ORIGINAL PAPER

Integration Process Simulator: a tool for performance evaluation of task scheduling of integration processes

Daniela L. Freire[a,b] and Rafael Z. Frantz[b], Fabricia Roos-Frantz[b] and Vitor Basto-Fernandes[a]

[a] University Institute of Lisbon (ISCTE-IUL) ISTAR-IUL, Lisbon, Portugal
[b] Unijuí University, Department of Exact Sciences and Engineering, Ijuí, Brazil

ARTICLE HISTORY
Compiled May 10, 2023

ABSTRACT

Due to large volumes of data from Cloud Computing and from the Internet of Things, the companies' software ecosystem requires an efficient integration of applications and services. Performance improvement from integration platforms' runtime systems is directly related to task scheduling strategies from integration processes. It is still a challenge to find the proper heuristic for a given integration process subject to high inbound data rates. This article proposes a simulation tool for the field of Enterprise Application Integration, which implements different scheduling heuristics and allows the extraction of performance metrics. Three task scheduling heuristics were simulated during the integration process, and the results were evaluated through statistical tests.

KEYWORDS
Application integration; task scheduling algorithm; workflow scheduling; simulation tool; high volume data

1. Introduction

Enterprises are being impacted by the vast amount of data to be processed by Cloud Computing (CC) services in the era of large data volume (Ritter, May, & Rinderle-Ma, 2017), having to meet new requirements from end-users, such as fast response and low latency time (Fernández-Cerero, Fernández-Montes, Jakóbik, Kołodziej, & Toro, 2018). The increasing number of heterogeneous devices connected to the Internet of Things (IoT) produce a large volume of information, providing knowledge and creating more business opportunities for enterprises. To cope with large data volumes, enterprises should integrate IoT with CC, so that they could create more value from generated data and develop smart applications for the users (Aazam, Huh, St-Hilaire, Lung, & Lambadaris, 2016).

Enterprise Application Integration (EAI) is the field of study that provides methodologies, techniques and tools for the design and implementation of integration processes (Frantz, Corchuelo, & Roos-Frantz, 2016; Ritter, Rinderle-Ma, Montali, & Rivkin, 2021). Integration platforms are tools which allow software engineers to

CONTACT Daniela L. Freire. Email: danielalfreirer@icmc.usp.br

design, implement, run and monitor integration processes. The conceptual model of an integration process is a workflow composed of segments of atomic tasks connected by communication channels desynchronising one task from another (Kanagaraj & Swamynathan, 2016). Data, wrapped in messages, are received from applications, processed by tasks in the workflows and delivered to destination applications. Currently, several open-source integration platforms support the integration patterns documented by Hohpe and Woolf (2004) and implement the Pipes-and-Filters architectural style (Alexander, Ishikawa, & Silvertein, 1977). Pipes depict message channels and filters represent atomic tasks that implement a particular integration pattern to process messages. In integration platforms, the runtime system is the component responsible for executing tasks; it manages jobs, which are composed of a number of tasks, and distributes computational resources among such tasks (Frantz et al., 2016). The runtime system primary function is task scheduling (Guo, Yu, Tian, & Yu, 2015; Hilman, Rodriguez, & Buyya, 2018). The pay-as-you-go charging model of Cloud Computing Services stimulates a task scheduling mechanism focused on minimising costs (Freire, Frantz, Roos-Frantz, & Sawicki, 2019) and able to handle with large volumes of data from IoT (Shoukry, Khader, & Gani, 2019). That comes in handy with emerging market demands for quality of software, flexibility and response times (Fan, Zhai, Li, & Wang, 2018).

In runtime systems, threads represent computational resources used to execute tasks of an integration process. There are two main execution models for runtime systems in the literature: process-based and task-based (Alkhanak, Lee, Rezaei, & Parizi, 2016; Blythe et al., 2005; Boehm, Habich, Preissler, Lehner, & Wloka, 2011; Frantz, Corchuelo, & Molina-Jiménez, 2012). In the former model, a thread was assigned to an instance of the integration process, so that the thread could execute every task of the workflow over an inbound message, so that this message would flow throughout the process. After every task in the workflow has been executed, the thread is released. In the latter 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 which reaches the task. When the task is finished, 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 depends on assigning an available thread to this very task. This article addresses the task-based execution model, where a performance degradation was detected in scenarios with high input rates of messages. Such scenarios were caused by the accumulation of execution requests from initial tasks in detriment of others, due to the heuristic algorithm for task scheduling, which uses a First-in-First-Out (FIFO) strategy (Frantz et al., 2012).

The heuristic algorithm chosen must improve the execution performance of runtime systems integration processes, especially where high input rates of messages are seen. The heuristic must guarantee an appropriate response time to the message processing and proper harnessing computational resources. However, a heuristic may be optimal for a given integration process and user-parameters, but inappropriate for some other integration process and user-parameters. Straightforward heuristics such as FIFO are usually adopted to achieve consistent results through all scenarios, which includes high and low input rates of messages. However, the performance may vary significantly depending on many factors (different input rates of messages, workloads, type of message traffic, among others (Qureshi, Shah, & Manuel, 2011)), once a single heuristic is rarely optimal for all cases. The simulation-approach can evaluate traditional and novel heuristics and measure their impact on the integration processes execution performance. Thus, a simulation tool able to reproduce the implementation

of integration processes - and subject to high input rates of messages - must be developed. It is noteworthy that no other proposal was found in the literature review. The closest one found was that of Haugg, Frantz, Roos-Frantz, Sawicki, and Zucolotto (2019), but the simulator is limited to a single heuristic and allows the monitoring of only one metric: the number of threads. There are several simulators in the commerce for task scheduling, but they are concerned either with the simulation in cloud computing environment or switch network distribution; they are not specific for integration processes with significant metrics to evaluate the execution performance of such integration processes. Such commercial simulators served as inspiration for the development of a simulator in the EAI context.

This article approaches simulation in the field of EAI by proposing a simulation tool, and developing an experiment aimed at answering the following research question:

RQ: Is it possible to extract performance metrics through simulation, by using different heuristics for task scheduling in the execution of integration processes under high workloads?

The hypothesis for this research question is that:

H: It is possible to develop a simulator to record performance metrics, which is also able to evaluate different heuristics for task scheduling in the execution of integration processes under high workloads.

The idea is an Integration Process Simulator (IPS) which can evaluate various heuristics for task scheduling of integration processes in runtime systems. IPS is a specific simulator for integration processes, with the possibility of monitoring metrics that are unique in this context, for instance: the number of messages processed, the number of incoming messages not processed, makespan and throughput. Besides, the simulator is adapted to receive specific parameters from integration processes, such as the number of input, outputs, communications channels, tasks (and their specific operations), threads, etc. Not only the traditional FIFO heuristic was implemented, but also two new heuristics: Multi-queue Round Robin (MqRR) and Queue Priority (qPrior) (Freire, Frantz, Roos-Frantz, & Basto-Fernandes, 2021). Such proposals look at a fair allocation of threads to integration process tasks in scenarios of high volumes of data. However, the architecture of the IPS makes the incorporation of other heuristic algorithms possible; it simulates several scenarios with varying message input rates, total and/or initial messages workload, simulation time and integration processes. Additionally, several performance metrics are provided by the IPS, such as throughput, the number of processed messages and remained messages. As a proof-of-concept, the heuristic was simulated in the execution of three integration processes. The results of the simulation were statistically validated with the ANOVA and the Scott & Knoot tests, which confirmed the hypothesis. Thus, the simulator allows the adequacy analysis of runtime systems from integration platforms, once it handles high workloads seen in modern EAIs.

This this article is organised as follows: Section 2 introduces works related to heuristic simulators for task scheduling; Section 3 describes the characteristics of scheduling in integration processes; Section 4 formulates the problem of task scheduling on integration processes; Section 5 presents the algorithms of the simulator; Section 6 presents a proof-of-concept to validate the heuristic simulator; while Section 7, the conclusions and future works.

2. Related Work

This section discusses tools which simulate heuritics for task scheduling in the Cloud. Once ideas can be borrowed and adapted in the context of EAI, they may be seen as related work. It is important to stress, though, that the target of these proposals is either task scheduling of services or tasks in the Cloud; on the other hand, this proposal targets task scheduling in integration processes. Most of the articles on heuristics for task scheduling develop their algorithms in a programming language, without taking into account benchmarking standardized methods, frameworks and tools - for instance, Gupta, Gupta, Choudhary, and Jana (2019); Riaz, Kazmi, Kazmi, and Shah (2018); T. Zhang, Pota, Chu, and Gadh (2018). Despite the lack of standardisation of the tools, some simulation tools were found, mainly regarding task scheduling for Cloud Computing. Table 1 lists the tools found in related works and indicates the research field where each of them is employed. In the following paragraphs, these proposals will be discussed.

GridSim is a simulator proposed by Buyya and Murshed (2002), built on top of SimJava library (Howell & McNab, 1998). It allows the modeling and simulation of heterogeneous Grid resources, users and application models, in the research field of Grid Systems. The simulator evaluates popular scheduling algorithms and new proposals. Some performance metrics provided by it include the average budget spent by each user for processing jobs; the average time at which the user experiment is terminated with a varying number of users competing for resources; and the number of jobs processed by each user when a varying number of users are competing for resources.

CloudSim, proposed by Calheiros, Ranjan, Beloglazov, Rose, and Buyya (2011), is based on SimJava (Howell & McNab, 1998) and GridSim (Buyya & Murshed, 2002) and focuses on Infrastructure as a Service (IaaS), and operation environment features in Cloud Computing. It supports the allocation of several virtual machines and migration policies. This tool allows the simulation of space and time-shared scheduling policies. Makespan is the performance metric, which computes the fastest completion time and cost.

WorkflowSim is a simulator, proposed by Kanagaraj and Swamynathan (2016), built on top of CloudSim (Calheiros et al., 2011), that offers support for workflow scheduling and execution in the Cloud Computing research field. This simulator allows the comparison of new heuristics with popular scheduling algorithms, such as the Critical Path Algorithm, the First Come First Serve, MaxMin and MinMin. Makespan is its performance metric.

TrueTime is a Matlab/Simulink-based simulation tool, proposed by Cervin and Årzén (2018), that has been developed at Lund University since 1999 (Eker & Cervin, 1999). It provides models of multi-tasking real-time kernels and networks used in simulation models for Networked Embedded Control Systems. TrueTime gathers metrics such as input-output latency and deadline overruns.

