Revista Brasileira de Computação Aplicada, November, 2025

DOI: 10.5335/rbca.v17i3.16747
Vol. 17, Nº 3, pp. 100–113
Homepage: seer.upf.br/index.php/rbca/index

ORIGINAL PAPER

Multi-queue Round Robin Scheduling for Enhanced Performance in Integration Platforms

Daniela L. Freire [1,2,3], Rafael Z. Frantz [2], Vitor Basto-Fernandes [3], Gerson Battisti [2], Sandro Sawicki [2], Fabricia Roos-Frantz [2]

[1]University of São Paulo, Brazil, [2]Unijuí University, Brazil, [3]University Institute of Lisbon, Portugal

*danielalfreire@icmc.usp.br;{rzfrantz,battisti,sawicki,frfrantz}@unijui.edu.br;vitor.basto.fernandes@iscte-iul.pt

Received: 2025-02-17. **Revised:** 2025-11-28. **Accepted:** 2025-12-10.

Abstract

Contemporary enterprise environments involve a large amount of information and heterogeneous applications that must exchange data in near real time. Integration platform-as-a-service (iPaaS) solutions support this scenario by executing integration processes composed of workflows of tasks. However, current task scheduling algorithms used in integration platforms, such as First-In, First-Out (FIFO), may lead to poor performance and unfair use of computational resources under high workloads. In this article we propose the Multi-queue Round Robin (MqRR) algorithm, a task scheduling heuristic tailored to the runtime systems of enterprise application integration platforms. MqRR organises tasks into multiple queues and applies a round-robin strategy with preemption to avoid starvation and to distribute the load more evenly among workflows. We evaluated MqRR against the traditional FIFO heuristic using an integration process simulator and three real-world integration workflows, under increasing message arrival rates. Regarding our research questions, the results show that: (RQ1) there is a workload threshold from which FIFO degrades its performance, leading the number of completed messages to approach zero; and (RQ2) MqRR improves task scheduling performance in high workload scenarios, keeping a linear growth of makespan and increasing the number of processed messages. These findings indicate that MqRR is more suitable than FIFO for integration platforms that must handle high message rates in cloud environments.

Keywords: Application integration; task scheduling; algorithm; workflow scheduling; integration patterns; round robin.

Resumo

Ambientes empresariais contemporâneos envolvem uma grande quantidade de informações e aplicações heterogêneas que precisam trocar dados em tempo quase real. Plataformas de integração como serviço (iPaaS) apoiam esse cenário executando processos de integração compostos por fluxos de tarefas. Entretanto, algoritmos de escalonamento de tarefas comumente utilizados nessas plataformas, como o First-In, First-Out (FIFO), podem levar a baixo desempenho e uso injusto dos recursos computacionais sob cargas de trabalho elevadas. Neste artigo propomos o algoritmo Multi-queue Round Robin (MqRR), uma heurística de escalonamento de tarefas voltada aos sistemas de tempo de execução de plataformas de integração de aplicações empresariais. O MqRR organiza as tarefas em múltiplas filas e aplica uma estratégia round robin com preempção para evitar inanição e distribuir a carga de forma mais justa entre os fluxos de trabalho. Avaliamos o MqRR em comparação com a heurística FIFO tradicional por meio de um simulador de processos de integração e de três fluxos de integração provenientes de cenários reais, variando a taxa de chegada de mensagens. Em relação às questões de pesquisa, os resultados mostram que: (RQ1) existe um limiar de carga a partir do qual o FIFO degrada seu desempenho, fazendo com que o número de mensagens concluídas tenda a zero; e (RQ2) o MqRR melhora o desempenho do escalonamento de tarefas em cenários de alta carga, mantendo um crescimento linear do makespan e aumentando o número de mensagens processadas. Esses resultados indicam que o MqRR é mais adequado que o FIFO para plataformas de integração que precisam lidar com altas taxas de mensagens em ambientes em nuvem.

Palavras-Chave: Integração de aplicativos; agendamento de tarefas; algoritmo; agendamento de fluxo de trabalho; padrões de integração; round robin.

1 Introduction

The large volume of information generated by the increasing number of heterogeneous devices connected to the Integration of the Internet of Things (IoT) produces knowledge and creates more business opportunities for enterprises. Integrating IoT with cloud computing has become primordial in tackling the increasing data and managing virtual resource utilisation and storage capacity. It also creates more usefulness from generated data and develops innovative applications for the users (Mohammad Aazam and Eui-Nam Huh and Marc St-Hilaire and Chung-Horng Lung and Ioannis Lambadaris, 2016).

The integration platform-as-a-service (iPaaS) is a cloud service that integrates services and applications to interchange data and functionalities to respond to business process requests quickly. In 2017, every two out of three application integration projects were directly developed using Cloud integration platforms, and iPaaS was the preferred deployment for the integration platform. The annual revenue for iPaaS grew higher than the traditional implementation of integration on-premise (Guttridge et al., 2017; Sharma, 2017). An iPaaS allows software engineers to design, run, and monitor integration processes. An integration process carries out a workflow comprising distinct atomic tasks, connected by communication channels that desynchronise one task from another (Pezzini et al., 2015; Kanagaraj and Swamynathan, 2016). Messages move through the workflow, encapsulating data from/to the integrated applications.

Many current open-source integration platforms support integration patterns documented by Hohpe and Woolf (2004) and follow the Pipes-and-Filters architectural style (Alexander et al., 1977). Pipes depict message channels, and filters represent atomic tasks implementing a particular integration pattern to process messages. The runtime system is the platform component responsible for executing integration processes (Frantz et al., 2016), and its primary function is task scheduling (Guo et al., 2015; Hilman et al., 2018). Task scheduling manages the schedule of the execution of tasks and the use of computational resources. In the context of cloud computing (where the charging model is pay-as-you-go), task scheduling should be aligned with the minimisation of costs (Freire, Frantz, Roos-Frantz and Sawicki, 2019), with the handling of large volumes of data from IoT (Shoukry et al., 2019), and with market demands for quality of software, flexibility, and response times (Fan et al., 2018).

In the literature, we found two main execution models for runtime systems: process-based and task-based (Blythe et al., 2005; Boehm et al., 2011; Frantz et al., 2012; Alkhanak et al., 2016). This classification concerns the granularity that the runtime system deals with regarding task execution (Freire, Frantz, Roos-Frantz and Sawicki, 2019). In the process-based model, the runtime system deals with process instances. In this model, a thread is assigned to an instance of the integration process; this thread is used to execute every task that composes the workflow over an inbound message so that this message may flow throughout the process. After every task in the workflow has been executed, the thread is released. In the task-based model, a thread is assigned to an instance of a task so that this thread is used to execute the task over the inbound message that reaches the task. When the task is completed, an outbound message is written to the channel that connects the current task to the next one in the workflow, so the thread is released. The execution of the message in the next task now depends on a new assignment of an available thread to this task. In this article, we discuss the task-based execution model. In this execution model, when messages have a high input rate, threads tend to execute initial tasks more frequently than other tasks due to the First-in-first-out (FIFO) policy, which is used in task scheduling. In the FIFO policy, the first arriving task will be executed in the first place, while the next will wait until the first task is finished. Thus, this model may lose efficiency when an integration process is submitted to high input rates of messages (Frantz et al., 2012; Freire, Frantz, Basto-Fernandes, Sawicki and Roos-Frantz, 2022; Freire, Frantz, Frantz and Basto-Fernandes, 2022b).

In a cloud computing environment, the task scheduling of various IoT applications is complicated due to the heterogeneous characteristics of IoT data. On the other hand, this scheduling needs to be efficient, and the load must be balanced to maximise performance while meeting constraints such as task dependencies (Basu et al., 2018). The increased volume, variety, and velocity of IoT data generated and their real-time processing require higher computational capacity than is usually offered on the cloud. However, the virtually unlimited computational capacity of the cloud entails a higher communication latency and a higher monetary cost. For this reason, the workload balance requires an effective and dynamic task scheduling heuristic (Stavrinides and Karatza, 2018).

Building fair scheduling while increasing performance is a significant concern for enterprises once they concurrently submit workflows for execution in different resources. Pietri et al. (2019) define fair scheduling as one that provides an adequate balancing from the workload to resources. In the domain of application integration, to obtain fair scheduling in contemporaneous environments, it is required to re-engineer integration platforms (Linthicum, 2017). The enhancement of the runtime system task scheduling can help enterprises take advantage of the scalability of Cloud computing, increase their productivity and reduce their costs by optimising computational resources.

