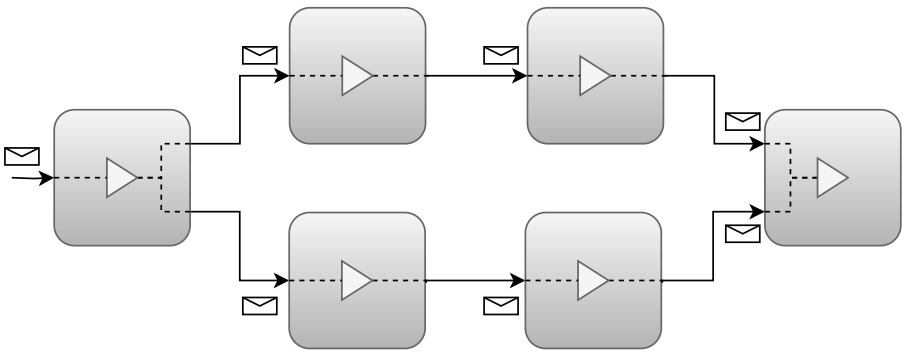
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores



Processamento Paralelo e Distribuído Baseado em *Workflows*

Pedro de Oliveira Fernandes

(Licenciado)

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Luís Manuel da Costa Assunção

Júri:

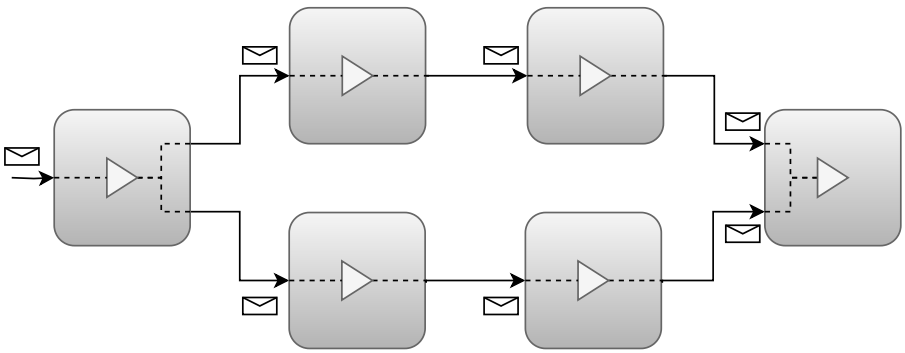
Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor Manuel Martins Barata
Doutor Luís Manuel da Costa Assunção

outubro, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores



Processamento Paralelo e Distribuído Baseado em *Workflows*

Pedro de Oliveira Fernandes

(Licenciado)

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Luís Manuel da Costa Assunção

Júri:

Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor Manuel Martins Barata
Doutor Luís Manuel da Costa Assunção

outubro, 2023

Em memória do meu avô, e em homenagem às minhas avós, aos meus pais e à minha irmã que sempre foram e sempre serão para mim símbolos de resiliência, coragem e força e também uma fonte inesgotável de inspiração e motivação. À minha namorada cujo amor, apoio e força são um pilar essencial na minha vida.

Agradecimentos

Gostaria de expressar a minha mais sincera gratidão às pessoas que tornaram possível a realização deste Trabalho Final de Mestrado e que me ajudaram no meu percurso académico.

Primeiramente, ao meu orientador, Professor Luís Assunção, pela sua inestimável orientação, pelo apoio incansável e dedicação inabalável ao meu sucesso. O seu aconselhamento e motivação, em momentos mais difíceis, foram fundamentais para a conclusão deste Trabalho Final de Mestrado, que não só me ajudou a crescer academicamente, mas também como pessoa.

Quero agradecer à empresa Google, através do Google Cloud Platform, Education Grants, pela grande flexibilidade e suporte na criação de uma infraestrutura de nós computacionais que permitiu a realização deste trabalho, nomeadamente na experimentação com casos reais de workflow.

Aos meus colegas da Critical Techworks, é com profundo apreço que vos agradeço por compreenderem e agilizarem muitas das minhas preocupações durante o meu percurso académico e por sempre me terem ajudado a crescer, não só profissionalmente, mas também a nível pessoal. Em especial consideração, os membros das equipas por onde passei, os "The 418's" desde o tempo tinham o nome "Atoms" e os "Dunkin Squad", vocês acrescentaram um valor inestimável ao meu trajeto. Agradeço pela vossa paciência e colaboração, que foram fundamentais para encontrar um equilíbrio entre o trabalho e os estudos. Cada gesto de compreensão não passou despercebido e fez toda a diferença. Obrigado por tornarem esta jornada mais suave e por fazerem parte do meu sucesso académico.

Quero também estender os meus agradecimentos aos meus colegas de curso, Sérgio e Pedro, cuja camaradagem e amizade enriqueceram não só o meu percurso académico, mas também a minha vida, pois ajudaram-me a crescer imenso durante esta nossa

jornada juntos. As nossas discussões e trocas de ideias foram verdadeiramente inspiradoras e sempre me fizeram refletir bastante.

Ao meu avô Júlio e às minhas avós Manuela e Felicidade, pelo carinho, motivação e amor incondicional que sempre me deram e que foi crucial não só durante todo o meu percurso académico, mas durante toda a minha vida.

À minha mãe Odete e ao meu pai António expresso a minha eterna gratidão pelo apoio incondicional, por acreditarem em mim, pela vossa paciência e por estarem sempre ao meu lado, mesmo nos momentos mais desafiadores. A vossa confiança, amor, ajuda e motivação é fundamental na minha vida e sem vocês não teria conseguido terminar o meu percurso académico.

À minha irmã Catarina, pela compreensão, confidência e apoio ao longo de não só todo o meu percurso académico, mas toda a minha vida. Obrigado por estares presente nos momentos mais importantes e por sempre acreditares em mim nos momentos difíceis.

Ao meu cunhado André, pelos conselhos e palavras sábias que sempre me motivaram em momentos mais difíceis.

Aos meus sobrinhos, que embora ainda não saibam ler, deixo aqui o vosso apreço e amor incondicional. A vossa inocência serve como um lembrete constante para encarar a vida com mais simplicidade, para apreciar as pequenas coisas e para compreender que o nosso percurso de aprendizagem é uma jornada contínua. Cada sorriso, cada abraço e cada momento compartilhado com vocês é uma lição de amor, gratidão e alegria.

A mi querida novia, Raquel, te agradezco por todo el amor, comprensión, cariño, paciencia, sacrificio y ayuda durante este recorrido académico. Tu presencia y apoyo me dieron fuerza en los momentos difíciles e hicieron que este logro académico sea aún más significativo.

A todos os meus amigos, quero expressar minha profunda gratidão pela vossa compreensão durante o meu percurso académico. O vosso apoio e paciência significaram muito para mim enquanto me dediquei às exigências deste desafio. Mesmo quando a distância física nos separou, o vosso entendimento e apoio foram uma bússola constante. Obrigado por compreenderem o meu distanciamento e por continuarem a ser uma parte valiosa da minha vida.

A todos vocês, mais uma vez, o meu mais profundo agradecimento. Este trabalho não teria sido possível sem o vosso apoio.

Resumo

As infraestruturas de computação distribuídas de larga escala suportadas em virtualização e plataformas de computação na nuvem (*Cloud*) permitem a execução de aplicações de processamento de dados modeladas por decomposição em múltiplas tarefas que interagem entre si, segundo modelos representados por grafos, usando um paradigma designado por *Scientific Workflows*. Na última década, surgiram imensas propostas de sistemas de *workflow*. No entanto, apesar da maior ou menor popularidade de alguns sistemas, continuam a persistir algumas questões em aberto, passíveis de serem melhoradas.

O trabalho descrito neste documento é uma contribuição para encontrar respostas para algumas questões em aberto, propondo um modelo de *workflow* com as seguintes características: i) descentralização do controlo de execução das múltiplas tarefas de um *workflow*; ii) execução de um *workflow* com múltiplas iterações; iii) possibilidade de especificar réplicas de uma tarefa para suportar equilíbrio de carga; iv) encapsulamento de tarefas em *containers*, possibilitando a sua execução em múltiplos nós computacionais; v) especificação flexível de *workflows* independentemente da infraestrutura tecnológica de execução.

A partir do modelo proposto é apresentada uma arquitetura conceptual de suporte ao modelo, culminado com o desenvolvimento de um protótipo que permite experimentação e validação do modelo, com estudo de casos de aplicações concretas modeladas segundo o paradigma de *workflow*.

O protótipo de experimentação executa-se numa infraestrutura computacional que utiliza máquinas virtuais, formando um *cluster* virtual com múltiplos nós computacionais (máquinas virtuais) alojados na *Google Cloud Platform*. Os múltiplos nós computacionais partilham ficheiros através do sistema de ficheiros distribuídos *Gluster*. As múltiplas tarefas de um *workflow*, ativadas no contexto de componentes autónomas designadas *Activities*, executam-se encapsuladas em *containers Docker*, permitindo uma

grande flexibilidade de desenvolvimento e reutilização dessas *Tasks* em múltiplos casos de aplicação.
O trabalho termina com uma forte componente experimental de casos concretos de aplicação, nomeadamente uma aplicação que permite detetar objetos e textos em imagens, uma aplicação que implementa o modelo *MapReduce* para realizar o histograma de ocorrência de palavras em ficheiros de texto.
A experimentação com o protótipo implementado permite concluir que o modelo é suficientemente genérico, desacoplado de detalhes tecnológicos, permitindo o desenvolvimento de *workflows* em múltiplas áreas das ciências e engenharia.

Palavras-chave: Processamento distribuído, *Scientific workflows*, Máquinas virtuais, *Containers*, *Publish/Subscribe*

Abstract

Distributed computing infrastructures with large scale and supported by virtualization and Cloud computing platforms enable the execution of data processing applications modeled by decomposition into multiple tasks that interact with each other, according to models represented by graphs, using the Scientific Workflows paradigm.

In the last decade, many proposals for workflow systems have emerged. However, despite the greater or lesser popularity of some systems, there remain some open issues that could be improved.

The work described in this document is a contribution to finding answers to some open issues, proposing a workflow model with the following characteristics: i) decentralization of control over the execution of multiple tasks in a workflow; ii) execution of a workflow with multiple iterations; iii) possibility of specifying replicas of a task to support load balancing; iv) encapsulation of tasks in containers, enabling their execution on multiple computing nodes; v) flexible specification of workflows regardless of the execution technological infrastructure.

Based on the proposed model, a conceptual architecture to support the model is presented, culminating in the development of a prototype that allows experimentation and validation of the model, with case studies of concrete applications, modeled according to the workflow paradigm.

The prototype for experimentation runs on a computing infrastructure that uses virtual machines, forming a virtual cluster with multiple computing nodes (virtual machines) hosted on the Google Cloud Platform. Multiple computing nodes share files through the Gluster distributed file system. The multiple tasks of a workflow, activated in the context of autonomous components called Activities, are executed encapsulated in Docker containers, allowing great flexibility in the development and reuse of these Tasks in multiple application cases.

The work ends with a strong experimental component of concrete application cases,

namely an application that allows detecting objects and texts in images, an application that implements the MapReduce model to create the histogram of word occurrence in text files.
Experimentation with the implemented prototype allows us to conclude that the model is sufficiently generic, decoupled from technological details, allowing the development of workflows in multiple areas of science and engineering.

Keywords: Distributed processing, Scientific workflows, Virtual machines, Containers, Publish/Subscribe

Índice

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Abreviaturas e Siglas

AMQP *Advanced Message Queue Protocol*. 54, 60
API *Application Programming Interface*. xviii, 15, 42, 43, 49, 50, 60, 62, 63, 64, 94, 106, 107, 109
AWA *Autonomic Workflow Activities*. 13
AWARD *Autonomic Workflow Activities Reconfigurable and Dynamic*. 12, 13, 16, 17, 104

DAG *Directed Acyclic Graph*. 2, 8, 11, 12, 15, 24
DAX *Directed Acyclic Graph in Xml*. 11, 12

FaaS *Function as a Service*. 3

GCP *Google Cloud Platform*. 53, 54, 55, 56, 81, 84, 89, 92, 96, 99, 106, 107, 108
GSURI *Google Cloud Storage URI*. 93, 94, 95
GUI *Graphical User Interface*. 8, 9, 10

HTTP *Hypertext Transfer Protocol*. xxiii, 42, 43, 44, 45, 54, 63, 64

IaaS *Infrastructure as a Service*. 38

JSON *JavaScript Object Notation*. 4, 16, 17, 36, 45, 65, 69, 71, 91, 94, 95, 96, 101, 104, 105, 107
JVM *Java Virtual Machine*. 53

MOM *Message Oriented Middleware*. 15, 23, 41, 54
MoML *Modeling Markup Language*. 11

PA Progressão Aritmética. 78, 79, 80
PN *Process Networks*. 13

REST *Representational State Transfer*. xviii, 42, 43, 49, 50, 54, 59, 62, 63, 64, 106, 107, 109

TFM Trabalho Final de Mestrado. 2, 3, 4, 7, 10, 14, 15, 16, 20, 103, 109

VM *Virtual Machine*. xxiii, 3, 10, 13, 14, 16, 35, 38, 53, 55, 56, 57, 60, 62, 81, 84, 89, 96, 99, 106, 107

WASA *Workflow-based Architecture for Scientific Applications*. 7
WfMC *Workflow Management Coalition*. 1
WfMS *Workflow Management System*. 1, 2, 7, 8, 9, 10, 11, 12, 13, 16

XML *Extensible Markup Language*. 11, 12, 13, 16, 36
XSD *Xml Schema Definition*. 13

YAML *YAML Ain't Markup Language*. 15, 16, 36

1

Introdução

Desde o início do século passado, essencialmente as empresas industriais, necessitaram de estabelecer claramente os seus processos de trabalho, tendo por objetivo a otimização e garantia que esses processos se podiam executar repetidamente através de sequências de tarefas e dos fluxos de informação entre as mesmas. Assim, surgiu o conceito de *workflow* para expressar fluxos de trabalho como um grafo, em que os vértices são as tarefas e os arcos indicam fluxos de dependência e comunicação entre os vértices.

Com o desenvolvimento da informática, na segunda parte do século passado, os sistemas de informação de suporte aos processos de negócio das organizações foram também modelados usando *workflows*, executados e coordenados por um sistema centralizado, designado por motor de *workflow* ou também designados por *Workflow Management System* (WfMS) [1].

Em 1993, algumas empresas, programadores, analistas de negócio e investigadores fundaram a *Workflow Management Coalition* (WfMC) [2], como uma comunidade global que teve como objetivo a definição de normas e a especificação de linguagens para suportar o desenvolvimento de sistemas de *workflow*, permitindo que as organizações se concentrassem no negócio e não nos sistemas de informação de suporte ao mesmo.

A existência de infraestruturas de computação distribuídas de larga escala, nomeadamente em ambientes de virtualização e plataformas de computação na nuvem (*Cloud*) permitem, cada vez mais, a execução de aplicações complexas de processamento de dados, que podem ser modeladas por decomposição em múltiplas tarefas. De acordo

com os fluxos de controlo e de dados inerentes ao contexto das aplicações, as múltiplas tarefas podem executar-se, interagindo entre si, segundo modelos representados por grafos com controlo mais ou menos centralizado, isto é, nos últimos anos o processamento intensivo de dados tem vindo também a ser realizado através de *workflows*.

Um *workflow* é representado por um grafo, tipicamente acíclico (*Directed Acyclic Graph* (DAG)), onde os vértices, representam a execução das tarefas (*Tasks*) e os arcos representam os fluxos de informação ou de controlo entre a sequência de tarefas. Por exemplo, o grafo da Figura 1.1 representa uma aplicação decomposta como uma sequência das tarefas *A*, *B* e *C*, em que só depois da execução de *A* se executa *B* e depois de *B* se executa *C*. As tarefas *B* e *C* podem usar eventuais resultados das suas tarefas predecessoras e a tarefa *C* produzirá o resultado final da aplicação.

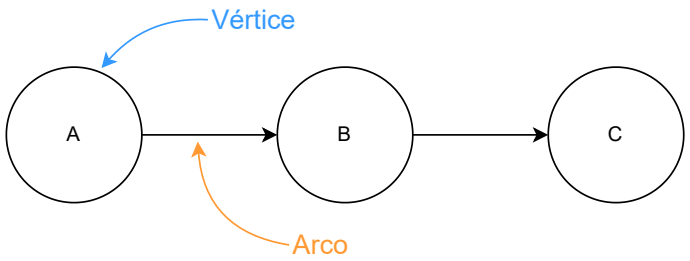


Figura 1.1: Representação de um grafo acíclico

As vantagens inerentes deste paradigma mostraram-se úteis não só no contexto comercial, mas também no contexto científico, surgindo as primeiras utilizações deste paradigma pela comunidade científica no final do século passado. No entanto, foi o advento do conceito de *e-science* e a sua popularidade na comunidade científica, que originou uma adoção em larga escala dos *workflows* para fins de investigação na área da ciência. Estes *workflows* são designados como *scientific workflows* [3], sendo muito utilizados como forma de representação e decomposição em tarefas de aplicações complexas de processamento paralelo e distribuído nos mais variados domínios da ciência.

Esta abordagem mostrou ser uma ferramenta flexível devido à sua capacidade de orquestração de computações distribuídas em larga escala e em várias áreas da ciência, tais como, entre outras, astronomia, bioinformática, física e investigação sísmica.

Para suprir a necessidade crescente, da comunidade científica, de modular aplicações, segundo o paradigma de *workflow*, foram adaptados alguns *Workflow Management System* (WfMS) e desenvolvidos novos sistemas de *workflow* de apoio à modulação e execução de aplicações nas várias áreas da ciência. No entanto, apesar da popularidade e diversidade, em geral os WfMS acarretam algumas características de execução e modelação passíveis de serem melhoradas. Assim, é na análise e proposta de melhoria dessas características que se foca este Trabalho Final de Mestrado.

1.1 Objetivo do Trabalho Final de Mestrado

Este Trabalho Final de Mestrado (TFM) teve como objetivo a definição de um modelo de *workflow*, bem como a implementação de um protótipo demonstrativo para a validação do modelo em questão e suportar a experimentação de casos concretos de aplicações. Adicionalmente, considerou-se importante possibilitar a modelação e execução de aplicações distribuídas, por composição de múltiplas tarefas, que se executam nas atuais infraestruturas de virtualização baseadas em máquinas virtuais (VM) e *containers*.

Uma tarefa de *workflow* (nó do grafo) deveria ser o mais autónoma possível podendo ser um simples processo de sistema operativo, um processo ou micro serviço interno a um *container*, sendo importante que estes processos ou micro serviços possam invocar serviços externos em qualquer *Cloud* pública ou privada, ou mesmo a ativação de funções *Function as a Service* (FaaS) segundo a tendência emergente de computação *serverless*.

Um objetivo importante foi, as tarefas interagirem entre si (arcos do grafo) com mensagens de dados (*data-flow*) ou simplesmente de controlo (*control-flow*) segundo o modelo *publish/subscribe*.

Em síntese, foram estabelecidos os seguintes objetivos para as principais características do modelo proposto neste Trabalho Final de Mestrado:

- Possibilidade de execução de *workflows* em infraestruturas distribuídas baseadas em tecnologias de virtualização, tais como VM e *containers*, permitindo que as tarefas de um *workflow* se executem em múltiplos nós de computação heterogéneos em termos de capacidades de processamento e de memória;
- Controlo descentralizado de execução das tarefas de forma a que um *workflow* tire partido de computação paralela e distribuída;
- Dado que as tarefas se executam em diferentes nós de computação, os fluxos de dados e de controlo entre as diversas tarefas são suportados por um mecanismo de comunicação por mensagens *publish/subscribe*;
- Especificação de *workflows* suportando padrões de *workflow* [4] básicos;
- Suporte para execução de *workflows* com múltiplas ou infinitas iterações;
- Suporte do padrão *load balancing*, com base em réplicas de tarefas que impliquem longos tempos de execução;

- Especificação dos *workflows* e desenvolvimento aberto das tarefas, o mais desacoplado possível de casos concretos de um determinado domínio das ciências ou de linguagens de programação;
- Especificação de *workflows* numa linguagem de representação de dados facilmente interpretada (*parsed*) por ferramentas de desenvolvimento de *software*, por exemplo *JavaScript Object Notation* (JSON).

Um objetivo fundamental consistiu em implementar um protótipo funcional e operacional que não só permitisse a experimentação para validação do modelo proposto, mas também a exploração e execução de casos de uso de aplicações concretas.

Um aspeto relevante nos sistemas de *workflow* são as ferramentas de ajuda ao desenho, execução e análise de resultados das aplicações modeladas como *workflows*, nomeadamente as interfaces de utilizador serem gráficas e amigáveis. Adicionalmente são também importantes requisitos de segurança nos pilares de confidencialidade e integridade, pois num ambiente onde se processam dados poderão existir implicações de sigilo ou mesmo de propriedade intelectual que interessa proteger. No entanto, desde o inicio deste Trabalho Final de Mestrado (TFM) não foram consideradas as dimensões de segurança e o desenvolvimento de interfaces gráficas de utilizador, nomeadamente de desenho e especificação dos *workflows*.

Ao longo deste documento será demonstrado que todos os objetivos atrás estipulados foram cumpridos.

1.2 Organização do Documento

Este documento está organizado em sete capítulos e um anexo.

Após esta introdução (Capítulo 1), no Capítulo 2 é feito um resumo do estudo e análise de trabalhos relacionados na área dos *scientific workflows*.

No Capítulo 3 é apresentada uma proposta de um modelo que suporta as características estipuladas na Secção 1.1. O modelo proposto também apresenta soluções e possibilidades de melhoria para algumas limitações dos trabalhos relacionados, indicados no Capítulo 2.

Tendo no Capítulo 3 sido definido um modelo de *workflow*, no Capítulo 4, encontra-se descrita a arquitetura conceptual de suporte à execução desse modelo.

No Capítulo 5, é detalhada a implementação de um protótipo de experimentação e execução de *workflows* segundo o modelo proposto no Capítulo 3 e seguindo a arquitetura

definida no Capítulo 4. São também apresentadas e justificadas as opções tecnológicas adotadas no processo de desenvolvimento do protótipo.

No Capítulo 6, é apresentada a experimentação e validação do modelo proposto, bem como a correta operacionalidade do protótipo implementado. São explorados casos de uso de aplicações concretas fazendo uma análise dos resultados obtidos.

Por fim, no Capítulo 7 são apresentadas conclusões do trabalho realizado bem como uma síntese de aspetos considerados relevantes a melhorar em trabalho futuro.