SCORE, proposed by Fernández-Cerero et al. (2018), is based on the Google Omega lightweight simulator (Schwarzkopf, Konwinski, Abd-El-Malek, & Wilkes, 2013). It tests energy-efficiency, security, and scheduling strategies in Cloud Computing environments. This simulator evaluates heuristics, such as Random, Spread tasks to the maximum, Greedy minimising energy, Greedy minimising makespan, Spread tasks to the minimum and Spread tasks to the minimum with randomness. It stores some performance metrics such as the time for a job to wait in the queue until its first task

is scheduled; the time a job remains in the queue until it is scheduled; and yet, the average scheduler utilisation rate.

Sphere, proposed by Fernández-Cerero, Fernández-Montes, Ortega, Jakóbik, and Widlak (2019), is based on SCORE (Cervin & Årzén, 2018) and follows the design and simplifications developed for the Google Omega simulator (Schwarzkopf et al., 2013). This tool enables the simulation of large-scale Edge Computing scenarios by focusing on resource management, scheduling and energy-efficiency policies. Some of the performance metrics provided are: makespan, the time jobs wait for their first task to be scheduled, and the time jobs wait for all their tasks to be scheduled.

GAME-SCORE, proposed by Fernández-Cerero, Jakóbik, Fernández-Montes, and Kołodziej (2019), is the extension of SCORE (Cervin & Årzén, 2018) and follows the design pattern: a hybrid approach between discrete-event and multi-agent simulation. Its main aim is simulating energy-efficient IaaS on Cloud Computing. Some of the performance metrics provided are: makespan, the time jobs wait for their first task to be scheduled and the time jobs remain in the system while all their tasks are scheduled.

Table 1. Related works summary.

Ref.	Name	Research field
Buyya and Murshed (2002)	GridSim	Grid Systems
Calheiros et al. (2011)	CloudSim	Cloud Computing
Kanagaraj and Swamynathan (2016)	WorkflowSim	Cloud Computing
Cervin and Årzén (2018)	TrueTime	Embedded Control Systems
Fernández-Cerero et al. (2018)	SCORE	Cloud Computing
Fernández-Cerero, Fernández-Montes, et al. (2019)	Sphere	Edge Computing
Fernández-Cerero, Jakóbik, et al. (2019)	GAME-SCORE	Cloud Computing
[Our Proposal]	IPS	Enterprise Application Integration

3. Background

This section presents an integration process that uses the integration patterns documented by Hohpe and Woolf (2004), and the Pipes-and-Filters (Alexander et al., 1977) architectural style. The elements of the task scheduling are defined, and the task-based model is described (also approached by this article).

An integration process is a computational program that connects applications, synchronising the exchange of data and functionalities amongst them. An example of an integration process, proposed by Hohpe (2005), is the «Processing Order», depicted in Figure 1. The «Processing Order» integration process connects five applications: «Ordering System», «Widget Inventory», «Gadget Inventory», «Invalid Items Log» and «Inventory System». «Ordering System» represents a source application that delivers new orders data to the integration process. Each order is wrapped inside a «message» which contains the order data. A message with a new order is split into individual messages, which must contain only one item. Messages are routed either to the «Widget Inventory» or the «Gadget Inventory» depending on their contents. Messages with items that do not belong in any of these inventories are then routed to «Invalid Items Log». The «Inventory System» application represents a final data sink that responds according to the availability of items. Processing of one order corresponds to one «job» instance. Generally, there are many jobs being processed at a particular point in time, reason why there are many job instances.

In the workflow of an integration process, a «path» is the set of uncoupled tasks connected by communication channels whereby an inbound message flows through,

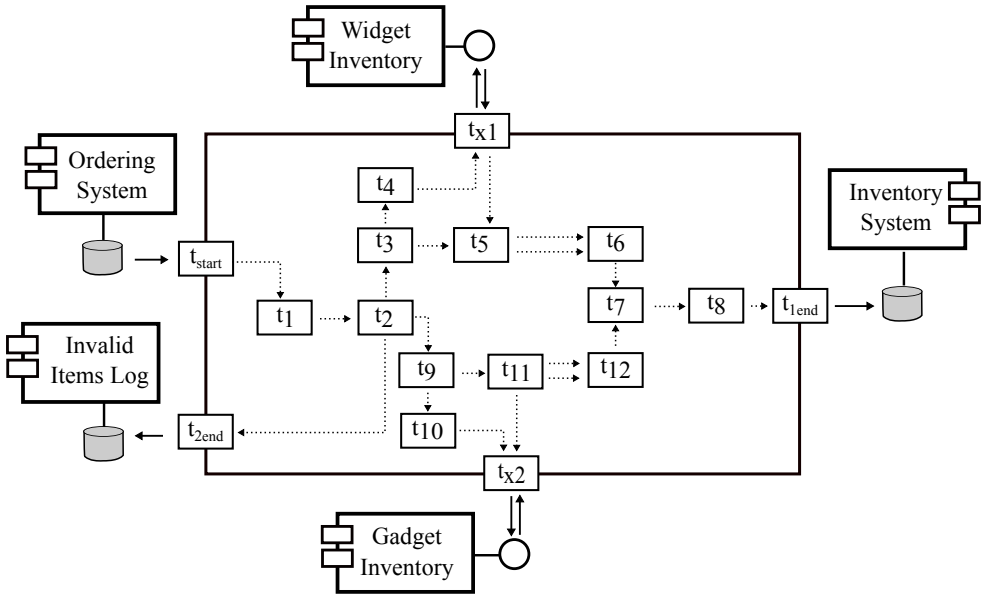


Figure 1. Processing Order conceptual model.

from source applications to sink applications. An integration process is a «workflow» composed of several segments of tasks sequentially arranged in parallel, or both ways. Parallel segments are parts of a path which can be executed in parallel. The inbound message process corresponds to the execution of all tasks of the path whereby the message flows from the source to the destination application.

A task implements an integration pattern and every pattern represents an atomic and specific operation on message processing, such as transforming, filtering, splitting, joining or routing. Depending on the integration pattern that a task implements, it can have one or more inputs and one or more outputs. There is an order of dependence for the tasks that determines their order of execution. Thus, a message can only be processed by a task after it has been processed by each predecessor task. An outbound message of a task is written to the communication channel that connects this task with the next successive task in the path.

Runtime systems supply services to applications implemented on top of them (Appel, n.d.). The «scheduler» is the key element that manages and orchestrates all activities of the elements of a runtime system, by accomplishing an inbound message processing. Computational resources to execute the tasks are managed by the scheduler. In runtime systems of integration platforms, these resources are «threads» and they are usually grouped in «thread pools» to avoid the creation of consecutive threads, and allow handling requests of tasks more quickly (Jeon & Jung, 2018). A thread is the smallest unit of a computational program that can be managed by the runtime system. It is also an abstraction piece of a physical and independent processing unit, or even a central processing unit (CPU) core.

The execution model of runtime systems establishes how tasks must be executed and how threads must be allocated during the processing of messages, in an integration process (Freire, Frantz, & Roos-Frantz, 2019). This article addresses the task-based model, once it allows treating task instances, i.e., tasks assigned to be executed by a thread. In this model, a task is ready to be executed when there are messages in all communication channels (inputs) to that task. The request of execution of a ready

task is annotated in a waiting queue. Ready tasks wait in the queue until there is an available thread to execute them. An available thread selects a task of the queue, following an execution policy, e.g. FIFO.

The scheduler creates, manages, and releases threads. It can also configure the pool by determining parameters such as the initial and the maximum number of threads, and the maximum lifetime of an idle thread. The scheduler assigns threads to execute instances of tasks; after an instance of the task has been executed, the thread is released and goes back to the pool. The processing of a message in the successive task now depends on a new assignment of an available thread from the pool to this very task. A message is processed according to the order of dependence of the tasks in the path. Tasks in sequential segments are sequentially executed, whereas tasks in parallel segments can be executed in parallel - after all, there is no dependency between them.

4. Problem Formulation

This section shall represent the task scheduling problem from the «Processing Order» integration process through a Directed Acyclic Graph (DAG), where the mathematical model is formulated. A DAG can represent the task scheduling model and the constraints of tasks execution. 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 $T_i = \{t_{i,1}, t_{i,2}, \cdots, t_{i,k}\}$ is the set of vertices and E is the set of edges. Every vertex in the graph represents a task of the process; each edge represents a communication channel between tasks, and indicates precedence constraints between tasks. Every edge has a weight, which represents the waiting time of the task in the queue (Saifullah, Li, Agrawal, Lu, & Gill, 2013). The «Processing Order» integration process is represented by DAG in Figure 2.

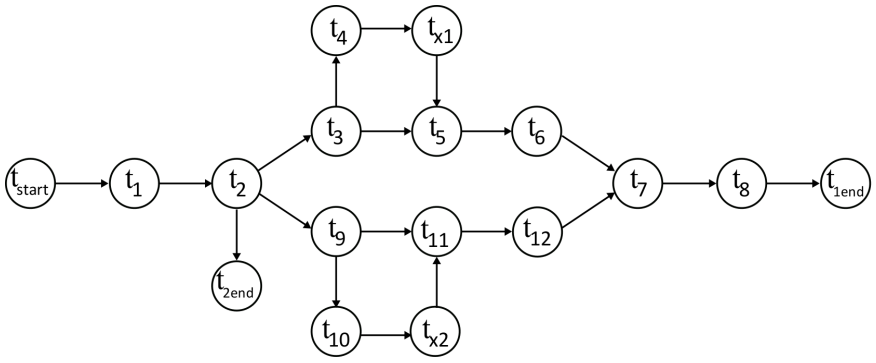


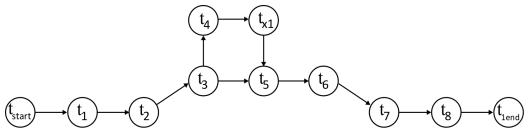
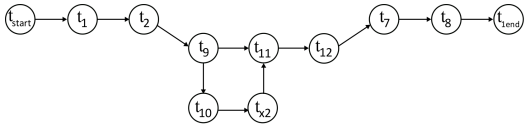
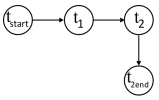
Figure 2. Processing Order represented in a DAG task model.

There are 17 nodes, which represent the following 16 tasks: t_{start}, 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_{x_1}, t_{x_2}, where t_{start} is a starting dummy node, and t_{1end} and t_{2end} are ending nodes. The nodes t_{x_1}, t_{x_2} represent tasks that send and receive information to/from applications. There are 19 edges, which represent the 19 communication channels.

There is one input task represented by t_{start}, which has no predecessor task and two output tasks represented by t_{1end} and t_{2end}, which have no successor task. The nodes t_{x1} and t_{x2} represent tasks that exchange messages with applications during runtime. Intermediary tasks are represented by t_i, where i ranges from 1 to 12.