In this article, we propose the Multi-queue Round Robin (MqRR), a new policy for task scheduling of integration processes carried out by runtime systems. First, we represented integration processes using Directed Acyclic Graphs, then classified them according to their conceptual models and logic integration. Next, we implemented our proposal through algorithms, which provide a fair allocation of threads to tasks from integration processes with high data volumes.

We executed a performance comparison of MqRR and FIFO heuristics under different load conditions. The results of this experiment proved that the FIFO is better than MqRR until a threshold of workload is reached.

Then, MqRR performance becomes higher, and FIFO stops delivering processed messages. MqRR revealed higher robustness in tackling high workloads and the uncertain environment of integration processes. The results were validated statistically with ANOVA and Scott & Knoot tests. Our proposal can adequately integrate the runtime systems of integration platforms into the contemporaneous integration environments and provide suitable solutions for the challenges raised by cloud computing and the Internet of Things.

The rest of this article is organised as follows: Section 2 discusses the related work regarding task scheduling; Section 3 describes the characteristics of scheduling in integration processes; Section 4 presents the problem formulation; Section 5 presents our algorithm; Section 6 presents an experiment to validate the proposal; and Section 7 presents our conclusions and future works.

2 Related Work

This section provides a literature review on recent task-scheduling approaches in cloud and edge computing. We group the selected works according to their main focus and optimisation goals.

- **Task scheduling in cloud computing environments.** Works in this group focus on minimising makespan and meeting deadlines by mapping tasks to heterogeneous virtual machines, often using variants of classic heuristics or metaheuristics.
- **Energy and cost efficiency.** These proposals aim to reduce energy consumption and monetary cost in clouds, e.g. by consolidating tasks, turning off idle machines or using multi-objective optimisation (such as PSO and GA) for resource allocation.
- **Resource utilisation and workflow management.** Here, authors concentrate on improving resource utilisation, balancing the load among servers and managing complex workflows and DAGs with heuristic scheduling strategies.
- **Scheduling in cloud and edge computing.** Some works extend the scheduling problem to hybrid cloud–edge scenarios, distributing tasks between central clouds and edge devices while minimising latency and makespan.
- **Novel and hybrid algorithms.** Other approaches combine heuristics, metaheuristics and graph-based models, for example scheduling workflows represented as DAGs and applying hybrid algorithms to improve performance in large-scale infrastructures.
- **Large-scale MapReduce clusters.** Finally, there are proposals specifically designed for large MapReduce clusters, often building on classical scheduling theory and adapting it to big-data workloads.

Table 1 summarises the main optimisation goals, application domains and methods used in the selected related work. Most existing works focus on minimising makespan and cost in cloud environments, while our proposal targets the runtime systems of enterprise application integration platforms.

3 Background

In this section, we shall discuss task scheduling for integration processes which adopt integration patterns documented by Hohpe and Woolf (2004) and the Pipes-and-Filters (Alexander et al., 1977) architectural style. We shall define the main elements involved in this type of scheduling and describe the task-based model, which is the model approached in this article.

An integration process is a computational program that supports exchanging data and functionalities amongst applications to perform a "job". A job is a user request. The accomplishment of a job consists of receiving input data from the user request and then processing these data to produce output data. Usually, one or more sources deliver data to an integration process, which goes through a segment of "tasks" uncoupled and connected by "communication channels". Then, the data is delivered to one or more data sinks. Sources and sinks of data can be applications, databases, sensors, etc. Data are wrapped in "messages". A message has a header and a body. The header contains custom properties, and the body has the payload data. A message can be split into one or more messages in the workflow; two or more messages can be merged into a unique message.

We use the following terminology:

- "**Workflow**" is a set of atomic tasks chained via communication channels inside an integration process.
- "**Segment**" is a piece of a workflow that can be composed of sequentially arranged tasks, in parallel, or both.
- "**Path**" refers to a specific segment connecting a source to a delivery application by which a message is entirely processed in an integration process.

A task can have either one or more inputs or outputs, depending on the integration pattern implemented. Every pattern represents an atomic operation with a specific operation (transforming, filtering, splitting, joining, or routing) on message processing. Tasks of a path have an order of dependence to be executed so that a message can only be processed by a task after each and every predecessor task has processed this message. An outbound message of a task is written to the communication channel that connects this task with the next one in the workflow path. Parts of the integration process may contain tasks that can be executed in parallel. The number of tasks executed in parallel is limited to the number of available cores, and the executions always obey the order of dependence in an integration process.

All activities required to accomplish the message processing are orchestrated by the "scheduler" that is the central element of the runtime system. The "scheduler" also manages the computational resources for the task execution. These resources are "threads" that are usually grouped in "thread pools". A thread is the smallest sequence of a computational program that the runtime system can manage. Execution threads are abstractions from pieces of physical threads, also called Central Processing Unit (CPU) cores, which are physical and independent processing units. In this article, we refer to execution threads as "threads", and physical threads as

Table 1: Related work summary.

Ref.	Goal	Research field	Method
Guo et al. (2015)	Minimize makespan	Cloud	Fuzzy
Rimal and Maier (2017)	Minimize makespan, cost, and tardiness	Cloud	-
Zhou et al. (2017)	Maximise completion ratio and minimize bandwidth consumption	Broadcast	EDS
Zaourar et al. (2018)	Minimize makespan and energy consumption	Manufacturing	PSO
Manasrah and Ali (2018)	Minimize makespan, cost and balancing load	Cloud	GA & PSO
Rodriguez and Buyya (2018)	Minimise makespan, cost, deadline, and Virtual Machine (VM)	Cloud	-
Anwar and Deng (2018)	Minimize makespan and cost	Cloud	BoTs & MIP
Sun et al. (2018)	Minimize makespan and resource utilisation rate	E-Stream	EFT & max-min fairness
Ghafouri et al. (2019)	Minimize makespan and cost	Cloud	back-tracking
Xie et al. (2019)	Minimize makespan and cost	Cloud & Edge	PSO
Eldesokey et al. (2021)	Minimize makespan, execution time and cost	Cloud	PSO & SSO
Attiya et al. (2022)	Minimize makespan	Cloud & IoT	MRFO & SSO
Kumar et al. (2019)	Minimize makespan	Cloud	B&B
Al-Maytami et al. (2019)	Minimize makespan and cost	Cloud	DAG
Gade et al. (2022)	Minimize makespan	Cloud	NALCA
Tarafdar et al. (2021)	Minimize makespan and energy consumption	Cloud	Greedy & ACO
Zhang and Shi (2021)	Minimize makespan and execution time	Cloud	ACO
Xia et al. (2022)	Minimize makespan	Cloud	TOPSIS & MT
Tian et al. (2016)	Minimize makespan	Cloud	JA & MapReduce
[Our Proposal]	**Minimize makespan**	**Enterprise Application Integration (EAI)**	**RR**

"cores".

Several jobs are typically processed at a particular time so that several job instances can be used. The processing of a job corresponds to executing all tasks of a path which results in the accomplishment of the job. The execution model of runtime systems establishes how they must execute tasks and allocate threads during the processing of messages in an integration process (Freire, Frantz and Roos-Frantz, 2019).

3.1 Task-Based Execution Model

In this section, we describe the interactions amongst a scheduler, a task, and threads in systems that adopt the task-based model. As shown in Fig. 1.

If there are messages in all communication channels, and if they are sources of a task, then this task is ready to be executed. Ready tasks depend on an available thread to execute them. Meanwhile, their executions are annotated in a waiting queue. Tasks are, therefore, instantiated and executed by following a First-in-first-out (FIFO) policy, in which the task that was first annotated is scheduled to be executed first. Threads are usually grouped in pools, so the creation of consecutive threads is avoided, and task requests are quickly handled (Jeon and Jung, 2018).

