**Nida Nur Efe**
**150220025**

# 1 – Introduction

Executing tasks at the correct time is critical in real time systems. Scheduling periodic, aperiodic and server based tasks must be executed safely in critical environments.
This project includes a comprehensive Real Time Simulator developed in C++ and supported by a Qt based graphical interface. The project can run from both command line and the Qt GUI. It supports a wide range of scheduling policies.
Moreover, the project has been packaged using Docker to make it platform independent.

# 2 – Project Summary

The project consists of two main components:

## 2.1 – CLI (Command Line) Simulator

- Written in C++.
- Loads tasks by reading an input file.
- Applies the selected algorithm.
- Produces a timeline, Gantt-like output and a deadline miss list.

## 2.2 – Qt GUI Application

- Takes an input file from the user.
- Allows the user to select the simulation duration.
- Provides algorithm selection.
- Displays the results textually and graphically with Qt Charts

## 2.3 – Docker Environment

- A multi-stage build environment containing Qt, C++, CMake dependencies has been created.
- Both the CLI and GUI are built within Docker.
- GUI forwarding can be done to the host screen.

# 3 – Supported Scheduling Algorithms

## 3.1 – Periodic Algorithms

- Earliest Deadline First
- Rate Monotonic Scheduling
- Deadline Monotonic Scheduling
- Least Laxity First

## 3.2 – Aperiodic Task Support

- Background Scheduling

### 3.3 – Server Based Algoritihms

- Polling Server
- Deferrable Server
- Sporadic Server

These servers work in conjunction with periodic tasks to provide a CPU budget for aperiodic tasks.

### 3.4 – Customizability of Rules

Rules for all server mechanisms:

- Budget consumption rule
- Replenishment rule

Rules can be defined in the settings.json file. This structure makes the project extensible.

## 4 – Project Architecture

rt-scheduler/

```
│
├── src/
│   ├── models.hpp
│   ├── parser.hpp
│   ├── policies.hpp
│   ├── sched_base.hpp
│   ├── sched_servers.hpp
│   ├── factory.hpp
│   └── main.cpp
│
├── gui/rt_gui/
│   ├── mainwindow.cpp
│   ├── mainwindow.h
│   ├── mainwindow.ui
│   └── CMakeLists.txt
│
├── settings.json
├── Dockerfile
└── examples/
```

### *Features*

- models.hpp: Holds task, job, and server structures.
- parser.hpp: Provides decimal input support and converts to integer schedules.
- sched_base.hpp: Basic periodic scheduler class.
- sched_servers.hpp: Implementation of server mechanisms.
- factory.hpp: Algorithm selection and scheduler creation.
- Qt GUI: A user-friendly scheduling interface.

