Lappeenranta University of Technology

School of Engineering Science

Degree Program in Computer Science

Tuomo Harjula

DESIGN AND IMPLEMENTATION OF A TASK SCHEDULING SERVICE

Examiners:	Professor Jari Porras
	M.Sc. Arto Hämäläinen
Supervisors:	Professor Jari Porras
	M.Sc. Arto Hämäläinen

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Tuomo Harjula

Design and implementation of a task scheduling service

Master's Thesis

2019

31 pages, 5 figures, 2 tables

Examiners: Professor Jari Porras
M.Sc. Arto Hämäläinen

Keywords: task scheduling, queue optimization, scheduling algorithm

Computing server resources are limited and the usage should be optimized. Literature review was conducted to identify various aspects of a scheduling service. Multiagent scheduling was identified as a solution to the problem of scheduling heterogeneous tasks using shared and limited resources. Unfairness, resource monitoring and load balancing were considered. The current systems were introduced and the requirements for the service identified. A scheduling service and scheduling algorithm were designed and implemented. A prototype of the service was built. The findings, insufficiencies of the design and built prototype were discussed.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Tuomo Harjula

Design and implementation of a task scheduling service

Diplomityö

2019

31 sivua, 5 kuvaa, 2 taulukkoa

Työn tarkastajat:	Professori Jari Porras
	M.Sc. Arto Hämäläinen

Hakusanat: tehtävien ajastus, jonon optimointi, aikataulutus algoritmi

Tietokonepalvelinten resurssit ovat rajalliset ja niiden käyttö pitäisi olla optimoitua. Suoritettiin kirjallisuuskatsaus ja aikataulutuspalvelun eri näkökulmat tunnistettiin. Usean agentin ajautus tunnistettiin ratkaisuna sekalaisten tehtäväryhmien aikatauluttamiseen jaetuilla resursseilla. Epäreiluutta, resurssien tarkkailua ja kuormantasausta harkittiin. Nykyiset järjestelmät esiteltiin ja uuden aikataulutusjärjestelmän vaatimukset tunnistettiin. Aikataulutusjärjestelmä ja aikataulutusalgoritmi suunniteltiin ja toteutettiin. Palvelusta toteutettiin prototyyppi. Löydöksistä, puutteista ja toteutuneesta prototyypistä keskusteltiin.

ACKNOWLEDGEMENTS

I would like to acknowledge my supervisors and coworkers for being supportive and providing critical feedback that enabled me to make this thesis.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application programming interface
CMS	Configuration management solution
CPU	Central processing unit
DEP	Device enrollment program
EDD	Earliest due date
EDF	Earliest deadline first
FIFO	First-in, first-out
HTTP	Hypertext transfer protocol
IT	Information technology
MDM	Mobile device management
MSMQ	Microsoft Message Queuing
OS	Operating System
QoS	Quality of service
RR	Round robin
SPT	Shortest processing time
TPL	Task Parallel Library
WSPT	Weighted shortest processing time

1 INTRODUCTION

In real world scenarios, server resources are limited and their usage should be optimized. Software should aim to optimize resource utilization for multiple reasons, mainly to avoid performance bottlenecks and to minimize costs. Task scheduling is needed to maximize throughput and to optimize resource usage. The purpose of such a service is to sort and queue incoming tasks and execute them in an optimized manner depending on variables such as required quality of service (QoS) and priority of tasks. To identify the main concerns that need to be considered when building such a service, a thorough investigation of existing literature on relevant topics needs to be conducted.

A schedule is a combination of feasible allocation of activities using given resources. The quality of a schedule is measured by optimality criterion such as minimizing completion time, resource usage and keeping due dates. Scheduling problems are problems where the task is to find optimal combination of heterogeneous tasks that achieve specified criterion. (Agnetis et al. 2014, 2) Environment monitoring is necessary for adapting to dynamically changing resource levels (Porras et al. 2009). Available resources need to be monitored and regulated to optimize throughput of tasks and to keep the load stable.

1.1 Background

This Master's thesis is conducted for a company that provides information technology (IT) asset and system management solutions to enterprises. The company has two main products: a cloud based mobile device management (MDM) solution and a configuration management solution (CMS). The MDM solution provides enterprises with device management functionalities such as application and configuration profile deployment. CMS offers enterprises a comprehensive IT asset lifecycle management including software asset, license, incident and security management capabilities.

The need for such a service originated from having suboptimal solutions in place to existing scheduling problems and partially from the absence of task scheduling in some parts of the company's current MDM product. Instead of implementing heterogeneous and disconnected scheduling in various locations, the idea for a centralized task scheduling service came forth.

The goal is to provide a shared task scheduling service for heterogeneous demands of various applications. To make the development of future products easier, an external service could be a viable solution for handling scheduling jobs. Monitoring of external resources is also a large part of the scheduling service. Company’s current solutions do not support this kind of functionality.

The service should also be able to accommodate dynamic changes to the schedules, such as the addition of new tasks, within a sufficient time span. Use cases include and are not limited to scheduling mobile device wake-ups, application deployments to devices, and creating and sending of periodical, instance specific reports.

The task scheduling service, or simply *the service*, is to be used as part of a product that works in a cloud environment. Cloud architectures represent a seemingly infinite amount of resources (Ould Deye et al. 2013). Despite the ample amount of potential resources, the service’s goal is to minimize overall resource usage and moderate load on resources.

1.2 Goals and delimitations

The goal of this thesis is to design and implement a prototype of a task scheduling service that can be used in various existing and future products. The idea is to make solid design decisions based on existing literature and internal experts’ opinions to enable designing a dynamic and portable scheduling service. This thesis focuses on the various aspects of such a service and provides argumentation on why particular design decisions were made and why they are better suited for the service in the given context than the alternatives.

Research questions are defined as following:

1. What has to be taken into consideration when designing a task scheduling service?
2. How to optimize performance of available resources?
3. How to avoid unfairness between different types of tasks?

Topics that have to be taken into consideration when designing a task scheduling service are investigated by conducting a literature review on existing research that addresses similar problems. Main concerns are disclosed when analyzing reviewed literature and by interviewing company’s internal field experts. A design for the service is crafted based on the resulting analysis and a prototype of the service is built. The prototype is created as a proof of concept for the usability of the scheduling and execution.

