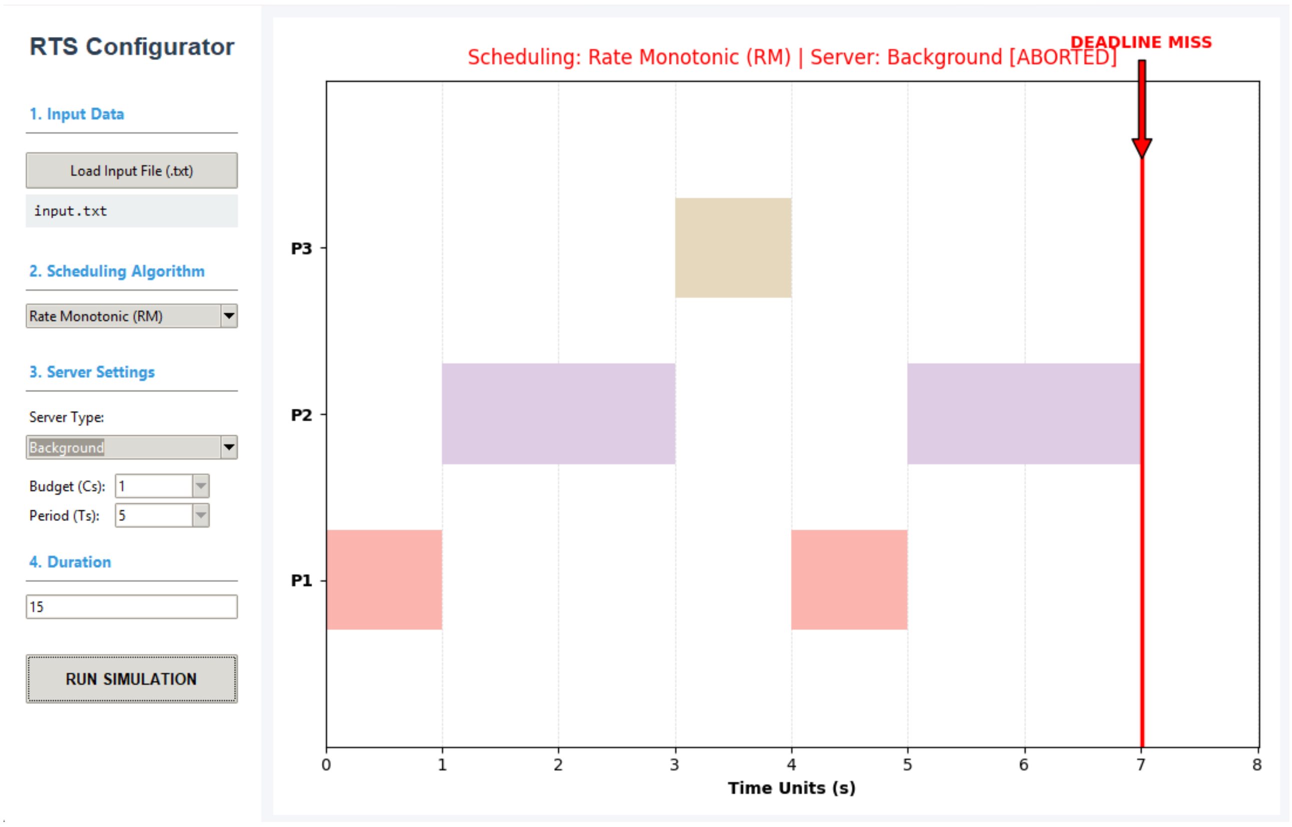
Figure 2: RM Schedule Aborted due to Deadline Miss at t=6.

**Analysis:** The total utilization is $U = 1.0$. While theoretically feasible under dynamic priority algorithms (like EDF), **Rate Monotonic (RM)** fails here. RM is a static priority algorithm ($P_1 > P_2 > P_3$). At the critical instant $t = 0$, higher priority tasks $P_1$ and $P_2$ consume the CPU. $P_3$, having the lowest priority, is starved and cannot complete its 1 unit of execution before its deadline at $t = 6$.