In the integration logic of this conceptual model, an order contains several items. An order is split into unitary items, which can exclusively belong either to the «Widget Inventory», to the «Gadget Inventory», or to none of these. The path for a unitary item belonging to the «Widget Inventory» is denoted by the task segment $s_1 = \{t_{start}, t_1, t_2, t_3, t_4, t_{x1}, t_5, t_6, t_7, t_8, t_{1end}\}$; for a unitary item belonging to the «Gadget Inventory» it is denoted by the task segment $s_2 = t_{start}, t_1, t_2, t_9, t_{10}, t_{x2}, t_{11}, t_{12}, t_7, t_8, t_{1end}$; and for a unitary item that does not belong to any inventory, it is denoted by the task segment $s_3 = \{t_{start}, t_1, t_2, t_{2end}\}$. Some tasks that can be executed in parallel in the «Processing Order» integration process are: $[t_3, t_9]$, $[t_4, t_{10}]$, $[t_{x1}, t_{x2}]$, $[t_5, t_{11}]$, $[t_6, t_{12}]$. Table 2 shows the paths and the classification from «Processing Order» integration process segments.

Table 2. Processing Order path characterisation.

ID	Path	Segment	
		Sequential	Parallel
s_1		t_{start}, t_1, t_2 t_6, t_7, t_8, t_{1end}	t_3, t_4, t_{x1}, t_5
s_2		t_{start}, t_1, t_2 $t_{12}, t_7, t_8, t_{1end}$	$t_9, t_{10}, t_{x2}, t_{11}$
s_3		t_{start}, t_1, t_{2end}	

Makespan is a metric of performance well-known by the integration community, defined as the total execution time of the integration process for a given message (Canon & Jeannot, 2007; Chirkin et al., 2017). It corresponds to the total execution time from all tasks in the path whereby the message must flow for its complete processing. It 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. It can be inferred that the execution time range of a task t_k is defined as $[te_{t_{kini}}, te_{t_{kfin}}]$. In this article, the total execution time of an integration process is computed as the elapsed time between the starting time of the first message entering the workflow ST_{m_1} and the ending time of the last message to leave the workflow ET_{m_np}.

Throughput is a performance measurement based on the amount of work a system can perform in a given time in a particular environment. The throughput of a computer system is a function of the environment and workload characteristics. Improvements resulting from system changes can be evaluated by throughput measurements (Thakur & Kumar., 2018; Wood & Forman, 1971). In the execution of integration processes, throughput corresponds to the number of messages processed per time unit and it is

calculated as the division of the number of total processed messages, np, by the total execution time, TET, cf. Equation 1.

$$Throughput = \frac{np}{ET_{m_{np}} - ST_{m_1}} \tag{1}$$

5. Proposal

This section describes the Integration Process Simulator, a heuristic simulator for task scheduling in integration processes. The simulator implements the following heuristics: First-In-First-Out (FIFO), Multi-queue Round Robin (MqRR) and Query-Priority (qPrior) (Freire et al., 2021). FIFO is a simple heuristic and is adopted in most integration platforms (Freire, Frantz, Roos-Frantz, & Sawicki, 2019), in which messages are processed in the same order they enter the workflow. Round Robin (RR) heuristic is popularly known by its simplicity and considered an efficient and effective scheduling technique in computing (Y. Zhang, Shen, & Song, 2018), in which messages are processed in a circular order. Here, two novel heuristics were proposed: MqRR and qPrior, both based on RR. These heuristics seek to increase the execution performance from integration processes in overload situations, by facing the dynamic environment of task scheduling in applications integration. Overload situations happen when the number of messages accumulated in communication channels is greater than the number of messages processed in a given time interval.

In FIFO scheduling, a single task queue holds the instances of all tasks. Tasks are kept in a queue and sorted in descending order of arrival time. The task with the furthest arrival time is placed in the head of the queue, while the first recently arrived task is placed in the tail of the queue. Available threads repeatedly poll the queue for tasks; if there are any pending tasks, threads shall execute them in a head-to-tail order.

In the MqRR scheduling heuristic, there are multiple task queues and each queue keeps the instances of a task. Thus, the number of queues is equal to the number of tasks of the integration process. However, tasks belonging to parallel segments can be maintained in the same queue to be executed in parallel. Tasks are kept in queues in descending order of arrival time, and available threads repeatedly poll the queue for tasks, executing existing tasks from head to tail. Threads recurrently poll task queues in a circular order, executing each time a predefined number of tasks for every queue.

In the qPrior heuristic there are also multiple task queues, but in this case, threads recurrently poll the queues following the order of priority. A task having more predecessor tasks has a higher priority for execution.

5.1. Algorithms

Figure 6 in Appendices A.2 presents the flowchart of the simulator. `Start` receives the simulation parameters and calls the heuristics to be used in the simulation. The `Integration Process Profile` contains the integration process information used as parameters by the other algorithms. `FIFO` provides task scheduling through the First-In-First-Out heuristic. `MqRR` uses Multi-Queue Round Robin heuristic. `qPrior` uses Queue Priority heuristic. `Allocate Thread` is responsible for managing and allocating threads that execute tasks of integration processes. The `Next Task` sends tasks to the

next task queue. `Operation` simulates the task execution. `Queue Add` adds tasks to a queue. `Queue Add Input` adds tasks to the first queue, by simulating the arrival of messages in the integration process. The pseudo-codes of these algorithms are presented and detailed as follows.

5.1.1. Start

`Start` coordinates the simulation of integration processes, cf. Activity 1. It receives the following parameters: the number of simulations, the maximum duration of the simulation, the maximum number of messages, the initial number of inbound messages, the heuristic, and the number of tasks performed at a time (called preemption) - in case of MqRR and qPrior heuristics. The last input parameter is the number of tasks executed every time the threads check a queue. `Start` returns the throughput, the number of processed messages, and the number of remaining messages. First, `Start` creates a vector for the starting time of messages and another for the final time of messages, besides task queues. If the heuristic to be used is FIFO, it creates a single queue; if the heuristic is RR, it shall create a queue for every task of the integration process. After that, it adds the first tasks to the queues and then calls the algorithm corresponding to the heuristic chosen. Finally, it calculates the throughput and writes the metrics results in a text file.

5.1.2. Integration Process Profile

`Integration Process Profile` deals with profiles of integration processes, cf. Activity 2. It receives the following vectors as parameters: the identification of tasks, parallel tasks, the execution time of the tasks, the operation of the tasks, the next tasks and the last tasks. It calculates the number of tasks of the integration process by measuring the length of the identification from the tasks vector.

5.1.3. First-In-First-Out

This algorithm carries out the task scheduling which uses FIFO heuristic, cf. Algorithm 3. It receives the following parameters: the task queue, the maximum duration of the simulation, and the starting time of the first task in the queue. The algorithm starts by initialising the auxiliary variable: *totsize* that corresponds to the queue total size. Afterwards, it repeatedly calls the `Allocate Thread` activity to allocate threads and perform tasks until the simulation time expires, or when there are no tasks left in the queue.

5.1.4. Multi-Queue Round Robin

This algorithm carries out the task scheduling through the MqRR heuristic, cf. Algorithm 4. It receives the following parameters: task queues, the maximum duration of the simulation, the starting time from the first task in the queue, the total number of tasks and the number of tasks to be performed at a time. This last input parameter is used to indicate the number of tasks that threads must execute every time a queue is checked. The algorithm starts by initialising two auxiliary variables: *totsize* and *preempt*. The former corresponds to the total size of the queue and the latter to the number of tasks performed at a time (preemption). The algorithm checks from the queue of the first task to the queue of the last task, and it continues checking them in a circular order. When the queue size is smaller than the *preempt*, the

Activity 1 *Start*

Input: Number of simulations: *numsimulation*
Input: Maximum duration of the simulation: *maxduration*
Input: Number of inbound messages: *messinbound*
Input: Heuristic: *policy*
Input: Number of tasks performed at a time: *preempttask*
Output: Throughput : *throughput*
Output: Number of processed messages: *messproc*
Output: Number of remained messages: *messrem*

```
1:  starttime[]                                   ▷ Creates a starting time vector
2:  endtime[]                                     ▷ Creates an end time vector
3:  while count ≤ numsimulation do                ▷ Execution of the simulations
4:      if policy = FIFO then
5:          queues[0]                             ▷ Creates a single queue
6:      else
7:          for [i] = 1 to numtasks do
8:              queues[i]                         ▷ Creates a queue to each task
9:          end for
10:     end if
11:     for [i] = 1 to messinbound do             ▷ Creates first tasks in queues
12:         starttime[i] ← current.Time           ▷ Adds starting times of the messages
13:         if policy = FIFO then
14:             queues[0] ← add(1)
15:         else
16:             queues[1] ← add(1)
17:         end if
18:     end for
19:     switch policy do
20:         case FIFO
21:             simulationFile ← SimulationFIFO.txt
22:             Fifo()                            ▷ Executes FIFO heuristic
23:         case MqRR
24:             simulationFile ← SimulationMqRR.txt
25:             MqRR( ))                          ▷ Executes MqRR heuristic
26:         case qPrior
27:             simulationFile ← SimulationqPrior.txt
28:             qPrior( ))                        ▷ Executes qPrior heuristic
29:     count ← count + 1
30: end while
31: messproc ← endtime.size();
32: if messproc > 0 then
33:     totmakespan ← (endtime[messproc] − starttime[1])   ▷ Calculates total total
    execution time
34:     throughput ← (messproc/(endtime[messproc] − starttime[1]))
35: end if
36: Record Archive(simulationFile)                ▷ Records metrics in a text file
```

Activity 2 *Integration Process Profile*

Input: Vector for tasks identification : $IdTask[\,]$
Input: Vector for parallel tasks : $ParallelTask[\,]$
Input: Vector for the execution time of the tasks : $TimeExec[\,]$
Input: Vector for tasks operation: $OperTask[\,]$
Input: Vector for the next tasks : $NextTask[\,]$
Input: Vector for the last tasks: $LastTask[\,]$

1: $NumTasks \leftarrow IdTask.length$

Algorithm 3 *First-In-First-Out*

Input: Task queue: $queues[0]$
Input: Maximum duration of the simulation: $maxduration$
Input: Starting time of the simulation: $start$

1: $totsize \leftarrow queues[0].size$ ▷ Initialises the queue size variable
2: **while** $totsize > 0$ & $duration < maxduration$) **do** ▷ Executes the tasks in the queue
3: **if** $queues[i] \neq \emptyset$ **then**
4: $Allocate\ Thread(queues[0], totsize)$ ▷ Allocates threads to tasks in the queue
5: **end if**
6: $duration \leftarrow$ current.Time $- start$ ▷ Updates the duration of the simulation
7: $totsize \leftarrow queues[0].size$ ▷ Updates the total size of the task queue
8: **end while**

algorithm executes all tasks in the queue; otherwise, only the number of tasks equals to the *preempt* will be executed. The algorithm calls the `Allocate Thread` activity to allocate threads and execute tasks while there are tasks in the queue, and the simulation duration is lower than the input parameter that stipulates maximum duration.