Various usage scenarios of the service are explained and their requirements and main challenges are reviewed. Their inner workings are only mentioned and are not discussed further. The goal is to keep the service sufficiently generic for extensibility to support new types of tasks.

The aim of this thesis is to build a design for a centralized scheduling service. The service focuses on optimizing limited resources such as central processing unit (CPU) usage and time. Distributed computing is a theme that appeared frequently during the literature overview process. Since scheduling in distributed computing deals with scheduling in a decentralized environment, it is excluded from the scope of this thesis.

Chapter 1 introduces the background and defines research questions. In chapter 2 a literature review is conducted on preceding research on similar problems and the various things that need to be taken into consideration when designing a task scheduling service are introduced. Chapter 3 describes the current systems, its issues and needs. Chapter 4 presents design and implementation of the scheduling service. Chapter 5 discusses thesis' findings and future research areas. Chapter 6 gives a conclusion of the thesis and its results.

2 PRECEDING RESEARCH

This section introduces the main preceding research findings on the topics that need to be decided on in order to design an effective task scheduling service. Prominent scheduling algorithms that could be taken into use in the new service are presented.

Literature review process begun with test searches using keywords that were identified in the beginning, or were identified in initial meetings with coworkers. These include terms such as *scheduling service, queue optimization, load balancing, concurrency, scalability* and *background job*. Keywords related to technologies in use were also identified: *HTTP, API* and *web service*. Main keywords were recognized and more literature was searched for using them. Found literature was reviewed for acquiring a better and larger understanding of the topic under scrutiny. Additional keywords were identified within found literature and more searches were conducted.

Discussions were held with co-workers to identify the main requirements for the service and additional keywords were identified. The interviews were conducted based on literature review findings and identified concepts were discussed openly. The experts were queried for their opinions on topics they thought are important in the context of the task scheduling service. New concepts were identified through this process. Discussions with co-workers led to the identification of multiple technical aspects concerning the service. The concept of multiagent scheduling was approached in the meetings, when considering how the various tasks should be grouped in the service. The current implementation environment provides specific restrictions that need to be considered in the implementation phase. Various alternative technical approaches were considered, such as using events instead of scheduling and if the existing codebase be used for building the service.

Scheduling theory has been discussed widely in books and other scientific literature since 1950s. Scheduling problems are combinatorial optimization problems where limited quantities of resources are to be shared among multiple activities in an optimized manner. Agnetis et al. (Agnetis et al. 2014) introduces multi-criteria and multi-agent scheduling problems and algorithms. Brucker et al. (Brucker et al. 2005) writes about complex scheduling problems and presents heuristic methods for solving such problems. Madni et al.

(Madni et al. 2017) compared different heuristic algorithms on various optimality criterion such as makespan and throughput for task scheduling in a cloud environment. Maurya et al. (Maurya et al. 2018) benchmarked some well-known scheduling algorithms for heterogeneous computing systems with multiple cores where applications can run different tasks concurrently. They also proposed a general framework for benchmarking task scheduling algorithms in heterogeneous computing systems.

2.1 Optimality

The quality of a schedule is measured by optimality criterion that are often based on total completion time of the schedule. A key issue in multiagent scheduling algorithms is the determination of non-dominated (or Pareto optimal) schedules. Pareto optimal schedule refers to a schedule that is globally optimal with respect to the optimality criterion, without sacrificing one agent's schedule over another. (Agnetis et al. 2014, 2) Each agent aims for a locally optimal schedule, which most often is globally sub-optimal at best, since it may be unfair in relation to other agents. Optimized schedule must resolve from a decision process where all agents are involved and must not result in a blockage of certain agents.

Optimality criterion implies to various metrics that can be used to measure the optimality of an algorithm. Agnetis et al (Agnetis et al. 2014, 9) lists the most important objective criterion in multi-agent scheduling problems as *makespan, maximum lateness, maximum tardiness, total (weighted) completion time, total (weighted) tardiness* and *(weighted) number of tardy jobs*. For the optimization algorithm used in the service, the most important task is to minimize the total completion time of tasks.

Due dates are important in QoS critical systems where waiting can cause significant problems. The service will not have functionality to support due date tasks. This means that a task can only have a scheduled start date; a deadline for the completion of a task cannot be defined. This will simplify the required scheduling algorithm, since it removes an additional element from the complexity. The MDM solution in general does not support defining a due date for functions; only the configuration of scheduled start date is supported. Most schedulable tasks in the system need only to start on the defined date; they do not need to be finished by a certain date.

2.2 Multiagent scheduling

Research on multiagent scheduling started approximately 10 years ago. Multiple agents interacting to perform their respective tasks while sharing a common resource pool on which the tasks are executed on is called multiagent scheduling (Agnetis et al. 2014, 3.) Multiagent scheduling approach was chosen for the thesis because it closely resembles the problem at hand, where various heterogenic tasks compete for limited amount of resources. The problem of handling heterogeneous group of tasks within a single service introduced its own limitations in terms of complexity.

Agnetis et al. (Agnetis et al. 2004) analyzed a scenario where two agents, each with their own set of jobs, schedule on a common resource while minimizing their own cost functions. They tackled the problem of computing schedules for sharing common resources in optimized way by first determining the best schedule for one agent and then determining set of pareto-optimal schedules for both agents. Agnetis et al. laid out a way to generate a minimum set of alternative schedules to provide a basis for negotiation between agents in pareto-optimization problems. Lee et al. (Lee et al. 2010) studied single-machine scheduling problems with two agents for minimizing total weighted completion time of jobs with linearly deteriorating jobs. They proposed a branch-and-bound algorithm and three different heuristic algorithms. Computational experimentations were conducted for the algorithms and the findings showed that the branch-and-bound algorithm could optimize up to 20 tasks in a reasonable amount of time, and one of the heuristic algorithms had an error percentage of less than 1.64%. Cheng et al. (Cheng et al. 2006) studied feasibility of minimizing total number of tardy jobs in multi-agent scheduling. They presented a fully polynomial-time approximation scheme for the problem. Rabelo et al. (Rabelo et al. 1999) discussed various multi-agent approaches to agile scheduling and how effective a negotiation-based aspect of multi-agent scheduling is to the development of agile scheduling systems. They built a prototype and showed that the approach is feasible in case of shop floor scheduling. Polanczyk (Polanczyk, 2012) extended semantics of scheduling priorities, where the writer argued that including priority over shared resources could lead to less contention for resources.