The "scheduler" creates, manages, and releases threads. It can also configure the pool by determining parameters, such as the initial thread number, the maximum number of threads, and the maximum lifetime of an idle thread. The "scheduler" assigns threads to execute instances of tasks, and after an instance of the task is executed, the thread is released back to the pool. The processing of a message in the next task now depends on a new assignment of an available thread from the pool to this task. A message is processed in an order dependent on the

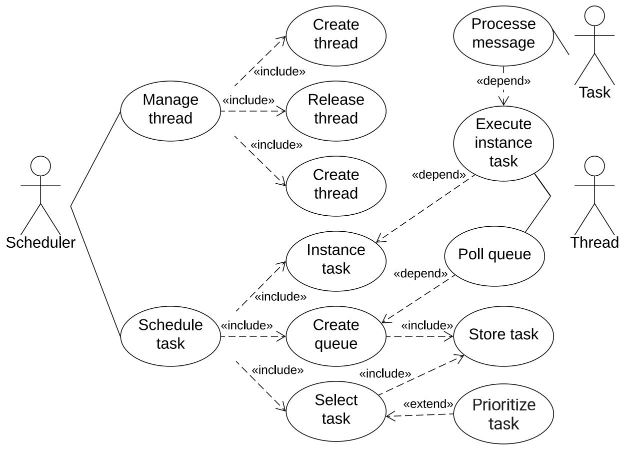

Figure 1: Actions involved in the scheduling of integration processes.

task of the path, which is composed of several segments. Tasks in sequential segments are sequentially executed, whereas tasks in parallel segments can be executed simultaneously once they are not interdependent.

4 Problem Formulation

In this section, we formulate our research problem. Firstly, we describe the software ecosystem and the integration process of a real-world problem. Then, we describe the terminology of the task scheduling problem in integration processes, which is based on the classic real-time scheduling theory and general-purpose parallel systems. Lastly, we present a mathematical formulation

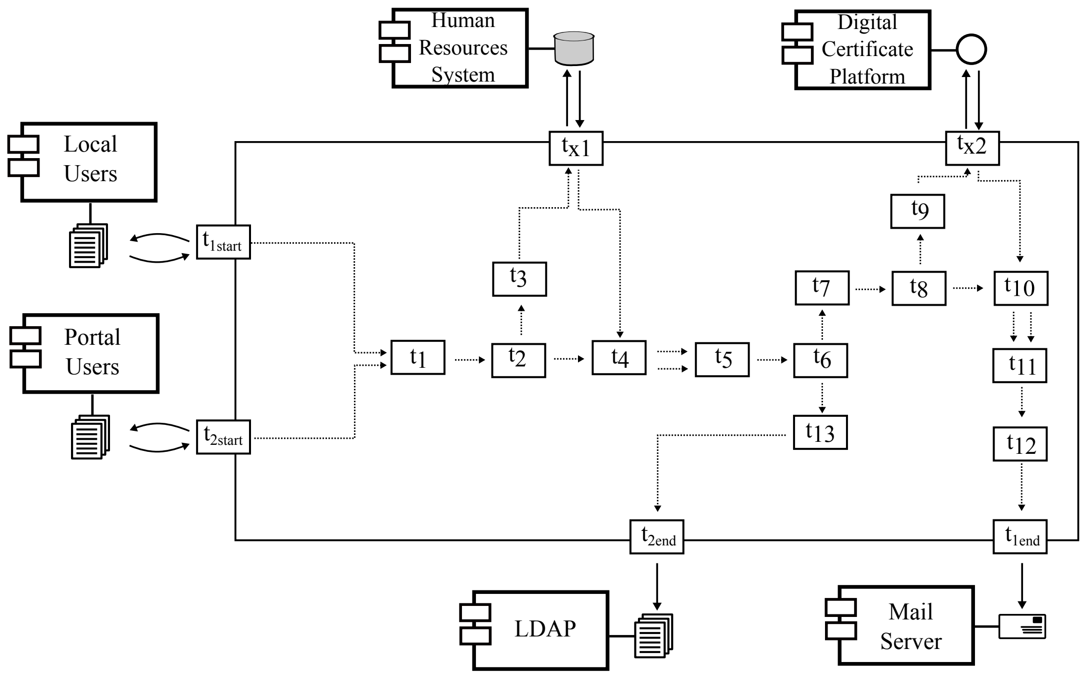

Figure 2: Huelva's County Council conceptual model (adapted from (Frantz et al., 2016)).

composed of the problem definition and the objective function. The former is the modelling and codification of the problem; the latter measures the adequacy of the heuristic in order to maximize the number of processed messages and minimize makespan.

4.1 The Software Ecosystem

The Huelva's County Council problem is a real-world integration process that automates user registration into a central repository (Frantz et al., 2016). Its conceptual model is depicted in Fig. 2.

Within this integration process, integrated applications are: "Local Users", "Portal Users", "Lightweight Directory Access Protocol (LDAP)", "Human Resources System", "Digital Certificate Platform", and "Mail Server". The "Local Users" are one of the source applications to manage data from users' information systems from the county council. The "Portal Users" is another source application which manages users in the web portal. The "Human Resources System" is the application that provides employees with personal information. Information such as name and e-mail are required to compose notification e-mails. The "Digital Certificate Platform" is the application that manages digital certificates. Finally, the "Mail Server" is the application that runs the e-mail service and is used exclusively for notification purposes.

4.2 Terminology

The task scheduling of integration processes can be represented as a set of jobs, $J = \{j_1, j_2, \cdots, j_n\}$ of the same capability on computational resources, consisting of m threads. Every job can have infinite paths of job instances j_i with $i = 1, \cdots n$. A path can have segments of tasks, which can be sequential, parallel, or both. Tasks in sequential segments are executed, obeying their order of dependence. Tasks of parallel segments can be simultaneously executed on different cores.

The DAG represents task models for real-time scheduling, allowing the description of constraints on tasks execution (Saifullah et al., 2013). In the DAG model, an integration process is described as a workflow W composed of k tasks, being an extension of the DAGs with weighted vertices (E_i, T_i), where $T_i = \left\{t_{i,1}, t_{i,2}, \cdots, t_{i,k}\right\}$ is the set of vertices and E is the set of edges. Every vertex in the graph represents a process task, and each edge represents a communication channel, which indicates precedence constraints among tasks. Every edge has a weight, representing the task's waiting time in the queue.

Ritter et al. (2018) represented the integration process as a directed graph called Integration Pattern Typed Graph (IPTG). IPTG was defined as a set of nodes T and edges $E \subseteq T \times T$ and a function $type : T \rightarrow F$, where $F = \{start, end, message\ \ processor, fork, join, condition, merge, external\ call\}$. For a node $t \in T \cdot t = \{t' \in T \mid (t' \cdot t) \in E\}$ for the set of direct predecessors of t, and $t\cdot = \{t'' \in T \mid (t \cdot t'') \in E\}$ for the set of direct successors of t.

The function *type* records what type of task each node represents. The first correctness condition claims that an integration pattern has at least one input and one output; the second condition indicates the cardinality of the involved tasks, i.e., the in-degrees and out-degrees of a node. The last condition states, "the graph (T, E) is connected and acyclic," indicating that a graph represents only a task and its relation with its predecessor and successor tasks and that messages do not loop back to previous tasks. From the IPTG representation, we adopted the condition of verification, the classification by task cardinality, and some terminologies, such as *type start*, *end*, *join*, *message processor*, and *external call*. Since we have considered the logic operation of the task, we then added *and*, *or*, and *or** function *types*. We called our representation of Integration Operation Typed Graph (IOTG).

An IOTG (T, E, *type*) is *correct* if the following conditions apply:

- $\exists\, t_1, t_2 \in T$ with $type\ (t_1) = start$ and $type\ (t_2) = end$;
- if $type\ (t) \in \{and\}$ then $|\cdot t| = 1$ and $|t\cdot| = n$ must produce messages to all n outputs;
- if $type\ (t) \in \{or\}$ then $|\cdot t| = 1$ and $|t\cdot| = n$ produce message in at least one of its outputs;
- if $type\ (t) \in \{or^*\}$ then $|\cdot t| = 1$ and $|t\cdot| = n$ produce message in only one of its outputs;
- if $type\ (t) \in \{join\}$ then $|\cdot t| = n$ and $|t\cdot| = 1$;
- if $type\ (t) \in \{message\ processor\}$ then $|\cdot t| = 1$ and $|t\cdot| = 1$;
- if $type\ (t) \in \{external\ call\}$ then $|\cdot t| = 1$ and $|t\cdot| = 2$;
- the graph (T, E) is connected and acyclic.