## 5.3 Case 3: Time Demand Analysis Failure

**Scenario:** (Slide 2, Page 25)

- $P_1$: $C = 1, T = 4$
- $P_2$: $C = 2, T = 5$
- $P_3$: $C = 2, T = 7$



RTS Configurator

1. Input Data

Load Input File (.txt)

input.txt

2. Scheduling Algorithm

Rate Monotonic (RM)

3. Server Settings

Server Type: Background

Budget (Cs): 1

Period (Ts): 5

4. Duration

15

RUN SIMULATION

Scheduling: Rate Monotonic (RM) | Server: Background [ABORTED]
DEADLINE MISS
P3
P2
P1
0
1
2
3
4
5
6
7
8
Time Units (s)

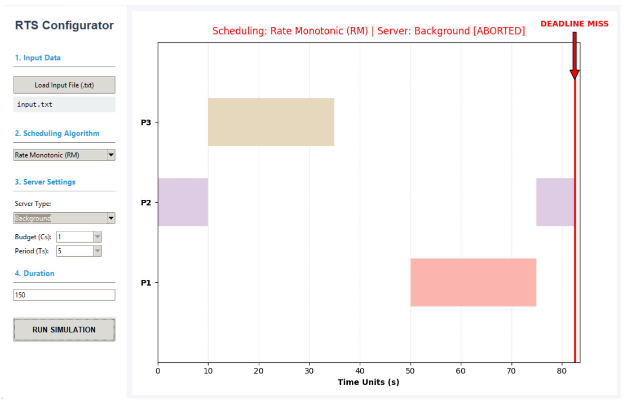
Figure 3: RM Failure detected around t=7.

**Analysis:** The utilization is $U \approx 0.93$. This is above the Liu & Layland bound for 3 tasks ($\approx 0.78$), so schedulability is not guaranteed. Time Demand Analysis (TDA) indicates that the demand $W_i(t)$ exceeds the available time $t$ for task $P_3$ near its deadline. The simulator visually confirms this theoretical result, showing $P_3$ missing its deadline at $t = 7$.

## 5.4 Case 4: Deadline Monotonic vs. Rate Monotonic

**Scenario:** (Page 120)

- $P_1$: $C = 50, T = 50, D = 100$ (Note: $D > T$)
- $P_2$: $C = 10, T = 62.5, D = 20$
- $P_3$: $C = 25, T = 125, D = 50$


RTS Configurator

1. Input Data

input.txt

2. Scheduling Algorithm: Rate Monotonic (RM)

3. Server Settings

Server Type: Background

Budget (Cs): 1

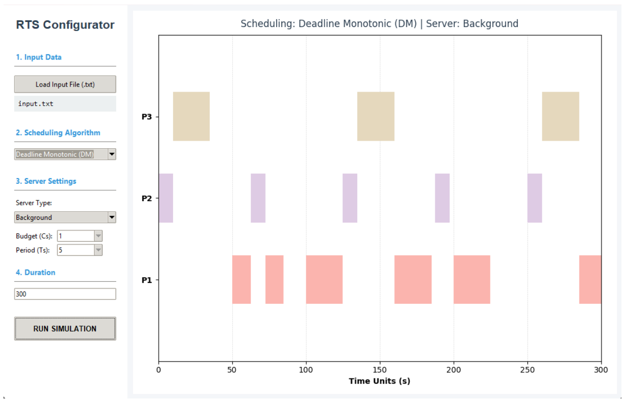
Period (Ts): 5

4. Duration: 150

Scheduling: Rate Monotonic (RM) | Server: Background [ABORTED]

DEADLINE MISS

(a) Rate Monotonic (Failed)


RTS Configurator

1. Input Data

input.txt

2. Scheduling Algorithm: Deadline Monotonic (DM)

3. Server Settings

Server Type: Background

Budget (Cs): 1

Period (Ts): 5

4. Duration: 300

Scheduling: Deadline Monotonic (DM) | Server: Background

(b) Deadline Monotonic (Success)

Figure 4: Comparison of RM vs DM.