Agnetis et al. (Agnetis et al. 2014, 13-14) classifies the different types of sharing resources between two agents into four distinct categories: competing agents, interfering sets, multi-criteria optimization and non-disjoint sets. Competing agents purely compete with each other for resources. In interfering sets, tasks are nested and need to be executed in a specific order. In multi-criteria optimization, which is a special case of interfering sets, allocation of resources happens based on multiple criterion. Non-disjoint set is the most general case, where two tasks may or may not intersect with each other. Set is disjoint if they have no common elements and are not dependent on each other.

2.3 Unfairness

Ajtai et al. (Ajtai et al. 1998.) explored fairness in scheduling in a scenario where multiple long-lived processes exist and the processes need to be scheduled repeatedly for various tasks across their lifecycle. Ajtai et al. studied the problem from the perspective that a process has a desired load level. In the context of the service, unfairness refers to the act of a specific agent reserving a larger quantity of resources for its own purposes while introducing the possibility of blocking other agents from executing due to lack of resources available to them.

Measures against unfairness should to be implemented in the scheduling algorithm to avoid situations where an agent blocks another agent’s execution for long periods of time. An example of such a rule is introducing minimum/maximum resource quotas to each agent which would promote fair treatment.

2.4 Job deterioration

Job deterioration refers to situations where the processing time of a job depends on its starting time in the queue (Alidaee et al. 1999.) For example, a fire-fighting situation where the fire spreads the longer it takes to be put off. Alidaee et al. reviewed existing research on scenarios where the processing time of a job is unknown and deteriorating. They found out that when the number of jobs is large, processing times can grow rapidly and suggested that this can be mitigated by increasing the amount of resources.

Mosheiov (Mosheiov, 2005) studied job deterioration when a task’s processing time is dependent on its position in the queue using exponential deteriorating function. He noted that the structure of an optimal schedule in job-position deterioration was similar to waiting-time deterioration. Using the exponential deteriorating function, Mosheiov found that an optimal schedule is V-shaped in respect to job processing times, favoring large jobs at the beginning of the schedule.

In the service, a task’s processing time is not dependent on its starting time, although the total makespan of the task is dependent on its position in the queue. Job deterioration can be ruled outside the consideration for the service.

2.5 Scheduling problem

Scheduling a heterogeneous group of tasks in an optimal sequence depends largely on the optimality criterion that needs to be fulfilled by the algorithm. For example, if the aim is to minimize total completion time or tardiness, the algorithm needs to adjust to meet the requirements of the scheduling.

Various scheduling algorithms exist that could be used in the scheduling scenario. Weighted Shortest Processing Time First (WSPT) is a scheduling algorithm where tasks are sorted in non-decreasing order by their weights and completion time estimates. Shortest Processing Time (SPT) is a simplified version of WSPT, where tasks are sorted according to their completion times. Both SPT and WSPT require accurate completion time estimates for maximum profit. In Round Robin (RR) scheduling algorithm, each process or task is given a slice of time in a circular order.

One of the most common cases of scheduling problems in literature is for agents to try and minimize their own total completion time. The problem can be classified as both non-disjoint set and multi-criteria optimization. It is considered NP-hard even when agent weights are not considered. (Agnetis et al. 2004) Lee et al. (Lee et al. 2009) proposed several multi-agent scheduling approximation algorithms for solving a weighted scheduling problem of minimizing total completion time.

2.6 Load balancing

In cloud environment, load balancing refers to the process of distributing load among virtual machines so that it improves execution ratio and resource utilization (Soni et al. 2018.) In the context of the service, the emphasis is on mitigating server load by limiting throughput. Monitoring server resources enables the service to react dynamically to load changes on the server. Thus, the service can either increase or decrease throughput of tasks to minimize resource throttling.

Optimally tasks that come through the service should only utilize a portion of a processing server's resources without reducing server's throughput to other processes. Task's utilization of server resources should be minimal. In case the server's capacities are below required level and the quantity of tasks grows faster than they can be processed, the processing servers' capacity needs to be increased.

Earliest Deadline First (EDF) is a dynamic scheduling algorithm where the task closest to its deadline is scheduled first. In contrast, in Earliest Due Date (EDD) scheduling tasks are ordered in non-decreasing order according to due dates. Casini et al. (Casini et al. 1998) proposed a semi-partitioned scheduling approach to scheduling real-time dynamic workload in a multiprocessor system. The proposed approach is simpler to implement and less heavy than global scheduling. Gracioli et al. (Gracioli et al. 2013) compared global and partitioned EDF strategies in scheduling in real-time environments with hard real-time tasks, where deadlines must be always met. The experiments showed that in hard real-time systems, partitioned EDF is always better than global EDF.

Ould Deye et al. (Ould Deye et al. 2013) suggested an approach to make load-balancing more dynamic to better suit QoS requirements in multi-instance cloud applications. The given approach used a new load balancing policy that was more scalable and took into account the interference on performance that occurs with resource sharing.

3 OVERVIEW OF EXISTING SYSTEMS

In this section, scheduling systems that are currently in place are introduced and their strengths and weaknesses are discussed. Usage scenarios of the products that will use the new service are introduced. The requirements for the new service are explained. The overall view of the existing systems was acquired from personal experience and discussions with experienced co-workers.

3.1 Microsoft Message Queuing

Existing scheduling in the MDM solution is accomplished by using Microsoft Message Queuing (MSMQ) technology. MSMQ operates in a first in, first out (FIFO) fashion, which executes the first item in queue and pushes new tasks to the end of the queue. It is a simple solution to scheduling a multitude of tasks.

MSMQ requires Windows platform, which limits the possible servers to Microsoft's own products and it cannot be operated on other platforms. MSMQ is widely used in the MDM solution, usage scenarios include such services as sending text messages, sending wake-ups to devices and updating inventory information.