## 4.1 – General Flow

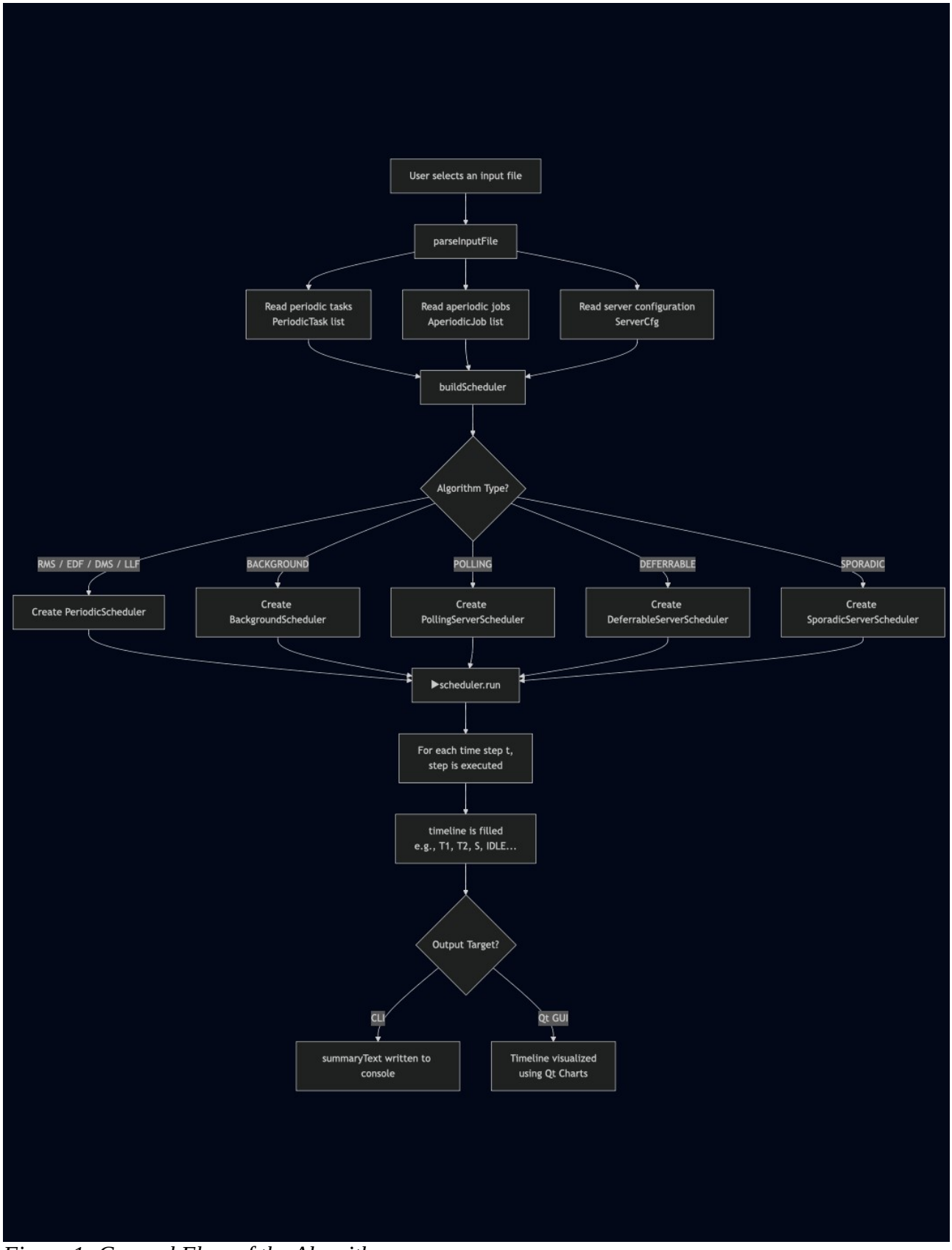


*Figure 1: General Flow of the Algorithm*

**1.** The user provides an input file (the CLI asks for a path, and Qt selects it with browse).

**2.** parseInputFile(...) reads the file and creates:

- Periodic Tasks (PeriodicTask)
- Aperiodic Jobs (AperiodicJob)
- Server Configuration (ServerCfg)

**3.** buildScheduler(...) looks at the selected algorithm name and creates the appropriate scheduler object:

- PeriodicScheduler
- BackgroundScheduler
- PollingServerScheduler, DeferrableServerScheduler, SporadicServerScheduler

**4.** scheduler->run() is called:

- For each time step (t), step(t) runs.
- Timeline is filled as a std::vector<std::string> (e.g., ["T1", "T1", "S", "IDLE", ...])

**5.** Output:

- CLI: SummaryText() string is written.
- Qt: Timeline() is sent to Qt Charts, and a colored chart is drawn.

## 4.2 – models.hpp – Data Models

```
struct PeriodicTask {
    std::string name;
    int arrival;
    int execTime;
    int period;
    int deadline;
};
```

*Figure 2: Periodic task data model*

Classic parameters for a periodic task:

- arrival: first arrival time
- execTime: execution time of each job
- period: period
- deadline: relative deadline (often the same as period)

```
struct PeriodicJob {
    const PeriodicTask* task;
    int releaseTime;
    int remaining;
    int absDeadline;
    std::string id;
};
```

*Figure 3: Periodic job data model*

The scheduler runs the job, not the task itself.

Job = instance created at a specific release time:

- task: which task it belongs to
- releaseTime: the time this job was created
- remaining: the remaining run time
- absDeadline: absolute deadline = releaseTime + deadline

```
struct AperiodicJob {
    std::string name;
    int releaseTime;
    int execTime;
    int remaining;
};
```

*Figure 4: Aperiodic job data model*

Aperiodic jobs are jobs served by the server or background.

```
struct ServerCfg {
    int Q;
    int T;
    int D;
};
```

*Figure 5: Server config data model*

Server parameters:

- Q: budget (how many ticks can run)
- T: period (budget renewal period)
- D: deadline (often taken as the same as T)

## 4.3 – parser.hpp – Reading the input file

File lines are read with std::getline. After the # sign, it is considered a comment and discarded. Empty lines are skipped. All numbers are first read as double and then converted to integer tick.

### *Comment Lines*

Lines starting with `#` are considered comments and are completely ignored.