4.3 Problem Definition

There are two input tasks represented by t_{1start} and t_{2start}, and two output tasks represented by t_{1end} and t_{2end}. Tasks which exchange messages with applications during runtime are represented by t_{x1} and t_{x2}. Intermediary tasks are represented by t_i, where i ranges from 1 to 13. In the integration logic of SC3, arriving data o users from t_{1start} and t_{2start} are replicated and one copy flows towards "Human Resources System" for information about the employee who has a user record. Further, on t_6, the message is replicated, and while one copy flows towards "LDAP", another one flows towards "Digital Certificate Platform". "Digital Certificate Platform" represents the application that manages digital certificates. The sending of the certificate and the notification to the employee about the inclusion in the "Lightweight Directory Access (LDA)" is done by "Mail Server". The path for a local user that has an e-mail address is the task segment $s_1 = t_{1start}$, t_1, t_2, t_3, t_4, t_{x1}, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{1end}. The path for a local user that has not an e-mail address is the following task segment $s_2 = t_{1start}$, t_1, t_2, t_3, t_4, t_{x1}, t_5, t_6, t_{13}, t_{2end}. The path for a web user that has an e-mail address follows the task segment $s_3 = t_{2start}$, t_1, t_2, t_3, t_4, t_{x1}, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{1end}. The path for a web user who has not an e-mail address is the task segment $s_4 = t_{2start}$, t_1, t_2, t_3, t_4, t_{x1}, t_5, t_6, t_{13}, t_{2end}. Examples of tasks that can be executed in parallel in the SC3 integration process are $[t_3, t_7]$, $[t_9, t_{10}]$, $[t_7, t_{18}]$. A DAG task model represents the Huelva's County Council integration process, as shown in Fig. 3.

There are 19 nodes representing the 19 tasks: t_{1start}, t_{2start}, t_{1end}, t_{2end}, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{x_1}, t_{x_2}. t_{1start} and t_{2start} are starting nodes, while t_{1end} andt_{2end} are *end* nodes. The nodes t_{x_1}, t_{x_2} are tasks that send and receive information to/from applications. There are 20 edges representing the 20 channels.

In the Integration Operation Typed Graph, t_{1start} and t_{2start} are *start* tasks; t_2, t_8 are *and* tasks; t_6 is the *or** task; t_1, t_4, t_{10} are *join*; t_3, t_5, t_7, t_9, t_{11}, t_{12}, t_{13} are the ones from the *message processor*; t_{x1}, t_{x2} are *external call* tasks; and t_{1end}, t_{2end} are *end* tasks.

4.4 Mathematical Formulation

The total processing time of a message in a given job instance, TP_{j_i}, is defined by the elapsed time interval between the time a message is entered and the time it leaves the workflow. TP_{j_i} is the sum of the execution time of all the path tasks by which the message must flow for its complete processing, as shown in Equation 1. We assume that the execution time of a task, TE_{t_k} includes all times involved, such as the total CPU time, the waiting time of the tasks in a queue, and the waiting time of the task in request and response operations with external applications. The number of tasks in the path is represented by *tot*. We also assumed that the range of a task execution time t_k is defined as $[te_{t_{kini}}, te_{t_{kfin}}]$.

$$TP_{j_i} = \sum_{1}^{tot} TE_{t_k},\ where\ \left\{TE_{t_k} \in \mathbb{R} \mid te_{t_{kini}} \leq TE_{t_k} \leq te_{t_{kfin}}\right\} \tag{1}$$

Makespan is calculated by the average of job instances accomplished during a given elapsed time, Δt, c.f. Equation 2. The total number of job instances accomplished during an elapsed time is $|j_i|$. This formulation is represented by the objective function shown in Equation 3.

$$Makespan_{\Delta t} = \frac{\sum_{1}^{|j_i|} TP_{j_i}}{|j_i|} \tag{2}$$

$$\min\{Makespan\} \tag{3}$$

Thus, the problem can be formulated as:

Find out an algorithm for task scheduling that minimizes makespan under high workloads and the uncertain environment from integration processes.

5 Our Proposal

We propose the Multi-queue Round Robin (MqRR) algorithm to tackle high workloads and the uncertain environment of integration processes. Our algorithm is lightweight and deals with the dynamicity of the application integration environment without harming the execution performance of integration processes. The main algorithm of our proposal uses the Round Robin heuristic, a classical scheduling algorithm popularly known for its simplicity, efficiency and effectiveness in computing (Sun et al., 2015; Zhang et al., 2018; Xie et al., 2019).

Two main components of the implementation are

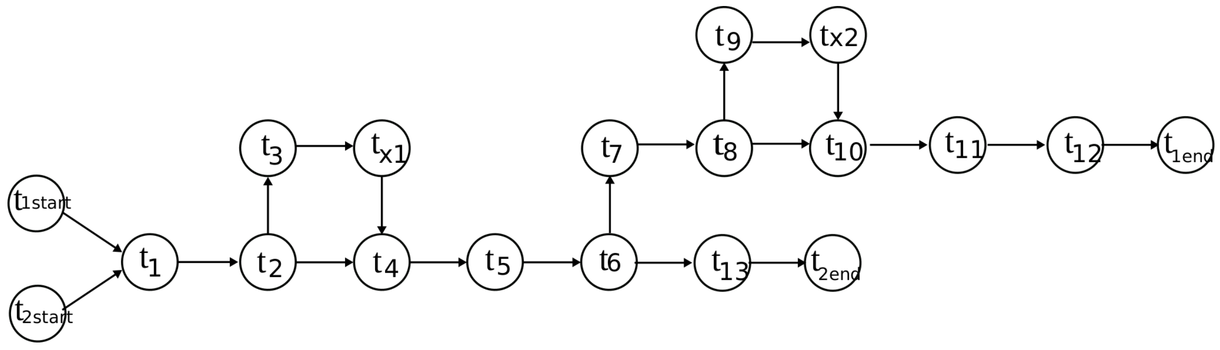

Figure 3: Huelva's County Council represent in a DAG task model.

worth describing in detail: Multi-queue Round Robin and Allocate Thread. The former applies the Round Robin heuristic to queues of tasks, which maintains tasks of an integration process. Allocate Thread manages the execution of tasks by threads and sending tasks to queues.

We analyze the proposed algorithms and discuss their computational complexity. It should be noted that the data input for the analysis (n) is defined by the number of tasks from the integration process. For the analysis purposes, we will consider only the worst-case execution of all algorithms.

5.1 Multi-queue Round Robin

The Multi-queue Round Robin algorithm performs the heuristic that coordinates the task scheduling. It configures multiple queues of tasks, and threads poll them in a circular order. Each queue maintains the instances of a task; however, tasks belonging to parallel segments can be maintained in the same queue, for they can be executed in parallel. Tasks are maintained in a queue in decreasing order of arrival time, in which the head of the queue has the first arriving task, whilst its tail has the last one to arrive. Available threads recurrently poll a queue of tasks; if there are any, they are executed by threads. At every polling, a fixed number of tasks (*preemp*) is caught to be executed.

This algorithm receives task queues, the total number of tasks, and the number of tasks that must be executed every time threads check a queue. The last input must be indicated when the execution is with preemption. The algorithm starts by initializing two auxiliary variables: *totsize* and *preemp*. The former corresponds to queue total size, and the latter to preemption.

The algorithm checks all queues from the first to the last task. The algorithm remains there to check queues in a circular order, while there are annotations of tasks to execute in any other queue. The algorithm checks the number of annotations of tasks inside the queue, and if there is no preemption or the queue size is smaller than the *preemp* variable, then the algorithm assigns the queue size to the *preemp*. The queue size is the number of tasks annotated until that time. Otherwise, *preemp* equals the number of tasks set in the preemption, which is an input of the algorithm. Afterwards, this algorithm calls the algorithm that allocates threads and executes tasks.

Algorithm 1 *Multi-queue Round Robin (MqRR)*

```
Input: queues of tasks: queues[ ]
Input: total number of tasks: numtasks
Input: number of tasks performed at a time (preemption): preemptask

 1: totsize ← 1                                ▷ Initialises the queue size
 2: preemp ← preemptask                         ▷ Initializes preemption
 3: while totsize > 0 do
                                      ▷ Execution of tasks of the queues
 4:     for [i] = 1 to numtasks do
 5:         if queues[i] ≠ ∅ then
                                   ▷ Checks whether there is preemption
                                            ▷ and compares with queue size
 6:             if (preemp = 0) or (queues[i].size < preemp) then
 7:                 preemp ← queues[i].size
 8:             else
 9:                 preemp ← preemptask
10:             end if
                                  ▷ Allocate threads to tasks of the queue
11:             Allocate Thread (queues[i], preemp)
12:         end if
13:     end for
14:     totsize ← 0 ▷ Calculates the total size o task queues
15:     for [i] = 1 to numtasks do
16:         totsize ← totsize + queues[i].size
17:     end for
18: end while
```

The algorithm is given the number of tasks, task queues and a preemption value. Line 3 presents the main loop, while there are tasks to be processed, complexity (O(n)). Line 4 presents a loop that goes through all the queues of the different types of tasks. In the worst case, we have one task for each queue, complexity (O(n)). The algorithm gives the complexity of line 11, "Allocate Thread" (O(n)). On line 15, step through all task queues to count existing tasks. In the worst case, each task is of a different type, stored in a specific queue, complexity (O(n)). The remaining lines inside the loop can be grouped in constant complexity (O(1)).