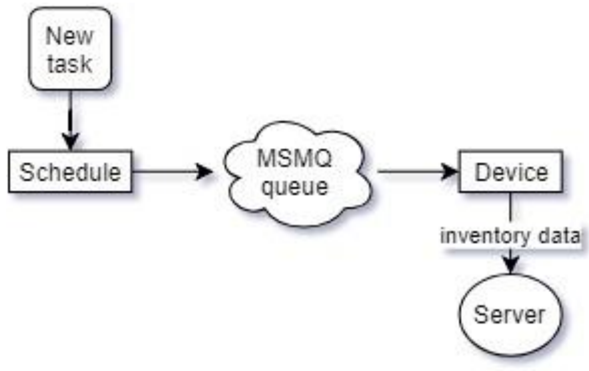

Fig. 1. Workflow process for device wake-up.

Figure 1 shows an example workflow diagram to help identify main concerns of the current implementations. Task is scheduled to MSMQ, which forwards it to the server that executes it. Task executer tells the device to send inventory data back to server. Tasks are scheduled before they are sent to MSMQ queue, since MSMQ does not support queuing tasks for a later date. Task types are discussed in more depth in section 3.3. As briefly mentioned in the introduction, there is no current implementation of server resource monitoring. By monitoring resource usage in the processing end, throughput to external services such as MSMQ could be regulated dynamically. In current solution, throughput is limited statically in the calling end, before it is sent to MSMQ.

Using the new service, resource usage of the inventory-processing server could be monitored and task throughput could be regulated accordingly. Figure 2 shows a workflow diagram when using the new service. Tasks can be scheduled in the service, removing the need to have them scheduled in an external queue. The service uses resource monitoring to directly acquire real-time usage data from the server. New solution would also have an algorithm that is more optimal than MSMQ's FIFO -approach.

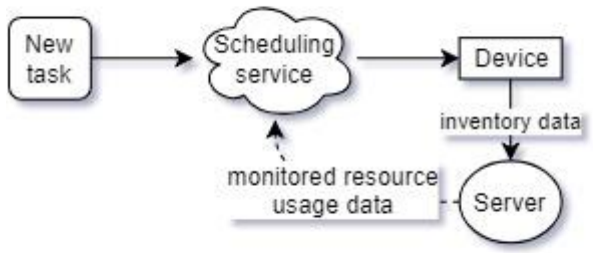

Fig. 2. Workflow process using the new service.

3.2 Apple Device Enrollment Program

Apple's Device Enrollment Program (DEP) is a solution offered by Apple Inc. used for enrolling new devices automatically to the MDM solution. Currently in the product there does not exist automatic retrieval of updated device lists. This process requires multi-phased processing and it consists of three steps: an update request is sent to the DEP service, a list of devices is sent back and lastly, predefined application configurations are deployed to new devices. This turns into a requirement for the service to provide support for multiple steps in the scheduling process. A DEP task would need to be able to wait for feedback after it's execution and execute again based on the feedback.

3.3 Tasks

Existing tasks that need to be scheduled are discussed here. Their main characteristics and challenges are explained to get an understanding of how the scheduling service should work internally.

Some of the tasks have subsequent actions that needs to be executed after getting a feedback from the task executor. For example, when updating Apple DEP devices, first, an update request is sent to the Apple Inc.'s service, a list is sent back to the caller in response and then a separate deployment order is sent to another internal service. Configuration and handling of tasks and subtasks is an essential part of the service and is discussed more closely in section 4.3.

3.3.1 Wake-ups

Wake-up is a way of monitoring available devices, their inventories and statuses. Woken devices send their inventories to a server to be processed. Executing a wake-up is a small and fast operation, but the inventory data processing that is required when devices send their inventories back to the server generates a large load on server resources.

Wake-ups are sent to all managed devices once a day. In the current implementation, all wake-ups are divided into 24 equally sized groups of tasks that are subsequently executed within a given time span of maximum of 1 hour, adding to a total of 24 hours. This kind of approach is needed in the current implementation, since there is no implementation to monitor the load that the inventory processing generates. By monitoring inventory processing server's resources, the time used to execute all wake-ups could potentially be cut drastically without burdening the servers any more than necessary.

A new recurring scheduled task needs to be created for all existing and new instances automatically. The new recurring task is added to the corresponding task agent. When predefined date and time elapses, the task is queued to the service. The priority of scheduled wake-up tasks is minimal but they need to be executed once a day at minimum. Wake-ups can also be dispatched manually by the user. In this case, the task is directly queued to the optimization process.

Android devices are polled in three-hour intervals. Other devices do not support polling and require a wake-up distribution. Wake-ups to iOS devices are sent through Apple Inc.'s Apple Push Certificate that establishes a trusted connection between the devices and the managing service.

3.3.2 Application deployments to devices

Applications can be deployed to multiple devices remotely and simultaneously using the MDM solution. Deploy process will use each device platform's corresponding application deployment services for deploying the applications.

An example of deployment needs is Apple DEP update process. Currently user can request an updated list of devices, and after acquiring information of new devices, user can deploy configuration profiles to them.

To use scheduling service for the automation, new devices should be checked periodically from the Apple's service. This would happen via a recurring scheduled task for Apple DEP updating with a subtask of deploying configuration profiles to new devices.

3.3.3 Reporting

A use case for the service is the creation and distribution of instance reports. The report creation process is time demanding can take minutes. Reporting tasks are of low priority and should be allowed to process slowly, or when the server has less load.

3.4 Demand for a new service/implementation

Current products do not have efficient scheduling solutions implemented. Scheduling is scattered across different functions or does not exist at all. Existing scheduling solutions are also simple and inefficient, for example, wake-up tasks are simply distributed in equal blocks across the day. The new service would allow easier implementation of scheduling automation to a wide variety of existing and future functionalities. There is a need for dynamic scheduling that is based on efficiency and monitoring of available server resources. The requirements and use cases for the new service were identified through discussions with co-workers. Scalability was a major concern to enable future usage of the service. Subsequent tasks need to be supported to enable scheduling tasks with multiple steps. Informing other components on the tasks was also identified as a requirement.

4 DESIGN AND IMPLEMENTATION

The design and implementation of the service is detailed in this section. A comprehensive picture of the service as a whole is given. The service is written using C# 7.3 and .NET Core framework 2.1, which at the time of writing are the latest versions. The service runs on Microsoft Corporation's Microsoft Azure cloud computing platform.

Section 4.1 shows the project's class structure. Scheduling process and implemented scheduling algorithm are discussed in section 4.2 Implemented scheduling algorithm is discussed in 4.3. Tasks are discussed in section 4.4. Resource monitoring of external servers is discussed in section 4.5. Section 4.6 introduces performance counters. Section 4.7. discusses task feedback. Other functions of the service are discussed in section 4.8.