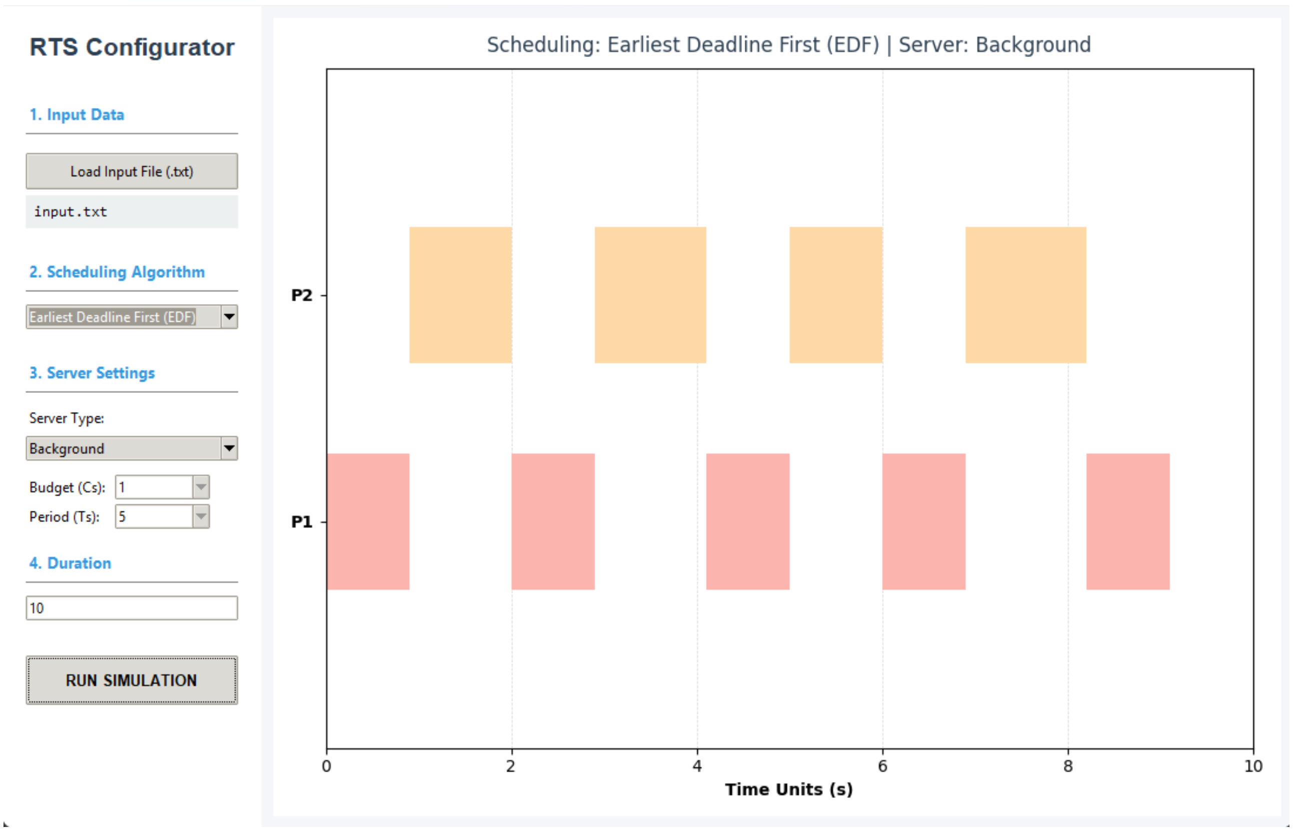
**Analysis:** This case highlights the superiority of **Deadline Monotonic (DM)** when Relative Deadlines ($D$) differ from Periods ($T$).

- **RM (Left):** Assigns highest priority to $P_1$ because it has the shortest period ($T = 50$), ignoring its large deadline ($D = 100$). This causes $P_2$, which has a very tight deadline ($D = 20$), to wait and miss its deadline.
- **DM (Right):** Assigns highest priority to $P_2$ because it has the shortest deadline ($D = 20$). This correct prioritization allows all tasks to meet their deadlines.