No final do documento é apresentado em anexo a especificação completa de um *workflow* para ser executado usando o protótipo de experimentação.

2

Trabalho Relacionado

Neste capítulo apresentamos uma definição dos conceitos *scientific workflow* e *Workflow Management System* (WfMS), e é feita posteriormente uma síntese de trabalhos relacionados. Os trabalhos escolhidos para a realização de uma pesquisa e análise foram WfMS tipicamente utilizados para o desenvolvimento de *workflows* científicos. A análise realizada teve como base algumas características que consideramos relevantes serem suportadas por um modelo de *workflow*. Para cada caso estudado, apresentamos as vantagens e as desvantagens das características que influenciaram a proposta de um novo modelo de *workflow*, na realização deste Trabalho Final de Mestrado.

2.1 Conceito de *Scientific Workflow*

Um *scientific workflow* [3] é um *workflow* criado com o propósito de alcançar um objetivo científico num determinado domínio da ciência, podendo ser representado, tal como referido no Capítulo 1, através de um grafo. Um dos primeiros projetos a enfatizar a importância do conceito de *workflow* no processamento de dados de teor científico foi o projeto *Workflow-based Architecture for Scientific Applications* (WASA) [5], introduzindo o termo "*scientific workflow*".

Com o aumento acentuado da quantidade e diversidade de informação em todos os domínios científicos, os *workflows* têm um papel cada vez mais importante em possibilitar cientistas e pesquisadores a desenvolver métodos de análise e processamento

de informação, partindo de múltiplos recursos de dados e tirando partido de várias plataformas de computação.

A grande vantagem do paradigma de *workflow* no processamento de dados é a possibilidade de processamento desses dados através da execução das diferentes tarefas em múltiplos nós de execução distribuídos e heterogéneos.

O paradigma dos *scientific workflows* possibilitou o desenvolvimento de diversos projetos relevantes no domínio científico, tais como: *Galactic Plane* [6], *RseqFlow* [7], *Montage* [8], *CyberShake* [9] e *LIGO Inspiral Analysis Workflow* [10], contribuindo para diversos avanços na área da ciência.

Tipicamente, o desenvolvimento e execução de *workflows* de um dado domínio científico é suportado por *Workflow Management System* (WfMS), existindo alguns mais vocacionados para determinados domínios da ciência. Por exemplo, o sistema *Triana* [11] nasceu como um ambiente de resolução de problemas (*problem-solving environment*) baseado em *workflows* para suportar a análise de dados num projeto de estudo de ondas gravitacionais. Outro sistema muito citado na bibliografia é o *Taverna* [12] cujo objetivo inicial era satisfazer as necessidades de desenhar *workflows* na área da bioinformática.

2.2 Conceito de *Workflow Management System*

Um *Workflow Management System* (WfMS) é um sistema e um conjunto de ferramentas de apoio ao desenvolvimento e execução de *workflows*. Muitos WfMS são utilizados no contexto de processo de negócio das organizações. No entanto existe uma grande oferta de WfMS *open-source*, concebidos com o propósito de suportarem processos de investigação cientifica.

Como referido anteriormente, um *workflow* é a automatização de um processo durante o qual é processada informação por diferentes tarefas. Os WfMS permitem a automatização destes processos gerindo os dados e a execução de um *workflow* numa infraestrutura computacional. Os WfMS também facilitam o mapeamento de aplicações complexas sobre o paradigma de *workflow*, expressando as tarefas de processamento e as suas dependências como um grafo tipicamente acíclico (*Directed Acyclic Graph* (DAG)). Este mapeamento é normalmente concretizado através de uma *Graphical User Interface* (GUI), tornando o processo de criar e compor *workflows* mais agradável para o utilizador. É também comum existirem WfMS com uma interface de utilização via comandos de linha num terminal, baseados em *scripts* e normalmente utilizados para a descrição de *workflows* mais complexos e de maior escala. No entanto, este tipo de interface com

comandos tem uma maior curva de aprendizagem face a interfaces GUI, devido a estas requererem competências de programação.

Os WfMS tipicamente também oferecem suporte à reutilização de *workflows*, não só os desenvolvidos anteriormente pelo mesmo utilizador, mas aos desenvolvidos por outros, promovendo a colaboração de cientistas e investigadores.

Com o aumento da necessidade de análise e computação de grande quantidade de dados, surgiram diversos tipos de sistemas dedicados a este problema, tais como sistemas baseados no modelo *MapReduce* [13], sistemas de apoio à execução de processamento concorrente e distribuído por nós de computação heterogéneos e sistemas de suporte ao processamento de *data streaming* em tempo real. No entanto, apesar das necessidades de processamento de dados em larga escala, oriundos de fontes distribuídas, o controlo dos WfMS é tipicamente centralizado, pelo que, a execução de *scientific workflows* em ambientes dinâmicos e distribuídos é ineficiente devido ao engarrafamento de dados num único ponto.

2.3 Pesquisa e Análise de *Workflow Management Systems*

Nesta secção apresentamos sucintamente alguns sistemas de apoio à modulação e execução de *workflows*, bem como as suas principais características que definimos como relevantes para serem suportadas por um modelo de *workflow*. A pesquisa e análise focou-se nos critérios que dessem resposta às seguintes questões:

- Qual o tipo de controlo de execução (*enactment engine*) de um *workflow*, centralizado por oposição a descentralizado?;
- É possível especificar *workflows* em linguagens de *script* facilmente interpretadas (*parsed*) por ferramentas de desenvolvimento de *software*?;
- Qual o grau de desacoplamento entre as aplicações concretas de um determinado domínio das ciências, o modelo de especificação, o modelo de desenvolvimento das tarefas e de execução dos *workflows*?;
- Suporta o desenvolvimento de *workflows* com longo tempo de execução e com múltiplas iterações?;
- Permite a especificação de padrões de *workflow* básicos e não básicos, como *load balancing* suportando replicação de tarefas?;

- Quais os mecanismos de suporte aos fluxos de dados e de controlo entre as diversas tarefas dos *workflows*?;
- Que tipo de infraestrutura de execução? Baseada em tecnologias de virtualização, tais como máquinas virtuais (VM) e *containers*? É permitido que as tarefas de um *workflow* possam ser executadas de forma distribuída em múltiplos nós de computação heterogéneos em termos de capacidades de processamento e de memória?.

2.3.1 *Kepler*

O Kepler [14] é um *Workflow Management System* (WfMS) *open-source* especializado para o desenvolvimento de *scientific workflows* [15], [16], [17]. É bastante popular devido a sua *Graphical User Interface* (GUI) ser *user-friendly*, o que torna a curva de aprendizagem do mesmo, menos acentuada quando comparado com outros WfMS do domínio científico. Este WfMS desenvolvido sobre a linguagem de programação *Java*, é uma extensão do projeto *Ptolemy II* [18], que por sua vez é um ambiente de apoio à modulação, desenho e simulação de sistemas concorrentes. O *Kepler* herda do *Ptolemy II* tanto a sua GUI como os componentes constituintes da definição de um *workflow*, tais como *Directors* e *Actors*.

No contexto do *Kepler*, um *workflow* é composto por vários *Actors* ligados entre si através de *channels*. Os *Actors* são os componentes que contêm as tarefas a serem executadas durante um *workflow* e os *channels* são o mecanismo de comunicação entre *Actors*. Também neste contexto, os *Directors* são o componente responsável pelo controlo da execução do *workflow*, podendo este assumir diferentes modelos de execução, tais como, *Synchronous Data Flow*, *Dynamic Data Flow*, *Process Network*, que mais interessam para o modelo a desenvolver neste Trabalho Final de Mestrado (TFM).

❖ Principais Características

O *Kepler* oferece um desenvolvimento de *workflows* sem estar acoplado a um determinado domínio das ciências, disponibilizando diferentes *Actors* especializados em diversos domínios, e permitindo também o desenvolvimento de raiz de novos *Actors*.

O *Kepler* disponibiliza *Directors* com diversos modelos de execução, incluindo iterações dependentes e independentes do resultado da iteração prévia. Apesar dos diversos modelos de execução, nenhum deles oferece a possibilidade de um controlo descentralizado. No *Kepler*, o controlo da execução de um *workflow* é centralizado num *Director* responsável pela orquestração da execução de todos os *Actors* de um *workflow*.

Os *Actors*, por omissão, são executados no contexto de *threads Java* dentro de um processo monolítico de um sistema operativo, causando dificuldades em executar um *workflow* de forma distribuída. Isto consequentemente obriga ao fluxo de dados entre os *Actors* a ser realizado na memória do processo de sistema, onde está a ser executado o *workflow*. Para além disto, o *Kepler* não suporta a execução de *Actors* em ambientes de virtualização por exemplo *containers*.

No *Kepler*, a especificação de um *workflow* é representada em *Extensible Markup Language* (XML) utilizando a *Modeling Markup Language* (MoML) também herdada do *Ptolomey II*. Nesta especificação são suportados padrões de *workflow* básicos e não-básicos.

2.3.2 *Pegasus*

O *Pegasus* [19] é um WfMS de mapeamento de aplicações sobre ambientes distribuídos tais como, *clusters*, *grids*, *Clouds* e supercomputadores. Uma aplicação é modelada através da definição de um *workflow* de forma abstrata como um *Directed Acyclic Graph* (DAG), utilizando uma linguagem de especificação própria, baseada em XML, cujo nome é *Directed Acyclic Graph in Xml* (DAX). De forma a executar um *workflow*, é feito um mapeamento da aplicação em causa sobre os recursos computacionais disponibilizados pelo programador.

O *Pegasus* é constituído principalmente por cinco componentes:

1. *Mapper* – ferramenta que gera um *workflow* executável a partir de uma definição abstrata de um *workflow*. Encontrando automaticamente o *software*, dados e recursos computacionais apropriados para possibilitar a execução do *workflow*. O *Mapper* pode ainda reestruturar o *workflow* de forma a otimizar o desempenho da sua execução;

2. *Local Workflow Execution Engine* – motor de execução do *workflow* providenciado pelo DAG-Man [20]. Este motor é responsável por submeter e monitorizar a execução das tarefas definidas no *workflow* mapeado pelo *Mapper*;

3. *Job Scheduler* – é uma componente de agendamento, providenciada pelo *HTCondor sched* [21] que gere as tarefas individuais, supervisionando a sua execução tanto localmente como num ambiente remoto;

4. *Remote Workflow Execution Engine* - proporciona o supervisionamento da execução de tarefas em recursos remotos;

5. *Monitoring Component* – componente que monitoriza o progresso da execução do *workflow*, guardando numa base de dados os *Logs* gerados durante a execução.

❖ **Principais Características**

O *Pegasus* é um WfMS, completamente desacoplado de qualquer área científica, que permite a execução de *workflows* em ambientes distribuídos heterogéneos, através de um controlo de execução centralizado (DAG-Man). A partir da versão 4.8.0, é suportada a utilização de *containers* para executar tarefas do *workflow*, permitindo assim serem executados *workflows* em infraestruturas que suportam tecnologias de virtualização.

Os *workflows* a serem mapeados pelo *Pegasus*, são descritos de forma abstrata através de um formato baseado em XML chamado *Directed Acyclic Graph in Xml* (DAX), suportando padrões de *workflow* básicos e não-básicos.

O *Pegasus* apenas suporta a especificação de *workflows* do tipo *Directed Acyclic Graph* (DAG), pelo que não suporta iterações, nem qualquer tipo de balanceamento de carga ou réplicas de tarefas.

O *Pegasus* providencia um fluxo de dados entre as tarefas do *workflow* através de sistemas de ficheiros partilhados.

2.3.3 AWARD

O modelo *Autonomic Workflow Activities Reconfigurable and Dynamic* (AWARD) [22] é um modelo de suporte ao desenvolvimento e execução de *workflows*, que teve como motivação propor soluções para alguns dos problemas existentes nos WfMS mais populares, tais como:

- A dependência de um controlo da execução do *workflow* centralizado;
- A falta de flexibilidade de suporte a *workflows* com tempo de execução longo e com múltiplas iterações, nomeadamente com réplicas para balanceamento de carga;
- A falta de oferta de suporte à reconfiguração dinâmica da estrutura e comportamento de um *workflow* em tempo de execução, permitindo alguns tipos de recuperação de falhas de execução;
- Não serem suportados padrões de *workflow* não básicos, tais como *feedback loops* e *load balancing*.

O modelo AWARD é baseado no modelo de computação *Process Networks* (PN) [23] e estende o conceito das tarefas de um *workflow*, designadas no modelo como *Autonomic Workflow Activities* (AWA), definindo-as como processos independentes, distribuídos, com controlo de execução autónomo, e com possibilidade de serem executados de forma paralela em ambientes de execução distribuídos, tais como *clusters* e *Clouds*.

Cada AWA executa um componente de *software* designado por *Task*. Esta *Task* é desenvolvida como uma classe *Java* que implementa uma interface genérica permitindo o seu desenvolvimento com abstração dos detalhes de baixo nível, nomeadamente do controlo de execução do *workflow*, bem como dos fluxos de dados entre as diferentes AWA de um *workflow*. O desenvolvimento de uma *Task* é feito de forma desacoplado da infraestrutura.

Durante a execução de um *workflow*, o fluxo de informação entre as AWA é realizado através de *Links*, como um mecanismo abstrato de passagem de informação entre as *Activities* de acordo com a estrutura em grafo do *workflow*. Este mecanismo de comunicação é concretizado através do conceito de *AWARD Space*, como um espaço global partilhado no âmbito do *workflow* e que é baseado no modelo *Linda* [24]. O *AWARD Space* é um *tuple space* que por sua vez é uma implementação do paradigma de memória associativa para computação paralela e distribuída, proporcionando um repositório de *Tuples* cujo acesso pode ser feito de forma concorrente. Assim as *Output Ports* de uma AWA escrevem *Tuples* neste *AWARD Space* e os *Input Ports* das AWA sucessoras no grafo do *workflow* consumem estes *Tuples*.

❖ **Principais Características**

O modelo AWARD oferece soluções para alguns dos problemas encontrados nos WfMS mais populares, oferecendo um forte desacoplamento às implementações concretas das aplicações. O modelo AWARD disponibiliza um tipo de controlo de execução de *workflows* descentralizado, através de *Activities* autónomas, suportando o fluxo de dados entre essas *Activities* através do *AWARD Space*, seguindo o modelo *publish/subscribe*.

Os *workflows* são especificados através da linguagem XML, obedecendo a um esquema bem definido, *Xml Schema Definition* (XSD), que permite validar se a especificação está conforme o modelo. Esta especificação suporta a definição de *workflows* com tempo de execução longo e com múltiplas iterações, padrões de *workflow* básicos e não básicos, tais como *feedback loops* e *load balancing* suportado pela possibilidade de replicação de *Activities*.

O modelo AWARD permite a execução dos *workflows* em ambientes heterogéneos distribuídos, nomeadamente em VM de *Clouds*. No entanto, não permite a execução de

tarefas suportadas por virtualização, como *containers*.

A utilização do paradigma de *tuple space* também introduz limitações de escala, pois na implementação do modelo o *AWARD Space* é um único processo de sistema operativo, o que introduz limitações de memória para armazenar os *Tuples* que pode limitar a execução de *workflows* com muitas tarefas e com muitas iterações.

2.3.4 Trabalhos e Sistemas de *Workflow* Recentes

Nos anos mais recentes, a crescente necessidade de processar dados nas mais variadas áreas das ciências continua a apostar no paradigma de *scientific workflows*, como forma de decompor problemas complexos em múltiplas tarefas. Contrariamente à primeira década do século XXI, onde requisitos de maiores capacidades de processamento requeriam a aquisição de *clusters* de máquinas físicas, as evoluções, nos últimos anos, das tecnologias de virtualização permitem atualmente criar ambientes flexíveis de execução de *workflows*, usando múltiplos nós de computação baseados em VM e *containers*.

Nos últimos anos, têm sido publicados imensos artigos relacionadas com *scientific workflows*. Como exemplos, entre 2020 e 2023, a pesquisa do termo "*Scientific workflows*" no *IEEE Xplore* dá como resultado 465 referências e na *ACM Digital Library* a mesma pesquisa resulta em 31904 referências.

De seguida apresenta-se uma pequena síntese de trabalhos relacionados recentes, que consideramos significativos e que ajudaram a motivar a realização deste Trabalho Final de Mestrado (TFM).

No artigo de 2018, [25], discute-se a vantagem de usar *Docker containers* como forma de encapsular as pilhas de *software*, facilitando a portabilidade e reprodutibilidade nos processos de investigação científica baseados em *workflows*. O artigo discute diferentes formas de realizar a composição de *containers*, mais estática ou mais dinâmica, para criar *workflows*, bem como realiza a discussão se um *container* deve encapsular unicamente a execução da tarefa do *workflows*, ou se deve encapsular também dados.

O artigo de 2020, [26], apresenta a implementação de um protótipo de um sistema de execução de *workflows* no contexto *Big Data*. Os *workflows* em discussão são caracterizados por uma sequência de passos (*steps*) que realizam a leitura de dados (*data ingestion*) a partir de ficheiros ou outras fontes, realizam a transformação de dados noutros formatos, terminando com passos de análise dos dados. O sistema propõe que cada passo (*step*) do *workflow* seja encapsulado num *Docker container* e a troca de dados entre os diferentes passos do *workflow* seja realizado através de um sistema de mensagens

Message Oriented Middleware (MOM) designado *KubeMQ*. As imagens dos *containers* de cada passo (*step*) derivam de uma imagem base genérica.

Em [27] apresenta-se um modelo com abstrações de programação para a gestão eficiente de dados em dispositivos de armazenamento com múltiplos níveis, propondo um modelo de espaço de armazenamento virtual para facilitar a gestão dos dados dos *scientific workflows*. Na avaliação são usados *workflows* conhecidos, por exemplo, o sistema *Montage* que requer que os dados a processar se localizem numa única diretoria. O artigo realça a importância de um sistema de *workflow* suportar estratégias de partilha de dados entre os múltiplos nós computacionais.

Em [28] é descrito o *framework DFLOW* que tem como principal objetivo permitir que *scientific workflows* complexos possam beneficiar da orquestração dinâmica dos recursos de execução em ambientes de *clusters Linux*. O artigo apresenta resultados experimentais com vários *workflows* desenvolvidos para estudar simulações complexas na área da Física, Biologia e Engenharia de materiais.

Entre o ano de 2020 e 2022 existem várias publicações, tais como, [29] e [30] onde são apresentadas várias questões em aberto que necessitam maior investigação na área dos *scientific workflows*, nomeadamente para facilitar a colaboração de múltiplos cientistas nos seus processos de experimentação. São identificados requisitos relacionados com a falta de informação sobre ferramentas e componentes de *software* em múltiplos ambientes e repositórios, bem como abordagens na gestão de dados num contexto *Big Data*.

No ano de 2023 já na parte final de concretização deste TFM no artigo [31] são descritas novas funcionalidades do sistema de *workflow Tapis*, [32], que assentam essencialmente na introdução do paradigma *publish/subscribe* com o sistema *RabbitMQ*. utilizado para comunicação entre a API do sistema *Tapis* e uma nova API proposta no artigo.

Outro trabalho relevante é o projeto *Argo Workflows* [33] que implementa um *workflow engine* para o sistema *Kubernets* [34], isto é, um *workflow* é um *Directed Acyclic Graph* (DAG), definido como um recurso *Kubernets*, permitindo criar e visualizar a execução dos *workflows* num *web browser*. Os diferentes passos de um *workflow* DAG são executados como *Pods* (um *Pod* é um grupo de um ou mais *containers*). Um *workflow* é especificado através de uma linguagem de *script* (*YAML Ain't Markup Language* (YAML)) compatível com os *scripts* de execução de tarefas em *Kubernets*. Um *workflow* pode ser especificado com *loops* que é um mecanismo equivalente a executar o mesmo *workflow* em múltiplas iterações. Embora o projeto refira a utilização em múltiplos projetos de investigação, numa análise prévia da documentação disponível mostra que a utilização deste sistema requer profundos conhecimentos dos sistemas *Docker*, *Docker Compose* e

kubernets e suas múltiplas ferramentas, nomeadamente linguagens de *script* YAML e interpretadores de comandos de linha.

2.4 Resumo

Após a pesquisa e estudo de trabalhos relacionados, dos quais se destacam os WfMS que foram detalhados neste capítulo, podemos concluir que existem vários aspetos que justificam o trabalho de definir um novo modelo, definir uma arquitetura de suporte ao mesmo e realizar a implementação de um protótipo que permita executar aplicações distribuídas por decomposição em atividades usando o paradigma de *workflow*. Os principais aspetos a considerar na definição do movo modelo são assim resumidos:

- Definição de um *workflow* sobre uma linguagem de especificação mais recente e fácil de interpretar, como *JavaScript Object Notation* (JSON);
- Suporte ao desenvolvimento de *workflows* com tempo de execução longo e com múltiplas iterações;
- Permite a especificação de padrões de *workflow* básicos, e o padrão de *load balancing* suportado por replicação de tarefas;
- Controlo de execução de um *workflow* descentralizado;
- Utilização de um *Broker* de comunicação por mensagens que permita o fluxo de dados entre as tarefas do *workflow* que se executam em diferentes nós distribuídos utilizando o padrão *publish/subscribe*;
- Suporte à execução de tarefas, em ambientes virtuais como *containers*, de forma distribuída em nós de computação heterogéneos, nomeadamente máquinas virtuais (VM) em *Clouds*.

Como podemos concluir pela pesquisa realizada, a maior parte dos WfMS não oferece um controlo de execução de *workflows* descentralizado. Apesar do modelo AWARD oferecer um modelo de execução descentralizado não permite a execução de tarefas como sendo *containers* e apenas suporta a especificação de um *workflow* em XML. Além disso o AWARD utiliza o mecanismo *Tuple Space* de suporte à comunicação de *Activities* sem implementações em *middlewares* recentes.

No entanto este Trabalho Final de Mestrado (TFM) tem como forte inspiração o modelo AWARD, que oferece soluções para alguns problemas levantados sobre os WfMS

mais populares. Assim uma motivação do trabalho é propor melhorias e contributos que podem beneficiar uma nova implementação do modelo AWARD, tendo em consideração as novas e melhores tecnologias disponíveis hoje em dia, tais como, permitir a especificação de um *workflow* utilizando uma linguagem de especificação mais moderna como o JSON, *middlewares* seguindo o modelo *publish/subscribe* e o suporte à virtualização através de *containers*.