5.1.5. Queue Priority

This algorithm carries out the task scheduling by the qPrior heuristic, cf. Algorithm 5. It receives the following parameters: the task queues, the maximum duration of the simulation, and the starting time of input of the first task in the queue, the total number of tasks and the number of tasks performed at a time. This last input parameter is used to indicate the number of tasks that the threads must execute every time a queue is checked. The algorithm starts by initialising two auxiliary variables: *totsize* and *preempt*. The former corresponds to queue total size and the latter to the number of tasks performed at a time (preemption). The algorithm checks queues from the highest to the lowest priority task queue. When the queue size is smaller than the *preempt*, the algorithm executes all tasks in the queue; otherwise, only the number of tasks equals to the *preempt* will be executed. The algorithm calls `Allocate Thread` activity to allocate threads and execute tasks while there are tasks in the queue, and the simulation duration is lower than the input parameter that stipulates maximum duration.

Algorithm 4 *Multi-Queue Round Robin*

Input: Task queues: $queues[\,]$
Input: Maximum duration of the simulation: $maxduration$
Input: Starting time of the simulation: $start$
Input: Total number of tasks: $numtasks$
Input: Number of tasks performed at a time: $preempttask$

1: $totsize \leftarrow 1$ ▷ Initialises the total queue sizes variable
2: $preeemp \leftarrow preempttask$ ▷ Initialises the $preempt$ variable
3: **while** $totsize > 0$ & $duration < maxduration$) **do** ▷ Execution of tasks in the queue
4: **for** $[i] = 1$ **to** $numtasks$ **do**
5: **if** $queues[i] \neq \emptyset$ **then**
▷ Checks whether there is preemption and compares with the queue size
6: **if** $(preempt = 0)$ **or** $(queues[i].size < preempt)$ **then**
7: $preempt \leftarrow queues[i].size$
8: **else**
9: $preempt \leftarrow preempttask$
10: **end if**
11: $Allocate\ Thread(queues[i], preempt)$ ▷ Allocates threads to tasks in the queue
12: **end if**
13: **end for**
14: $totsize \leftarrow 0$
15: **for** $[i] = 1$ **to** $numtasks$ **do** ▷ Updates the task queues total size
16: $totsize \leftarrow totsize + queues[i].size$
17: **end for**
18: $duration \leftarrow$ current.Time $- start$ ▷ Updates the duration of the simulation
19: **end while**

Algorithm 5 *Queue Priority*

Input: Task queues: $queues[\ ]$
Input: Maximum duration of the simulation: $maxduration$
Input: Starting time of the simulation: $start$
Input: Total number of tasks: $numtasks$
Input: Number of tasks performed at a time: $preempttask$

1: $totsize \leftarrow 1$ ▷ Initialises the total queue sizes variable
2: $preeemp \leftarrow preempttask$ ▷ Initialises the variable $preempt$
3: $qprior \leftarrow numtasks$ ▷ Initialises the variable $qprior$
4: **while** $totsize > 0$ & $duration < maxduration$) **do** ▷ Execution tasks in the queue
5: **for** $[i] = numtasks$ **to** 1 **step**−1 **do**
▷ Selects queue of higher priority
6: **if** $queues[i] \neq \emptyset$ **then**
7: $qprior \leftarrow i$
8: $i \leftarrow 1$
9: **end if**
10: **end for**
11: **if** $(preempt = 0)$ **or** $(queues[qprior].size < preempt)$ **then**
12: $preempt \leftarrow queues[qprior].size$
13: **else**
14: $preempt \leftarrow preempttask$
15: **end if**
16: *Allocate Thread*$(queues[qprior], preempt)$ ▷ Allocates threads to tasks in the queue
17: **end while**
18: $totsize \leftarrow 0$
19: **for** $[i] = 1$ **to** $numtasks$ **do** ▷ Updates the task queues total size
20: $totsize \leftarrow totsize + queues[i].size$
21: **end for**
22: $duration \leftarrow$ current.Time $- start$ ▷ Updates the duration of the simulation

5.1.6. Allocate thread

`Allocate Thread` manages and allocates threads to execute tasks of a queue, cf. Activity6. It receives the following parameters: a task queue, the maximum duration of the simulation, the maximum number of messages, the number of tasks performed at a time and a vector containing the ending tasks. It initialises the auxiliary variable *preempt* and then starts with the creation of a thread pool, which must be elastic, i.e, allowing a variable sized pool to take advantage of multi-core CPUs. `Allocate Thread` selects tasks in the queue, from head to tail, until it achieves the number of tasks equals to the size of the preemption variable (*preempt*). If the queue size is smaller than the number of tasks, the algorithm assigns the queue size to the *preempt*. Then, it submits the task to the thread pool and calls `Operation` activity, which is responsible for the task operation. After the execution, `Allocate Thread` removes the task from the current queue and then checks if the task belongs to the vector containing the ending tasks. If the task is not an end task and the heuristic used has multiple task queues, the `Allocate thread` activity calls the `Next queue` activity that is responsible for storing the task in the next queue, according to the logic of the integration process. Finally, it destroys the thread pool.

5.1.7. Next queue

`Next Task` sends the task to its next queue, in case of heuristics that use multiple queues, cf. Activity 7. It receives the following parameters: a task, a vector containing the next task queues, and a vector containing the logic operations of the tasks. It initialises the auxiliary variable *operleng* that corresponds to the length of the vector containing the tasks logic operations. If the length of this vector equals zero, it means that the task is sequential and must be sent to only one queue. Otherwise, the task must be sent to one or more queues, depending on the logic operation. Logic operations of a task can be `AND`, `OR*`, and `OR`. If the operation equals `AND`, the task must be sent to all the next queues. If the operation equals `OR*`, the task must only be sent to the next queue. The operation `OR` acts as both `AND` and `OR*`, i.e., the task can be sent to one or more queues. `Next Task` sends the task to the queues according to the vector from the next tasks, which contains the information regarding the next task of the integration process.

5.1.8. Operation

`Operation` simulates the processing of messages by a task, cf. Activity 8. It receives the following parameters: a task and the vector from the tasks execution time. It randomly selects an execution time from this vector, and then waits during this time. The operation of processing message can be: split, translate, filter, aggregate, route, etc.

5.1.9. Queue Add

`Queue Add` adds a task to a queue, cf. Activity 9. It receives a task, and the vector of parallel tasks as parameters. The latter input indicates when a task can be executed in parallel with another task. `Queue Add` checks the heuristic and if it equals FIFO, the task is added to the FIFO queue; otherwise, the vector of parallel tasks is checked. If there is a parallel task to the task, it is added to the queue of parallel tasks; otherwise, it is added to the task in its own queue.

Activity 6 *Allocate Thread*

Input: Task queue: $queues[i]$
Input: Maximum duration of the simulation: $maxduration$
Input: Maximum number of messages: $maxmessages$
Input: Number of tasks performed at a time: $preempt$
Input: Vector for the last tasks: $LastTask[\ ]$

1: $preeemp \leftarrow preempttask$ ▷ Initialises the variable $preempt$
2: Creates elastic thread pool
3: **for** $[j] = 1$ **to** $preempt$ **do**
4: **if** $duration < maxduration$) **then**
5: $task \leftarrow queues.head$ ▷ Selects the task from the head of the queue
6: **if** $task \neq null$ **then**
7: Submits $Operation(task)$ to a thread pool ▷ Executes $task$
8: **for** $[j] = 0$ **to** $LastTask[\ ].length$ **do**
9: **if** $task \neq LastTask[j]$ **then** $lasttask = 1$
10: **end if**
11: **end for**
12: **if** $lasttask = 0$ **then**
13: $Next\ queue(task)$ ▷ Stores $task$ in next queue
14: **else**
15: $endtime[] \leftarrow current.Time$
16: **end if**
17: Removes $task$ of the queue[i] ▷ Removes $task$ of the queue
18: Shutdown thread pool
19: **end if** ▷ Compares the queue size with the preemption
20: **if** $queue[i].size < preempt$ **then**
21: $preempt \leftarrow queue[i].size$
22: **else**
23: $preempt \leftarrow preempttask$
24: **end if**
25: **else**
26: $i \leftarrow preempt + 1$
27: **end if**
28: $duration \leftarrow current.Time - start$
29: **if** $maxmessages$>starttime.size() **then**
30: $Queue\ Input\ Add()$ ▷ Adds messages in first queue every 100 tasks executed
31: **else**
32: $duration \leftarrow maxduration$
33: **end if**
34: **end for**

Activity 7 *Next queue*

Input: Task : *task*
Input: Vector of next tasks : $NextTask[\,]$
Input: Vector of operation tasks: $OperTask[\,]$

1: $operleng \leftarrow OperTask[task].length$ ▷ Initialises the auxiliary variable with the vector size
2: **switch** *operleng* **do**
3: **case** 0 ▷ Sequential task
4: $nexttask \leftarrow NextTask[task][1]$
5: *Queue Add*(*nexttask*) ▷ Adds the task in its next queue
6: **case** 1 **or** 2 ▷ Fork task
7: **if** $operleng = 2$ **then**
8: $operrad =$ random $OperTask[task][\,]$ ▷ Randomly selects one of the operations
9: **else**
10: $operrad = OperTask[task]$ ▷ Selects the task operation
11: **end if**
12: **if** $operrad =$ `or` **then** ▷ Can choose one of the next tasks
13: $nexttask = randomNextTask[task][1]$ ▷ Randomly selects one of the next task
14: *Queue Add*(*nexttask*) ▷ Adds the task to its next queue
15: **else**
16: **if** $operrad =$ `and` **then** ▷ Send to all next tasks
17: **for** $[i] = 1$ **to** $NextTask[task].length$ **do**
18: $nexttask = NextTask[task][i]$
19: *Queue Add*(*nexttask*) ▷ Adds the task to next queue
20: **end for**
21: **end if**
22: **end if**

Activity 8 *Operation*

Input: Task : *task*
Input: Vector for execution time tasks : $TimeExec[\,]$

1: $time \leftarrow$ random $TimeExec[task][\,]$ ▷ Randomly selects one from the execution time to the task
2: Waits *time* ▷ Simulates the execution of the task

Activity 9 *Queue Add*

Input: Task : *task*
Input: Vector for parallel tasks : $ParallelTask[\,]$

1: **if** *policy* = `FIFO` **then**
2: $queues[0] \leftarrow add(task)$ ▷ Adds tasks to FIFO queue
3: **else** ▷ Checks if tasks are parallel
4: **if** $ParallelTask[task] \neq 0$ **then**
5: $parallel = ParallelTask[task]$
6: $queues[parallel] \leftarrow add(task)$ ▷ Adds tasks to the queue of the parallel task
7: **else**
8: $queues[task] \leftarrow add(task)$ ▷ Adds tasks to the respective queue
9: **end if**
10: **end if**

5.1.10. Queue Input Add

`Queue Input Add` adds the first tasks to the queue, cf. Activity 10. It receives the number of input tasks as a parameter, then, it checks the heuristic and if the heuristic is FIFO, it then adds the tasks to the FIFO queue. Otherwise, it adds the task to the queue of the first task.