Asymptotically we have:

$$O(n) + O(n) + O(n) + O(n) + O(1) = O(n)$$

Algorithm 2 *Allocate Thread*

```
Input: task queue: $queues[i]$
Input: number of tasks performed at a time: $preemp$
Input: last task vector: $LastTask[\ ]$

1:                                              ▷ Configures thread pool
2:  Creates elastic thread pool
3:  for $[j] = 1$ to $preemp$ do
                                  ▷ Assigns the task of the queue head to task
4:      $task \leftarrow queue[i].head$
5:      if task is not null then
                                                        ▷ Executes task
6:          Submits task to thread pool
                                                   ▷ Store task in next queue
7:          for $[j] = 1$ to $LastTask[\ ].length$ do
                             ▷ Assigns to lasttask the element of LastTask[ ]
8:              $lasttask \leftarrow LastTask[i]$
9:              if $task \neq lasttask$ then
10:                 Stores task in successor task queue
11:             end if
12:         end for
13:     end if
                                                 ▷ Removes task of the queue
14:     Removes task of the $queue[i]$
                                                        ▷ Releases threads
15:     Shutdown thread pool
                              ▷ Compares queue size with the preemption
16:     if $queue[i].size < preemp$ then
17:         $preemp \leftarrow queue[i].size$
18:     end if
19: end for
```

The algorithm has linear complexity.

5.2 Allocate Thread

The allocate Thread algorithm receives a task queue, the preemption, and a vector containing the end tasks.

The algorithm begins with creating a thread pool, which must be elastic and use a specific type of thread pool provided by a multithreading programming language. The algorithm submits the execution of every task to the thread pool, which executes the task operation. Then, the algorithm checks if the task annotation belongs to the vector containing the end tasks. If the task is not an end task, the algorithm stores the annotation of the task execution in the next queue according to the logic of the integration process. Finally, the algorithm removes the annotation from the task of the current queue and destroys the thread pool.

The algorithm receives a queue of tasks, a preemption value and an array containing the final tasks. Line 3 contains a loop with a fixed number of iterations defined by the preemption value; that is, it does not depend on the amount of data in the input of the problem. In this case, the complexity of the loop is defined by the complexity of the body. In other words, the total time complexity or efficiency of the loop is not solely dependent on the number of iterations it performs but is also heavily influenced by how complex or resource-intensive the individual operations performed inside each loop iteration are. The complexity of conditional tests (line 5) depends on the evaluation of its branches. In line 6, we have a loop that runs through the entire vector of final tasks; here, the worst case occurs when all tasks are final tasks; the size of the vector is equal to the number of tasks, so the complexity is O(n). The other lines of the algorithm have constant processing and can be grouped in O(1).

Asymptotically we have:
$O(n) + O(1) = O(n)$
The algorithm has linear complexity.

6 Proof-of-Concept Experiment

In this segment, we present an experiment to compare task scheduling performance using the algorithms FIFO and MqRR. Makespan was the performance metric to evaluate our proposal against the current FIFO implementation at integration platforms. Makespan is a well-known performance metric in the integration community, and it is defined as the total execution time of the integration process for a given message (Canon and Jeannot, 2007; Chirkin et al., 2017).

This experiment is classified in the literature as a termination simulation, in which the output is a function of the initial conditions. To conduct this experiment, we followed a protocol based on Jedlitschka and Pfahl (2005), Wohlin et al. (2012), and Basili et al. (2007), with procedures for controlled experiments in the field of software engineering.

6.1 Research Questions and Hypothesis

This experiment aims to answer the following research questions:

- RQ1: Is there a workload threshold from which the FIFO heuristic decreases its performance to nearly zero?
- RQ2: Is it possible to improve task scheduling performance in high workload executions of integration processes with the use of the MqRR algorithm?

Our hypotheses to such research questions are that:

- H1: There is a workload threshold from which the FIFO heuristic does not process messages.
- H2: MqRR can improve task scheduling performance in high workload executions of integration processes.

6.2 Variables

Independent variables controlled in the execution of the algorithm are:

- `Heuristic`
  The heuristic used to task scheduling. The values tested for this variable were: FIFO and MqRR.

- `Integration process`
  The conceptual model of the integration process. The

value tested for this variable was "Huelva's County Council".

`- Elapsed time (`Δt`)`
The time interval the algorithm is executed. The value tested for this variable was 60 seconds.

`- Workload`
The number of input messages when the algorithm executes the integration process. The values tested for this variable were: 100, 500,000, 1,000,000, 2,000,000, 2,200,000, 2,500,000, and 5,500,000.

`- Rate of message input`
The number of input messages periodically added to the integration process. The value tested for this variable was 100.

The dependent variable measured in the execution of the algorithm was:

`- Makespan`
This variable corresponds to the average processing time of job instances accomplished during the time interval of the experiment.

6.3 Environment and Supporting Tools

The experiments were carried out on a machine equipped with 16 Intel processors, Xeon CPU E5-4610 V4, 1.8 GHz, 32GB of RAM, and Windows Server 2016 Datacenter 64-bit operating system. The programming language used to implement and execute the algorithms was Java, version 8.0 update 152. We chose Java for its platform independence, robust performance, and extensive ecosystem, aligning well with our project's needs for scalability, concurrency management, and compatibility across diverse environments. Afterwards, we used a simulation tool for Enterprise Application Integration, which implements different scheduling heuristics and allows the extraction of performance metrics (Freire, Frantz, Frantz and Basto-Fernandes, 2022a). The Genes software (Cruz, 2006), version 2015.5.0, was used to process descriptive statistics; ANOVA and Scott & Knoot tests were employed to measure makespan in this study.

6.4 Execution and Data Collection

The experiments were conducted by a simulator built on Java, which simulates the execution of integration processes. The simulation starts with a workload of random input messages and receives an average of 100 new random input messages at every execution task. The term "random" means that the time a task spends to process a message varies within an interval. We configured the simulation time to 60 seconds so the simulator interrupts the current task executions after this time. Then, the simulator collects makespan and stores it in a text file. Afterwards, we handled and analyzed data to finally apply statistical tests.

The tasks from integration processes which can be executed in parallel are kept in the same queue. Execution

Table 2: Execution time range by types of tasks in microseconds.

Function *type*	$\Delta TE_{t_{type}}$
start, *end*	$1-2$
and, *or*, *or**	$2-3$
join	$3-4$
message processor	$1-2$
external call	$1-2$

time intervals (in microseconds) that each task can vary are shown in Table 2.

The results are usually statistically analyzed by the method of executions, by which 20-30 executions are sufficient to obtain a population average, in the use of the distribution with more extreme values than a normal distribution (Sargent, 2013); our experiments were repeated 25 times. For each integration process, we repeated the execution 25 times for each heuristic under seven different workload conditions, resulting in 350 different scenarios, summarized in Table 3.

Table 3: Scenarios.

Heuristics	FIFO and MqRR	2
Integration Processes	Huelva's County Council	1
Elapsed time	60 seconds	1
Workloads	100, 500000, 1000000, 2000000, 2200000, 2500000, and 5500000	7
Rate of message input	100	1
Repetitions		25
Scenarios	2 x 1 x 1 x 7 x 1 x 25	**350**

6.5 Results

We present the results of metrics collected during the simulation in tables and charts for each integration process. The statistical theory is indicated to analyze data from experiments on performance (Georges et al., 2007) because it deals with non-determinism in computational systems, such as runtime systems of integration platforms (Frantz et al., 2011). We used the variance analysis (ANOVA) and Scott & Knott statistical tests to evaluate the results.

We used scatter charts to present the makespan average of all repetitions for every workload value, as shown in Fig. 4. The x-axis represents workloads, whilst the y-axis represents makespan average values.

Regression analysis was used to estimate the relation between the dependent variable (makespan) and the independent variable (workload) (Yao and Liu, 2018). In regression analysis, the square of Pearson product-moment correlation coefficient R^2 is a parameter that determines the degree of linear correlation of variables, defined by $R^2 = 1 - \frac{SSE}{SST}$, where SSE is the sum of squared error and SST is the sum of squared total (Kaytez et al., 2015). Thus, R^2 tends to 1 when $SSE \ll SST$.