3

Proposta de um Modelo de *Workflow*

Neste capítulo apresenta-se a proposta de um modelo de *workflow* de acordo com os objetivos definidos no Capítulo 3 e das conclusões retiradas no estudo de trabalhos relacionados apresentado no Capítulo 2. Assim nas secções seguintes serão abordados os seguintes tópicos:

- Definição de termos do modelo de suporte ao desenvolvimento e execução de *workflows*;
- *Workflows* com balanceamento de carga suportado por réplicas de atividades que passaremos a designar por *Activities*;
- Modelo e execução de uma *Activity* de *workflow*, como uma componente computacional autónoma;
- Contrato genérico de uma tarefa, que passaremos a designar por *Task*, no contexto de uma *Activity* de um *workflow*;
- Linguagem de especificação de *workflows*;
- Especificação dos padrões de *workflow* suportados pelo modelo;
- Conclusões sobre o modelo proposto.

3.1 Modelo de Execução de um *Workflow*

Um problema complexo pode ser decomposto em múltiplas *Activities* que se executam em sequência e ou em paralelo num ambiente distribuído. Assim, desenvolver uma aplicação distribuída para solucionar um problema pode ser realizado através da composição de *Activities* recorrendo ao paradigma de *workflow*, isto é, através de um grafo.

Por exemplo, na Figura 3.1 apresenta-se um caso de uma aplicação distribuída decomposta como um grafo com 3 vértices. No modelo aqui proposto, os vértices de um grafo serão designados como *Activities* e os arcos são suportados por um mecanismo de comunicação, entre essas *Activities*, designado por *Channel*.

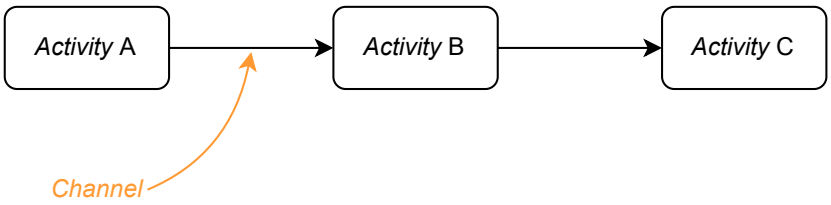


Figura 3.1: *Workflow* de *Activities*

Certos casos de uso exigem a execução de *workflows* de forma consecutiva (múltiplas iterações), como apresentado na Figura 3.2. Tal como foi analisado no Capítulo 2, o suporte ao desenvolvimento e execução de *workflows* com múltiplas iterações é uma das limitações/possibilidades de melhoria identificadas em alguns dos trabalhos relacionados que foram analisados. Sendo assim, o modelo proposto neste Trabalho Final de Mestrado (TFM) suporta esta característica. Um exemplo de um caso aplicacional que beneficia da modulação como um *workflow*, com múltiplas iterações, é uma aplicação complexa de processamento de ficheiros, onde em cada iteração é processado um ficheiro diferente.

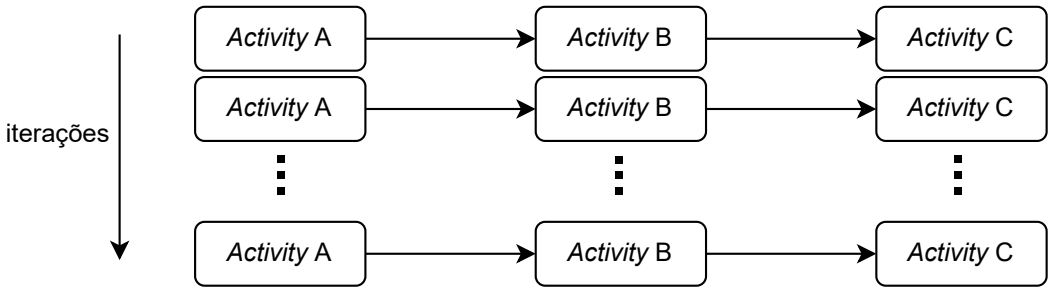


Figura 3.2: *Workflow* com múltiplas iterações

De forma a possibilitar o mecanismo de comunicação entre *Activities*, um *Channel* permite receber e enviar dados entre *Activities*. Cada *Activity* expõe *Ports* de comunicação com o exterior, podendo ter múltiplos *Ports* de entrada (*Input Ports*) e múltiplos *Ports* de saída (*Output Ports*), isto é, cada arco do grafo (*Channel*) é uma associação concetual entre *Ports*, representada por um par (*Output Port*, *Input Port*).

Os dados produzidos e recebidos nos *Ports* são designados por *Token*. Um *Token* é um objeto que contém dados compostos por um ou mais campos, num formato de acordo com as especificidades da aplicação modelada como *workflow* e informações de controlo, por exemplo, a iteração do *workflow* quando foi produzido o *Token*.

Uma *Activity*, tal como demonstrado na Figura 3.3, é composta pela tarefa (*Task*) a ser executada pela *Activity*, por *Input Ports* e *Output Ports* e por componentes de mapeamento entre esses *Ports* e a *Task*. Os *Tokens* recebidos nos *Input Ports* são mapeados para os argumentos da *Task* e os resultados produzidos pela *Task* serão mapeados para *Tokens* e enviados para os *Output Ports* da *Activity*.

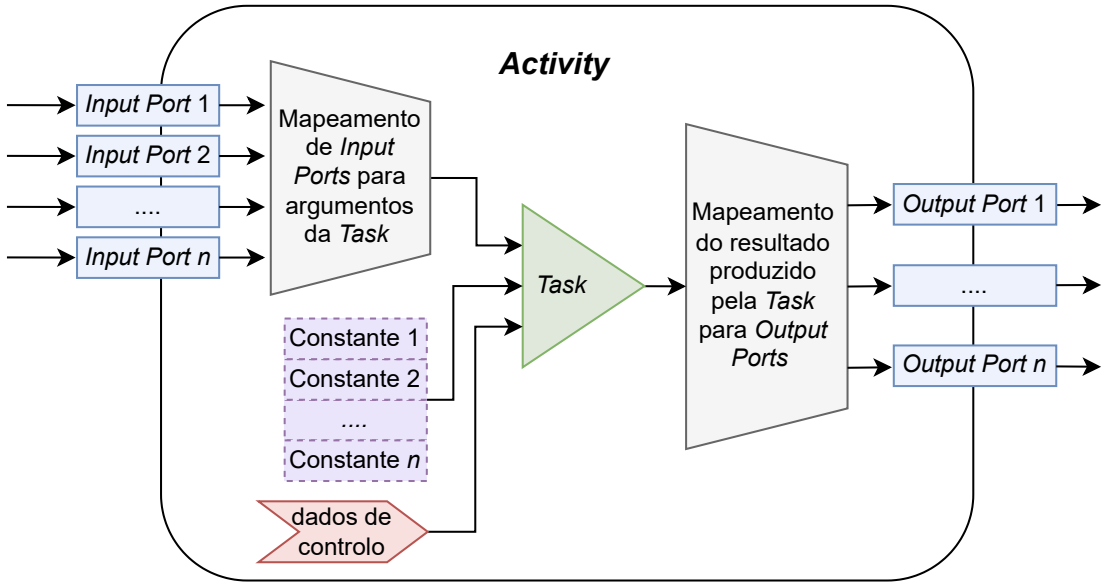


Figura 3.3: Componentes internas de uma *Activity*

Para além dos *Tokens* recebidos e mapeados para os argumentos da *Task*, esta pode também receber valores estáticos constantes e dados de controlo. Os valores constantes são definidos na especificação da *Activity*, e tal como os *Tokens*, são objetos que contém dados num formato de acordo com as especificidades da aplicação modelada como *workflow*. Os dados de controlo contêm informação relativa à execução da *Activity* que podem ser úteis à execução da *Task*, como por exemplo, o número de iteração corrente no momento de ativação da *Task*

Os algoritmos executados pelas *Tasks* estão comprometidos com o formato dos dados

dos *Tokens* e das constantes especificadas, isto é, o modelo é transparente ao conteúdo e formato dos *Tokens* e constantes.

O número de *Ports* de uma *Activity*, tanto de *Input* como de *Output*, é configurável de forma a acomodar os requisitos do caso aplicacional modelado segundo o novo modelo de *workflow* proposto.

O facto de o modelo permitir executar múltiplas *Activities* num ambiente computacional distribuído introduz características importantes nesta definição:

i) O controlo da execução das *Activities* é efetuado de forma descentralizada, pelo que cada *Activity* é autónoma no sentido em que deve ser capaz de ser iniciada, de se executar e terminar de forma independente das outras *Activities*. A execução de iterações por parte de cada *Activity* também é autónoma e independente, tal como representado na Figura 3.4. Uma *Activity* é capaz de executar *n* iterações, independentemente da iteração corrente das outras *Activities*, desde que se cumpram as condições de execução a cada iteração, como a chegada de *Tokens* a todos os *Input Ports*. Por exemplo, uma *Activity* de inicialização do *workflow* que tenha um tempo de execução reduzido, pode executar *n* iterações enquanto outra *Activity* subsequente, com um tempo de execução elevado, executa apenas a primeira iteração;

ii) A comunicação entre *Activities* é feita, através de *Channels*, recorrendo à intermediação de um sistema (*Broker*) baseado no modelo *publish/subscribe*. Assim, os *Input Ports* das *Activities* são registados no intermediário como subscritores de um *Channel* e os eventos com origem em *Output Ports* serão publicados em *Channels* no intermediário, onde irão permanecer até serem consumidos pelos subscritores do *Channel*, tal como está representado na Figura 3.5, onde no *workflow* com múltiplas iterações a *Activity A* publicou 2 *Tokens* e em paralelo a *Activity B* publicou 1 *Token*;

iii) Uma *Activity* pode não ter *Input Ports*, por exemplo uma *Activity* de inicialização do *workflow*, significando que é responsabilidade da *Task* aceder a eventuais dados num repositório, por exemplo ficheiros cujos nomes são especificados como constantes de acordo com a aplicação concreta a desenvolver. Por exemplo no *workflow* da Figura 3.5 a *Activity A* não tem *Input Ports*;

iv) Uma *Activity* pode também não ter *Output Ports*, por exemplo uma *Activity* final do *workflow*, como a *Activity C* na Figura 3.5, significando que a *Task* da *Activity* produzirá eventualmente dados em repositórios ou ficheiros de acordo com a aplicação concreta a desenvolver.

A utilização de um mecanismo de intermediação baseado no modelo *publish/subscribe* (*Broker*) permite o desacoplamento entre *Activities* e que cada *Activity* produza ou consuma *Tokens* autonomamente em ritmos diferenciados.

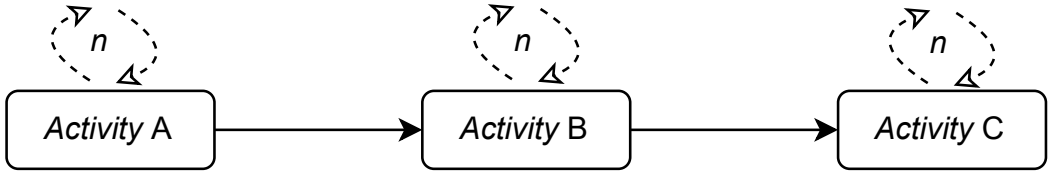


Figura 3.4: Cada *Activity* executa *n* iterações de forma autónoma

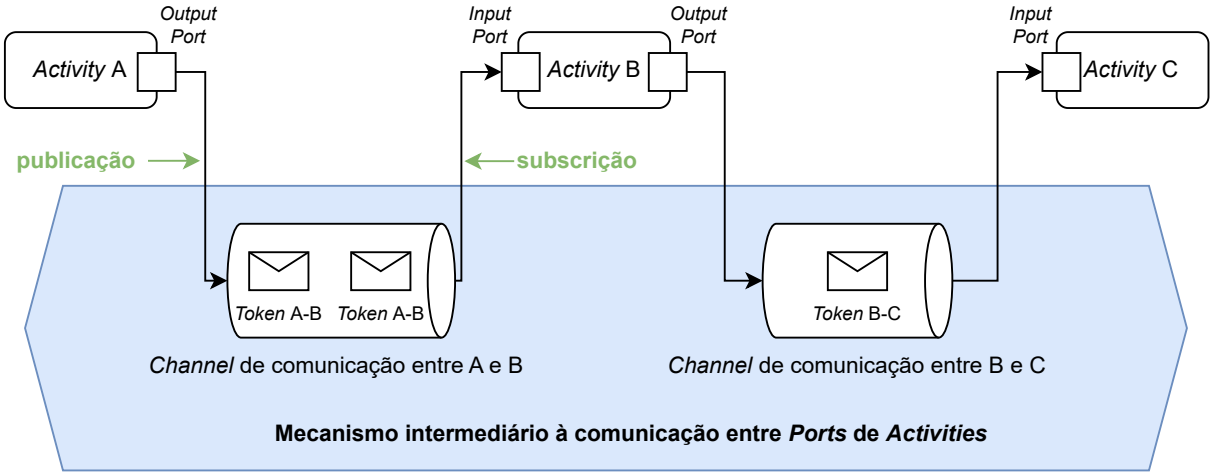


Figura 3.5: Exemplo de intermediação (*Broker*) na comunicação entre *Activities* de um *workflow*

Com o crescimento atual da dimensão dos dados a serem processados pelos *workflows* é ineficiente considerar que esses dados, entre *Activities*, são enviados como conteúdo dos *Tokens*. De facto, tipicamente os *Message Oriented Middleware* (MOM) têm limitações quanto à dimensão das mensagens. Assim, é proposto que os dados de grande dimensão sejam partilhados entre *Activities*, através de ficheiros, usando um sistema de ficheiros distribuídos, acessível em toda a infraestrutura computacional existente para a execução de *workflows*. Nesse caso os *Tokens* transportam o nome dos ficheiros existentes a partir de um *pathname* de uma diretoria base associada a cada *workflow* (*workingDirectory*).

3.2 Réplicas e Balanceamento de Carga

O desenvolvimento e execução de *workflows* com múltiplas iterações, segundo o modelo proposto, aumenta significativamente os casos de uso suportados relativamente

a *workflows* do tipo *Directed Acyclic Graph* (DAG). No entanto, o tempo de execução de um *workflow* com múltiplas iterações pode ser bastante elevado, especialmente no caso de serem utilizadas *Tasks* que impliquem longos tempos de execução. Nestes casos, será vantajoso especificar réplicas para as *Activities* que tenham tempos de execução longos, conforme está representado na Figura 3.6 com k réplicas na *Activity B*. Neste exemplo cada réplica executa n/k iterações.

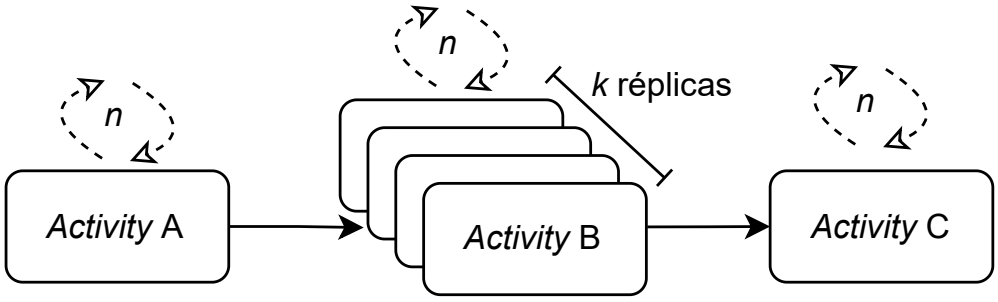


Figura 3.6: *Workflow* configurado com n iterações e com k réplicas na *Activity B*

O modelo proposto suporta a especificação de um número de réplicas menor ou igual ao número de iterações definidas para todo o *workflow*, possibilitando que a execução do *workflow* tire partido da distribuição dos *Tokens* (*load balancing*) pelas diferentes réplicas de uma *Activity* ao longo das iterações.

As réplicas de uma *Activity* são criadas com os mesmos *Ports* e consequentemente os mesmos *Channels*, pelo que todas recebem os mesmos *Tokens*. No entanto, o balanceamento de carga é conseguido, deixando à responsabilidade de cada *Activity* o processamento de *Tokens* marcados com a iteração igual à iteração corrente da sua execução. Ou seja, uma *Activity* que esteja na terceira iteração, apenas aceita *Tokens* marcados com essa iteração. Os restantes *Tokens* de outras iterações serão ignorados até ao seu eventual processamento nessas iterações.

A Figura 3.7 apresenta um *workflow* com 4 iterações onde é possível observar os *Tokens* criados pela *Activity A* serem distribuídos equitativamente pelas 2 réplicas da *Activity B*, isto é, cada réplica processa 2 *Tokens*. Assim, os *Tokens* das iterações 2 e 3 ainda no *Channel*, serão processados pelas réplicas 1 e 2 da *Activity B* respetivamente.

No modelo proposto assume-se uma política rígida de distribuição de *Tokens* pelas múltiplas réplicas como se descreve a seguir.

Cada *Activity* recebe no início da sua execução, uma estratégia de iteração com os seguintes valores: i) o número de iteração inicial; ii) o valor a incrementar à iteração corrente após cada iteração; iii) o número máximo de iterações do *workflow*.

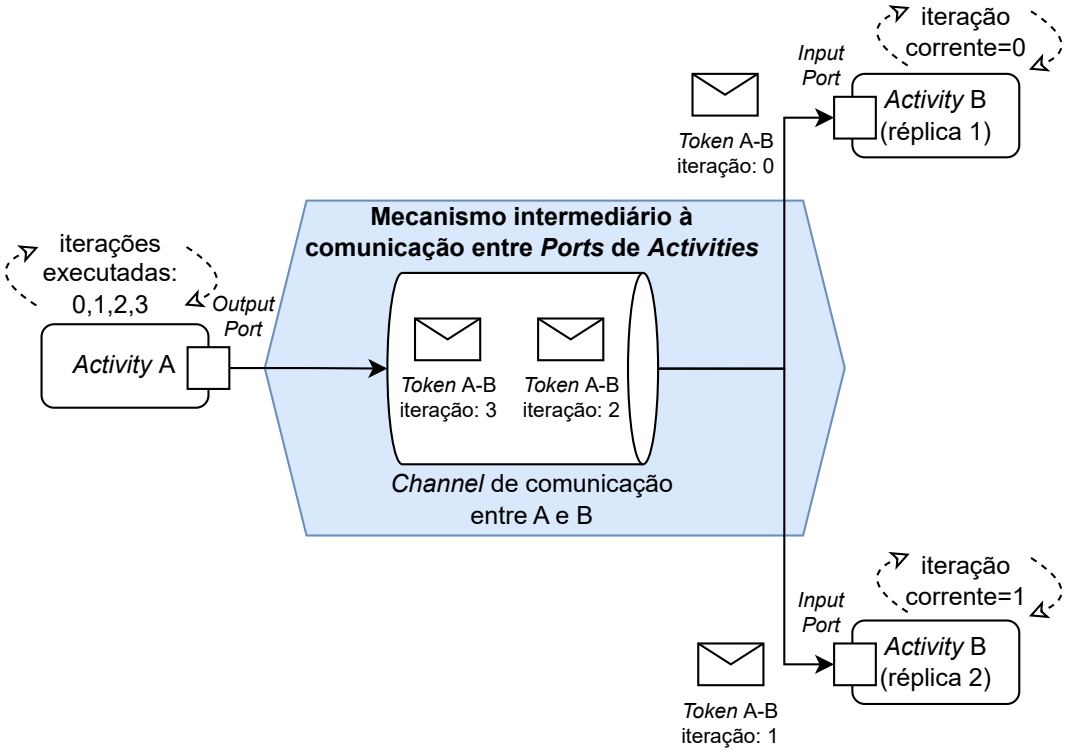


Figura 3.7: Exemplo de distribuição de *Tokens* por réplicas de uma *Activity*

Na Figura 3.8 apresenta-se um exemplo de *workflow* com um número máximo de iterações igual a 5, onde as *Activities A* e *C*, sem réplicas, executam essas 5 iterações com um valor inicial de iteração igual a 0 e um valor de incremento de iteração igual a 1. A réplica 1 da *Activity B* recebe um valor inicial de iteração igual a 0 e um valor de incremento de iteração igual a 2. A réplica 2 da *Activity B* recebe um valor inicial de iteração igual a 1 e um valor de incremento de iteração igual a 2. Assim, a réplica 1 da *Activity B* executa as iterações 0, 2 e 4 e a réplica 2 da *Activity B* as iterações 1 e 3.

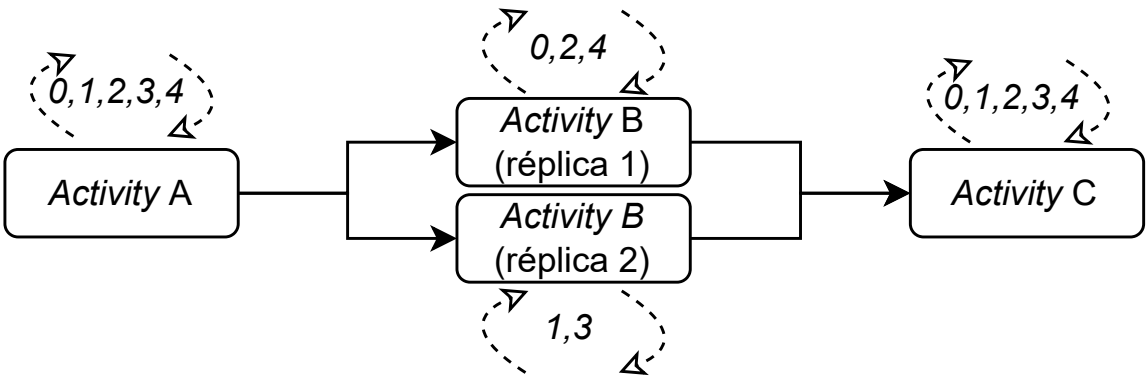


Figura 3.8: Exemplo de execução de 5 iterações com 2 réplicas da *Activity B*

3.3 Modelo de Execução de uma *Activity* Autónoma

O ciclo de vida de uma *Activity* tem as seguintes fases:

1. Inicia-se como uma execução autónoma;
2. Para cada iteração:
   2.1. Aguarda a chegada de *Tokens* nos seus *Input Ports*;
   2.2. Realiza o mapeamento desses *Tokens* para os argumentos da *Task*;
   2.3. Ativa a execução da *Task*;
   2.4. Após terminar a execução da *Task* é feito o mapeamento dos resultados para os *Output Ports*;
   2.5. Incrementa o número de iteração corrente;
   2.6. Se não atingiu a última iteração repete a fase 2;
   2.7. Se atingiu a última iteração termina a execução da *Activity*.