## 5.5 Case 5: Earliest Deadline First (EDF) Success

**Scenario:** (Page 120)

- $P_1$: $C = 0.9, T = 2$
- $P_2$: $C = 2.3, T = 5$



RTS Configurator

1. Input Data

input.txt

2. Scheduling Algorithm: Earliest Deadline First (EDF)

3. Server Settings

Server Type: Background

Budget (Cs): 1

Period (Ts): 5

4. Duration: 10

Scheduling: Earliest Deadline First (EDF) | Server: Background

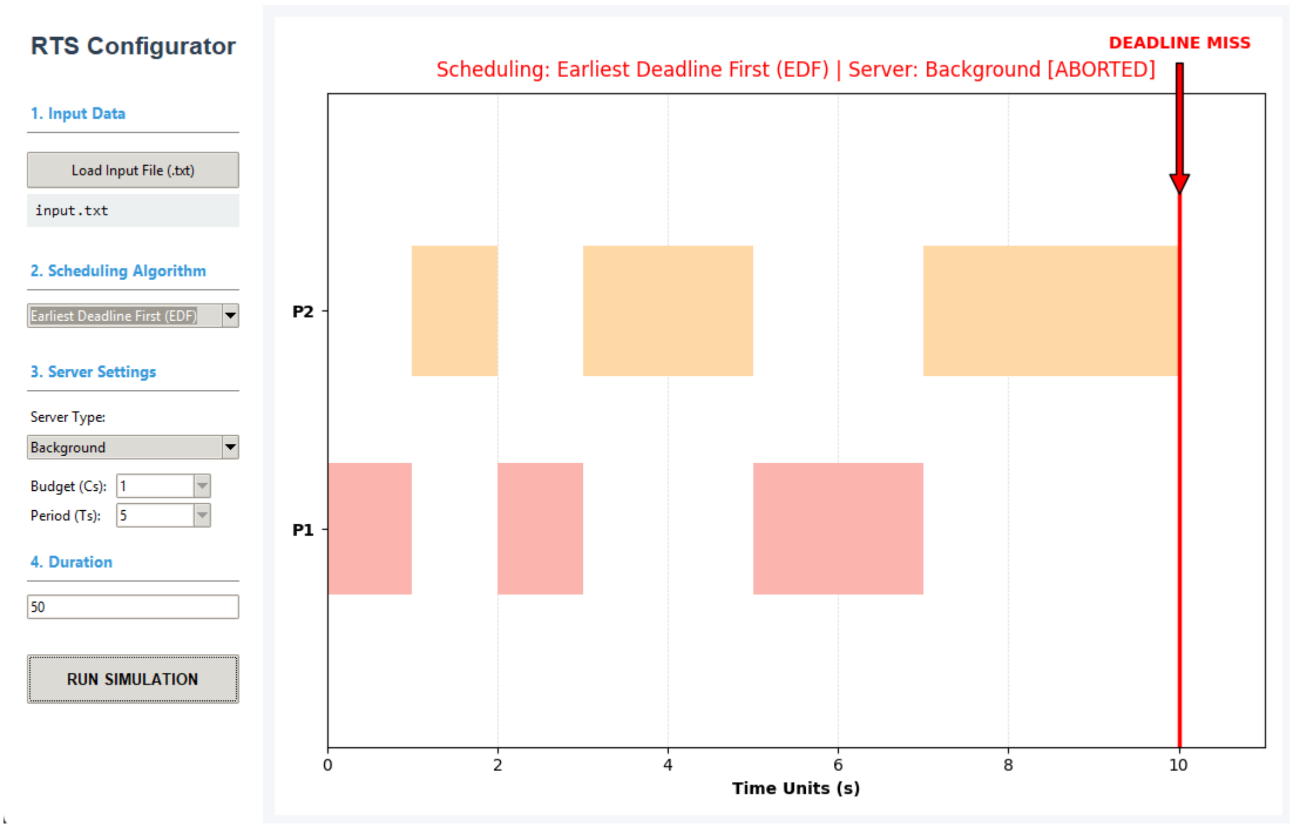
Figure 5: EDF scheduling successfully under high utilization.

**Analysis:** Total Utilization $U = 0.91$. Since $U \leq 1$, **EDF** is optimal and guaranteed to find a feasible schedule. The Gantt chart confirms that deadlines are met by dynamically adjusting priorities based on absolute deadlines, even under this high load where static algorithms might fail.