Activity 10 *Queue Input Add*

Input: Number of input tasks : *numinputtask*

1: **for** $[i] = 1$ **to** *numinputtask* **do** ▷ Creates first tasks in the queue
2: $starttime[] \leftarrow current.Time$
3: **if** *policy* = `FIFO` **then**
4: $queues[0] \leftarrow add(1)$
5: **else**
6: $queues[1] \leftarrow add(1)$
7: **end if**
8: **end for**

6. Validation

This section uses the IPS simulator to compare the performance of FIFO, MqRR and qPrior (Freire et al., 2021), when it comes to the execution of integration processes under high workloads (over 1,000,000 messages). Performance metrics such as throughput, the number of processed messages and the number of remained messages were used. The integration processes employed in the simulations were Processing Order (IP1), Huelva's County Council (IP2), and Real Estate (IP3). Their profiles are described in Appendices A.1. Introduced by Hohpe (2005), the Processing Order problem is a classic example of an integration process, which conceptual model is depicted in Figure 1. The Huelva's County Council problem is a real-world integration process (Frantz et al., 2016), consisting of the automatisation of the user registration into a central repository (Huelva, Spain). The Real State problem is a

real-world integration process (Freire, Frantz, & Roos-Frantz, 2019) consisting of the automatisation of the real state tax management system in the city of Ijuí (Brazil).

The simulation is classified as termination simulation because the output is a function of the initial conditions, for which a protocol based on Jedlitschka and Pfahl (2005), Wohlin et al. (2012) and Basili, Rombach, Kitchenham, Selby, and Pfahl (2007) was followed, with procedures for controlled experiments in the field of software engineering. The literature suggests that a statistical analysis of performance data from simulations must be carried out (Georges, Buytaert, & Eeckhout, 2007), since this type of simulation deals with non-determinism in computational systems (Frantz, Corchuelo, & Arjona, 2011). The ANOVA test was used to verify the influence of random factors (called errors) in the measurements of the dependent variables. The comparison of means, according to the test of Scott & Knott, groups the average values of the dependent variables to check if there is a statistical difference between the results from the heuristics. The source code and data sets used in the simulations is publicly available for download [1].

These are the independent variables:

`Heuristic`. The heuristic used for task scheduling. The values tested for this variable were FIFO, MqRR, and qPrior.

`Integration process`. The conceptual model of the integration process. The values tested for this variable were IP1, IP2, and IP3.

`Duration`. The time interval during which the algorithm keeps running. The value tested for this variable was 60 seconds.

`Total workload`. The total number of inbound messages. The value tested for this variable was 2,000,000 messages.

`Initial workload`. The initial number of inbound messages. The value tested for this variable was 1,000 messages.

`Rate of inbound messages`. The number of inbound messages added periodically to the integration process. The value tested for this variable was 1,000 messages.

And these are the dependent variables:

`Throughput`. This variable corresponds to the average of processed messages per millisecond, cf. Equation 1.

`Processed messages`. This variable corresponds to the number of inbound messages that were entirely processed by the integration process.

`Remained messages`. This variable corresponds to the number of inbound messages that remained unprocessed by the integration process.

6.1. Environment and Supporting Tools

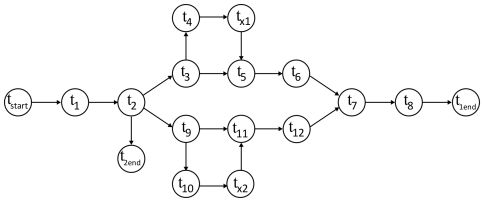
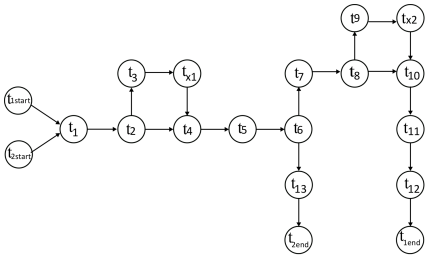
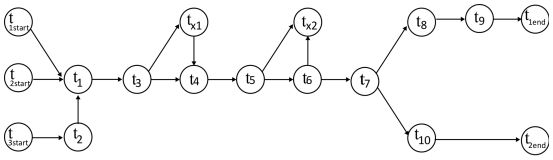
The simulations were carried out on a machine equipped with 16 processors Intel Xeon CPU E5-4610 V4, 1.8 GHz, 32GB of RAM, with a Windows Server 2016 Datacenter 64-bits operating system. The programming language Java, version 8.0 update 152, was used to implement and execute the IPS simulator. The Genes (Cruz, 2006) software, version 2015.5.0, was used to process the descriptive statistic, ANOVA and Scott & Knoot tests were used to calculate the performance metrics in this study.

[1] `https://github.com/gca-research-group/simulation-qprior`

6.2. Execution and Data Collection

The experiment was conducted using the IPS simulator, which simulates the execution of the IP1, IP2, IP3 integration processes. Table 3 describes the main differences between the integration processes tested.

Table 3. Integration process characterisation.

ID	DAG	Nodes number	Starting nodes	Ending nodes
IP1		17	1	2
IP2		19	2	2
IP3		17	3	2

The simulation starts with a workload of 1,000 inbound messages; it receives 1,000 inbound messages to every 100 tasks executions. The execution time of each task varies within an interval (in seconds) according to the profile of the integration process. For MqRR and qPrior heuristics, the preemption variable is set to 100 tasks.The simulation time is configured to 60 seconds, after which, the simulation stops. Next, the simulator collects the values from dependent variables and stores them in a text file. Afterwards, data were handled and analysed, and then statistical tests were applied.

The literature suggests that 20 to 30 executions are sufficient to obtain a population average (Sargent, 2013). In the experiment, each heuristic was simulated 25 times for each integration process, resulting in 225 different scenarios summarised in Table 4.

6.3. Results

The scatter charts show average values of dependent variables obtained in 25 independent runs of the simulation, for every type of heuristic, cf. Figures: 3, 4,

Table 4. Scenarios of the simulations.

Heuristics	FIFO, MqRR, qPrior	3
Integration Process	IP1, IP2, IP3	1
Duration	60 seconds	1
Total workload	2,000,000	1
Initial workload	1000	1
Rate of inbound messages	1000	1
Repetitions		25
Scenarios	3 x 3 x 1 x 1 x 1 x 1 x 25	**225**

and 5. In these charts, the x-axis represents the heuristics and the y-axis represents the average throughput, the average number of processed messages and the average number of remained messages, respectively.

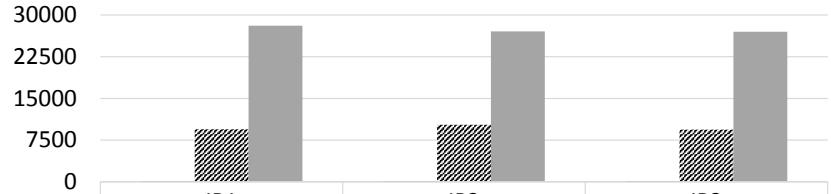
The throughput achieved in the simulation of IP1 (in messages per milliseconds) was 8.44 msg/ms with FIFO; 9,455.17 msg/ms with MqRR; and 28,076.67 msg/ms with qPrior. The average number of processed messages was 502.40 with FIFO; 22,607 messages with MqRR; and 199,800 messages with qPrior. Lastly, the average number of remained messages was 1,999,498 with FIFO; 1,977,393 messages with MqRR; and 1,800,200 messages with qPrior. So, qPrior was the most efficient algorithm, while FIFO was the least.

The throughput achieved in the simulation of IP2 (in messages per milliseconds) was 64.26 msg/ms with FIFO; 10,250.02 msg/ms with MqRR; and 27,001.52 msg/ms with qPrior. The average number of processed messages was 3,821.52 with FIFO; 25,515 messages with MqRR; and 199,800 messages with qPrior. Lastly, the average number of remained messages was 1,996,178 with FIFO; 1,974,485 messages with MqRR; and 1,800,200 messages with qPrior. So, qPrior was again the most efficient algorithm, while FIFO was the least.

The throughput achieved in the simulation of IP3 (in messages per milliseconds) was 8.49 msg/ms with FIFO; 9,381.09 msg/ms with MqRR; and 27,001.52 msg/ms with qPrior. The average number of processed messages was 506.28 with FIFO; 22,607 messages with MqRR; and 199,800 messages with qPrior. Lastly, the average number of remained messages was 1,999,494 with FIFO; 1,977,393 messages with MqRR; and 1,800,200 messages with qPrior. Yet again, qPrior was the most efficient algorithm, while FIFO was the least .

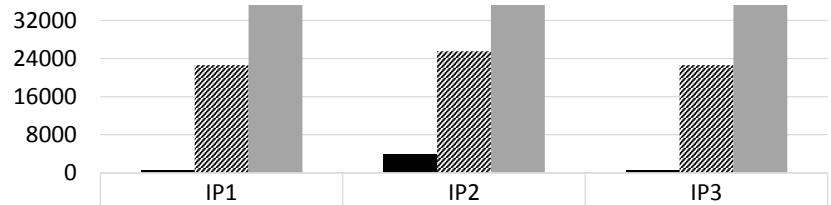
Table 5 shows the analysis of variance for IP1, which is the mean square of the throughput. The mean square was 509,9275,548 for the heuristics and 1,704,696 for error. The overall mean was 12,513.42 and the coefficient of variation, 10.43%. In the variance analysis of the number of processed messages, the mean square was 298,355,654,464 for the heuristics and 3,415 for error. The overall mean was equal to 1,925,696.70 and the coefficient of variation was 3.03%. Lastly, in the variance analysis of the number of remained messages, the mean square was 298,355,654,464 for the heuristics and 3,415 for error. The overall mean was equal to 74,303.29 and the coefficient of variation was 0.07%. Thus, there was a relation between the treatments (heuristics) and the result of an experiment (performance).