A passagem entre cada uma das fases do ciclo de vida é controlada por uma máquina de estados e são as interações entre os componentes e os acontecimentos despoletados em cada fase que vão acionar as transições de estado. As fases atrás enunciadas, são descritas com mais detalhe de seguida:

1. Inicia a máquina de estados de acordo com a especificação da *Activity*, criando os eventuais *Input Ports* e *Output Ports* associados aos *Channels* necessários. Inicia também o controlo de iterações de acordo com a estratégia definida para a *Activity*;
2. Em cada iteração executa as seguintes subfases:
   2.1. Aguardar a chegada de *Tokens* nos seus *Input Ports* consiste primeiramente na subscrição dos *Input Ports* aos *Channels* que lhes foram atribuídos. O seguimento para a próxima subfase apenas será desencadeado quando forem recebidos *Tokens* em todos os *Input Ports*;
   2.2. Os *Tokens* recebidos, marcados com o valor de iteração igual ao valor de iteração corrente da *Activity*, são mapeados para argumentos da *Task*. A informação relativa à posição do argumento para qual deverá ser mapeado o *Token* recebido num certo *Input Port* é definido na especificação da *Activity*. Após este mapeamento a *Activity* passará à próxima subfase;

2.3. De acordo com a especificação do *workflow*, é feita a ativação da *Task* a ser executada pela *Activity*. A *Task* é executada com os argumentos mapeados na subfase anterior, as constantes definidas na especificação da *Activity* e os dados de controlo. Esta execução irá produzir uma coleção de resultados passando à subfase seguinte;

2.4. O mapeamento da coleção de resultados da *Task* para os *Output Ports* é feito de acordo com a especificação da *Activity*, onde a cada resultado individual está associado um ou mais *Output Ports*. Esta associação é baseada na posição numérica que o resultado ocupa na coleção de resultados gerados na subfase anterior. Os resultados são mapeados em *Tokens* e enviados para os *Channels* associados aos *Ports*, passando à próxima subfase;

2.5. É incrementado o valor da iteração corrente de acordo com a estratégia definida na especificação da *Activity* e passa à próxima subfase;

2.6. Se o valor de iteração corrente da *Activity* for menor ou igual ao valor máximo de iterações do *workflow*, repete a fase 2;

2.7. Se o valor da iteração corrente for maior que o valor máximo de iterações do *workflow* termina a execução da *Activity* libertando todos os recursos alocados antes e durante a sua execução;

3.4 Contrato Genérico de uma *Task*

A *Task* é um componente o mais desacoplado possível do modelo de *workflow*, isto é, uma componente de *software* desenvolvida de acordo com a especificidade e domínio da aplicação modulada como *workflow*. A ativação de uma *Task* deve ser genérica, pelo que esta deve seguir um contrato padrão. Este contrato encontra-se representado na Figura 3.9, onde podemos observar, que uma *Task* recebe conjuntos de argumentos, constantes e dados de controlo, produzindo um conjunto de resultados. Os argumentos são o mapeamento resultante dos *Tokens* recebidos nos *Input Ports* de uma *Activity*, as constantes são valores definidos na especificação de uma *Activity*, e os dados de controlo remetem a informação de execução da *Activity* como o valor de iteração corrente no momento de ativação da *Task*, ou o nome identificador da *Activity* que a executa. Os resultados da execução da *Task* são transformados em *Tokens* e mapeados para os *Output Ports* correspondentes.

Desta forma, qualquer *Activity* consegue ativar a execução de uma *Task* apenas necessitando de providenciar-lhe os argumentos, as constantes e os dados de controlo, e no final recolher os resultados.

É de notar que uma *Task* durante a sua execução pode abstrair-se completamente da existência dos outros componentes do *workflow*, nomeadamente outras *Tasks*.

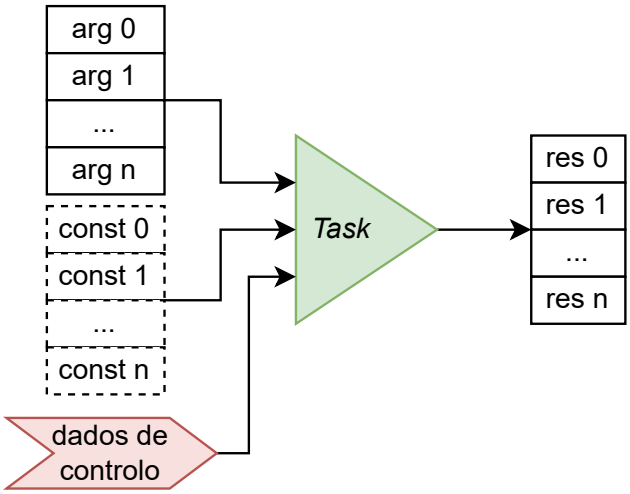


Figura 3.9: Contrato genérico de uma *Task*

3.5 Especificação do *Workflow*

O modelo proposto pressupõe uma especificação do *workflow* antes da sua execução, isto é, a definição de um nome único para o *workflow*, a atribuição do número de iterações e a especificação de cada uma das *Activities* existentes. De seguida apresenta-se a estrutura da especificação de *workflows*:

1. **Nome do *workflow***: nome único do *workflow*;
2. **Número de iterações**: valor inteiro de 0 a n, representando o número de iterações com que o *workflow* será executado. Caso o valor seja 0, o *workflow* é executado em permanente repetição (iterações infinitas);
3. **Diretoria base**: *pathname* de diretoria para partilha de ficheiros entre *Activities*;
4. ***Activities***: lista da especificação das *Activities* que constituem o *workflow*.

A especificação de cada *Activity* tem a seguinte estrutura:

1. **Nome da *Activity***: nome único identificador da *Activity* no contexto global do *workflow*;
2. **Número de réplicas**: número de réplicas a serem criadas para a *Activity*;

3. ***Task***: informação necessária à execução da *Task* que contém as seguintes definições:

   3.1. **Executável**: especificação da componente de *software* que implementa a *Task* a ser ativada para execução pela *Activity*;

   3.2. **Constantes**: conjunto de valores estáticos constantes, que são passados à *Task* no momento da sua ativação e que contêm dados num formato de acordo com as especificidades da aplicação modulada como *workflow*;

4. ***Ports***: especificação dos *Input Ports* e dos *Output Ports* da *Activity*. Cada *Port* tem as seguintes definições:

   4.1. **Nome**: nome único do *Port* no contexto global do *workflow*;

   4.2. **Tipo**: tipo de *Port* que assume os valores *IN* ou *OUT*, respetivamente se é um *Input Port* ou um *Output Port*;

   4.3. ***Channel***: Se o tipo do *Port* for *IN* indica o nome do *Port* origem, isto é, um *Output Port* de outra *Activity*. Se o tipo do *Port* for *OUT* indica o nome do *Port* destino, isto é, um *Input Port* de outra *Activity* para onde vai ser publicado o *Token* resultante da execução da *Task*;

5. Mapeamento de *Input Ports* para argumentos da *Task*, associando o nome dos *Input Ports* com a posição relativa (índice de 0 a *n*) dos argumentos da *Task*;

6. Mapeamento de resultados da *Task* para *Output Ports*, associando a posição relativa (índice de 0 a *n*) dos resultados da *Task* com o nome dos *Output Ports*.

É da responsabilidade de quem especifica o *workflow* fazer coincidir as especificações das *Activities* com as necessidades das *Task* associadas a cada *Activity*.

Como exemplo, na Tabela 3.1 apresenta-se a especificação do *workflow* representado na Figura 3.10. O *workflow* tem o nome *WF1* e apenas uma iteração. A *Activity* de nome *A* não tem *Input Ports*. No entanto, tem definido um valor constante "*Valor x*". A *Activity A* executa uma *Task* de nome *Task-A*, cujo resultado de execução (índice 0) é mapeado para o *Output Port* designado por *AOut*, publicando um *Token* para o *Input Port* indicado no *Channel* de nome *BIn*. A *Activity* de nome *B* tem configurado um *Input Port*, de nome *BIn*, que recebe um *Token* proveniente do *Channel* com origem do *Output Port* de nome *AOut*. A *Activity B* executa a *Task* de nome *Task-B* com o mapeamento de argumentos (índice 0) recebidos no *Channel* associado à *Input Port* de nome *BIn*.

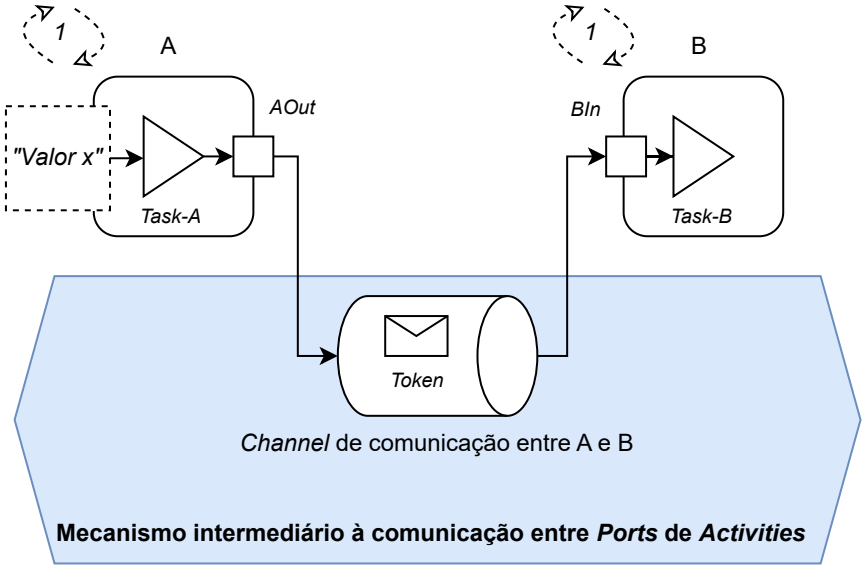


Figura 3.10: Exemplo de *workflow* constituído por duas *Activities*

Tabela 3.1: Especificação do *workflow* ilustrado na Figura 3.10

Workflow			
Nome do *workflow*	*WF1*		
Número de iterações	1		
Diretoria base			
Activities	**Nome da *Activity***	*A*	
	Número de réplicas	1	
	Task	**Executável**	*Task-A*
		Constantes	*"Valor x"*
	Ports	**Nome**	*AOut*
		Tipo	*OUT*
		Channel	*BIn*
	Mapeamento de argumentos		
	Mapeamento de resultados	*AOut*	0
	Nome da *Activity*	*B*	
	Número de réplicas	1	
	Task	**Executável**	*Task-B*
		Constantes	
	Ports	**Nome**	*BIn*
		Tipo	*IN*
		Channel	*AOut*
	Mapeamento de argumentos	*BIn*	0
	Mapeamento de resultados		

Na Figura 3.11, apresenta-se a especificação de uma *Activity* pertencente a um *workflow* de nome *WF2*, com três iterações, com duas constantes (valores 1 e 2), 3 *Input Ports* e 3 *Output Ports*, onde se pode verificar a flexibilidade dos mapeamentos desses *Ports* para

argumentos e resultados da *Task*. Os *Input Ports* podem ser mapeados para qualquer uma das posições dos argumentos da *Task*. Por exemplo, o primeiro *Input Port* de nome *AIn0* mapeia para o argumento na posição 3 (*arg 2*), da mesma forma que a terceira *Input Port* de nome *AIn2* mapeia para a segunda posição (*arg 1*). Os resultados da *Task* podem também ser mapeados para qualquer um dos *Output Ports*. Por exemplo, o segundo resultado (*res 1*) replicado para ser mapeado para dois *Output Ports* de nome *AOut0* e *AOut2*.

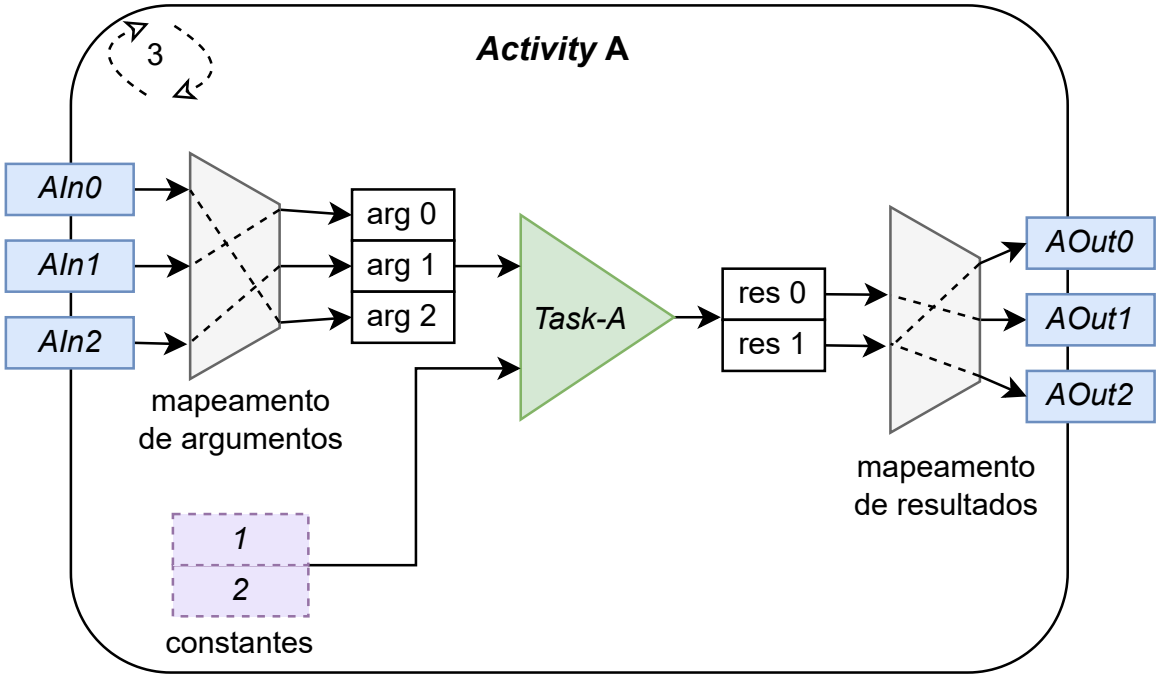


Figura 3.11: Exemplo detalhado do fluxo de informação dentro de uma *Activity*

Na Tabela 3.2 apresenta-se a especificação da *Activity* representada na Figura 3.11

Tabela 3.2: Especificação da *Activity* ilustrada na Figura 3.11

Workflow			
Nome do *workflow*	*WF2*		
Número de iterações	3		
Diretoria base			
Activities	**Nome da *Activity***	*A*	
	Número de réplicas	1	
	Task	**Executável**	*Task-A*
		Constantes	*1*
			2

(continua)

Tabela 3.2: Especificação da *Activity* ilustrada na Figura 3.11 (continuação)

Activities	***Ports***	**Nome**	*AIn0*
		Tipo	*IN*
		Channel	*SomeOutPort0*
		Nome	*AIn1*
		Tipo	*IN*
		Channel	*SomeOutPort1*
		Nome	*AIn2*
		Tipo	*IN*
		Channel	*SomeOutPort2*
		Nome	*AOut0*
		Tipo	*OUT*
		Channel	*SomeInPort0*
		Nome	*AOut1*
		Tipo	*OUT*
		Channel	*SomeInPort1*
		Nome	*AOut2*
		Tipo	*OUT*
		Channel	*SomeInPort2*
	Mapeamento de argumentos	*AIn0*	2
		AIn1	0
		AIn2	1
	Mapeamento de resultados	*AOut0*	1
		AOut1	0
		AOut2	1
	...		

Tendo em conta as regras de especificação, é possível modular aplicações concretas utilizando o modelo de *workflow* proposto. Como exemplo, consideremos a realização de cálculos da equação apresentada em Expressão (3.1), onde o valor de K é constante e o valor de x corresponde a um número aleatório maior que 0.

$$\frac{K \times x}{K + x} \tag{3.1}$$

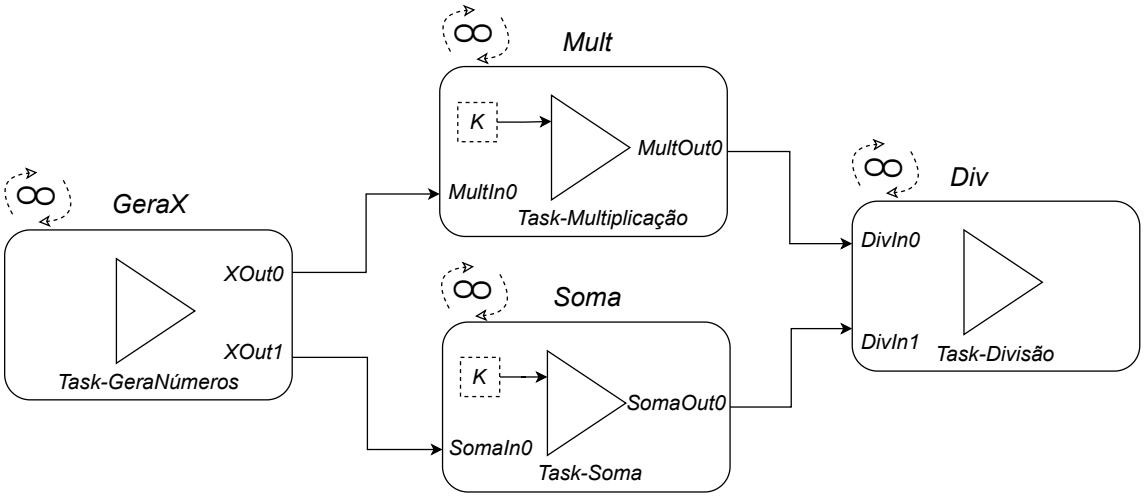


Figura 3.12: Expressão (3.1) mapeada sob o paradigma de *workflow*

Dado que pretendemos calcular a expressão com um indeterminado número de valores de x, então o *workflow* apresentado na Figura 3.12 tem um número de iterações infinitas. A *Activity GeraX*, contém a *Task* de nome *Task-GeraNúmeros*, cujo resultado, em cada iteração, é um número inteiro aleatório (valor de x), que através das suas *Output Ports* publicará *Tokens* nos respetivos *Channels* associados a *Input Ports* de outras *Activities* (*Activity Mult* e *Activity Soma*). Para realizar as operações aritméticas necessárias, são utilizadas as *Tasks* de nomes *Task-Multiplicação*, *Task-Soma* e *Task-Divisão*, utilizadas respetivamente nas *Activities Mult*, *Soma* e *Div*.

Na Tabela 3.3 apresenta-se a especificação completa para o *workflow* representado na Figura 3.12.

Tabela 3.3: Especificação do *workflow* ilustrada na Figura 3.12

Workflow			
Nome do *workflow*	*WF3*		
Número de iterações	0		
Diretoria base			
Activities	**Nome da *Activity***	*GeraX*	
	Número de réplicas	1	
	Task	**Executável**	*Task-GeraNúmeros*
		Constantes	
	Ports	**Nome**	*XOut0*
		Tipo	*OUT*
		Channel	*MultIn0*

(continua)

Tabela 3.3: Especificação do *workflow* ilustrada na Figura 3.12 (continuação)

Activities	***Ports***	**Nome**	*XOut1*
		Tipo	*OUT*
		Channel	*SomaIn0*
	Mapeamento de argumentos		
	Mapeamento de resultados	*XOut0*	0
		XOut1	1
	Nome da *Activity*	*Mult*	
	Número de réplicas	1	
	Task	**Executável**	*Task-Multiplicação*
		Constantes	*K*
	Ports	**Nome**	*MultIn0*
		Tipo	*IN*
		Channel	*XOut0*
		Nome	*MultOut0*
		Tipo	*OUT*
		Channel	*DivIn0*
	Mapeamento de argumentos	*MultIn0*	0
	Mapeamento de resultados	*MultOut0*	0
	Nome da *Activity*	*Soma*	
	Número de réplicas	1	
	Task	**Executável**	*Task-Soma*
		Constantes	*K*
	Ports	**Nome**	*SomaIn0*
		Tipo	*IN*
		Channel	*XOut1*
		Nome	*SomaOut0*
		Tipo	*OUT*
		Channel	*DivIn1*
	Mapeamento de argumentos	*SomaIn0*	0
	Mapeamento de resultados	*SomaOut0*	0

(continua)

Tabela 3.3: Especificação do *workflow* ilustrada na Figura 3.12 (continuação)

Activities	**Nome da *Activity***	*Div*	
	Número de réplicas	1	
	Task	**Executável**	*Task-Divisão*
		Constantes	
	Ports	**Nome**	*DivIn0*
		Tipo	*IN*
		Channel	*MultOut0*
		Nome	*DivIn1*
		Tipo	*IN*
		Channel	*SomaOut0*
	Mapeamento de argumentos	*MultIn0*	0
	Mapeamento de resultados	*MultOut0*	0

3.6 Padrões de *Workflow*

De forma a possibilitar a modulação de um problema sob o paradigma de *workflow*, o modelo proposto suporta os principais padrões de *workflow* básicos [4] tais como o padrão *Sequence*, o padrão *Synchronization* e o padrão *Parallel Split*. Todos estes padrões são utilizados no exemplo da Figura 3.12 como apresentado na Expressão (3.2).

$$\underline{Sequence}\ \{GeraX, \underline{Paralel\ Split}\ \{Mult, Soma\}, \underline{Synchronization}\ Div\ \} \tag{3.2}$$

3.7 Resumo

Refletindo sobre os objetivos definidos no Capítulo 1 e salientando a listagem das características designadas como importantes para um modelo de apoio à modulação e execução de um *workflow*, podemos tirar as seguintes conclusões sobre o modelo proposto:

- O modelo não limita a execução de *Activities* a nenhum ambiente computacional específico, podendo estas serem executadas em qualquer tipo de infraestrutura distribuída baseada em tecnologias de virtualização, tais como máquinas virtuais (VM) e *containers*, permitindo que as *Activities* de um *workflow* se executem em

múltiplos nós de computação heterogéneos em termos de capacidades de processamento e de memória;

- O modelo garante um controlo de execução de *workflows* descentralizado. Este objetivo é cumprido através da definição do controlo de uma *Activity* como sendo autónomo, permitindo às *Activities* executarem-se automaticamente quando receberem *Tokens* em todos os seus *Input Ports*;
- O fluxo de dados entre as *Activities* de um *workflow* é suportado através de um mecanismo intermediário de comunicação (*Broker*), seguindo o modelo *publish/-subscribe*;
- O modelo permite a especificação de *workflows* com padrões de *workflow* básicos;
- O modelo permite a execução de *workflows* com múltiplas ou infinitas iterações;
- O modelo permite a utilização de réplicas de *Activities* de longos tempos de execução, suportando assim *load balancing* entre essas réplicas ao longo das sucessivas iterações;
- A especificação de *workflows* e o desenvolvimento aberto de *Tasks* é desacoplado de aplicações concretas de um determinado domínio das ciências ou de linguagens de programação, permitindo ao programador focar-se na implementação dos algoritmos de cada *Task* associada a cada *Activity*, sem ter de conhecer outros detalhes do modelo de *workflow*;
- A especificação de um *workflow* segundo o modelo é facilmente concretizável através de linguagens de representação de dados, tais como *Extensible Markup Language* (XML), *YAML Ain't Markup Language* (YAML) ou *JavaScript Object Notation* (JSON).