The Makespan average of the integration process which was tested is a polynomial of degree 6 for the FIFO heuristic, represented by: $\overline{makespan} = -0.42 \cdot w^6 + 9.54 \cdot$

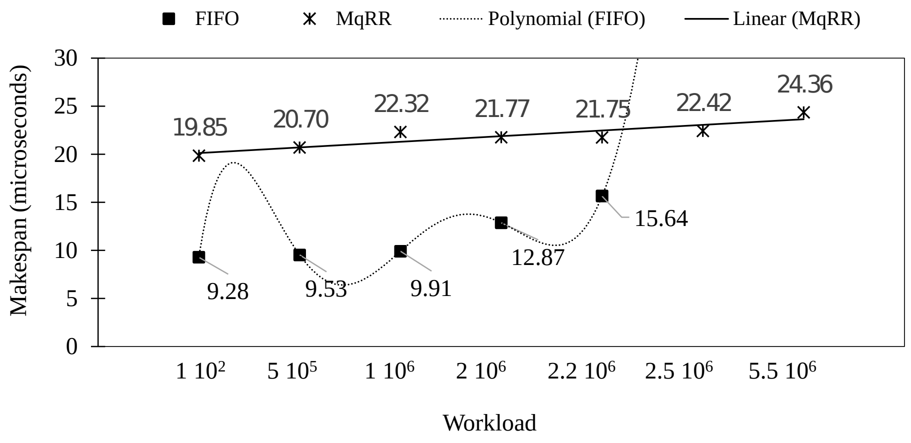

Figure 4: Makespan average of the Huelva's County Council integration process.

$w^5 - 683,81 \cdot w^4 + 367,2 \cdot w^3 - 838.46 \cdot w^2 + 933.47 \cdot w - 378,23$, with $R^2 = 1$. For MqRR heuristic, it is linear, represented by $\overline{makespan} = 0.59 \cdot w + 20$, with $R^2 = 0.79$; as depicted in Fig. 4.

We employed the ANOVA test to differentiate amongst the variations we found in a set of results derived from random factors called error and influenced by the dependent variable. The Scott & Knoot test is considered more rigorous because it only considers relevant differences between independent variables. It is usually adopted in experiments connected to performance due to its simplicity. Table 4 presents a makespan analysis of variance. The ANOVA from makespan shows that the average square was 47,430.58 for the heuristics and 292.81 for error in the integration process we tested. The overall average was equal to 53.22 microseconds and the coefficient of variation was 32.15%.

Table 4: Variance analysis of makespan.

Sources of variation	Degree of freedom	Average square
Heuristics	1	47,430.58 †
Error	48	292.81
Total	49	
Overall average	53.22	
Coefficient of variation (%)	32.15	

† *significant statistical by Fisher-Snedecor's Probability and error level of 5%.*

The Makespan comparison test by Scott & Knott is present in Table 5. The heuristics are in the first column, the means of the dependent variables are in the second one, and the group of Scott & Knott test is in the third column. These test groups of heuristics can check if there is a statistical difference among them. There were two groups: "a" and "b". Group "a" refers to the heuristic in which makespan was the highest, while group "b" refers to the heuristic in which the makespan was the lowest. For the integration process we tested, with the workload of 2,500,000, FIFO was in group "a", and MqRR was in "b".

Table 5: Average of makespan by Scott & Knott test.

Heuristic	Makespan average	Group
FIFO	84.02	a
MqRR	22.42	b

Error level of 5% by Scott & Knoot model.

6.6 Discussion and Comparison

For Huelva's County Council integration process, the average makespan is lower using FIFO than by MqRR to a certain threshold. Then it becomes higher, as shown in Fig. 4. A concave upward represents this behaviour with FIFO, and this behaviour change occurs with a workload between 2,200,000 and 2,500,000 messages. There were more than 250,000 messages and some executions did not process them. In the case of MqRR, the average makespan varies between 19.85 to 24.36 microseconds, with workloads between 100 and 5,500,000 messages. This behaviour is represented by a linear function, whose growth is slow compared to the growth of FIFO with workloads above 2,200,000 messages.

Different heuristics generate a significant difference in makespan average, as shown in Table 4. The coefficient of variation was reduced, indicating that the experiment was adequate and reliable. The average comparison test from Scott & Knott showed that MqRR achieved the best performance, in which the makespan was 22.42 microseconds in the integration process with a workload of 2,500,000 messages. There were two different groups in this test; thus, there was a statistical difference between the two heuristics.

Regarding the research questions and hypotheses:

- **RQ1:** There is a threshold from which the FIFO heuristic does not process any messages: it is above 2,500,000 messages.
- **RQ2:** MqRR improved performance of the task scheduling in high workload execution of integration processes. Executions performed better through MqRR than FIFO when an integration process tested over 2,500,000 messages. Scott & Knott test proved there

was a statistical difference between MqRR and FIFO in those cases.

6.7 Threats to Validity

Threats to validity are usual in any empirical research (Cruzes and ben Othman, 2017), and some of those threats are more specific to optimization studies (Wohlin et al., 2012). Next, We describe how we evaluated some factors that could influence the experiment results, and how we tried to mitigate them.

6.7.1 Constructor Validity

Constructor validity discusses whether the planning and execution of the study are well adequate to answer research questions. We planned the experiment according to procedures from empirical software engineering (Jedlitschka and Pfahl, 2005; Basili et al., 2007; Wohlin et al., 2012). Our primary steps were to define our research question, formulate our hypothesis, and define both independent and dependent variables. Next, we provided information about the execution environment, supporting tools, execution and data collection. Then, we performed our simulation in 350 different scenarios and used statistical techniques to evaluate the results.

6.7.2 Conclusion Validity

As reported by Wohlin et al. (2012), conclusion validity "concerns issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment". We used statistical techniques to ensure that the actual outcome observed in our experiment would relate to the heuristics employed and that there was a significant difference among them.

6.7.3 Internal Validity

Internal validity aims to ensure that the treatment caused the outcome by mitigating the effects of other uncertain, unmeasured factors (Feldt and Magazinius, 2010). Instrumentation and source of noise are possible threats. We experimented with the same machine, which was in security mode, with minimal features and disconnected from the Internet during the executions, in order to minimize interference in the execution time of the algorithm. We built our algorithm in Java. The first executions of codes are usually slower, and it is advisable to let the VM eventually perform code optimization (Pinto et al., 2014). We executed the algorithm only once to warm up Java's VM. Additionally, the researchers accurately inspected the procedures and used statistical tests to validate the measures.

6.7.4 External Validity

External validity focuses on the generalization of results out of the scope of our study (Feldt and Magazinius, 2010). This study is generalized for integration platforms that adopt the integration patterns by Hohpe and Woolf (2004), the Pipes-and-Filters style, and the task-based model. We reported this study following an empirical guideline (Wohlin et al., 2012) to make repetition possible. The experiment is valid to test other parameters, such as integration processes, message arrival rate, and simulation duration. In future work, we intend to experiment with an extensive data set to evaluate the generalization of results.

7 Conclusion

IoT and cloud computing expand the possibilities for enterprises, but they also increase the volume of data and the complexity of integration processes. Integration platforms support software engineers by providing runtime systems to orchestrate workflows of tasks that implement the communication between heterogeneous applications and services.

This article proposed the Multi-queue Round Robin (MqRR) algorithm, a task scheduling heuristic designed for the runtime environment of enterprise application integration platforms. The main goal of MqRR is to improve the behaviour of the traditional FIFO heuristic under high workloads, avoiding starvation and making better use of computational resources.

We conducted a proof-of-concept experiment using three real-world integration workflows and an integration process simulator, comparing MqRR with FIFO under increasing message arrival rates. The results answer our research questions as follows. For RQ1, we observed that FIFO achieves a lower makespan when the workload is low, but as the input rate increases its performance degrades sharply, until it reaches a threshold from which it practically stops processing messages. For RQ2, the experiments show that MqRR improves task scheduling performance in high workload scenarios: the makespan grows approximately linearly with the input rate, and the number of completed messages is significantly higher than with FIFO.

As future work, we plan to extend this study in several directions. First, we intend to evaluate MqRR in additional integration platforms and in multi-tenant scenarios, considering not only makespan but also monetary cost and energy consumption. Second, we aim to compare MqRR with more advanced scheduling heuristics and metaheuristics commonly used in cloud computing, such as PSO, GA and hybrid approaches. Finally, we envision incorporating MqRR into a production-ready integration platform and conducting case studies with real enterprise workloads, in order to assess its impact on the quality of service perceived by end users.