4.1 Class structure

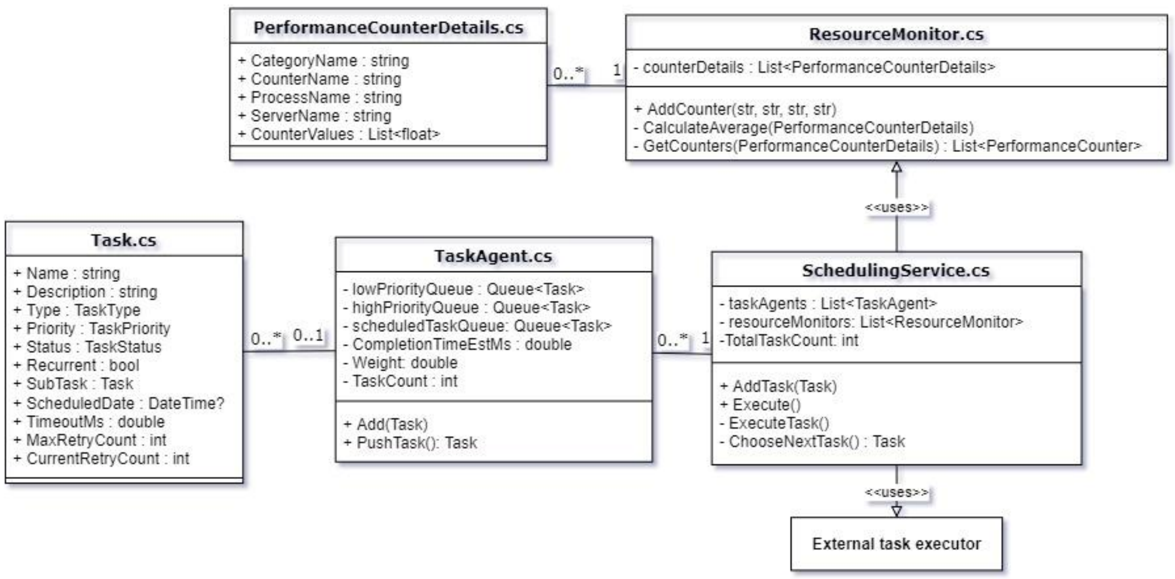
PerformanceCounterDetails.cs
+ CategoryName : string
+ CounterName : string
+ ProcessName : string
+ ServerName : string
+ CounterValues : List<float>
0..*
1
ResourceMonitor.cs
- counterDetails : List<PerformanceCounterDetails>
+ AddCounter(str, str, str, str)
- CalculateAverage(PerformanceCounterDetails)
- GetCounters(PerformanceCounterDetails) : List<PerformanceCounter>
<<uses>>
Task.cs
+ Name : string
+ Description : string
+ Type : TaskType
+ Priority : TaskPriority
+ Status : TaskStatus
+ Recurrent : bool
+ SubTask : Task
+ ScheduledDate : DateTime?
+ TimeoutMs : double
+ MaxRetryCount : int
+ CurrentRetryCount : int
0..*
0..1
TaskAgent.cs
- lowPriorityQueue : Queue<Task>
- highPriorityQueue : Queue<Task>
- scheduledTaskQueue: Queue<Task>
- CompletionTimeEstMs : double
- Weight: double
- TaskCount : int
+ Add(Task)
+ PushTask(): Task
0..*
1
SchedulingService.cs
- taskAgents : List<TaskAgent>
- resourceMonitors: List<ResourceMonitor>
-TotalTaskCount: int
+ AddTask(Task)
+ Execute()
- ExecuteTask()
- ChooseNextTask() : Task
<<uses>>
External task executor

Fig. 3. Class diagram of the service. The service consists of two main components, scheduling service and resource monitor.

Class diagram of the service is shown in figure 3. The scheduling consists of two main components, scheduling service and resource monitor. SchedulingService.cs contains the main logic of the service including handling task flow during task lifecycle and schedule optimization. SchedulingService.cs contains all tasks inside TaskAgent –objects, which in turn contain Task-objects.

TaskAgent.cs defines a task agent that handles specific types of tasks that are assigned to it. It stores tasks with different parameters, such as priority, to be later called by the scheduling service for execution. Task.cs contains the definition for a task. It defines each task's behavior including recurrence, timeout, status and subtasks.

ResourceMonitor.cs contains the handling of performance counters, including the calculations of counter values, and supplies these values to the users of a resource monitor. Individual counter details are held in an object of type PerformanceCounterDetails.

4.2 Scheduling process

The scheduling process consists of 5 main processes:

- Adding a new task to the service: moving it to its corresponding agent
- Scheduling algorithm: Cost assessment: algorithm picks tasks for execution according to criterion such as monitoring data
- Execution: task is executed
- Handling feedback
- Environment monitoring: check server availability, resource levels

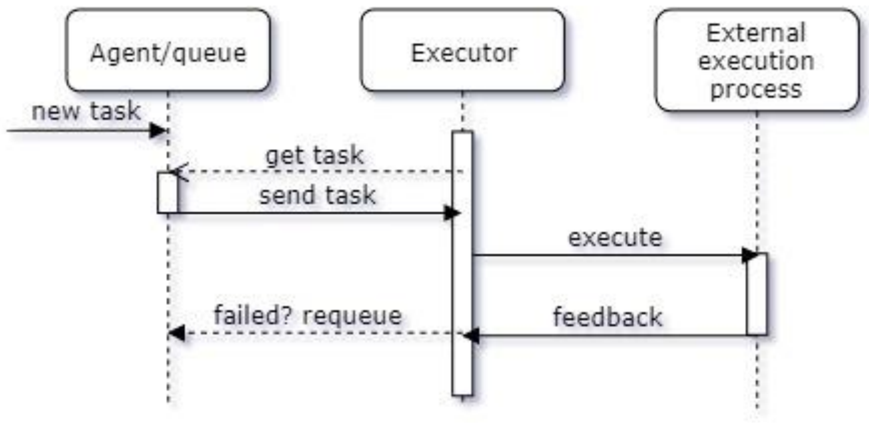

Fig. 4. Sequence diagram of the inner workings of the service.