4

Arquitetura de Suporte à Implementação do Modelo

Neste capítulo apresenta-se a arquitetura de suporte à implementação do modelo proposto no Capítulo 3 e a descrição dos aspetos de interação entre as componentes envolvidas, servindo de referência à implementação de um protótipo que permita a experimentação com a execução de *workflows*. Assim, nas secções seguintes serão abordados os seguintes tópicos:

- Infraestrutura computacional utilizada no suporte à execução de *workflows*;
- Mecanismo de intermediação (*Broker*) de comunicação entre *Activities* durante a execução de *workflows*;
- Interlocutor (*Activity Bucket*) de apoio ao lançamento e monitorização de *Activities* em execução num nó computacional;
- Pressupostos de execução da *Task* de uma *Activity*;
- Controlo de execução de uma *Activity* (*Activity Engine*);
- Ambiente de execução de uma *Activity* (*container*);
- Lançamento para execução de um *workflow* (*WorkflowLauncher*).

4.1 Infraestrutura Computacional

De forma a suportar a execução de *workflows* num ambiente distribuído, considerou-se a infraestrutura de um nó de computação como sendo uma máquina virtual (VM) disponibilizada numa plataforma de *Cloud* seguindo o modelo *Infrastructure as a Service* (IaaS).

Conforme o modelo proposto no Capítulo 3, a execução de um *workflow* em múltiplos nós computacionais (*cluster*) implica colocar em execução em cada nó um processo de apoio ao lançamento e monitorização de execução das *Activities* (*Activity Bucket*). As múltiplas *Activities* existentes no *workflow* são distribuídas pelos vários nós. O intermediário de troca de *Tokens* (*Broker*) deve ser colocado em execução num dos nós. Na Figura 4.1, apresenta-se um exemplo de um único nó computacional utilizado para executar o *Broker*, as *Activities* de um *workflow* e o *Activity Bucket*.

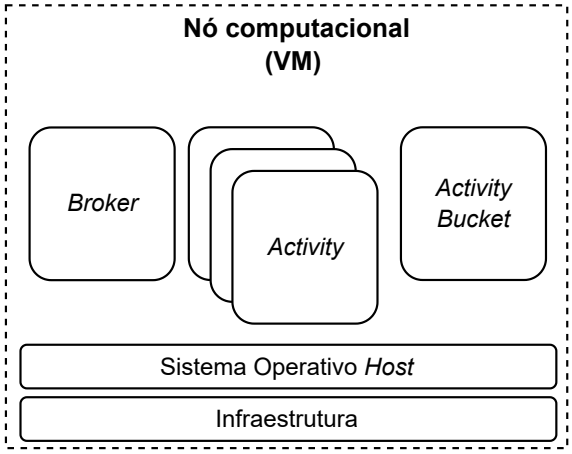


Figura 4.1: Arquitetura de um nó computacional

Na Figura 4.2 apresenta-se um requisito importante da infraestrutura dos nós computacionais, isto é, a existência de um sistema de ficheiros distribuídos de forma a possibilitar a partilha de dados entre os nós computacionais e consequentemente entre as *Activities* de um *workflow*. Assim em cada nó computacional existe um ponto de acesso ao sistema de ficheiros distribuídos como uma diretoria designada *sharedDirectory* que tem replicados os dados que se pretendem partilhar. Por exemplo, um caso aplicacional onde a utilização do sistema de ficheiros distribuídos se torna mais eficiente é quando um *workflow* necessita realizar o processamento de ficheiros de grande dimensão e as suas *Activities* se executam em diferentes nós.

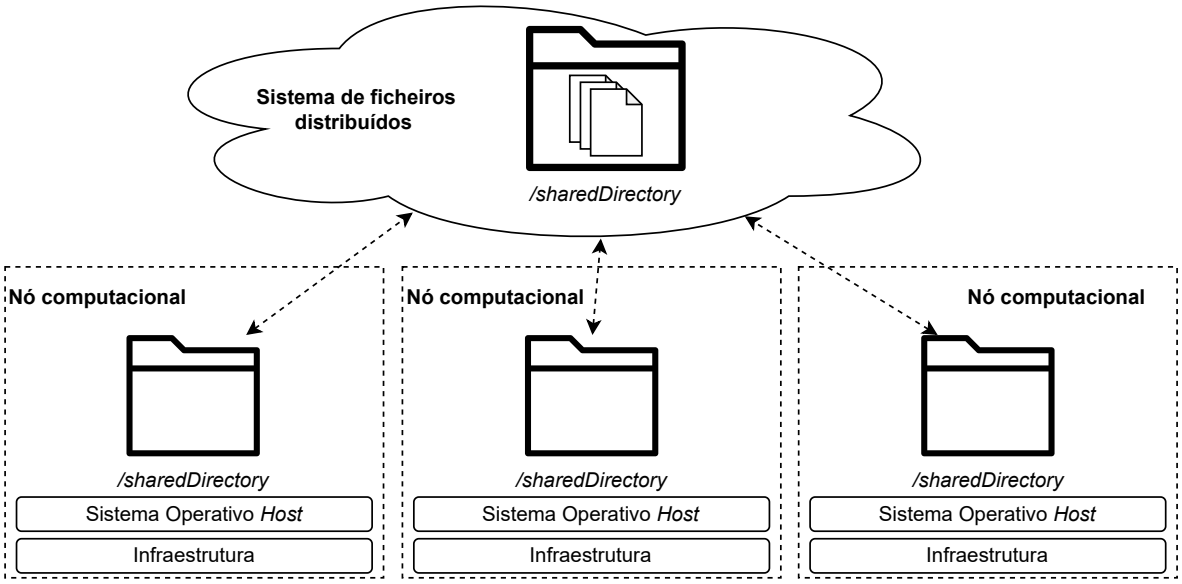


Figura 4.2: Sistema de ficheiros distribuídos

4.2 Mecanismo de Comunicação entre *Activities*

O fluxo de *Tokens* entre as *Activities* de um *workflow* é descrito no modelo (Capítulo 3) como sendo suportado por um *Broker*, seguindo um modelo de comunicação *publish/-subscribe* suportado por *queues*. Embora o *Broker* possa executar-se no mesmo nó computacional das *Activities*, como apresentado na Figura 4.1, é recomendável por questões de desempenho que se execute num nó computacional dedicado, acessível nos outros nós onde se executam as *Activities* do *workflow*.

Na Figura 4.3, está ilustrado um *workflow* composto pelas *Activities A*, *B* e *C*, seguindo o padrão *Sequence* pela ordem de execução *A*, *B* e *C*, com o *Channel* entre *A* e *B* e o *Channel* entre *B* e *C* suportados por um *Broker*. Como podemos observar na figura, a comunicação entre as *Activities* é realizada através de *queues* no *Broker*. Estas *queues* correspondem à concretização do conceito *Channel* entre um *Output Port* e um *Input Port* como definido no modelo proposto no Capítulo 3.

De acordo com a especificação do *workflow* e para suportar a sua respetiva execução, o *Message Broker* deve expor operações que permitam: i) criação de *queues*; ii) subscrição de *queues* para consumo de *Tokens*; iii) publicação de *Tokens* em *queues*; iv) eliminação de *queues*.

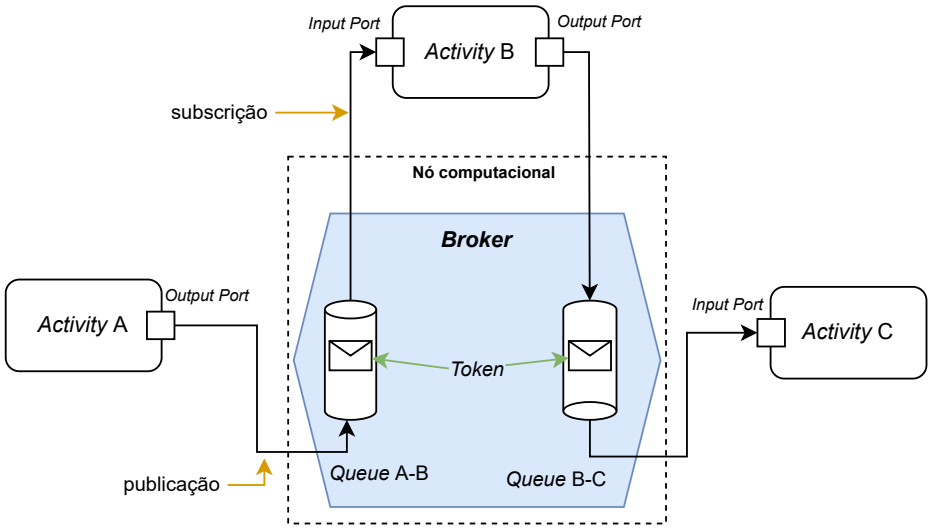


Figura 4.3: Comunicação entre *Activities* via *Broker*

4.3 *Activity Bucket*

A partir da especificação do *workflow* e em função do número de nós computacionais disponíveis, as *Activities* para execução são distribuídas pelos diferentes nós computacionais, podendo várias *Activities* serem executadas no mesmo nó computacional. Isto requer que exista em cada nó computacional um interlocutor designado *Activity Bucket* capaz de lançar, eliminar e disponibilizar o estado de execução das *Activities*, como ilustrado na Figura 4.4, onde se ilustra a execução do *workflow* da Figura 4.3, com três *Activities*, a serem supervisionadas por dois *Activity Buckets*. Neste caso, as *Activities A* e *B* encontram-se em execução sobre o mesmo nó computacional e a *Activity C* noutro nó computacional. Esta flexibilidade é útil, por exemplo, se o nó que contém as duas *Activities*, tiver maiores recursos computacionais, nomeadamente capacidade de processamento e memória. Neste caso, o *Broker* ocupa um terceiro nó computacional dedicado.

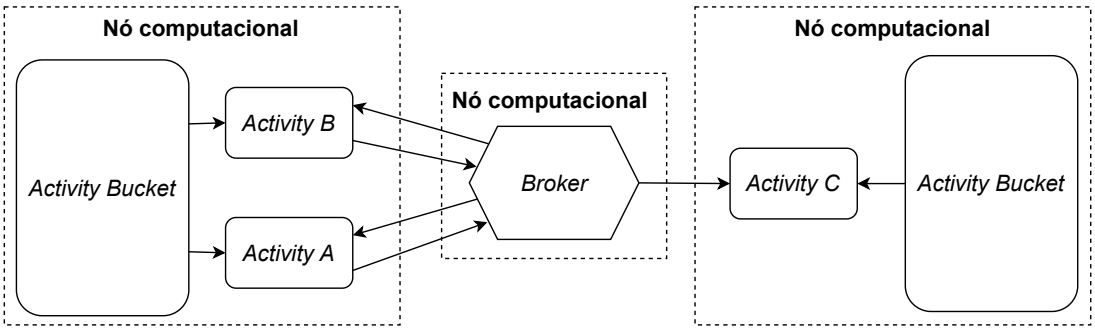


Figura 4.4: *Activity Bucket* no suporte à execução de *Activities* num nó computacional

Na Figura 4.5, apresenta-se um nó computacional com a execução de um *Activity Bucket* e duas *Activities*. O *Activity Bucket* é uma aplicação que se executa como um processo em *background* (*daemon*) do sistema operativo. Uma *Activity* consiste num *container* com um sistema operativo *Guest* onde se executa o *Activity Engine*, responsável pelo controle de execução da *Activity* nomeadamente a ativação da respetiva *Task*.

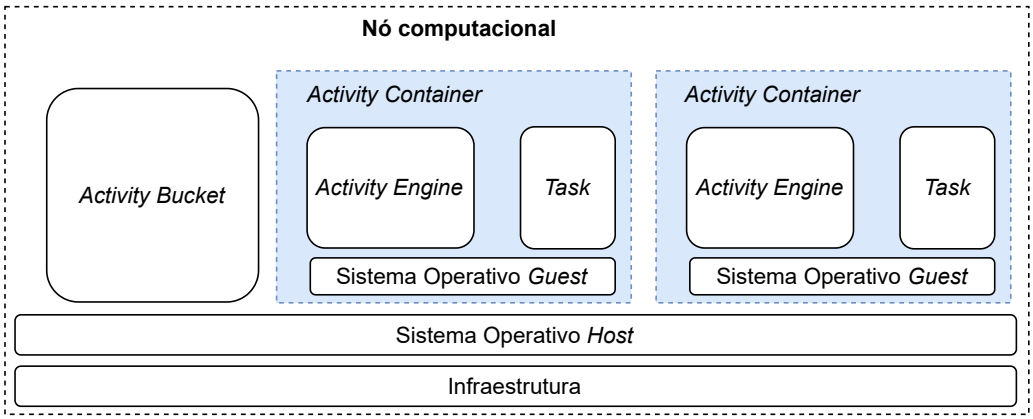


Figura 4.5: Arquitetura de execução de um nó computacional de uma *Activity*

Como já mencionado, certos *workflows* podem requerer o envio de dados de grande dimensão, entre *Activities*. Se estes dados fossem enviados através de *Tokens*, iriam sobrecarregar o processamento no *Broker*, podendo inclusive limitar a dimensão dos dados, pois tipicamente os *Message Oriented Middleware* (MOM) têm limitações quanto à dimensão das mensagens. Assim, quando existem dados de grande dimensão devem ser colocados em ficheiros utilizando o sistema de ficheiros distribuídos entre os nós computacionais, passando unicamente como conteúdo dos *Tokens* os nomes desses ficheiros.

Como já ilustrado na Figura 4.2 e com mais detalhe na Figura 4.6, para que as *Activities* tenham acesso ao sistema de ficheiros distribuídos, no momento de criação de uma *Activity*, o *Activity Bucket* do nó computacional, cria um mecanismo de partilha de ficheiros (*shared volume*) entre o sistema de ficheiros do sistema operativo *Host* e o sistema de ficheiros do sistema operativo *Guest* do *container* onde se executa a *Activity*. Este *shared volume* é criado sempre entre a diretoria *workingDirectory* do sistema operativo *Guest* e uma subdiretoria específica ao *workflow* a partir da *sharedDirectory* do sistema operativo *Host*, já partilhada no sistema de ficheiros distribuídos. De forma a separar a partilha de ficheiros entre *Activities*, por cada distinto *workflow* é configurado na sua especificação uma subdiretoria a partir da *sharedDirectory*. Os *Activity Bucket* utilizam o *shared volume* associado à diretoria *workingDirectory* do sistema operativo *Guest* para transferir ficheiros de e para o *container* que suporta a execução da *Activity*.

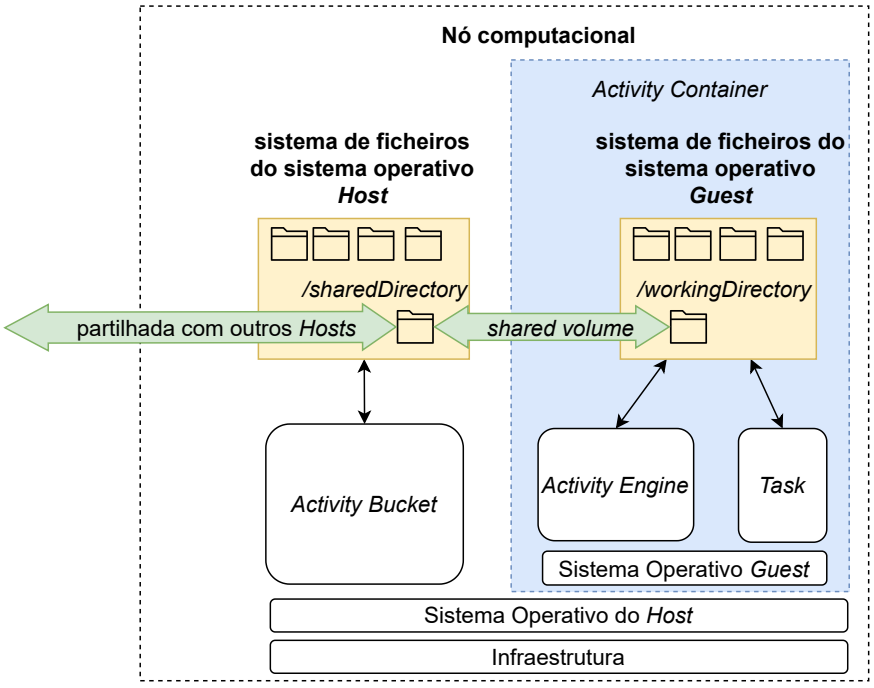


Figura 4.6: Partilha de dados entre o nó computacional e um *container*

Como exemplo de uma aplicação, que necessita partilhar ficheiros para o seu funcionamento, consideremos um *workflow* onde uma *Task* aplica um algoritmo de processamento de um ficheiro cujo conteúdo é uma fotografia e que resultou do processamento da *Task* de uma *Activity* anterior. Para a *Task* processar o ficheiro, este deve ser acessível dentro do *container*, na *workingDirectory* associada ao *shared volume* da *sharedDirectory* do sistema operativo *Host*. A *Task* poderá produzir um outro ficheiro na *workingDirectory* que ao estar mapeada para a *sharedDirectory*, tornará o novo ficheiro acessível em qualquer outra *Activity* em execução em qualquer outro nó computacional.

Um *Activity Bucket* ao lançar uma *Activity* fornece também informação relativa à infraestrutura de onde se executa o *workflow*, nomeadamente a informação relativa à localização do *Broker*. Por isso no momento de inicialização da execução de um *Activity Bucket*, este necessita dos seguintes argumentos: i) o *path* para a base da diretoria partilhada (*sharedDirectory*); ii) o IP do nó computacional e o porto TCP/IP de acesso ao *Broker*.

Para supervisionar a execução de *Activities* num nó computacional, o *Activity Bucket* expõe um contrato com uma especificação aberta para facilitar o desenvolvimento de ferramentas de criação, execução e monitorização de *workflows*. Assim propõe-se uma REST API (*Representational State Transfer* [35], *Application Programming Interface*) baseada no protocolo *Hypertext Transfer Protocol* (HTTP) [36]. Como representado na Figura 4.7, a REST API deverá ter as operações de lançar uma *Activity*, consultar o estado

de execução de *Activities* de um *workflow* e de eliminar as *Activities* de um *workflow*.

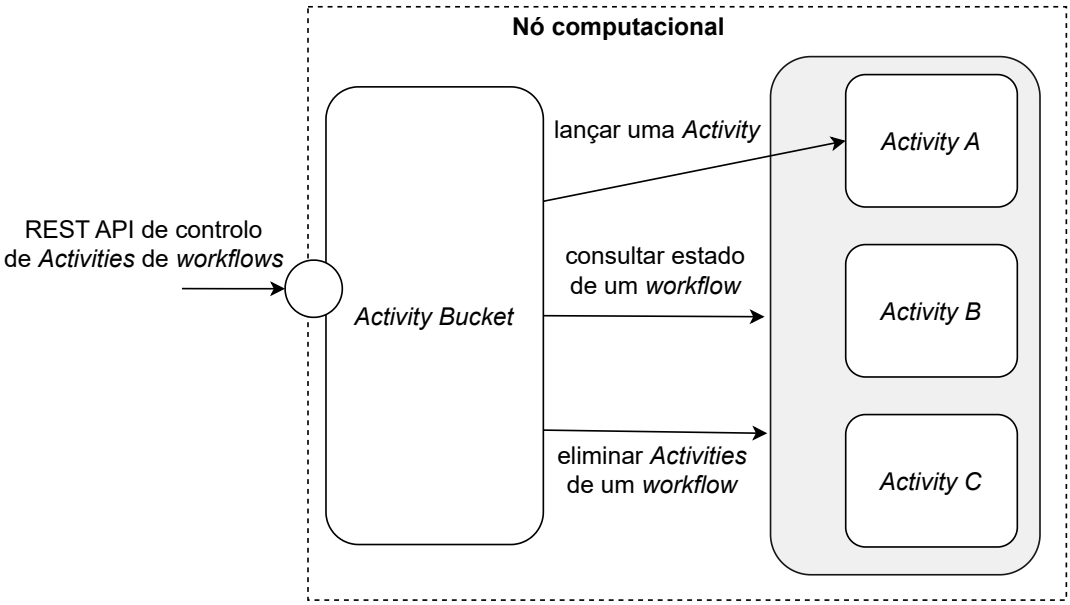


Figura 4.7: Operações do *Activity Bucket* expostas numa interface REST API

De seguida descreve-se detalhadamente as operações da REST API disponibilizada pelo *Activity Bucket*.

❖ **Lançar uma *Activity***

A operação de iniciar a execução de um *Activity* é requisitada através de um pedido HTTP *POST* com o *endpoint* de nome *activity*. A mensagem (*Body*) contém informações relacionadas com a especificação global do *workflow* e outras como se descreve a seguir:

1. ***workflowName***: nome único do *workflow*, proveniente da especificação do *workflow*;

2. ***workingDirectory***: caminho (*path*) relativo de uma subdiretoria a partir da *sharedDirectory*, a partir da qual é criado o *shared volume* para a *workingDirectory* do sistema operativo *Guest* das *Activities*;

3. ***iterations***: estratégia de iterações: número máximo de iterações; número de iteração inicial; valor a incrementar à iteração corrente após cada iteração.