Table 6 shows the analysis of variance for IP2, which is the mean square from the throughput. The mean square was 4,638,534,208 for the heuristics and 2,231,170.0 for error. The overall mean was 12,451 and the coefficient of variation, 11.99%. In the analysis of variance of the number of processed messages, the mean square was 288,556,580,878 for the heuristics and 182,858 for error. The overall mean was equal



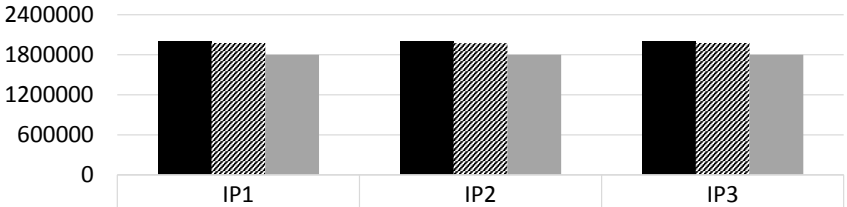
	IP1	IP2	IP3
Fifo	8.44	64.26	8.49
RR	9455.17	10250.02	9381.09
qPrior	28076.67	27038.91	27001.52

Figure 3. Average throughput



	IP1	IP2	IP3
Fifo	502.4	3821.52	506.28
RR	22607.48	25514.52	22607.48
qPrior	199800	199800	199800

Figure 4. Average number of processed messages



	IP1	IP2	IP3
Fifo	1999497.6	1996178.48	1999493.72
RR	1977392.52	1974485.48	1977392.52
qPrior	1800200	1800200	1800200

Figure 5. Average number of remained messages

to 1,923,621.32 and the coefficient of variation was 2.22%. Lastly, in the variance analysis of the number of remained messages, the mean square was 288,556,580,878 for the heuristics and 182,858 for error. The overall mean was equal to 76,378.68, and the coefficient of variation was 0.55%. So, there was a relation between the treatments and the result of an experiment.

Table 7 shows the analysis of variance for IP3, which is the mean square of the throughput. The mean square was 4,695,619,987 for the heuristics and 3,282,183 for error. The overall mean was 12,130.36 and the coefficient of variation, 14.93%. In the variance analysis of the number of processed messages, the mean square was 298,348,495,903 for the heuristics and 2,398 for error. The overall mean was equal to 1,925,695.41 and the coefficient of variation was 2.54%. Lastly, in the variance analysis of the number of remained messages, the mean square was 298,348,495,903 for the heuristics and 2,398 for error. The overall mean was equal to 74,304 and the coefficient of variation was 6.59%. So, there was a relation between the treatments and the result of an experiment.

Table 5. ANOVA test for IP1.

Sources of variation	**Degree of freedom**	**Mean square**		
		`Throughput`	`Processed messages`	`Remained messages`
Heuristics	2	5099275548 †	298355654464 †	298355654464 †
Error	72	1704696	3415	3415
Total	74			
overall mean		12513.42	1925696.70	74303.29
Coefficient of variation (%)		10.43	3.03	0.07

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

Table 6. ANOVA test for IP2.

Sources of variation	**Degree of freedom**	**Mean square**		
		`Throughput`	`Processed messages`	`Remained messages`
Heuristics	2	4638534208 †	288556580878 †	288556580878 †
Error	72	2231170	182858	182858
Total	74			
overall mean		12451	1923621.32	76378.68
Coefficient of variation (%)		11.99	2.22	0.55

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

Table 7. ANOVA test for IP3.

Sources of variation	**Degree of freedom**	**Mean square**		
		`Throughput`	`Processed messages`	`Remained messages`
Heuristics	2	4695619987 †	298348495903 †	298348495903 †
Error	72	3282183	2398	2398
Total	74			
overall mean		12130.36	1925695.41	74304
Coefficient of variation (%)		14.93	2.54	6.59

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

The results of the Scott & Knoot test are shown in Tables 8, 9 and 10, for IP1, IP2, and IP3, respectively. The heuristics are identified in the first column. For each dependent variable, there is a column for the average value and another for the group of Scott & Knott. There were three groups: «a», «b», and «c». Group «a» refers to the heuristic with the highest average, while group «b» refers to the heuristic with the second-highest average, and group «c» refers to the heuristic with the lowest average.

The results of the Scott & Knoot test for IP1 regarding throughput show that FIFO was in group «c» with the lowest average rate, which was 8.44 msg/ms; MqRR in «b» with an average of 9,455.17 msg/ms; and qPrior in «a» with the highest average, which was 28,076,67 msg/ms. Regarding the number of processed messages, FIFO was in group «c» with the lowest average: 502 messages. MqRR was in group «b» with an average of 22,607 messages, and qPrior in «a» with the highest average: 199,800 messages. Regarding the number of remained messages, FIFO was in group «a» with the highest average: 1,999,498 messages. MqRR was in group «b» with an average of 1,977,393 messages; and qPrior in «c», with the lowest average: 1,800,200 messages cf. Table 8. Therefore, there was a statistical difference among the three treatments.

The results of the Scott & Knoot test for IP2 regarding throughput show that FIFO was in group «c» with the lowest average: 64.26 msg/ms. MqRR was in group «b» with and average of 10,250.02 msg/ms, and qPrior in «a» with the highest average: 27,038.91 msg/ms. Regarding the number of processed messages, FIFO was in group «c» with the lowest average: 3,821 messages. MqRR was in «b» with an average of 25,515 messages, while qPrior was in «a» with the highest average: 199,800 messages. Regarding the number of remained messages, FIFO was in group «a» with the highest average: 1,996,178 messages. MqRR was in «b» with an average of 1,974,485, and qPrior was in «c» with the lowest average: 1,800,200 messages cf. Table 9. Thus, there was statistical difference among the three treatments.

The results of the Scott & Knoot test for IP3 regarding throughput show that FIFO was in group «c» with the lowest average: 8.49 msg/ms. MqRR was in «b» with an average of 9,381.09 msg/ms, while qPrior was in «a» with the highest average: 27,001,52 msg/ms. Regarding the number of processed messages, FIFO was in group «c» with the lowest average: 506 messages. MqRR was in «b» with an average of 22,607 messages, and qPrior in «a» with the highest average: 199,800 messages. Regarding the number of remained messages, FIFO was in group «a» with the highest average: 1,999,4948 messages. MqRR was in «b» with an average of 1,977,393 messages, and qPrior was in «c» with the lowest average: 1,800,200 messages cf. Table 10. So, there was statistical difference among the three treatments.

Table 8. Scott & Knott test for IP1.

Heuristic	Throughput		Processed messages		Remained messages	
	Average	**Group**	**Average**	**Group**	**Average**	**Group**
FIFO	8.44	c	502	c	1999498	a
MqRR	9455.17	b	22607	b	1977393	b
qPrior	28076.67	a	199800	a	1800200	c

Error level of 5% by the Scott & Knoot model.

Table 9. Scott & Knott test for IP2.

Heuristic	Throughput		Processed messages		Remained messages	
	Average	**Group**	**Average**	**Group**	**Average**	**Group**
FIFO	64.26	c	3821	c	1996178	a
MqRR	10250.02	b	25515	b	1974485	b
qPrior	27038.91	a	199800	a	1800200	c

Error level of 5% by the Scott & Knoot model.

Table 10. Scott & Knott test for IP3.

Heuristic	Throughput		Processed messages		Remained messages	
	Average	**Group**	**Average**	**Group**	**Average**	**Group**
FIFO	8.49	c	506	c	1999494	a
MqRR	9381.09	b	22607	b	1977393	b
qPrior	27001.52	a	199800	a	1800200	c

Error level of 5% by the Scott & Knoot model.

6.4. Summary

In the execution simulation of IP1, the best average throughput was 28076.67 msg/ms with qPrior, whereas the worst-case scenario was with FIFO, when it reached an average throughput of 8.44 msg/ms. The qPrior heuristic managed to process more messages, reaching an average of 199,800 messages and the number of 1,800,200 remained messages. The worst-case scenario for processed messages occurred with FIFO, where only 502.40 messages managed to be processed, and 1,999,498 remained unprocessed.

In the execution simulation of IP2, the best average throughput was 27,038.91 msg/ms with qPrior, whereas the worst-case scenario was with FIFO, when it reached a throughput of 64.26 msg/ms. The qPrior heuristic managed to process more messages, reaching an average of 199,800 messages and the number of 1,800,200 remained messages. The worst-case scenario for processed messages happened with FIFO, where only 3,821.52 messages were processed, and 1,996,178 remained unprocessed.

In the execution simulation of IP3, the best average throughput was 27,001.52 msg/ms with qPrior, whereas the worst-case scenario was with FIFO, when it reached a throughput of 8.49 msg/ms. The qPrior heuristic managed to process more messages, reaching an average of 199,800 messages and the number of 1,800,200 remained messages. The worst-case scenario for processed messages occurred with FIFO, where only 506.28 messages were processed, and 1,999,494 remained unprocessed.

The MqRR and qPrior heuristics provided a stable scenario considering the average number of processed messages for all three integration processes evaluated in this study. This stability is an indication that these heuristics display a predictable number of processed messages, independently of the integration process executed. For all integration processes, the qPrior heuristic provided the best average throughput, whereas FIFO provided the worst.

The use of different heuristics generates a significant difference in the average throughput values, in the number of processed and remained messages, cf. Table 5. Low coefficients of variation indicate the adequacy and reliability of the simulations. The results obtained in the test of Scott & Knott applied for comparison of means, show that there were three different groups for throughput, for the number of processed and remained messages, cf. Table 8. The statistical differences among the three heuristics resulting from the experiment with simulations provided support to answer the research question (RQ) and confirm the hypothesis (H). Thus, it is possible to simulate heuristics from task scheduling in an integration process with a simulator, and to find the throughput, the number of processed messages, and performance metrics in the execution of an integration process under high workloads.

6.5. Threats to Validity

Threats to validity can be found in any empirical research (Cruzes & ben Othman, 2017). There are some of them specific to optimisation studies (Wohlin et al., 2012). Factors that could influence the results of the experiments were evaluated and ways to mitigate them were suggested.

6.5.1. Constructor Validity

Constructor validity discusses whether the planning and execution of the study is adequate and able to answer the research question. The experiment was planned according to procedures from empirical software engineering (Basili et al., 2007; Jedlitschka & Pfahl, 2005; Wohlin et al., 2012). Firstly, the research question was defined, the hypothesis was formulated, and the dependent and the independent variables were defined. Next, it was provided information about the execution environment, supporting tools, execution algorithms and data collection. Then, the simulation was carried out in seventy-five different scenarios, and statistical techniques were applied to evaluate the results.