Scheduling process from the viewpoint of a task is shown as a sequence diagram in figure 4. A new task is scheduled to the service and is moved to the corresponding task agent. If the task is scheduled to start immediately, it is added to its agent's queue. If the task is queued for later, it is added to its agent's waiting queue. When the scheduled date has elapsed, the task is queued in the agent's main queue. The executor requests tasks from agents according to the algorithm.

Tasks inside an agent's queue are in a FIFO order based on their arrival to the queue. A single task is requested from an agent based on each agent's weight. Agent then sends the first task in its high priority task queue. If no tasks exist in the high priority task queue, a task from the low priority task queue is sent instead. If both queues are empty, the executor is signaled that no tasks exist in the agent. The executor then proceeds to execute the task in an external service that is responsible for processing the specific task. The executor waits for a feedback. When feedback is received, it is processed according to predefined criterion. For example, if task has been configured for recurrence, it is sent back to its responsible agent and is queued again. If no feedback is received by predefined timeout duration, the task is deemed as failed.

Microsoft's task parallel library (TPL) is used for adding concurrent functionality to the service. With TPL, each job can be allocated to an independent thread that processes apart from other jobs. Task execution is divided into multiple threads that each can wait for status feedback on their execution without halting other tasks from completing. Concurrency in the service is handled using C# language's Thread class offered in System.Threading namespace. Signaling between different parts of the service execution is implemented using Thread's EventWaitHandle.WaitOne() method for waiting, and EventWaitHandle.Set() method for resuming. Signaling is used to notify other parts when new tasks are needed and when new tasks arrive to the service. When no tasks exist in the service, the service can stop and wait for new tasks.

4.3 Implemented scheduling algorithm

A task has three criteria on which the scheduling optimization is based on:

- Task quantity of agent k, t_k
- Task completion time estimate c_k

- Resource load on executing server *x*, s_x

Implemented algorithm is based on each agent's weight proportional of the total weight:

$$W = \sum_{k=0}^{n} w_k = \frac{c_k K_k}{s_x}$$

, where W is the total weight of all agents, w_k is the weight of agent *k*, c_k is the completion time estimate of agent *k*, K_k is the coefficient of agent *k* and s_x is the executing server x's load. In calculations, c_k is limited to maximum of 0.95 to allow tasks to execute even on overloaded machines. Available resources for task execution are split into concurrent threads with each thread capable of handling one task at a time. The amount of threads each task type has depends on its weight. For example, if agent A has relative weight of 90% and 20 tasks can be run concurrently, it is executed on maximum of $20 * 0.9 = 18$ threads.

The weights for each agent are re-calculated frequently to keep the proportions up-to-date. Resources allocated to agent *k* are calculated as follows:

$$r_k = \frac{w_k}{W} R$$

, where R is the total amount of available resources (e.g. *x* concurrent threads), w_k is the weight of agent *k* and W is the total weight of all agents.

To factor in agent's total task quantity into the agent's weight, a manually defined coefficient K_t can be introduced to the algorithm. It can be defined as a threshold-based value

$$K_t = t_k c_k$$

Depending on the quantity and size of tasks in an agent. Thresholds can be defined when deemed necessary. Thresholds for the K_t can be defined as:

$$K_t < x \rightarrow 1$$

$$x \leq K_t < y \rightarrow a$$

$$K_t \geq y \rightarrow b$$

, where $0 \leq x \leq y$ and $1 \leq a \leq b$. Value *a/b* is the wanted coefficient for multiplying the weight of an agent based on the threshold value. For example, threshold values are initialized with as $x = 1000, y = 3000, a = 1.5, b = 2$ and the coefficient K_t is 2500, the weight of the agent *t* is 1.5 times higher than regularly, since $1000 \leq 2500 < 3000 \rightarrow 1.5$. This is a simplified but relatively robust way of accounting in agent's task quantity to the algorithm.

The emphasis on task quantities between various agents is highly subjective value and cannot be used as is.

Pseudocode for the scheduling algorithm is shown in algorithm 1. Resources to execute each agents' tasks are divided between agents according to their weights. Minimum and maximum quotas have been introduced to avoid agents having too much or too little resources to execute tasks. Quota limitations are not active when only one type of task is handled in the service or when no tasks are available for execution.

```
WHILE service is active
        Calculate relative weights of each agent and assign resources accordingly
        Loop through agents and execute tasks according to assigned resources
        Re-calculate weights frequently
ENDWHILE
```

Algorithm 1. The pseudocode for the algorithm.

4.4 Task

A new task is created by specifying its properties. Table 1. introduces each of the task properties. All the properties of a task are initialized when a new task is created. Task name and description are free-text fields that state general information about the task. Task type defines how the task is handled and executed. Task priority ranges from high to low. Estimated completion time is a dynamic estimation of last x executions that is used to optimize the scheduling algorithm's accuracy. When the task is scheduled for later date, scheduled date can be defined. Recurrence states whether the task can be retried in case of a failure to execute and maximum retry count states how many times it can be retried. Current retry count tells how many times the task has been retried. If task has a subsequent task that should be executed after the main task has finished executing, a subtask is defined. Timeout duration states how long the service will keep the task execution running before it is terminated and the task status is set to fail.

Table 1. Properties of a task.

Property	**Description**
Name	Name of the task
Description	Description for the task
Type	Task type can be e.g. wake-up/DEP update/reporting
Priority	Priority has two values: high/low
Status	Task status can be New/Succeeded/Failed
Estimated completion time	Estimation for the time that a task takes to process from that to finish.
Schedule	Task can be scheduled for a specific date/time
Recurrence	Task is recurring or not
Maximum retry count	The maximum count of retries for a task
Current retry count	The current amount of retries
Subtask	A subsequent task can be defined for a task
Timeout duration	The time a task tries to execute before timeout

Task's flow in the service is shown as a flowchart diagram in figure 5. When a new task is added to the service, it is moved to the corresponding task agent. Each task agent holds specific types of tasks, for example all wake-up tasks can be held in the same task agent. Executor fetches the task from task agent. In case an execution fails for any reason, if recurrence has been enabled and the current retry count has not reached the maximum retry count, the task is moved back to the task agent from where it is re-queued.