Acknowledgements

This research is partially funded by the Co-ordination for the Brazilian Improvement of Higher Education Personnel (CAPES) and the Brazilian National Council for Scientific and Technological Development (CNPq) under the following project grants 311011/2022-5, 309425/2023-9, 402915/2023-2. We would like to thank Dr Maria do Rosário Laureano and Dr Sancho M. Oliveira from the Instituto Universitário de Lisboa (ISCTE-IUL) ISTAR-IUL, Lisboa, Portugal, for their helpful comments in earlier versions of this article.

References

Al-Maytami, B. A., Fan, P., Hussain, A., Baker, T. and Liatsis, P. (2019). A task scheduling algorithm with improved makespan based on prediction of tasks computation time algorithm for cloud computing, *IEEE Access* **7**: 160916–160926. https://doi.org/10.1109/ACCESS.2019.2948704.

Alexander, C., Ishikawa, S. and Silvertein, M. (1977). *A pattern language: towns, buildings, construction*, Oxford University Press, Oxford, United Kingdom. https://doi.org/10.2307/1574526.

Alkhanak, E. N., Lee, S. P., Rezaei, R. and Parizi, R. M. (2016). Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues, *Journal of Systems and Software* **113**: 1–26. https://doi.org/10.1016/j.jss.2015.11.023.

Anwar, N. and Deng, H. (2018). Elastic scheduling of scientific workflows under deadline constraints in cloud computing environments, *Future Internet* **10**(5): 1–23. Elasticschedulingofscientificworkflowsunderdeadlineconstraintsincloudcomputingenvironments.

Attiya, I., Elaziz, M. A., Abualigah, L., Nguyen, T. N. and ElLatif, A. A. A. (2022). An improved hybrid swarm intelligence for scheduling iot application tasks in the cloud, *IEEE Transactions on Industrial Informatics* . https://doi.org/10.1109/TII.2022.3148288.

Basili, V. R., Rombach, D., Kitchenham, K. S. B., Selby, D. and Pfahl, R. W. (2007). *Empirical Software Engineering Issues*, Springer Berlin/Heidelberg, Berlin, Germany. https://doi.org/10.1007/978-3-540-71301-2_10.

Basu, S., Karuppiah, M., Selvakumar, K., Li, K.-C., Islam, S. K. H., Hassan, M. M. and Bhuiyan, M. Z. A. (2018). An intelligentcognitive model of task scheduling for IoT applications in cloud computing environment, *Future Generation Computer Systems* **88**: 254–261. https://doi.org/10.1016/j.future.2018.05.056.

Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A. and Kennedy, K. (2005). Task scheduling strategies for workflow-based applications in grids, *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Vol. 2, pp. 759–767. https://doi.org/10.1109/CCGRID.2005.1558639.

Boehm, M., Habich, D., Preissler, S., Lehner, W. and Wloka, U. (2011). Cost-based vectorization of instance-based integration processes, *Information Systems* **36**(1): 3–29. https://doi.org/10.1016/j.is.2010.06.007.

Canon, L.-C. and Jeannot, E. (2007). A comparison of robustness metrics for scheduling DAGs on heterogeneous systems, *International Conference on Cluster Computing (IEEE Cluster)*, pp. 558–567.

Chirkin, A. M., Belloum, A. S. Z., Kovalchuk, S. V., Makkes, M. X., Melnik, M. A., Visheratin, A. A. and Nasonov, D. A. (2017). Execution time estimation for workflow scheduling, *Future Generation Computer Systems* **75**: 376–387. https://doi.org/10.1016/j.future.2017.01.011.

Cruz, C. D. (2006). *Programa Genes: estatística experimental e matrizes*, Editora Universidade Federal de Viçosa, Viçosa, Brazil. https://arquivo.ufv.br/dbg/genes/gdown1.htm.

Cruzes, D. S. and ben Othman, L. (2017). Threats to validity in empirical software security research, *Empirical Research for Software Security*, pp. 295–320. https://www.taylorfrancis.com/books/mono/10.1201/9781315154855/empirical-research-software-security?refId=7830b03b-b9d3-4b27-9625-c08766da71e2&context=ubx.

Eldesokey, H., abd elatty, S., El-Shafai, W., Amoon, M. and ElSamie, F. A. (2021). Hybrid swarm optimization algorithm based task scheduling in cloud environment, *International Journal of Communication Systems* **34**. https://doi.org/10.1002/dac.4694.

Fan, K., Zhai, Y., Li, X. and Wang, M. (2018). Review and classification of hybrid shop scheduling, *Production Engineering* **12**(5): 597–609. https://doi.org/10.1007/s11740-018-0832-1.

Feldt, R. and Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey., *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 374–379. https://api.semanticscholar.org/CorpusID:12670942.

Frantz, R. Z., Corchuelo, R. and Arjona, J. L. (2011). An efficient orchestration engine for the cloud, *International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 711–716. https://doi.org/10.1109/CloudCom.2011.110.

Frantz, R. Z., Corchuelo, R. and Molina-Jiménez, C. (2012). A proposal to detect errors in enterprise application integration solutions, *Journal of Systems and Software* **85**(3): 480–497. https://doi.org/10.1016/j.jss.2011.10.048.

Frantz, R. Z., Corchuelo, R. and Roos-Frantz, F. (2016). On the design of a maintainable software development kit to implement integration solutions, *Journal of Systems and Software* **111**(1): 89–104. https://doi.org/10.1016/j.jss.2015.08.044.

Freire, D. L., Frantz, R. Z., Basto-Fernandes, V., Sawicki, S. and Roos-Frantz, F. (2022). Queue-priority optimized algorithm: a novel task scheduling for runtime systems of application integration platforms, *Journal of Supercomputing* **78**(1): 1501—-1531. https://doi.org/10.1007/s11227-021-03926-x.

Freire, D. L., Frantz, R. Z., Frantz, R.-F. and Basto-Fernandes, V. (2022a). Integration process simulator: A tool for performance evaluation of task scheduling of integration processes, *Journal of Simulation* **16**(6): 604–623. https://doi.org/10.1080/17477778.2022.2041989.

Freire, D. L., Frantz, R. Z., Frantz, R.-F. and Basto-Fernandes, V. (2022b). Task scheduling characterisation in enterprise application integration, *The Journal of Supercomputing* pp. 1–39. https://doi.org/10.1007/s11227-021-04119-2.

Freire, D. L., Frantz, R. Z. and Roos-Frantz, F. (2019). Towards optimal thread pool configuration for run-time systems of integration platforms, *International Journal of Computer Applications in Technology* **62**(2): 129–147. https://doi.org/10.1504/IJCAT.2020.104692.

Freire, D. L., Frantz, R. Z., Roos-Frantz, F. and Sawicki, S. (2019). Survey on the run-time systems of enterprise application integration platforms focusing on performance, *Software: Practice and Experience* **49**(3): 341–360. https://doi.org/10.1002/spe.2670.

Gade, A., Bhat, M. N. and Thakare, N. (2022). Task pattern identification and scheduling using equal opportunity model for minimization of makespan and task diversity in cloud computing, *Pattern Recognition and Image Analysis* **32**: 67–77. https://doi.org/10.1134/S1054661821040088.

Georges, A., Buytaert, D. and Eeckhout, L. (2007). Statistically rigorous java performance evaluation, *ACM SIGPLAN Notices* **42**(10): 57–76. https://doi.org/10.1145/1297027.1297033.

Ghafouri, R., Movaghar, A. and Mohsenzadeh, M. (2019). A budget constrained scheduling algorithm for executing workflow application in infrastructure as a service clouds, *Peer-to-Peer Networking and Applications* **12**(1): 241–268. https://doi.org/10.1007/s12083-018-0662-0.

Guo, F., Yu, L., Tian, S. and Yu, J. (2015). A workflow task scheduling algorithm based on the resources' fuzzy clustering in cloud computing environment, *International Journal of Communication Systems* **28**(6): 1053–1067. https://doi.org/10.1002/dac.2743.

Guttridge, K., Pezzini, M., Golluscio, E., Thoo, E., Iijima, K. and Wilcox, M. (2017). Magic quadrant for enterprise integration platform as a service 2017, *Technical report*, Gartner, Inc. https://www.gartner.com/en/documents/3645397.