4. ***containerImage***: nome da imagem a ser utilizada como base do *container* da *Activity*;

5. ***task***: informação necessária à execução da *Task* que contém as seguintes definições:

5.1. ***executionCommand***: comando base para executar a componente que implementa a *Task*, por exemplo *java -jar* ou *python*;

5.2. ***executable***: componente de *software* que implementa a *Task* a ser executada com o comando *executionCommand*;

5.3. ***constants***: conjunto de valores estáticos constantes, que são passados à *Task* no momento da sua ativação e que contêm dados num formato de acordo com as especificidades do *workflow*;

6. ***ports***: especificação dos *Input Ports* e dos *Output Ports* da *Activity*, onde cada *Port* tem as seguintes definições:

6.1. ***name***: nome único do *Port* no contexto global do *workflow*;

6.2. ***type***: tipo de *Port* que assume os valores *IN* ou *OUT*, respetivamente se é um *Input Port* ou um *Output Port*;

6.3. ***channel***: nome do *Port* origem ou o nome do *Port* destino, dependendo do tipo do *Port*;

7. ***mappingTaskArguments***: conjunto de mapeamento dos nomes dos *Input Ports* com as posições relativas (índice de 0 a *N*) dos argumentos da *Task*;

8. ***mappingTaskResults***: conjunto de mapeamento das posições relativas (índice de 0 a *N*) dos resultados da *Task* com os nomes dos *Output Ports*;

As informações atrás descritas são passadas ao *Activity Engine* excepto a *workingDirectory* que é apenas relevante ao *Activity Bucket* para a criação do *shared volume*. Adicionalmente o *Activity Engine* recebe também a informação relativa à informação do *Broker*.

❖ **Eliminar *Activities***

A operação de eliminar *Activities* pode ser utilizada a qualquer momento de execução de um *workflow*. Esta operação vai eliminar os *containers* de todas as *Activities* de um determinado *workflow*. A eliminação das *Activities* é requisitada através de um pedido HTTP com o método *DELETE* no *endpoint* de nome *allActivities*, com o *query parameter* de nome *workflowName* que indica o nome do *workflow*. Esta operação é útil para remover os *containers* inativos das *Activities* depois da sua execução, ou de forma a terminar abruptamente a execução de um *workflow*.

❖ **Consultar estado de *Activities***

A operação de consultar o estado de *Activities* permite observar o estado de execução de todas as *Activities* de um determinado *workflow*. Esta operação é requisitada através

de um pedido HTTP com o método *GET* no *endpoint* de nome *allActivities* com o *query parameter* de nome *workflowName*, indicando o nome do *workflow*. O resultado da operação contém uma lista com informação sobre o estado de execução de cada *Activity*: i) nome do *workflow*; ii) nome da *Activity*; iii) o *timestamp* de criação da *Activity*; iv) o estado corrente de execução da *Activity*, que pode ter um dos seguintes valores, *running* (em execução) ou *exited* (término de execução).

4.4 Contrato de Ativação de uma *Task*

Uma *Task* é um processo em execução no sistema operativo *Guest* do *container*. Este processo de execução é iniciado pelo *Activity Engine* a partir de um ficheiro executável presente no sistema de ficheiros do sistema operativo *Guest*, fazendo parte da imagem base do *container*. Na Figura 4.8 apresenta-se o contrato de execução de uma *Task*, indicando que uma *Task* é ativada com uma lista de argumentos, uma lista de constantes e dados de controlo. O conceito de dados de controlo, introduzido no Capítulo 3, concretiza-se em dois valores que poderão ser úteis na funcionalidade da *Task*: i) o número da iteração corrente da *Activity*; ii) o nome identificador da *Activity*.

A *Task* produz um ficheiro com uma lista de resultados em que cada resultado contém dados num formato de acordo com o caso concreto da aplicação. Por questões de generalidade e simplicidade assume-se que o ficheiro de resultados tem um formato genérico, a ser processado pelo *Activity Engine*, por exemplo *JavaScript Object Notation* (JSON). A localização do ficheiro com os resultados é especificada numa variável de ambiente no sistema operativo *Guest* do *container* onde se executa a *Task*.

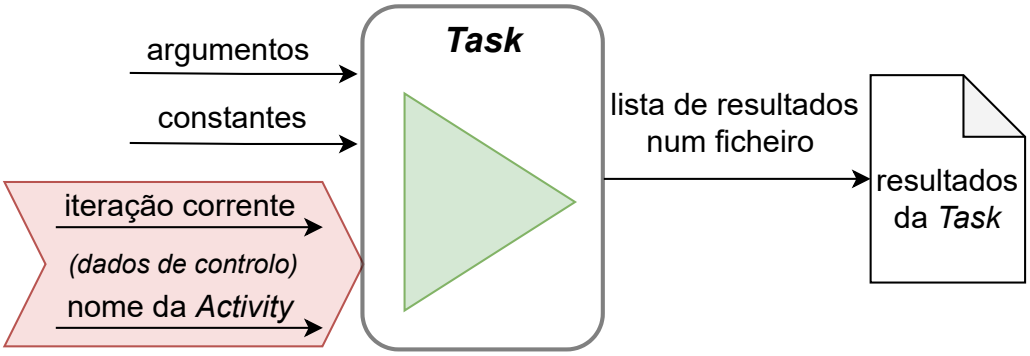


Figura 4.8: Contrato de execução de uma *Task*

4.5 *Activity Engine*

Quando o *Activity Bucket* inicia o *container* de uma *Activity*, este recebe como argumentos a especificação completa da *Activity* de acordo com a informação no *Body* da mensagem da operação lançar *Activity* executada pelo *Activity Bucket*, bem como informação essencial à comunicação com o *Broker* (IP e porto TCP/IP). No *container* de imediato fica em execução o processo *Activity Engine* que será responsável pelo ciclo de vida da *Activity*, nomeadamente a ativação da *Task* em cada iteração.

Na Figura 4.9 são ilustradas as principais ações executadas pelo *Activity Engine* ao longo das iterações de uma *Activity*.

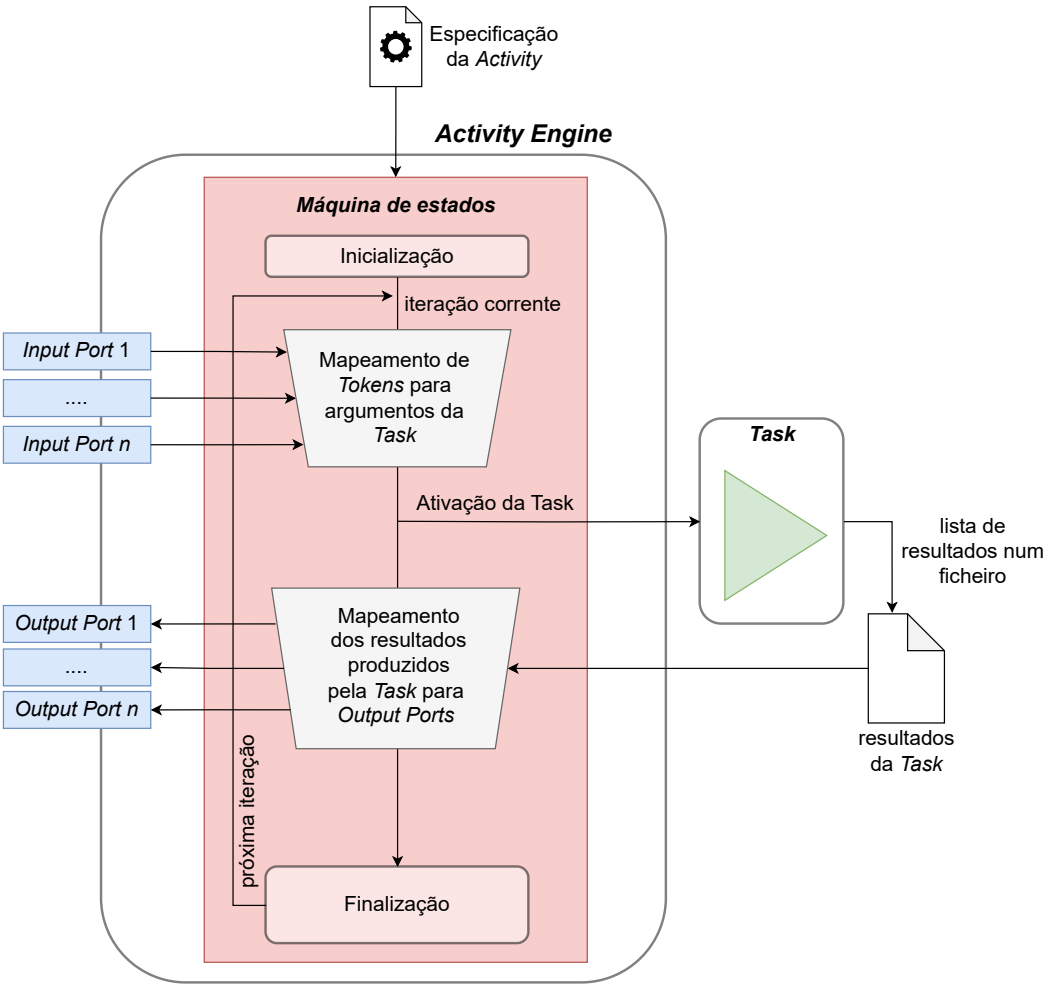


Figura 4.9: Principais ações do *Activity Engine*

O ciclo de vida de uma *Activity*, como detalhada na Secção 3.3, é controlado por uma máquina de estados cujo diagrama de transição de estados é ilustrado na Figura 4.10.

De seguida são descritos os vários estados e respetivas transições.

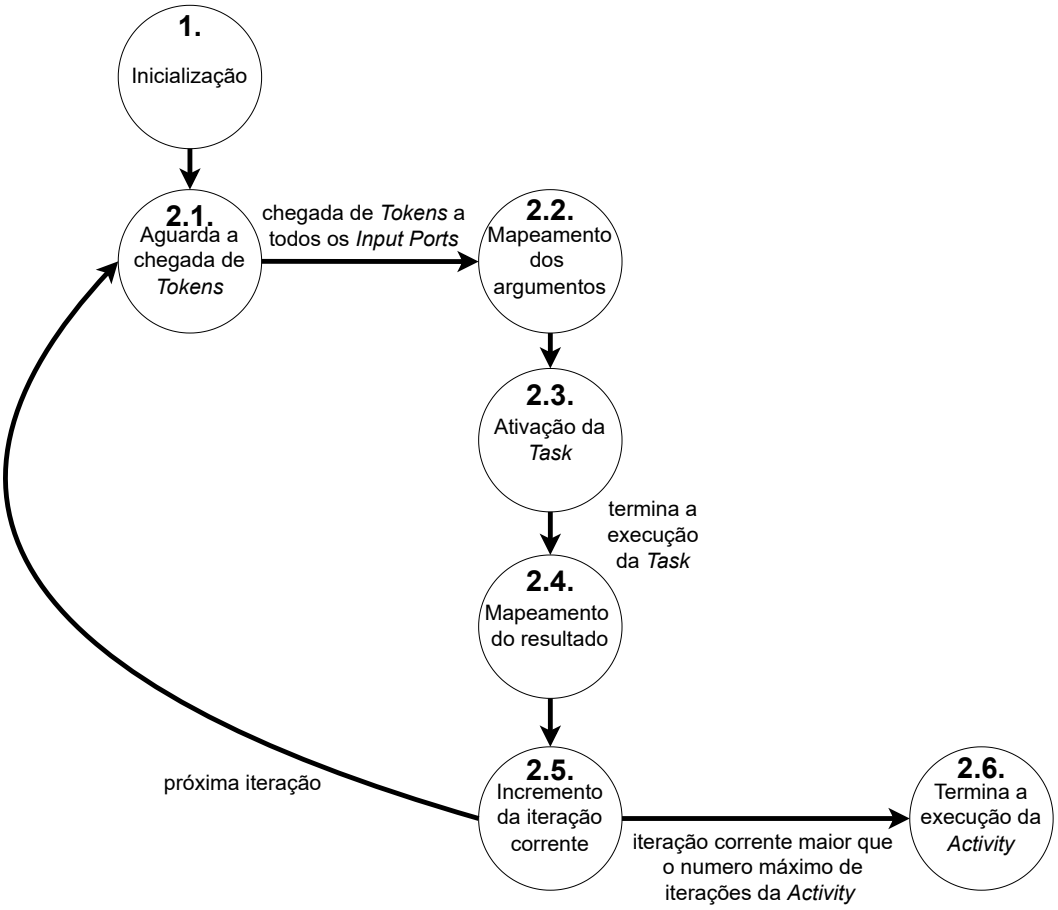


Figura 4.10: Diagrama de transição de estados

1. **Inicialização**: Inicia a máquina de estados de acordo com a especificação da *Activity*, criando caso existam, os *Input Ports* e os *Output Ports* associados aos respetivos *Channels*. Inicia também o controlo de iterações de acordo com a estratégia definida para a *Activity*;

2. Por cada iteração executam-se os seguintes estados:

   2.1. **Aguarda a chegada de *Tokens***: Aguarda a chegada de *Tokens* nos *Input Ports*. O seguimento para o próximo estado apenas será desencadeado quando forem recebidos todos os *Tokens* marcados com o valor de iteração igual ao valor de iteração corrente da *Activity*. Quando tal ocorrer passa ao próximo estado;

   2.2. **Mapeamento dos argumentos**: Os *Tokens* recebidos são mapeados para argumentos da *Task*. A informação relativa à posição do argumento para qual deverá ser mapeado o *Token* recebido num certo *Input Port* é definido na especificação da *Activity*. Após este mapeamento, a *Activity* passará ao próximo estado;

2.3. **Ativação da *Task***: De acordo com a especificação do *workflow*, é feita a ativação da *Task* a ser executada pela *Activity*. A *Task* é executada com os argumentos mapeados no estado anterior e as constantes definidas na especificação da *Activity*. Esta execução irá produzir uma coleção de resultados passando ao estado seguinte;

2.4. **Mapeamento do resultado**: O mapeamento da coleção de resultados da *Task* para os *Output Ports* é feito de acordo com a especificação da *Activity*, onde a cada resultado individual está associado um ou mais *Output Ports*. Esta associação é baseada na posição numérica que o resultado ocupa na coleção de resultados gerados no estado anterior. Os resultados são mapeados em *Tokens* e publicados nos *Channel* associados aos *Output Ports*, passando ao próximo estado;

2.5. **Incremento da iteração corrente**: É incrementado o valor da iteração corrente de acordo com a estratégia definida para a *Activity*. Se o valor de iteração corrente da *Activity* for menor ou igual ao valor máximo de iterações do *workflow*, passa ao estado 2.1 (próxima iteração) para aguardar novamente por *Tokens*. Se o valor de iteração corrente for maior que o máximo de iterações do *workflow* passa ao estado 2.6 de terminação da *Activity*;

2.6. **Termina a execução da *Activity***: Termina a execução da *Activity* libertando todos os recursos alocados para a sua execução.

4.6 Construção da Imagem de um *Activity Container*

De forma a possibilitar a inicialização dos *containers* das *Activities* por parte dos *Activity Bucket*, o programador do *workflow* tem de construir imagens a serem utilizadas como base pelos *containers* de execução das *Activities*. Na Figura 4.11 apresentam-se as camadas existentes na imagem base que suporta a execução de um *container* de uma *Activity*. Como ilustrado na figura, existem 3 camadas:

- **Camada 1**: contém apenas o sistema operativo *Guest*;
- **Camada 2**: contém o ficheiro executável necessário para inicializar o processo *daemon* do *Activity Engine*;
- **Camada 3**: acrescentada pelo programador, contém o ficheiro executável necessário para a execução da *Task*.

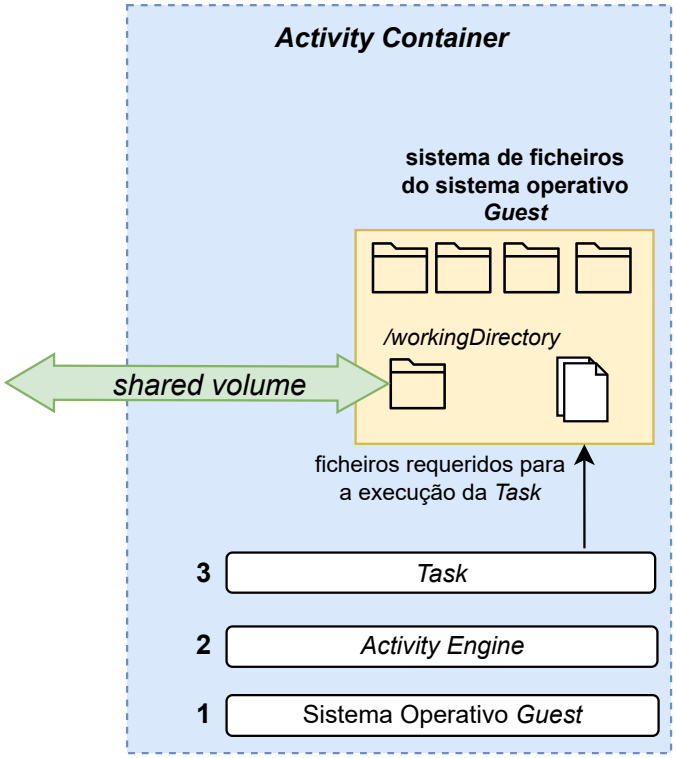


Figura 4.11: Camadas constituintes de um *Activity Container*

4.7 Esquema de Execução de um *Workflow*

O esquema de execução de um *workflow* é constituído por 3 fases tal como está representado na Figura 4.12.

A primeira fase (1) consiste em colocar no sistema de ficheiros distribuídos, entre os nós de execução, os eventuais ficheiros de dados a serem processados na execução de um *workflow*.

Na segunda fase (2), uma aplicação designada *WorkflowLauncher*, parametrizada com a especificação global do *workflow* e a localização dos nós computacionais onde em cada um já se executa um *Activity Bucket*. O *WorkflowLauncher* calcula a estratégia de iterações a serem executadas pelas *Activities*, em função do número de réplicas especificadas para cada *Activity* e do número de iterações global do *workflow*. De seguida o *WorkflowLauncher* distribui para execução as *Activities* nos diferentes nós computacionais, recorrendo à REST API dos *Activity Bucket*.

Na terceira fase (3) obtêm-se os resultados do *workflow* que são flexíveis de acordo com a especificidade do *workflow*. Por exemplo, podem ser ficheiros produzidos no sistema de ficheiros distribuídos entre os nós computacionais, ou outros repositórios, como uma base de dados cujo acesso é encapsulado na programação da *Task*.

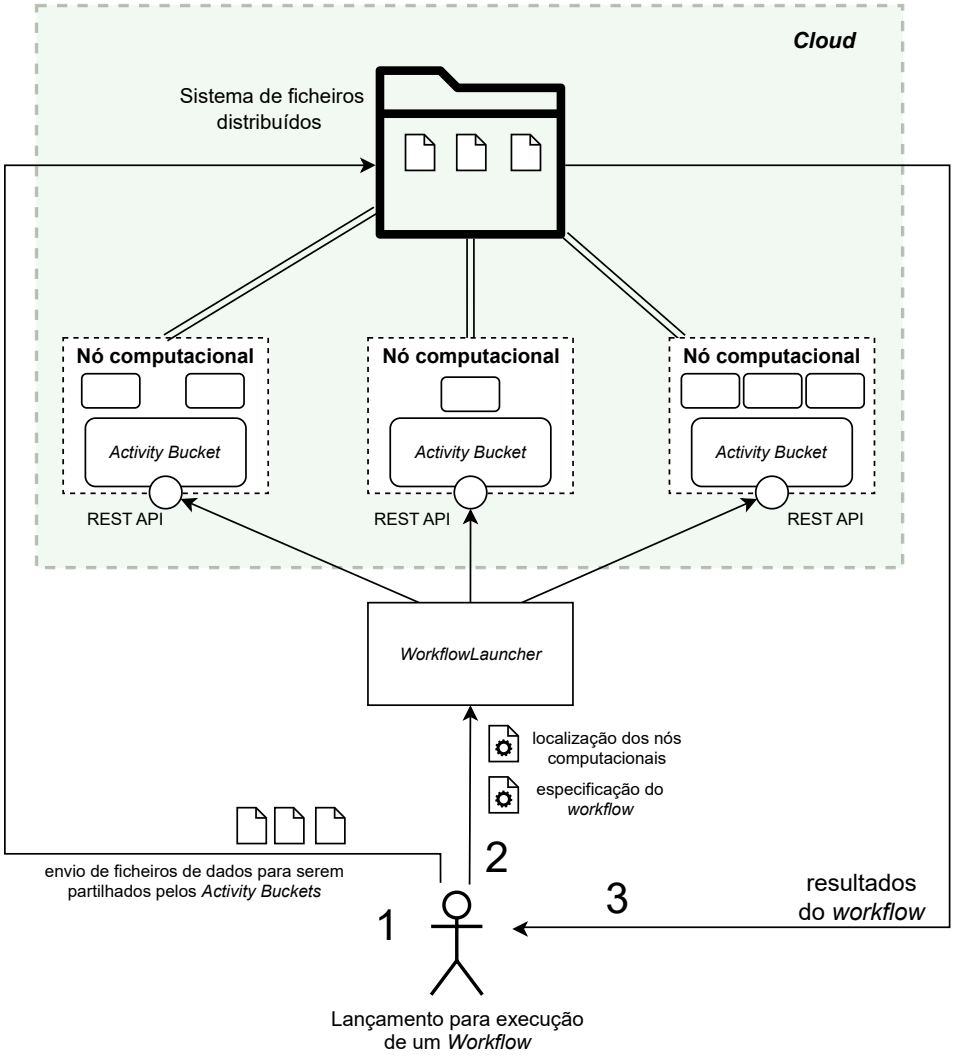


Figura 4.12: Lançamento de um *workflow*

A partir da especificação global do *workflow* e da especificação da infraestrutura computacional, o *WorkflowLauncher*, através da REST API dos *Acticity Bucket*, também possibilita a consulta do estado global de execução de um *workflow*, bem como a eliminação de todas as *Activities* de um *workflow*.

4.8 Resumo

Neste capítulo foi apresentada uma arquitetura de suporte à implementação de um sistema que permita a execução de *workflows* segundo o modelo proposto no Capítulo 3. Esta arquitetura consiste na especificação das interações e funcionalidades dos seguintes componentes:

- Infraestrutura computacional utilizada no suporte à execução de *workflows*;

- Mecanismo intermédio (*Broker*) de comunicação entre as *Activities* durante a execução de *workflows*;
- Interlocutor (*Activity Bucket*) de apoio ao lançamento e monitorização de *Activities* em cada nó computacional;
- Pressupostos de execução da *Task* de uma *Activity*;
- Controlo de execução de uma *Activity* (*Activity Engine*);
- Ambiente de execução de uma *Activity* (*container*);
- Lançamento para execução de um *workflow* (*WorkflowLauncher*).

Considera-se que a partir desta arquitetura é possível fazer a implementação de protótipos de experimentação e de execução de *workflows*, utilizando as mais recentes linguagens e tecnologias de desenvolvimento de *software*.