6.5.2. Conclusion Validity

As reported by Wohlin et al. (2012), conclusion validity "is concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment". Statistical techniques were used to assure that the actual outcome observed in the experiment would be related to the heuristics employed and that there was a significant difference amongst them.

6.5.3. Internal Validity

Internal validity aims to ensure that the treatment leading to the outcome mitigate the effects of other uncertain or not measured factors (Feldt & Magazinius, 2010). In this case, treatments are the heuristics. All experiments were performed in the same machine, on security mode, with minimal features and disconnected from the Internet, in order to minimise any influence on the execution time of the algorithms. The algorithms were implemented in Java and a few executions were run before starting the experiments, letting the virtual machine stabilise and eventually perform code optimisation (Pinto, Castor, & Liu, 2014). Additionally, the procedures were accurately inspected and statistical tests were used to validate the performance metrics employed.

6.5.4. External Validity

External validity focuses on the generalisation of results outside the scope of the study (Feldt & Magazinius, 2010). This study is generalised for integration platforms that adopt the integration patterns by Hohpe and Woolf (2004), the Pipes-and-Filters style and task-based model. This study was reported following a practical guideline (Wohlin et al., 2012), in order to make possible its exact reproduction. The experiment is valid to test other parameters, such as integration processes, message arrival rate, and the duration of the simulation. For future works other integration processes shall be experimented, in order to further evaluate the generalisation of results.

7. Conclusion

There are many business possibilities for enterprises in the era of Big Data, but dealing with a large volume of data and extracting exact information on the business is still a challenge (Shoukry et al., 2019). Enterprise Application Integration (EAI) is a field of study that faces the complex task of integrating such data, providing new methodologies, techniques, and tools for the design and implementation of integration processes (Frantz et al., 2016; Ritter et al., 2021). In this article, a tool for simulating task scheduling algorithms was proposed: the Integration Process Simulator (IPS). Three heuristics were evaluated by the use of IPS: First-In-Fist-Out (FIFO), Multi-queue Round Robin (MqRR) and Queue Priority (qPrior), although other scheduling algorithms might be easily implemented in IPS. Performance metrics, such as throughput, the number of processed messages, and the number of remained messages can be obtained from IPS. It also allows to configure the time of the simulation, the initial and the total workload of messages, the rate of inbound messages and the integration process. The simulator was tested with three benchmark integration processes. The results of the simulations proved that qPrior was the best heuristic, providing the highest throughput in high workloads. On the other hand, FIFO was the worst heuristic for it provided the lowest throughput. The statistical analysis confirmed the above and showed that there is a significant difference in performance metrics when task scheduling is performed with FIFO, MqRR, and qPrior. In the future, more metrics and more heuristics shall be included into the simulator, in order to extend the simulations, test other integration processes and compare other workload parameters. Regarding the research questions of the experiment: *the simulator is able to evaluate heuristics used in task scheduling of integration processes. For the three integration processes tested, the qPrior heuristic was the one which was able to process more messages per time unit.*

In the literature review, a single domain-specific tool developed to execute simulations of integration processes in enterprise application integration was found. The simulator found was proposed by Haugg et al. (2019), and is limited to a single heuristic, besides allowing the monitoring of only one metric: the number of threads. Despite other commercial tools which allow simulating task scheduling, which are either concerned with the simulation of tasks in the cloud computing environment and switch network distribution, or they are tools that provide a set of general-purpose building blocks to model practically any kind of system. In this way, to be useful in the field of application integration, software engineers first would have to spend some effort building a model that represents the task-based execution model (a theoretical execution model implemented by some runtime system of integration platforms) and endow it with the capacity to incorporate different heuristics and monitor the required metrics. Only after developing this simulation model, they would be ready to simulate a given integration process. Note that this very tool already provides this simulation model with the support for different heuristics, ready to be used. For this reason, this proposal can be considered a "domain-specific simulato". Besides, the majority of commercial tools have prohibitive costs for small and even mid-size companies and, more often, infeasible for scientific research. Such commercial simulators served as inspiration for the development of this simulator. The simulator has the possibility of monitoring metrics that are unique in this context, such as the number of messages processed, the number of messages that entered the integration process and were not processed, makespan, and throughput. Yet, the simulator is adapted to receive the specific parameters of integration processes, such as the number of inputs, outputs,

communications channels, tasks (and their specific operations), threads, etc. It is worth noting that this simulator has no intention of competing commercially, considering that current commercial tools have great potential in simulating task scheduling.

Acknowledgements

This work was supported by the Coordination for the Improvement of Higher Education Personnel (CAPES), under grant numbers 88881.119518/2016-01 and 88881.136207/2017-01; the Research Support Foundation of the State of Rio Grande do Sul (FAPERGS) under grant number 17/2551-0001206-2; and, the National Council for Scientific and Technological (CNPq) under grant number 309315/2020-4. We would like to thank Dra. Maria do Rosário Laureano and Dr. Sancho M. Oliveira from the Instituto Universitário de Lisboa (ISCTE-IUL) ISTAR-IUL, Lisboa, Portugal and Dra. Iryna Yevseyeva from the De Montfort University, United Kingdom, for their helpful comments in earlier versions of this article.

References

Aazam, M., Huh, E.-N., St-Hilaire, M., Lung, C.-H., & Lambadaris, I. (2016). Cloud of Things: Integration of IoT with Cloud computing. In *Robots and sensor clouds* (pp. 77–940). Springer International Publishing.

Alexander, C., Ishikawa, S., & Silvertein, M. (1977). *A pattern language: towns, buildings, construction*. Oxford University Press.

Alkhanak, E. N., Lee, S. P., Rezaei, R., & 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.

Appel, A. W. (n.d.). A runtime system. *LISP and Symbolic Computation*.

Basili, V. R., Rombach, D., Kitchenham, K. S. B., Selby, D., & Pfahl, R. W. (2007). *Empirical software engineering issues*. Springer Berlin/Heidelberg.

Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., & Kennedy, K. (2005). Task scheduling strategies for workflow-based applications in grids. In *IEEE international symposium on cluster computing and the grid (CCGrid)* (Vol. 2, pp. 759–767).

Boehm, M., Habich, D., Preissler, S., Lehner, W., & Wloka, U. (2011). Cost-based vectorization of instance-based integration processes. *Information Systems*, *36*(1), 3–29.

Buyya, R., & Murshed, M. (2002). Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency Computation Practice and Experience*, *14*(13–15), 1175–1220.

Calheiros, R. N., Ranjan, R., Beloglazov, A., Rose, C. A. F. D., & Buyya, R. (2011). Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, *41*(1), 23–50.

Canon, L.-C., & Jeannot, E. (2007). A comparison of robustness metrics for scheduling DAGs on heterogeneous systems. In *International conference on cluster computing (ieee cluster)* (pp. 558–567).

Cervin, A., & Årzén, K.-E. (2018). Truetime: Simulation tool for performance analysis of real-time embedded systems. In *Model-based design for embedded systems* (pp. 169–200). CRC Press.

Chirkin, A. M., Belloum, A. S. Z., Kovalchuk, S. V., Makkes, M. X., Melnik, M. A., Visheratin, A. A., & Nasonov, D. A. (2017). Execution time estimation for workflow scheduling. *Future Generation Computer Systems*, *75*, 376–387.

Cruz, C. D. (2006). *Programa genes: estatística experimental e matrizes*. Editora Universidade Federal de Viçosa.

Cruzes, D. S., & ben Othman, L. (2017). Threats to validity in empirical software security research. In *Empirical research for sofware security* (pp. 295–320).

Eker, J., & Cervin, A. (1999). A matlab toolbox for real-time and control systems co-design. In *Proceedings sixth international conference on real-time computing systems and applications (RTCSA)* (pp. 320–327).

Fan, K., Zhai, Y., Li, X., & Wang, M. (2018). Review and classification of hybrid shop scheduling. *Production Engineering*, *12*(5), 597–609.

Feldt, R., & Magazinius, A. (2010). Validity threats in empirical software engineering research-an initial survey. In *International Conference on Software Engineering and Knowledge Engineering (SEKE)* (pp. 374–379).

Fernández-Cerero, D., Fernández-Montes, A., Jakóbik, A., Kołodziej, J., & Toro, M. (2018). Score: Simulator for cloud optimization of resources and energy consumption. *Simulation Modelling Practice and Theory*, *82*, 160–173.

Fernández-Cerero, D., Fernández-Montes, A., Ortega, F. J., Jakóbik, A., & Widlak, A. (2019). Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption. *Simulation Modelling Practice and Theory*, *101*, 101966.

Fernández-Cerero, D., Jakóbik, A., Fernández-Montes, A., & Kołodziej, J. (2019). GAME-SCORE: Game-based energy-aware cloud scheduler and simulator for computational clouds. *Simulation Modelling Practice and Theory*, *93*, 3–20.

Frantz, R. Z., Corchuelo, R., & Arjona, J. L. (2011). An efficient orchestration engine for the cloud. In *International Conference on Cloud Computing Technology and Science (CloudCom)* (pp. 711–716).

Frantz, R. Z., Corchuelo, R., & Molina-Jiménez, C. (2012). A proposal to detect errors in enterprise application integration solutions. *Journal of Systems and Software*, *85*(3), 480–497.

Frantz, R. Z., Corchuelo, R., & Roos-Frantz, F. (2016). On the design of a maintainable software development kit to implement integration solutions. *Journal of Systems and Software*, *111*, 89–104.

Freire, D. L., Frantz, R. Z., & Roos-Frantz, F. (2019). Towards optimal thread pool configuration for run-time systems of integration platforms. *International Journal of Computer Applications in Technology*, *XX*(in-press), 1–18.

Freire, D. L., Frantz, R. Z., Roos-Frantz, F., & Basto-Fernandes, V. (2021). Queue-priority optimized algorithm: a novel task scheduling for runtime systems of application integration platforms. *The Journal of Supercomputing*, 1–31.

Freire, D. L., Frantz, R. Z., Roos-Frantz, F., & 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.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, *42*(10), 57–76.

Guo, F., Yu, L., Tian, S., & 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.

Gupta, I., Gupta, S., Choudhary, A., & Jana, P. K. (2019). A hybrid meta-heuristic approach for load balanced workflow scheduling in iaas cloud. In *International conference on distributed computing and internet technology (ICDCIT)* (pp. 73–89).

Haugg, I. G., Frantz, R. Z., Roos-Frantz, F., Sawicki, S., & Zucolotto, B. (2019). Towards optimisation of the number of threads in the integration platform engines using simulation models based on queueing theory. *Revista Brasileira de Computação Aplicada*, *11*(1), 48–58.