Hilman, M. H., Rodriguez, M. A. and Buyya, R. (2018). Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions, *ACM Computing Surveys* **1**(1): 1–33. https://doi.org/10.48550/arXiv.1809.05574.

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*, Addison-Wesley Professional, Oxford, United Kingdom. https://dl.acm.org/doi/book/10.5555/940308.

Jedlitschka, A. and Pfahl, D. (2005). Reporting guidelines for controlled experiments in software engineering, *International Symposium on Empirical Software Engineering (ESEM)*, pp. 95–104. https://doi.org/10.1109/ISESE.2005.1541818.

Jeon, S. and Jung, I. (2018). Experimental evaluation of improved IoT middleware for flexible performance and efficient connectivity, *Ad Hoc Networks* **70**: 61–72. https://doi.org/10.1016/j.adhoc.2017.11.005.

Kanagaraj, K. and Swamynathan, S. (2016). A study on performance of dominant scheduling algorithms on standard workflow systems in cloud, *International Conference on Informatics and Analytics (ICIA)*, pp. 45:1–45:6. hîÃps://doi.org/10.1145/2980258.2980358.

Kaytez, F., Taplamacioglu, M. C., Cam, E. and Hardalac, F. (2015). Forecasting electricity consumption: A comparison of regression analysis, neural networks and least squares support vector machines, *International International of Electrical Power and Energy Systems* **67**: 431–438. https://doi.org/10.1016/j.ijepes.2014.12.036.

Kumar, A. M. S., Parthiban, K. and Shankar, S. S. (2019). An efficient task scheduling in a cloud computing environment using hybrid genetic algorithm - particle swarm optimization (ga-pso) algorithm, *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, pp. 29–34. https://doi.org/10.57159/gadl.jcmm.2.4.23076.

Linthicum, D. S. (2017). Cloud Computing Changes Data Integration Forever: What's Needed Right Now, *IEEE Cloud Computing* **4**(3): 50–53. https://doi.org/10.1109/MCC.2017.47.

Manasrah, A. M. and Ali, H. B. (2018). Workflow scheduling using hybrid GA-PSO algorithm in cloud computing, *Wireless Communications and Mobile Computing* **2018**: 1–16. https://doi.org/10.1155/2018/1934784.

Mohammad Aazam and Eui-Nam Huh and Marc St-Hilaire and Chung-Horng Lung and Ioannis Lambadaris (2016). Cloud of Things: Integration of IoT with Cloud computing, *in* A. Koubaa and E. Shakshuki (eds), *Robots and Sensor Clouds*, Springer International Publishing, Cham, pp. 77–94. https://doi.org/10.1007/978-3-319-22168-7_4.

Pezzini, M., Natis, Y. V., Malinverno, P., Iijima, K., Thompson, J., Thoo, E. and Guttridge, K. (2015). Magic quadrant for enterprise integration platform as a service, *Gartner, Stamford* pp. 1–35. https://www.mendix.com/press/gartner-2015-magic-quadrant-for-enterprise-application-platform-as-a-service-worldwide/.

Pietri, I., Chronis, Y. and Ioannidis, Y. (2019). Fairness in dataflow scheduling in the cloud, *Information Systems* **83**: 118–125. https://doi.org/10.1016/j.is.2019.03.003.

Pinto, G., Castor, F. and Liu, Y. D. (2014). Understanding energy behaviors of thread management constructs, *ACM SIGPLAN Notices*, Vol. 49, pp. 345–360. https://doi.org/10.1145/2714064.266023.

Rimal, B. P. and Maier, M. (2017). Workflow scheduling in multi-tenant cloud computing environments, *IEEE Transactions on parallel and distributed systems* **28**(1): 290–304. https://doi.org/10.1109/TPDS.2016.2556668.

Ritter, D., Forsberg, F. N. and Rinderle-Ma, S. (2018). Optimization strategies for integration pattern compositions, *International Conference on Distributed and Event-based Systems (DEBS)*, pp. 88–99. https://doi.org/10.1145/3210284.3210295.

Rodriguez, M. A. and Buyya, R. (2018). Scheduling dynamic workloads in multi-tenant scientific workflow as a service platforms, *Future Generation Computer Systems* **79**: 739–750. https://doi.org/10.1016/j.future.2017.05.009.

Saifullah, A., Li, J., Agrawal, K., Lu, C. and Gill, C. (2013). Multi-core real-time scheduling for generalized parallel task models, *Real-Time Systems* **49**(4): 404–435. https://doi.org/10.1109/RTSS.2011.27.

Sargent, R. G. (2013). Verification and validation of simulation models, *Journal of simulation* **7**(1): 12–24. https://doi.org/10.1109/WSC.2010.5679166.

Sharma, S. (2017). Ovum decision matrix highlights the growing importance of ipaas and api platforms in hybrid integration, *Technical report*, Ovum Consulting. https://boomi.com/content/report/ovum-decision-matrix-report.

Shoukry, A., Khader, J. and Gani, S. (2019). Improving business process and functionality using IoT based E3-value business model, *Electronic Markets* **1**: 1–10. https://doi.org/10.1007/s12525-019-00344-z.

Stavrinides, G. L. and Karatza, H. D. (2018). A hybrid approach to scheduling real-time IoT workflows in fog and cloud environments, *Multimedia Tools and Applications* . https://doi.org/10.1007/s11042-018-7051-9.

Sun, D., Yan, H., Gao, S., Liu, X. and Buyya, R. (2015). An integrated approach to workflow mapping and task scheduling for delay minimization in distributed environments, *Journal of Parallel and Distributed Computing* **84**: 51–64. https://doi.org/10.1016/j.jpdc.2015.07.004.

Sun, D., Yan, H., Gao, S., Liu, X. and Buyya, R. (2018). Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams, *The Journal of Supercomputing* **74**(2): 615–636. https://doi.org/10.1007/s11227-017-2151-2.

Tarafdar, A., Debnath, M., Khatua, S. and Das, R. K. (2021). Energy and makespan aware scheduling of deadline sensitive tasks in the cloud environment, *Journal of Grid Computing* **19**. https://doi.org/10.1007/s10723-021-09548-0.

Tian, W., Li, G., Yang, W. and Buyya, R. (2016). Hscheduler: an optimal approach to minimize the makespan of multiple mapreduce jobs, *The Journal of Supercomputing* . https://doi.org/10.1007/s11227-016-1737-4.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A. (2012). *Experimentation in software engineering*, Springer Science & Business Media, New York, United Kingdom. https://link.springer.com/book/10.1007/978-3-662-69306-3.

Xia, Y., Zhan, Y., Dai, L. and Chen, Y. (2022). A cost and makespan aware scheduling algorithm for dynamic multi-workflow in cloud environment, *The Journal of Supercomputing* . https://doi.org/10.1007/s11227-022-04681-3.

Xie, Y., Zhu, Y., Wang, Y., Cheng, Y., Xu, R., Sani, A. S., Yuan, D. and Yang, Y. (2019). A novel directional and non-local-convergent particle swarm optimization based workflow scheduling in cloud-edge environment, *Future Generation Computer Systems* **97**: 36–378. https://doi.org/10.1016/j.future.2019.03.005.

Yao, K. and Liu, B. (2018). Uncertain regression analysis: an approach for imprecise observations, *Soft Computing-A Fusion of Foundations, Methodologies and Applications* **22**(17): 5579–5582. https://doi.org/10.1007/s00500-017-2521-y.

Zaourar, L., Aba, M. A., Briand, D. and Philippe, J.-M. (2018). Task management on fully heterogeneous micro-server system: Modeling and resolution strategies, *Concurrency and Computation: Practice and Experience* **30**(23): e4798. https://doi.org/10.1002/cpe.4798.

Zhang, R. and Shi, W. (2021). A makespan-optimized task-level scheduling strategy for cloud workflow systems, *2021 2nd International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, pp. 712–720. https://doi.org/10.1109/AINIT54228.2021.00145.

Zhang, Y., Shen, Z.-J. M. and Song, S. (2018). Exact algorithms for distributionally β-robust machine scheduling with uncertain processing times, *INFORMS Journal on Computing* **30**(4): 662–676. ttps://doi.org/10.1287/ijoc.2018.0807.

Zhou, Q., Li, G., Li, J., Shu, L., Zhang, C. and Yang, F. (2017). Dynamic priority scheduling of periodic queries in on-demand data dissemination systems, *Information Systems* **67**: 58–70. https://doi.org/10.1016/j.is.2017.03.005.