5

Implementação de um Protótipo

Neste capítulo descreve-se a implementação de um protótipo de experimentação e de execução de *workflows* com base no modelo descrito no Capítulo 3 e na arquitetura apresentada no Capítulo 4. Assim nas secções seguintes serão abordados os seguintes tópicos:

- Considerações sobre escolhas tecnológicas;
- Infraestrutura computacional e partilha de ficheiros distribuídos num *cluster* virtual formado por máquinas virtuais (VM);
- Desenvolvimento das componentes que implementam a arquitetura apresentada no Capítulo 4 e correspondente implementação de um protótipo de experimentação para a execução de *workflows* concretos.

5.1 Escolha de Tecnologias

Os artefactos de *software*, constituintes do protótipo de experimentação e validação do modelo de *workflow*, foram desenvolvidos utilizando a linguagem de programação *Java*, mais especificamente utilizando a implementação *open-source* da versão 11, que foi selecionada devido à robustez e estabilidade da *Java Virtual Machine* (JVM) em execução nas máquinas virtuais da *Google Cloud Platform* (GCP), em *containers* ou mesmo para efeitos de testes em computadores pessoais.

A opção pela plataforma da GCP, deveu-se às facilidades concedidas pela *Google* através do programa *Education Grants* que oferece 50 dólares para a utilização de uma vasta gama de recursos, nomeadamente máquinas virtuais com endereço IP público e com configurações flexíveis com múltiplos CPU e dezenas de GB de memória.

Como tecnologia de criação, execução e gestão de *containers*, foi utilizado o *Docker* [37], que embora seja um produto de *software* proprietário para fins comerciais, tem uma licença de uso livre para uso pessoal, para fins educativos e projetos não comerciais. Apesar de existirem outras alternativas *open-source* de *containers*, o *Docker* é bastante popular e amplamente adotado, existindo uma grande comunidade de utilizadores, grande quantidade de recursos nomeadamente documentação e suporte.

Para ambiente de desenvolvimento foi utilizado o *IntelliJ* [38], por ser muito flexível e já ser uma ferramenta familiar ao longo do curso de mestrado. Como curiosidade, este documento foi produzido em *LaTeX*, utilizando também o *IntelliJ*.

O *RabbitMQ* [39] foi utilizado como tecnologia de *Message Oriented Middleware* (MOM), segundo o modelo de comunicação *publish/subscribe* suportado por *queues*. As razões para a escolha do *RabbitMQ* foram, o facto de ser *open-source*, de suportar o protocolo *Advanced Message Queue Protocol* (AMQP) que é *standard*, significando que existe uma vasta quantidade de informação. Para além disso existe uma imagem *Docker* oficial disponibilizada no *Docker Hub* [40], que permite instanciar um servidor *RabbitMQ* como um *container*, tornando a sua utilização bastante simples e flexível.

Dado que um *Activity Bucket* se executa como um processo em todos os nós computacionais, escolheu-se um *framework* leve, *SparkJava* [41] que permite implementar um servidor HTTP à escuta em qualquer porto TCP/IP e com uma biblioteca *Java* que facilita a implementação de operações que seguem a filosofia REST. Para representação de dados de pedidos e respostas utilizou-se a biblioteca *Gson* [42] desenvolvida pela empresa *Google*.

De forma a providenciar um sistema de ficheiros distribuídos, entre os nós computacionais, usou-se o *GlusterFS* (*Gluster File System*) [43], que é um *software open-source*, desenvolvido para oferecer grande disponibilidade e escalabilidade no armazenamento de ficheiros distribuídos.

No repositório *https://github.com/TheOneAndOnlyPedroFernandes/tfm-code.git* existe toda a implementação do protótipo que poderá ser consultada por solicitação através do *e-mail a41427@alunos.isel.pt*. Nas secções seguintes apresenta-se o detalhe do desenvolvimento de *software* considerado mais relevante.

5.2 Infraestrutura Computacional de Máquinas Virtuais

Para experimentação e validação utilizou-se uma infraestrutura de 4 máquinas virtuais (VM), instanciadas na *Google Cloud Platform* (GCP), com o tipo *e2-standard-4*, com 4 vCPU e memória de 16GB, com o sistema operativo com a distribuição de *Linux*, *Ubuntu*. Estas máquinas virtuais estão interligadas através de uma rede de comunicações, interna da *Google*, o que permite encarar as 4 máquinas como se fossem um *cluster* virtual de processadores gerido a partir do computador pessoal através de sessões SSH ou diretamente na interface *web* do GCP. Cada VM é configurada com os pacotes de *software* base: i) ambiente *Java* com a instalação do *OpenJDK* 11 [44]; ii) ambiente de execução *Docker*.

Como se descreve na secção seguinte, os quatro nós, do *cluster* virtual, são configurados de acordo com o *GlusterFS*, no modo *Replicated Glusterfs Volume* que implica a replicação total de ficheiros em todos os nós.

5.2.1 Partilha de Ficheiros entre os Nós do *Cluster* Virtual

A instalação e configuração do *GlusterFS* com um *volume* de nome *gv0* partilhado entre as 4 VM é ilustrado Figura 5.1.

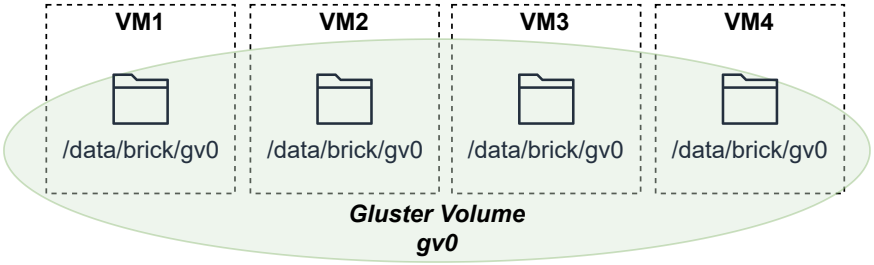


Figura 5.1: Exemplo de um *volume* do *Gluster*

A Listagem 5.1 indica a sequência de comandos para instalar o *GlusterFS* e a inicialização do respetivo *deamon* (*glusterd*) em cada máquina virtual.

```
1 $ sudo add-apt-repository ppa:gluster/glusterfs-7
2 $ sudo apt update
3 $ sudo apt install glusterfs-server
4 $ sudo service glusterd start
```

Listagem 5.1: Comandos de instalação do *GlusterFS*

Ao criar as 4 VM na plataforma GCP temos os seguintes pares (*hostname*, IP): (*VM1*, 10.128.0.1); (*VM2*, 10.128.0.2); (*VM3*, 10.128.0.3); (*VM4*, 10.128.0.4). Os IP atribuídos

pela plataforma GCP são os IP da rede interna, são estáticos e acessíveis por todas as VM, possibilitando que a configuração seja feita apenas uma única vez.

De forma a possibilitar a comunicação dos processos *daemon* do *Gluster*, cada VM é configurada com o comando *Linux* da Listagem 5.2, que permite que as VM aceitem todo o tipo de tráfego.

```
1 $ sudo iptables -I INPUT -p all -j ACCEPT
```

Listagem 5.2: Abertura da *firewall* de uma VM para todo o tipo de tráfego

De forma a agilizar a configuração do *cluster* virtual de VM, com o *GlusterFS*, adicionou-se em todas as VM, no ficheiro */etc/hosts*, as associações dos *hostnames* e dos IP internos. Para configurar o emparelhamento entre os processos *daemon* do *GlusterFS*, basta executar numa VM, por exemplo *VM1*, os comandos apresentados na Listagem 5.3.

```
1 $ sudo gluster peer probe VM2
2 $ sudo gluster peer probe VM3
3 $ sudo gluster peer probe VM4
```

Listagem 5.3: Emparelhamento dos processos *daemon* do *Gluster*

De forma a criar e iniciar a execução do *volume gv0* foram executados os comandos apresentados na Listagem 5.4.

```
1 $ sudo gluster volume create gv0 replica 4 VM1:/data/brick/gv0 VM2:/data/
    ↪ brick/gv0 VM3:/data/brick/gv0 VM4:/data/brick/gv0 force
2 $ sudo gluster volume start gv0
```

Listagem 5.4: Criação e iniciação do *volume gv0* do *Gluster*

Em cada VM é criada uma diretoria (*sharedDirectory*) que é *mounted* para o *volume gv0* do *Gluster* como se indica na Listagem 5.5.

```
1 $ sudo mkdir sharedDirectory
2 $ sudo mount -t glusterfs VM1:/gv0 /data/sharedDirectory
```

Listagem 5.5: *Mount* da diretoria */data/sharedDirectory* para o *volume gv0* do *Gluster*

5.2.2 Inicialização Automática das Máquinas Virtuais

A configuração das máquinas virtuais na GCP permite indicar um *start-up script*. O *script /var/init/init.sh*, para o caso da máquina virtual *VM1*, encontra-se descrito na Listagem 5.6, que inicia o *daemon* do *GlusterFS*, espera 10 segundos para que fique inicializado, realiza o *mount point* do *volume gv0* para a diretoria */data/sharedDirectory* e executa o *daemon ActivityBucket* que recebe como argumento a localização do *Broker RabbitMQ*.

```
1 sudo service glusterd start
2 #esperar que o daemon do gluster esteja inicializado
3 sleep 10
4 sudo mount -t glusterfs VM1:/gv0 /data/sharedDirectory
5 sudo java -jar /var/activity-bucket/ActivityBucket.jar /data/
    ↪ sharedDirectory 10.128.0.4:5672
```

Listagem 5.6: *Script* de inicialização de uma VM

5.3 Diagrama Geral da Implementação do Protótipo

Na Figura 5.2 apresenta-se um diagrama geral de interação entre as diferentes componentes envolvidas na arquitetura descrita no Capítulo 4 e que foram implementadas no desenvolvimento do protótipo de experimentação e validação do modelo.

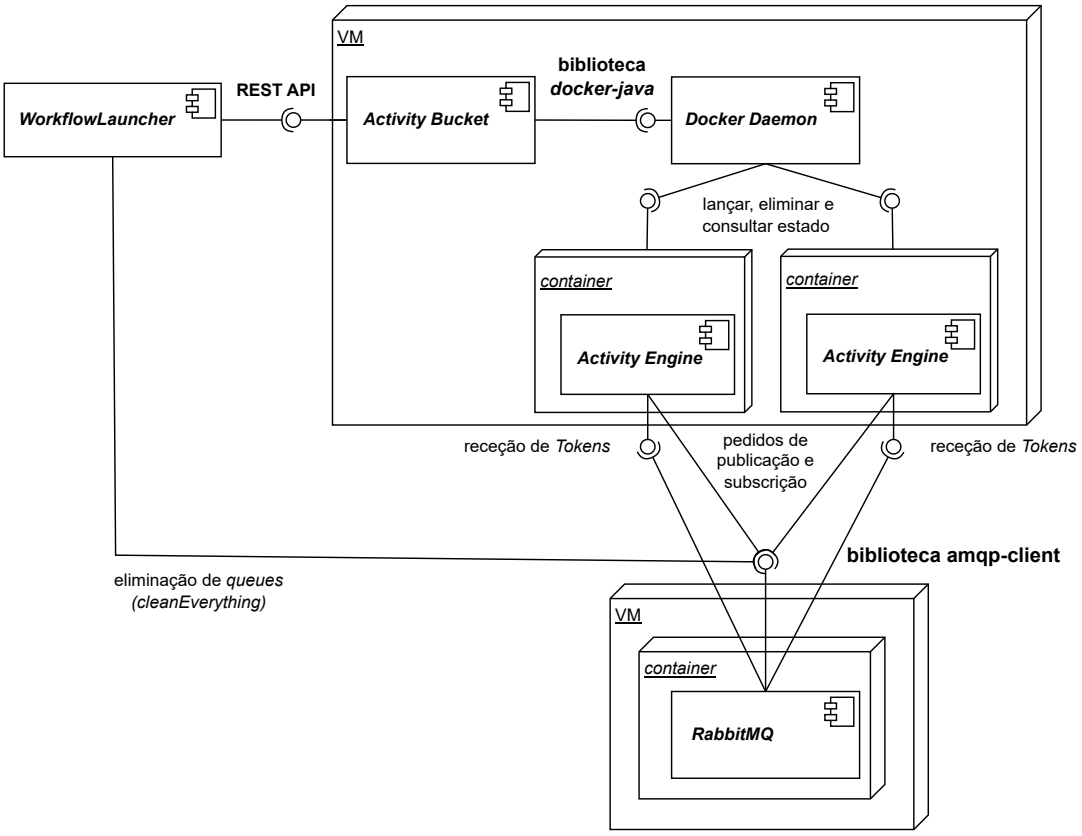


Figura 5.2: Diagrama de arquitetura do protótipo de experimentação

As componentes do diagrama e respetivas interações da Figura 5.2 são descritas com mais detalhe nas secções seguintes.

5.4 Execução de *Workflows* (*WorkflowLauncher*)

A componente *WorkflowLauncher* é uma aplicação executada numa linha de comandos com a sintaxe apresentada na Listagem 5.7 e descrita de seguida.

```
1 $ WorkflowLauncher <infraspec.json> <workflowspec.json> <launch [<
    ↪ mappingspec.json>] | status | deleteActivities | cleanEverything>
```

Listagem 5.7: Sintaxe de execução do *WorkflowLauncher*

<infraspec.json>: *path* para o ficheiro de especificação da infraestrutura computacional incluíndo a localização do *RabbitMQ*.

<workflowspec.json>: *path* para o ficheiro com a especificação do *workflow*.

launch [<mappingspec.json>]: lança em execução o *workflow*, recebendo opcionalmente o *path* para um ficheiro <*mappingspec.json*> de mapeamento entre *Activities* e nós computacionais.

status: obtém o estado *{RUNNING | EXITED}* de execução dos *containers* de todas as *Activities* do *workflow*.

deleteActivities: elimina todos os *containers* onde se executam as *Activities* do *workflow*.

cleanEverything: elimina todos os *containers* das *Activities*, bem como as *queues* no *Broker RabbitMQ* de comunicação entre as *Activities* do *workflow*.

Na Listagem 5.8 apresentam-se exemplos de ficheiros de especificação de infraestrutura (2 nós computacionais em que o *Broker* se executa no *node1*), de especificação do *workflow* e de mapeamento dos nós para *Activities*.

De acordo com o ficheiro de especificação do *workflow* (*workflowspec.json*) as *Activities* são distribuídas equitativamente pelos diferentes nós computacionais cuja localização é descrita no ficheiro *infraspec.json*. No entanto, pode haver situações em que o utilizador queira mapear determinadas *Activities* para nós computacionais específicos, podendo nesse caso fazer o mapeamento através do ficheiro *mappingspec.json*. Por exemplo, para uma *Activity* de nome *A* com réplicas é possível indicar neste ficheiro uma lista de nós onde as réplicas de *A* se irão executar. Se uma *Activity* não tiver réplicas, pode indicar-se qual o nó mais adequado à sua execução consoante requisitos específicos, por exemplo, número de processadores e memória de um nó computacional.

O *WorkflowLauncher* realiza o cálculo da estratégia de iteração de cada *Activity*. No caso de uma *Activity* com réplicas, a estratégia de iteração depende do número de réplicas e de iterações do *workflow*.

```
Excerto do ficheiro infraspec.json
{
  "broker": {
    "host": "34.31.60.226", "port": 5672
  },
  "nodes": [
    { "id": "node1", "ip": "34.31.60.226", "port": 8081 },
    { "id": "node2", "ip": "34.68.72.200", "port": 8081 },
    . . .
  ]
}

Excerto do ficheiro workflowspec.json
{
  "workflowName": "WorkflowExample",
  "workingDirectory": "/workflowExampleDirectory",
  "iterations": 5,
  "activities": [
    {
      "activityName": "A",
      "task": { "executionCommand": "java -jar", "executable": "ATask.jar"
    ↪ },
      . . .
}]
}

Excerto do ficheiro mappingspec.json
{
  "mappingActivitiesToNodes": [
    {
      "activityName": "A",
      "nodeNames": ["node1"]
    },
    . . .
  ]
}
```

Listagem 5.8: Excertos de ficheiros de especificação do *workflow*, de infraestrutura e mapeamento de *Activities* para nós computacionais

O *WorkflowLauncher* é uma aplicação desenvolvida em *Java*, cujo propósito é interagir com as interfaces de comunicação do *Activity Bucket* e do *Broker RabbitMQ*. Na Listagem 5.9 apresenta-se um excerto de código para lançar uma *Activity* utilizando a REST

API do *Activity Bucket*. Na Listagem 5.10 apresenta-se um excerto de código para eliminar uma *queue* no *RabbitMQ*.

```
String jsonActivity="{  'workingDirectory':'...',
                        'workflowName':'...',
                        'activityName':'...',
                        'iterations':{'total':100,'start': 0,'step': 1},
                        'containerImage': '...',
                        . . .
                     }";

HttpRequest request =
      HttpRequest.newBuilder()
          .uri(URI.create("http://" + activityBucketUri + "/activity"))
          .POST(HttpRequest.BodyPublishers.ofString(jsonActivity))
          .build();
```

Listagem 5.9: Lançamento de uma *Activity*

```
ConnectionFactory factory = new ConnectionFactory();
//Inicializar a conexão com o Broker
ConnectionFactory factory = new ConnectionFactory();
factory.setHost(infraspec.broker.host);
        factory.setPort(infraspec.broker.port);
Connection connection = factory.newConnection();
//Obter o canal de comunicacao com a API disponibilizada pelo RabbitMQ
var rabbitMQ = connection.createChannel();

//Apagar uma queue
rabbitMQ.queueDelete("ChannelA-B");
```

Listagem 5.10: Eliminação de uma *queue* no *RabbitMQ*

5.5 *Broker RabbitMQ*

O *Broker* que suporta a comunicação de *Tokens* entre *Activities* foi implementado recorrendo ao servidor *RabbitMQ*.

5.5.1 Inicialização do Servidor *RabbitMQ*

O servidor *RabbitMQ* é instanciado como um *container* numa VM, a partir de uma imagem oficial existente no *Docker Hub*. O servidor é lançado através do comando apresentado na Listagem 5.11, expondo assim o protocolo *standard* AMQP no porto TCP/IP

5672 e uma interface *web* de administração no porto TCP/IP 15672.

```
$ docker run --rm --name rabbidmq -d -p 15672:15672 -p 5672:5672 rabbitmq:
    ↪ 3.11.3-management
```

Listagem 5.11: Execução do servidor *RabbitMQ* num *Docker container*

5.5.2 Biblioteca de Interação

Todos os componentes que interagem com o *RabbitMQ*, utilizam a biblioteca *Java amqp-client* [45] que permite encapsular o código de comunicação com o *Broker* de forma fácil e intuitiva, como apresentado na Listagem 5.12, onde se realça o *handler* de receção de mensagens (*DeliverCallback*) que, de acordo com o *currentIteration*, aceita os *Tokens* marcados com essa iteração (*basicAck*), rejeitando e devolvendo à *queue* os que não estão conformes (*basicNack*).

```
//Inicializar a conexão com o Broker
ConnectionFactory factory = new ConnectionFactory();
factory.setHost(infraspec.broker.host);
factory.setPort(infraspec.broker.port);
Connection connection = factory.newConnection();
//Obter o canal de comunicação com a API disponibilizada pelo RabbitMQ
var rabbitMQ= connection.createChannel();

//Criar de uma queue
rabbitMQ.queueDeclare("ChannelA-B", true, false, false, null);

//Callback de uma subscrição para receção de Tokens
DeliverCallback deliverCallback = (consumerTag, message) -> {
  var token = message.getBody();
  long delivaryTag = message.getEnvelope().getDeliveryTag();

  if(token.iteration==currentIteration){
    //Aceitar uma mensagem com acknowlege positivo
    rabbitMQ.basicAck(deliveryTag);
  } else{
     //Rejeitar uma mensagem com acknowlege negativo, repondo a mensagem na
    ↪  queue
     rabbitMQ.basicNack(deliveryTag, true);
  }
};

//Subscrever uma queue para consumo de mensagens
rabbitMQ.basicConsume(
```

```
28    "ChannelA-B",
29    false,//Modo de acknowlege explícito
30    deliverCallback,
31    consumerTag -> {
32      // Cenário de erro
33    }
34  );
35
36  //Publicar uma mensagem para um exchange default, enviando a mensagem
        ↪ diretamente para a queue com o nome ChannelA-B
37  String token= "{'content':'24','iteration':12}";
38  rabbitMQ.basicPublish("","ChannelA-B", null, token.getBytes());
39
40  //Criar uma exchange de tipo fanout
41  rabbitMQ.exchangeDeclare("ExchangeName", "fanout", true, false, null);
42
43  //Publicar uma mensagem para uma exchange específica
44  rabbitMQ.basicPublish("ExchangeName", "", null, message.getBytes());
45
46  //Apagar uma queue
47  rabbitMQ.queueDelete("ChannelA-B");
```

Listagem 5.12: Interação com o *RabbitMQ* através da biblioteca *amqp-client*

5.6 *Activity Bucket*

O *ActivityBucket* é uma aplicação executada como um *daemon* lançado através da linha de comandos da VM com a sintaxe apresentada na Listagem 5.13 e que se descreve de seguida.

```
1  $ ActivityBucket <sharedDirectory> <brokerlocalization> [<REST API port>]
```

Listagem 5.13: Sintaxe de execução do *ActivityBucket*

<sharedDirectory>: *path* para a *sharedDirectory* configurada na VM;

<brokerlocalization>: IP do nó computacional onde se executa o *broker* e o seu porto TCP/IP de comunicação, por exemplo 34.121.196.80:5672;

[*<REST API port>*]:porto TCP/IP onde é exposta a REST API do *Activity Bucket*, que por omissão é o porto 8081;

Como descrito no Capítulo 4, o *ActivityBucket* expõe uma REST API com as operações de lançar uma *Activity*, consultar o estado de execução de todas *Activities* de um

workflow e de eliminar todas *Activities* de um *workflow*. A REST API foi desenvolvida utilizando o *framework SparkJava*. Na Listagem 5.14 apresenta-se um exemplo de implementação da operação de lançamento de uma *Activity* (*POST /activity*), e na Listagem 5.15 apresenta-se o detalhe do pedido (HTTP *Body*).

```
public class RestApi {
  private final DockerApi dockerApi;

  public RestApi(DockerApi dockerApi) {
    this.dockerApi = dockerApi;
  }
  public void start() {
    Spark.port(8081);
    Spark.post("/activity", "application/json", this::postActivity);
  }
  public String postActivity(Request request, Response response) {
    String activityName = request.body().activityName;
    try {
      dockerApi.createActivity(request.body());
    } catch (Exception e) {
      response.status(500);
      return "Could not create Activity: " + activityName;
    }
    return "Successfully created Activity: " + activityName;
  }
}
```

Listagem 5.14: Lançamento de uma *Activity* (HTTP *POST /activity*)

```
{
  "workingDirectory": "/exampleWorkflow",
  "workflowName": "ExampleWorkflowName",
  "activityName": "A",
  "iterations": { "total": 100, "start": 0, "step": 1 },
  "containerImage": "my-namespace/example-workflow-activity:latest",
  "task": {
    "executionCommand": "java -jar", "executable": "ATask.jar",
    "constants": [ "a" ]
  },
  "ports": [{ "name": "AOut0", "type": "OUT", "channel": "BIn0" }],
  "mappingsTaskArguments": [],
  "mappingsTaskResults": [{ "portName": "OPort0RampX", "index": 0 }]
}
```

Listagem 5.15: Pedido (HTTP *Body*) de lançamento da *Activity A*

Por cada pedido, o *Activity Bucket* interage com o *Docker runtime* (*Docker Daemon*) através da biblioteca *docker-java* disponível num repositório *GitHub* de acesso público [46]. Esta biblioteca encapsula os pedidos HTTP à REST API do *Docker runtime*. Na Listagem 5.16 é apresentado o exemplo de utilização da biblioteca *docker-java* para lançar um *container* de uma *Activity*.

```
1  public class DockerApi {
2
3    public void createActivity(CreateActivityRequest request) {
4      DockerClient dockerClient =
5          DockerClientBuilder
6            .getInstance()
7            .withDockerHttpClient(new ApacheDockerHttpClient
8                                  .Builder()
9                                  .dockerHost(URI.create("unix:///var/run/
     ↪ docker.sock"))
10                                 .build())
11           .build();
12
13     var imageBuilder =
14         dockerClient
15                   .pullImageCmd(request.containerImage)
16                   .withTag(request.containerImageTag)
17                   .start()
18                   .awaitCompletion();
19
20     CreateContainerResponse containerResponse =
21         dockerClient
22           .createContainerCmd(request.containerImage)
23           .withName(extractContainerName(request))
24           .withBinds(Bind.parse("/data/sharedDirectory:/var/
     ↪ workingDirectory"))
25           .withCmd(request.getActivityArgumentAsJson())
26           .exec();
27
28     dockerClient
29           .startContainerCmd(containerResponse.getId())
30           .exec();
31   }
32 }
```

Listagem 5.16: Lançamento de um *container* de uma *Activity*

Como exemplo, na Figura 5.3 é ilustrado o diagrama de sequência de um pedido de lançamento de uma *Activity*, mostrando a interação dos diferentes componentes do

protótipo, envolvidos nessa operação.