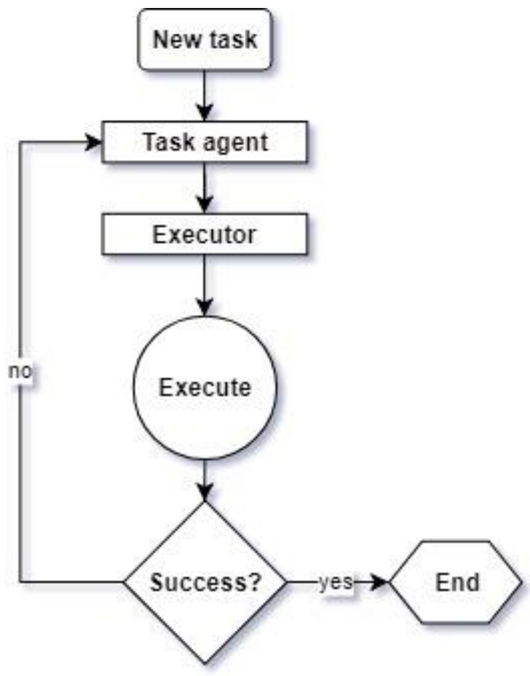

Fig. 5. Flowchart shows the workflow for a task's transition in the service from creation to execution.

4.4.1 Task configuration

When creating a new task, basic information is given to the initialization. Task-specific functionality has been configured inside the service and tasks are handled accordingly.

Configuring new tasks needs to be relatively lightweight to make adding them to the service straightforward. When creating a new task type, the configuration needs to be coherent across all locations where the task is created in. All callers of the service using the configured task need to be sure that they are using it correctly.

Two locations for configuring tasks were considered: inside the service itself and in the calling end of the service. In case the configuration would be located in the calling end, the service itself could remain more generic and fewer modifications would be required to the service itself. However, if the task would be configured outside the service, uncertainties would arise. If definition of the task would change, modifications would be required to every location where the task has been configured and it would be harder to manage. The volatility of any step of the task changing later would affect maintenance of task configurations negatively.

To configure tasks inside the service would risk the genericity of the service. The pros of configuring tasks in a single location at the cost of lessening genericity seemed a good trade-off, and the decision was made to centralize the configuration of tasks inside the service for the benefit of easier modifications later on.

4.4.2 Subtask

Subtask is a task that is scheduled for execution after the main task has been executed successfully. Subtasks are useful when a task has multiple steps that need to be executed sequentially in order to do all the work that needs to be done. DEP update is an example of a use case when a subtask is required; first new devices are fetched from Apple Inc.'s server and then another task is run to deploy configurations to the discovered devices. After successful execution of the main task and a feedback is acquired, if a subtask has been defined, it is added to the service as a new task.

4.5 Monitoring of external resources

Monitoring external servers' resources that the service uses to execute its tasks is an essential part of the service. Monitoring is needed to avoid resource overload, throttling and bottlenecks. By monitoring the service, performance on both ends can be optimized, limiting the throughput to react to resource usage. Monitoring provides the service with up-to-date data on server resources.

Microsoft provides Performance Counters that are used to provide performance information of an operating system (OS), application, server or a driver (Microsoft, 2018.) They are used to supply the service with real-time data of different tasks' execution loads. Performance counter data can be used to limit task throughput in the scheduler so that the resource throttle of the server can be stabilized. The monitoring is implemented as a separate service and the resource usage values are retrieved to the service.

Microsoft provides various performance counters that are preinstalled in every Windows machine, which makes using them in Windows environment relatively simple. In the thesis only existing, default performance counters are used, the creation of additional counters is not needed. In the C# project, a NuGet package called System.Diagnostics.PerformanceCounter is used to utilize pre-existing code for interacting with performance counters. Used performance counters are discussed in following section 4.4.1.

Resource monitors for each server and task are initialized in the service and run on background. While active, a monitor collects counter values periodically and when requested, provides an average counter value based on recent values.

4.6 Performance counters

Used performance counters include counters for the total server CPU usage, total memory usage and the process times used by distinct processes. Used performance counters are listed in the table 2.

Table 2. Used performance counters: category name, counter name and process name.

Category	**Counter**	**Process name**	**Value description**
Memory	% Committed Bytes in Use	_Total	Total memory used by the machine in bytes.
Processor information	% Processor Time	_Total	A measure of how often the system is doing nothing subtracted from 100%, a calculation that gives the time that the processor is busy working on productive threads.
Process	% Processor Time	<Process>, e.g. Chrome	The amount of time the process spends using the processor resource.

Counters “Memory (% Committed Bytes in Use)” and “Processor information (% Processor Time)” provide the amount of resources used in percentages, which makes using them straightforward. These values are used in the service as limiting factors. When server resources are nearing their maximum limit, service task throughput is slowed until the server can handle more load again. When using counter “Process (% Processor Time)” additional calculations are required to make the provided value understandable. The value is calculated over the baseline of (No of logical CPUs * 100). The value may be larger than 100 percent per core. This indicates that the process is using more than its capacity via multithreading.

By calculating the sum of distinct processes with the same name, we can get the resource usage of a distinct type of task. Multiple instances of same process are named similarly, for example in the case of Chrome; the processes are named as follows: ‘Chrome’, ‘Chrome#1’, ‘Chrome#2’, and so forth. Machines are searched for process names matching this scheme, the values are summed up and a calculated value is sent back to the caller.

4.7 Handling task feedback

Task execution in most cases returns a value of some sort. In case of a successful execution, a notification to the caller of the service can be sent. If a task fails to execute it is removed or, if concurrence and maximum retry count have been provided, moved back to its corresponding task agent and an error message is sent to the caller. Tasks that exceed their defined timeout duration are terminated automatically.

4.8 Other functions

The service needs a multitude of additional functionalities to operate. In addition to handling task lifecycle and executing tasks in optimally, several functionalities are needed. A list of additional functionalities to be added:

- Cancel (scheduled) tasks
- Clear queues
- Modify task parameters dynamically
- Copy tasks

5 DISCUSSION

Research questions were answered in the course of this thesis. Design of the task scheduling service consists of multiple aspects that needed to be addressed, such as what the optimality criterion for the service are. Finding a fitting approach to the problem was achieved by studying existing literature. Multiagent scheduling was approached because it closely resembled the aim of the service: scheduling heterogeneous tasks using shared and limited resources. Other elements such as unfairness, resource monitoring and load balancing were discovered. The optimization of the service depended on task properties and the defined optimality criterion. Safeguards against unfairness were implemented to avoid bottlenecks and resource throttling. Resource usage of the service and of the executing servers were considered when implementing the service.