### *P lines*

**P ri ei pi di**

Defines a periodic task.

- ri: task's initial arrival time (release time)
- ei: execution time
- pi: period
- di: task's relative deadline

**P ri ei pi**

When no deadline is given, the relative deadline is assumed to be equal to the period: di = pi.

**P ei pi**

When no arrival time is specified, ri = 0 is assumed, and the deadline is again taken to be equal to the period: di = pi.

All three of these forms are converted to the same `PeriodicTask` data structure. Missing parameters are filled in using the above assumptions.

### *D lines*

**D ei pi di**

Defines the server configuration required for server based algorithms (Polling Server, Deferrable Server, Sporadic Server). The parameters in this line are internally converted to the ServerCfg structure:

- ei -> server execution budget Q
- pi -> server period T
- di -> server relative deadline D

### *A lines*

**A ri ei**

Defines aperiodic jobs that can be used by the background scheduler and server-based algorithms.

- ri: job arrival time
- ei: execution time

The application operates according to the integer based time axis requested in the assignment.

If decimal values are given in the input file, these values are first read and rounded to the nearest integer before being transferred to the system. This preserves the format specified in the presentation and provides the user with more flexible input.

## 4.4 – policies.hpp – Algorithm Logic and Priority

The scheduler uses the policy_->key(job, t) value when deciding which job to run. A smaller value means a higher priority.

- RMS: key = task.period
- DMS: key = task.deadline
- EDF: key = job.absDeadline
- LLF: key = (job.absDeadline - t - job.remaining) (laxity)

## 4.5 – sched_base.hpp – Periodic Scheduler

The entire timeline is initially filled with "IDLE". At each tick,  if t >= arrival and (t - arrival) is a multiple of the full period, create a new PeriodicJob and add it to the ready_ list.

Jobs that have passed the deadline but still have remaining > 0 are moved to the missed_ list.

The scheduler looks at all ready jobs and selects the smallest one with policy_->key(...).

Step by step:

1. New jobs are created.
2. Jobs that have passed the deadline are moved to "missed."
3. The job to be run is selected.
4. The remaining time of the selected job is reduced by 1.
5. T1, T2, etc. are written to the timeline.
6. If the job is finished, it is added to "finished_" and removed from the "ready" list.

## 4.6 – sched_servers.hpp – Server Mechanisms

A common BaseServerScheduler is defined and servers are inherited from it. It starts PeriodicScheduler with an empty task list and creates a "server" task named serverTask_:

- ExecTime = Q
- Period = T
- Deadline = D

Then it puts the regular periodic tasks + serverTask into tasks_.

In step(t) override,

1. Normal periodic jobs are released/checked first.
2. The server job competes with other periodic jobs, such as RMS/EDF, etc.
3. If the selected job is serverTask_:
    - If the budget is > 0 and aperiodic ready is available, run the aperiodic job.
    - If there is no budget or job, remove the server from the ready list and select another job.
4. Otherwise, run the normal periodic job.

### *4.6.1 - Polling Server*

At the beginning of each period (t % T == 0), it checks:

- If there is aperiodic work: serverBudget_ = Q
- Otherwise: budget 0 -> that period is wasted and budget cannot carried over.

### *4.6.2 - Deferrable Server*

It is reset to Q at the beginning of each period.

- Unused budget can be carried forward (deferred) within the period.

### *4.6.3 - Sporadic Server*

Each consumed unit of budget is recorded to be replenished after T.  Thus, the Q/T ratio is maintained on average.

## 4.7 – factory.hpp – Producing Correct Scheduler

algName is converted to uppercase. If:

- “RMS”, “DMS”, “EDF”, “LLF” -> PeriodicScheduler
- “BACKGROUND” -> BackgroundScheduler
- “POLLING” -> PollingServerScheduler
- “DEFERRABLE” -> DeferrableServerScheduler
- “SPORADIC” -> SporadicServerScheduler

## 4.8 – main.cpp – CLI Flow

1. Input path is retrieved.
2. parseInputFile() is called.
3. Hyperperiod is calculated, the user is prompted for the simulation time (hyperperiod if left blank).
4. Algorithm name is requested.
5. Scheduler is created with buildScheduler(...).
6. run() and summaryText() are called, and the result is printed to the screen.

# 5 – Timeline Output and Results

The simulator produces the following outputs:

- Timeline (seconds-based)
- Gantt-like ASCII output
- Completed jobs
- Deadline miss list

This project brings together a wide range of scheduling algorithms used in real-time systems and makes them executable from both the command line and a graphical interface.