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

Hohpe, G. (2005). Your coffee shop doesn't use two-phase commit [asynchronous messaging architecture]. *IEEE software*, *22*(2), 64–66.

Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Howell, F., & McNab, R. (1998). Simjava: A discrete event simulation package for java with applications in computer systems modelling. *Proceedings of the First International Conference on Web-based Modelling and Simulation*, 51–56.

Jedlitschka, A., & Pfahl, D. (2005). Reporting guidelines for controlled experiments in software engineering. In *International Symposium on Empirical Software Engineering (ESEM)* (pp. 95–104).

Jeon, S., & Jung, I. (2018). Experimental evaluation of improved IoT middleware for flexible performance and efficient connectivity. *Ad Hoc Networks*, *70*, 61–72.

Kanagaraj, K., & Swamynathan, S. (2016). A study on performance of dominant scheduling algorithms on standard workflow systems in cloud. In *International conference on informatics and analytics (ICIA)* (pp. 1–6).

Pinto, G., Castor, F., & Liu, Y. D. (2014). Understanding energy behaviors of thread management constructs. In *ACM SIGPLAN Notices* (Vol. 49, pp. 345–360).

Qureshi, K., Shah, S. M. H., & Manuel, P. (2011). Empirical performance evaluation of schedulers for cluster of workstations. *Cluster computing*, *14*(2), 101–113.

Riaz, R., Kazmi, S. H., Kazmi, Z. H., & Shah, S. A. (2018). Randomized dynamic quantum cpu scheduling algorithm. *Journal of Information Communication Technologies and Robotic Applications*, 19–27.

Ritter, D., May, N., & Rinderle-Ma, S. (2017). Patterns for emerging application integration scenarios: A survey. *Information Systems*, *67*, 36–57.

Ritter, D., Rinderle-Ma, S., Montali, M., & Rivkin, A. (2021). Formal foundations for responsible application integration. *Information Systems*, *101*, 101439.

Saifullah, A., Li, J., Agrawal, K., Lu, C., & Gill, C. (2013). Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, *49*(4), 404–435.

Sargent, R. G. (2013). Verification and validation of simulation models. *Journal of simulation*, *7*(1), 12–24.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013). Omega: Flexible, scalable schedulers for large compute clusters. *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 351–364.

Shoukry, A., Khader, J., & Gani, S. (2019). Improving business process and functionality using IoT based E3-value business model. *Electronic Markets*, *1*, 1–10.

Thakur, V., & Kumar., S. (2018). A pragmatic study and analysis of load balancing techniques in parallel computing. In *Information and decision sciences* (pp. 447–454).

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Wood, D. C., & Forman, E. H. (1971). Throughput measurement using a synthetic job stream. In *Fall joint computer conference* (pp. 51–56).

Zhang, T., Pota, H., Chu, C.-C., & Gadh, R. (2018). Real-time renewable energy incentive system for electric vehicles using prioritization and cryptocurrency. *Applied Energy*, *226*, 582–594.

Zhang, Y., Shen, Z.-J. M., & Song, S. (2018). Exact algorithms for distributionally β-robust machine scheduling with uncertain processing times. *INFORMS Journal on Computing*, *30*(4), 662–676.

Appendices

A. Profiles of the integration processes

A.1. Profiles of integration processes

A.1.1. Processing Order (IP1)

- Identification of the integration process tasks:
  VectorIdTask = {1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,16,17};

- Identification of the next task of each task of the integration process:
  VetorNextTask = {{2}, {3}, {4,12,17}, {5,7}, {6}, {7}, {8}, {9}, {10}, {11}, {}, {13,15}, {14}, {15}, {16}, {9}, {}};

- Identification of the execution time range of each task of the integration process:
  VetorTimeExec = {{1,2}, {2,3}, {2,3}, {2,3}, {1,2}, {1,2}, {3,4}, {1,2}, {3,4}, {1,2}, {1,2}, {2,3}, {1,2}, {1,2}, {3,4}, {1,2}, {1,2}};

- Identification of the logic operation type of each task of the integration process:
  VetorOper= {{}, {}, {or}, {and}, {}, {}, {}, {}, {}, {}, {}, {and}, {}, {}, {}, {}, {}};

- Identification of the parallel tasks of the integration process:
  VetorParallelTask = {0,0,0,0,0,0,0,0,0,0,0,4,5,6,7,8,4};

- Identification of the last tasks of the integration process:
  LastTask = {11,17}.

A.1.2. Huelva's County Council (IP2)

- Identification of the integration process tasks:
  VectorIdTask = {1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,16,17,18,19};

- Identification of the next task of each task of the integration process:
  VetorNextTask = {{3}, {3}, {4}, {5,7}, {6}, {7}, {8}, {9}, {10,18}, {11}, {12,14}, {13}, {14}, {15}, {16}, {17}, {}, {19}, {}};

- Identification of the execution time range of each task of the integration process:
  VetorTimeExec = {{1,2}, {1,2}, {3,4}, {2,3}, {1,2}, {1,2}, {3,4}, {1,2}, {2,3}, {1,2}, {2,3}, {1,2}, {1,2}, {3,4}, {1,2}, {1,2}, {1,2}, {1,2}, {1,2}};

- Identification of the logic operation type of each task of the integration process:
  VetorOper= {{}, {}, {}, {and}, {}, {}, {}, {}, {or}, {}, {and}, {}, {}, {}, {}, {}, {}, {}, {}};

- Identification of the parallel tasks of the integration process:
  VetorParallelTask = {0,1,0,0,0,0,5,0,0,0,0,0,0,12,0,0,0,16,17};

- Identification of the last tasks of the integration process:
  LastTask = {17,19}.

A.1.3. Real Estate (IP3)

- Identification of the integration process tasks:
  VectorIdTask = {1,2,3,4,5,6,4,8,9,10,11,12,13,14,15,16,17};

- Identification of the next task of each task of the integration process:
  VetorNextTask = {{5}, {5}, {4}, {5}, {6}, {7,8}, {8}, {9}, {10,11}, {11}, {12}, {13,16}, {14}, {15}, {}, {17}, {}};

- Identification of the execution time range of each task of the integration process:
  VetorTimeExec = {{1,2}, {1,2}, {1,2}, {1,2}, {3,4}, {2,3}, {1,2}, {3,4}, {2,3}, {1,2}, {3,4}, {2,3}, {1,2}, {1,2}, {1,2}, {1,2}, {1,2}};

- Identification of the logic operation type of each task of the integration process:
  VetorOper= {{}, {}, {}, {}, {}, {and}, {}, {}, {and}, {}, {}, {or}, {}, {}, {}, {}, {}};

- Identification of the parallel tasks of the integration process:
  VetorParallelTask = {0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,13,14};

- Identification of the last tasks of the integration process:
  LastTask = {15,17}.

A.2. Flowchart of the simulator

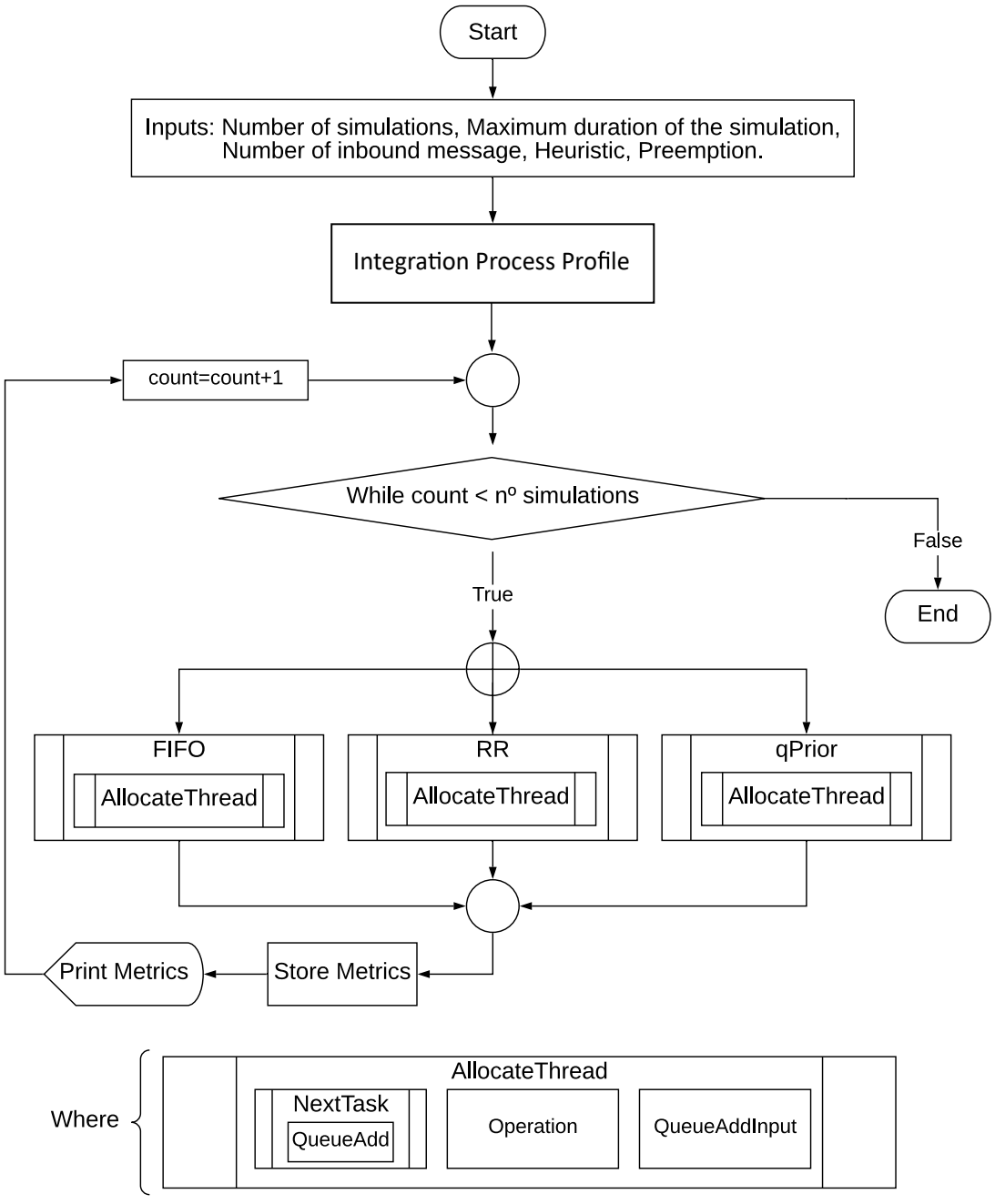


Figure 6. Flowchart of the IPS.