A working prototype of the service and the scheduling algorithm was built based on the design laid out in section 4. The built prototype was very limited in features due to pressing schedule of the thesis, it was not implemented in a real test environment and no comparisons were tested against more traditional approaches due to lack of time. Initial localized tests with the prototype showed that the approach is feasible from performance perspective. To make the service work in real-life situations, more development is needed to support more of the features listed and described in the thesis.

The designed algorithm leaves a lot of space for improvement. The implemented algorithm was built to match the needs of the company and its performance is sub-optimal at best. A lot of improvements and simplifications were made to it during the writing of this thesis.

The service has a few insufficiencies concerning different types of tasks and their handling. If the type of task that is used in the service cannot return any feedback on its status of completion, having recurrence rules is not possible. In addition, since task’s average completion time is of vital value for the optimization process, if the execution of task cannot be tracked a manual value must be given for the completion time. Another insufficiency concerns the availability of monitoring external resources. If resources aimed at a specific task cannot be monitored outside the service, the value of a task’s weight cannot be calculated reliably. All of the above-mentioned scenarios will lead to inaccurate estimates for a task’s weight and the optimality of the schedule cannot be guaranteed. The algorithm’s optimality can only be assured when all of the preceding factors can be acquired reliably. A few measures against these insufficiencies were implemented, but no real testing was conducted to ascertain their effectiveness.

Security is a valid concern which has not been addressed within the scope of this thesis. Since the service consists of variety of different parts that need to communicate with each other, trust between the processes is required. Future work consists mainly of testing the prototype in real test environment and refining the variables and weights according to real-life usage for optimal performance.

6 CONCLUSION

Requirements for a new task scheduling service were defined and discussed. Literature review was conducted and various considerable aspects relevant to creating a task scheduling service for the given purpose were introduced. Existing scheduling problems and algorithms were introduced. A new task scheduling service was designed and a prototype was implemented. Various design choices were made with the aim of optimizing the service and available resources. An algorithm was built for executing heterogeneous group of tasks while aiming to minimize total completion time and resource usage. Minimizing unfairness between task agents was a key feature when designing the algorithm. A prototype was built. Several key insufficiencies and problems were discussed and further research and development targets were identified.

REFERENCES

1. Agnetis, A., Billaut, J-C., Gawiejnowicz, S., Pacciarelli, D., Soukhal, A., Multiagent scheduling – models and algorithms, Springer, Berlin, 2014.
2. Agnetis, A., Mirchandani, P. B., Pacciarelli, D., Pacifici, A., Scheduling problems with two competing agents, Operations Research 52(2), Informs, 2004, pp. 229-242.
3. Ajtai, M., Aspnes, J., Naor, M., Rabani, Y., Schulman, L. J., Waarts, O., Fairness in scheduling, Journal of Algorithms 29, 1998, pp. 306-357.
4. Alidaee, B., Womer, N. K., Scheduling with time dependent processing times: review and extensions, Journal of the Operational Research Society 50, 1999, pp. 711-720.
5. Brucker, P., Knust, S., Complex scheduling, 2nd edition, GOR-Publications, Springer, 2012.
6. Casini, D., Biondi, A., Buttazzo, G., Semi-partitioned scheduling of dynamic real-time workload: a practical approach based on analysis-driven load balancing, C.3 Real-Time and Embedded Systems, ACM, 1998.
7. Cheng, T. C. E., Ng, C. T., Yuan, J. J., Multi-agent scheduling on a single machine to minimize total weighted number of tardy jobs, Theoretical Computer Science 362, Elsevier, 2006, pp. 273-281.
8. Gracioli, G., Fröhlich, A. A., Pellizzoni, R., Fischmeister, S., Implementation and evaluation of global and partitioned scheduling in real-time OS, Real-Time Systems, Vol. 49, Issue 6, 2013, pp. 669-714.
9. Lee, K., Choi, B. C., Leung, J. Y. T., Pinedo, M. L., Approximation algorithms for multi-agent scheduling to minimize total weighted completion time, Information Processing Letters 109, Elsevier, 2009, pp. 913-917.
10. Lee, W. C., Wang, W. J., Shiau, Y. R., Wu, C. C., A single-machine scheduling problem with two-agent and deteriorating jobs, Applied Mathematical Modelling 34, Elsevier, 2010, pp. 3098-3107.
11. Madni, S. H. H., Latiff, M. S. A., Abdullahi, M., Abdulhamid, S. M., Usman, M. J., Performance comparison of heuristic algorithms for task scheduling in IaaS cloud computing environment, PLOS ONE 12(5), 2017.

12. Maurya, A. K., Tripathi, A. K., On benchmarking task scheduling algorithms for heterogeneous computing systems, The Journal of Supercomputing Vol. 74, Issue 7, Elsevier, 2018, pp. 3039-3070.

13. Microsoft, Performance counters, available at https://docs.microsoft.com/en-us/windows/desktop/perfctrs/performance-counters-portal, 2018, retrieved 8.1.2019.

14. Mosheiov, G., A Note on Scheduling deteriorating jobs, Mathematical and Computer Modeling 41, Elsevier, 2005, pp. 883-886.

15. Ould Deye, M. M., Slimani, Y., Sene, M., Load balancing approach for QoS management of multi-instance applications in clouds, *2013 International Conference on Cloud Computing and Big Data*, 2013, pp. 120-126.

16. Polanczyk, R. V., Extending the semantics of scheduling priorities, ACM Queue, Vol. 10, Issue 6, 2012.

17. Porras, J., Riva, O., Darø Kristensen, M., Dynamic Resource Management and Cyber Foraging, Middleware for Network Eccentric and Mobile Applications, Springer, Berlin, Heidelberg, 2009, pp. 349-368.

18. Rabelo, R. J., Camarinha-Matos, L. M., Afsarmanesh, H., Multi-agent-based agile scheduling, Robotics and Autonomous Systems, Vol. 27, Issues 1-2, Elsevier, 1999, pp. 15-28.

19. Soni, S., Sangwan, S., Load balancing in cloud computing: a review, International Journal of Advanced Research in Computer Science, Vol. 9 No. 2, 2018.