## 5.6 Case 6: System Overload (EDF Failure)

**Scenario:** (Page 123)

- $P_1$: $C = 1, T = 2$
- $P_2$: $C = 3, T = 5$



input.txt

Earliest Deadline First (EDF)

Server Type: Background

Budget (Cs): 1

Period (Ts): 5

50

Scheduling: Earliest Deadline First (EDF) | Server: Background [ABORTED]

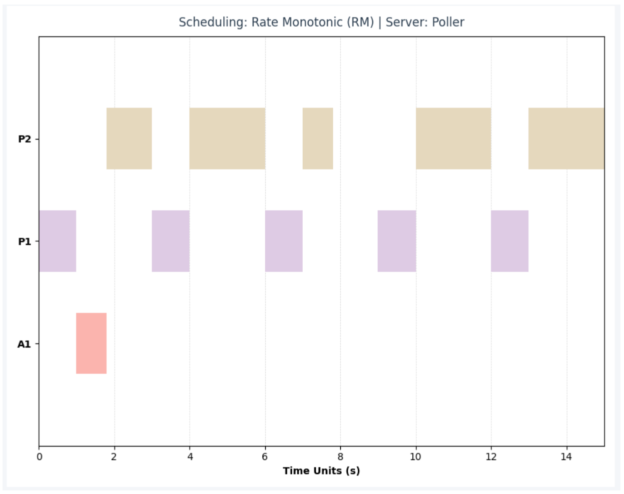
Figure 6: EDF aborts due to overload at t=10.

**Analysis:** Total Utilization $U = \frac{1}{2} + \frac{3}{5} = 1.1$. Since $U > 1$, the system is strictly **overloaded**. Even an optimal algorithm like EDF cannot tackle this. The simulation shows a deadline miss at $t = 10$, confirming that execution requirements exceed the available processor time.

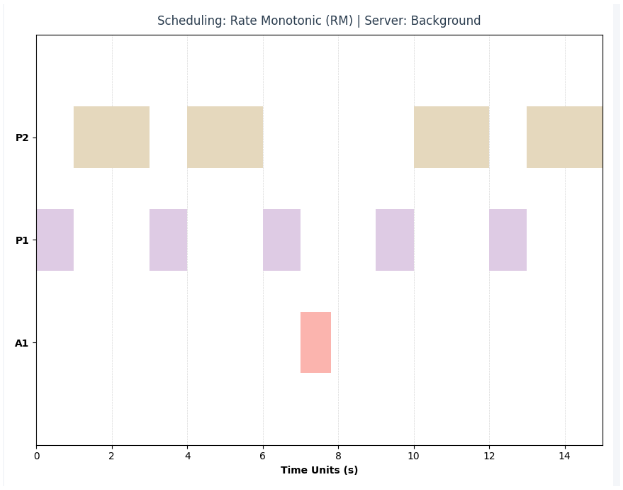
## 5.7 Case 7: Polling Server vs Background (Under RM)

**Scenario:** (Page 193)

- $P_1$: $C = 1, T = 3$
- $P_2$: $C = 4, T = 10$
- $A_1$: Arrival=0.1, $C = 0.8$



(a) Polling Server (RM)



(b) Background (RM)

Figure 7: Comparison: Poller vs Background under Rate Monotonic.

**Analysis:** This comparison illustrates the benefit of a Server mechanism under Rate Monotonic scheduling:

- **Polling Server (Left):** The server has reserved capacity. $A_1$ arrives at $t = 0.1$ and is serviced relatively quickly (around $t = 1$) using the server's budget.
- **Background (Right):** $A_1$ is treated with the lowest priority. It must wait until all periodic tasks are done. The chart shows $A_1$ is delayed significantly, executing much later (around $t = 7$) compared to the Poller approach.

# 6 Conclusion

The project successfully demonstrates the fundamental differences between fixed-priority and dynamic-priority scheduling. It highlights the importance of Server algorithms (Poller vs. Deferrable) in mixed-criticality systems. The simulation confirms that while algorithms like EDF are optimal for CPU utilization, they can be unpredictable under overload, whereas static algorithms like RM provide predictability at the cost of lower maximum utilization.