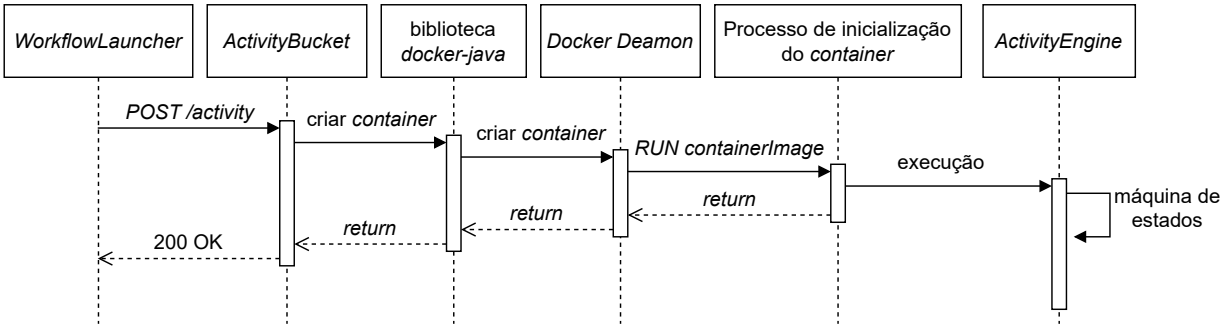


Figura 5.3: Diagrama de sequência de um pedido de lançamento de uma *Activity*

5.7 *Activity Engine*

O *Activity Engine* é uma aplicação executada a partir da linha de comandos, com a sintaxe apresentada na Listagem 5.17 e descrita de seguida.

```
$ ActivityEngine <specjsonstring>
```

Listagem 5.17: Sintaxe de execução do *ActivityEngine*

<specjsonstring>: objeto serializado numa *string* JSON com informação relacionada com a especificação do *workflow*, em particular a especificação da *Activity* e a localização do *Broker*.

Na Listagem 5.18, é apresentado um exemplo do conteúdo JSON de um <*specjsonstring*> passado como parâmetro no lançamento do *Activity Engine*.

```
{ Informação oriunda do pedido de criação da Activity
  "workflowName": "ExampleWorkflowName",
  "activityName": "A",
  "iterations": { "total": 100, "start": 0, "step": 1 },
  "task": {
    "executionCommand": "java -jar", "executable": "ATask.jar",
    "constants": [ "a" ]
  },
  "ports": [{ "name": "AOut0", "type": "OUT", "channel": "BIn0" }],
  "mappingsTaskArguments": [],
  "mappingsTaskResults": [{ "portName": "OPort0RampX", "index": 0 }],
  localização do Broker
  "brokerInfo": { "host": "34.121.196.80", "port": 5672 }
}
```

Listagem 5.18: Informação passada como parâmetro no lançamento do *Activity Engine*

O *Activity Engine* implementa a máquina de estados descrita no Capítulo 4 baseando-se nos padrões de *software*, *State Pattern* [47] e *Service Locator* [48].

Os estados cruciais da máquina de estados, são a criação de *Input Ports*, a espera de *Tokens*, a execução da *Task* e a publicação de *Tokens*. Cada *Input Port* é criado com uma subscrição a uma *queue*, passando-lhe o *handler* cujo excerto é apresentado na Listagem 5.19.

O *handler*, de acordo com o *currentIteration* aceita os *Tokens* marcados com essa iteração (*basicAck*), ou rejeita devolvendo à *queue* os *Tokens* que não estão conformes (*basicNack*).

O *handler* permite que a máquina de estados, para cada *Input Port*, obtenha um *CompletableFuture* (*getTokenFuture*) para ser sinalizado quando existir um *Token* nesse *Input Port*. Quando o *handler* aceita um *Token* válido na iteração corrente, sinaliza (*complete*) a máquina de estados da receção desse *Token*.

```
1  public class TokenReceptionCallback implements DeliverCallback {
2
3    private final IterationController iterationController;
4    private CompletableFuture<Token> tokenFuture;
5
6    @Override
7    public void handle(String consumerTag, Delivery message) {
8      var token = message.getBody();
9      long delivaryTag = message.getEnvelope().getDeliveryTag();
10
11     if(token.iteration==currentIteration){
12       //Aceitar uma mensagem com acknowlege positivo
13       rabbitMQ.basicAck(deliveryTag);
14       tokenFuture.complete(token);
15     } else{
16       //Rejeitar uma mensagem com acknowlege negativo, repondo a mensagem
     ↪ na queue
17       rabbitMQ.basicNack(deliveryTag, true);
18     }
19   }
20
21   public CompletableFuture<Token> getTokenFuture() {
22     return tokenFuture;
23   }
24 }
```

Listagem 5.19: Excerto do *handler* de receção de *Tokens*

Para generalizar a implementação dos vários estados da máquina de estados existe uma classe base de nome *EngineState*, que tem um método *execute* que recebendo a

informação do contexto do estado corrente, executa as ações inerentes a cada estado e um método *setNextState* que recebe a implementação do próximo estado e a informação do contexto.

Por exemplo, na Listagem 5.20 é apresentado um excerto da classe *ExecuteTask* que implementa o estado da máquina de estados (Figura 4.10) que ativa para execução a *Task* de uma *Activity*, onde se usa o contexto para obter a informação da especificação da *Activity* e após o lançamento e terminação do processo que executa a *Task* é feita a transição para o próximo estado, onde são obtidos os resultados da *Task*, implementado com a classe *CollectTaskResultsState* que também estende a classe base *EngineState*.

```
1  public class ExecuteTask extends EngineState {
2    private final String taskParameters;
3
4    public ExecuteTask (String taskParameters) {
5      this.taskParameters = taskParameters;
6    }
7    @Override
8    public void execute(IStateContext context) {
9      //Obtenção da especificação da Activity
10     Specification spec = context.get(Specification.class);
11
12     //Obtenção do path para a diretoria de executáveis da Task
13     String taskExecutableDirectory =
14         System.getenv("TASK_EXECUTABLE_DIRECTORY");
15
16     //Argumentos do processo
17     String[] cmdArray=new String[]{
18       spec.executionCommand,
19       taskExecutableDirectory + "/" +spec.executable,
20       taskParameters
21     };
22
23     //Execução e espera do processo
24     Process p = Runtime.getRuntime()
25                        .exec(cmdArray)
26                        .waitFor();
27
28     //Configurar o próximo estado
29     setNextState(new CollectTaskResultsState(), context);
30   }
31 }
```

Listagem 5.20: Execução da *Task*

Como se apresenta na Figura 5.4, ao longo da execução do *Activity Engine*, são enviados *Logs* de execução para uma *Exchange* do tipo *Fanout* do *RabbitMQ* de nome *logs*, permitindo serem realizadas aplicações de escuta desses *Logs* muito úteis para fazer *debugging*, análise ou estudo de desempenho, nomeadamente tempos de execução dos *workflows* e de *overheads* associados.

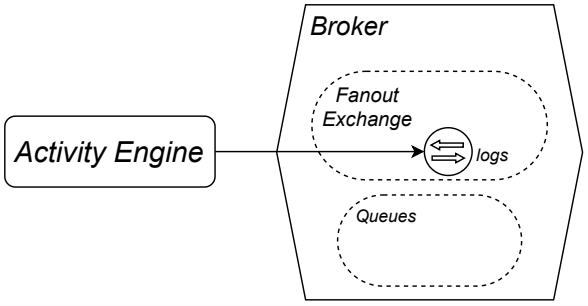


Figura 5.4: Envio de *Logs* de execução de uma *Activity* para o *Exchange* de nome *logs*

5.8 *Activity Container*

Como ilustrado na Figura 5.5, as imagens *Docker* de uma *Activity* necessitam de ter como camada base, a imagem *activity-base-image* que já contém o *Activity Engine*. Para criação dessa imagem apresenta-se em Listagem 5.21 o ficheiro *Dockerfile*.

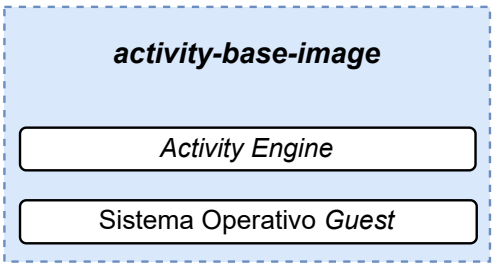


Figura 5.5: Camadas da imagem *activity-base-image*

```
1 FROM alpine:3.17
2 RUN apk add openjdk11
3 COPY ActivityEngine.jar /var/activity/ActivityEngine.jar
4 ENV PATH_TO_WORK_DIRECTORY="/var/workingDirectory"
5 ENV PATH_TO_TASK_RESULTS_FILE="/var/results/taskResults.json"
6 ENTRYPOINT ["java", "-jar", "/var/activity/ActivityEngine.jar"]
```

Listagem 5.21: *Dockerfile* de criação da imagem *activity-base-image*

A imagem *activity-base-image* utiliza o sistema operativo *alpine* (*FROM alpine:3.17*) devido a esta ser uma distribuição *Linux* minimalista e estar disponível numa imagem

Docker oficial publicada no *Docker Hub*. De forma a poder executar o *Activity Engine*, é necessário declarar no *Dockerfile* a instalação do *OpenJDK* (*RUN apk add openjdk11*), e copiar o artefacto executável *ActivityEngine.jar* para a imagem. De forma a proporcionar o *path* para a *workingDirectory* e o *path* do ficheiro onde serão colocados os resultados da *Task*, são utilizadas as variáveis de ambiente *PATH_TO_WORK_DIRECTORY* ("*/var/workingDirectory*") e *PATH_TO_TASK_RESULTS_FILE* ("*/var/results/taskResults.json*") respetivamente. No final é definido o *entrypoint* do *container* (*ENTRYPOINT ["java", -jar", "/var/activity/ActivityEngine.jar"]*) com o comando de ativação do *Activity Engine* a ser despoletada de imediato no momento de execução do *container*. Sendo a imagem *Docker* definida com este *ENTRYPOINT*, é possível no momento de inicialização de um *container*, passar mais argumentos, providenciando ao *Activity Engine* as informações necessárias de execução de uma *Activity*.

A imagem *Docker activity-base-image* está disponibilizada num repositório público de uma conta pessoal do *Docker Hub* referida como *my-namespace*. Assim o nome da imagem para ser reutilizada (*pull*) é *my-namespace/activity-base-image*.

5.9 Especificação de um *Workflow*

Tal como referido no Capítulo 1, um dos objetivos para as principais características deste projeto foi a especificação de *workflows* numa linguagem de representação de dados facilmente interpretada por ferramentas de desenvolvimento de *software*. Assim para a especificação de *workflows* e de acordo com o modelo apresentado no Capítulo 3, utilizou-se a linguagem de representação de dados JSON.

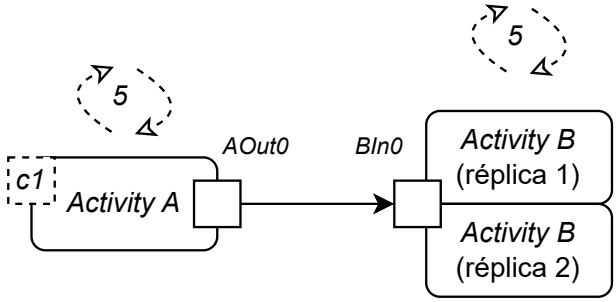


Figura 5.6: Exemplo de um *workflow* com 2 réplicas da *Activity B*

Na Listagem 5.22 apresenta-se um exemplo de especificação do *workflow* com o nome *WorkflowExample*, ilustrado na Figura 5.6 e configurado com 5 iterações. A *Activity A*, tem uma *Task* desenvolvida com a linguagem de programação *Java* e a *Activity B* configurada com 2 réplicas tem uma *Task* desenvolvida na linguagem de programação *Python*.

```
{
  "workflowName": "WorkflowExample",
  "workingDirectory": "/workflowExampleDirectory",
  "iterations": 5,
  "activities": [
    {
      "activityName": "A",
      "containerImage": "my-namespace/workflow-example-activities:latest",
      "task": {
        "executionCommand": "java -jar", "executable": "ATask.jar",
        "constants": ["c1"]
      },
      "ports": [{ "name": "AOut0", "type": "OUT","channel": "BIn0" }],
      "mappingsTaskResults": [{ "portName": "AOut0", "index": 0}]
    },
    {
      "activityName": "B",
      "containerImage": "my-namespace/workflow-example-activities:latest",
      "replicas": 2,
      "task": { "executionCommand": "python3", "executable": "b-task.py" },
      "ports": [{ "name": "BIn0", "type": "IN", "channel": "AOut0" }],
      "mappingsTaskResults": [{ "portName": "BIn0", "index": 0 }]
    }
  ]
}
```

Listagem 5.22: Especificação do *workflow* com o nome *WorkflowExample*

5.10 Desenvolvimento de uma *Task*

O programador de uma *Task* tem a liberdade de a desenvolver em qualquer linguagem de programação, apenas necessita de ter em consideração os seguintes pressupostos do ambiente de onde será executada a *Task*.

WorkDirectory

Existência de uma variável de ambiente de nome *PATH_TO_WORK_DIRECTORY*, que contém o *path* para a diretoria *workingDirectory*.

Ficheiro de resultados

Existência de uma variável de ambiente de nome *PATH_TO_TASK_RESULTS_FILE*, que contem o *path* para o ficheiro onde são escritos os resultados da *Task*.

Parâmetro de chamada

Como foi descrito na Secção 4.4, uma *Task* é executada como um comando de linha para lançar um processo de acordo com os pressupostos da linguagem onde foi desenvolvida, tendo como parâmetro uma *string* em formato JSON como se apresenta na Listagem 5.23 com os seguintes valores: i) ***arguments***: array de *strings*, resultantes do mapeamento de *Tokens* recebidos nos *Input Ports* da *Activity*; ii) ***constants***: array de *strings* com valores constantes definidos na especificação do *workflow*; iii) ***currentIteration***: valor inteiro da iteração corrente da *Activity* no momento de ativação da *Task*; iv) ***activityName***: nome da *Activity* que executa a *Task*;

```
{
  "arguments":["a","b"],
  "constants":["c"],
  "currentIteration":1,
  "activityName":"ActivityXpto"
}
```

Listagem 5.23: Exemplo de parâmetro de ativação de uma *Task*

Resultados

Os resultados da *Task* necessitam de ser escritos no ficheiro de resultados, em formato JSON, cujo *path* é providenciado pela variável de ambiente *PATH_TO_TASK_RESULTS_FILE*, composto de um campo de nome *results* cujo valor é um *array* de *strings*. Um exemplo de uma *Task* que produz 3 resultados encontra-se apresentado na Listagem 5.24.

```
{"results":["res1","res2","res3"]}
```

Listagem 5.24: Exemplo de resultados produzidos por uma *Task*

Na Listagem 5.25 apresenta-se um exemplo de uma *Task* desenvolvida em *Java*, com interpretação da *string* JSON recebida como parâmetro e a escrita de resultados num ficheiro.

```
public class App {
  public static final String PATH_TO_WORK_DIRECTORY =
      System.getenv("PATH_TO_WORK_DIRECTORY");
  private static final String PATH_TO_TASK_RESULTS_FILE =
      System.getenv("PATH_TO_TASK_RESULTS_FILE");

  public static void main(String[] args) {
    ArgumentsModel argumentsModel = new Gson().fromJson(args[0],
    ↪ ArgumentsModel.class);

    // ...  código específico da Task no contexto da aplicação
```

```
11      writeToOutputFile(results);
12    }
13
14    public void writeToOutputFile(String... results) {
15      try (FileWriter fileWriter = new FileWriter(PATH_TO_TASK_RESULTS_FILE))
        ↪  {
16        fileWriter.write(new Gson().toJson(new ResultsModel(results)));
17        System.out.println("JSON data written to file successfully!");
18      } catch (IOException e) {
19        System.out.println("An error occurred while writing JSON data to file
        ↪ .");
20      }
21    }
22
23    public class ArgumentsModel {
24      public String[] arguments;
25      public String[] constants;
26      public int currentIteration;
27      public String activityName;
28    }
29
30    public class ResultsModel {
31      public String[] results;
32
33      public ResultsModel(String... results) {
34        this.results = results;
35      }
36    }
37  }
```

Listagem 5.25: Exemplo de uma *Task* desenvolvida em *Java*

Um programador de *workflow* tem de criar as imagens das *Activities* utilizando como camada base a imagem *activity-base-image*, adicionando o executável da *Task* e as configurações necessárias para suportar a sua execução, tal como se ilustra na Figura 5.7.

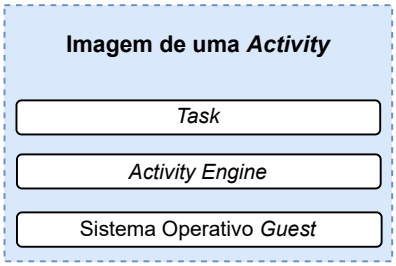


Figura 5.7: Camadas da imagem de uma *Activity*

Por exemplo, o *Dockerfile* para uma *Activity* que necessite executar uma *Task* desenvolvida como um *script Python*, encontra-se na Listagem 5.26 e é descrito de seguida.

O *Dockerfile* necessita primeiramente da especificação da camada base da *Activity* (*FROM my-namespace/activity-base-image:latest*), de seguida, e visto que a linguagem base apenas suporta a execução de artefactos desenvolvidos em *Java*, é necessário adicionar à imagem a instalação do ambiente de execução de *scritpts Python* (*RUN apk add python3*). Um requisito obrigatório da imagem de uma *Activity* é a definição de uma variável de ambiente de nome *TASK_EXECUTABLE_DIRECTORY* com o *path* da diretoria para onde é copiado o ficheiro executável da *Task*, sendo através desta variável de ambiente que o *Activity Engine* consegue localizar o executável da *Task* definido na especificação do *workflow*.

```
1 FROM my-namespace/activity-base-image:latest
2 RUN apk add python3
3 RUN mkdir /var/tasks
4 ENV TASK_EXECUTABLE_DIRECTORY="/var/tasks"
5 COPY task.py /var/tasks/task.py
```

Listagem 5.26: *Dockerfile* de uma *Activity* com uma *Task* desenvolvida em *Python*

É possível definir uma imagem de *Activity* com várias *Tasks*, proporcionando a sua reutilização por várias *Activities*, bastando copiar todos os executáveis para a diretoria definida em *TASK_EXECUTABLE_DIRECTORY* e podendo definir qual *Task* a ser utilizada numa determinada *Activity* através do campo *executable* da definição da *Task* definido na especificação do *workflow*.

5.11 Recolha de Informação de *Log*

Como ilustrado na Figura 5.8, foi desenvolvida uma aplicação *WorkflowLogListener* que cria e subscreve uma queue do *RabbitMQ* associada à *Exchange* de nome *logs* criada pelo *Activity Engine*, escrevendo num ficheiro as mensagens de *Log* recebidas.

O *WorkflowLogsListener* é uma aplicação cuja execução é lançada através da linha de comandos com a sintaxe apresentada na Listagem 5.27 e descrita de seguida.

```
1 $ WorkflowLogListener -broker=<brokerinfo> -filterByWorkflowName=<
    ↪ workflowName> -outputFile=<outFile>
```

Listagem 5.27: Sintaxe de execução do *WorkflowLogsListener*

<brokerinfo>: localização do *Broker RabbitMQ*, por exemplo 34.121.196.80:5672.

<*workflowName*>: nome do *workflow* cujas mensagens de *Log* se pretende recolher.

<*outFile*>: *path* para o ficheiro a escrever (*write append*) as mensagens de *Log* recebidas.

De salientar que os *Logs* se referem unicamente à execução completa do *workflow* no contexto de um *Activity Engine*, como representado na Figura 5.9.

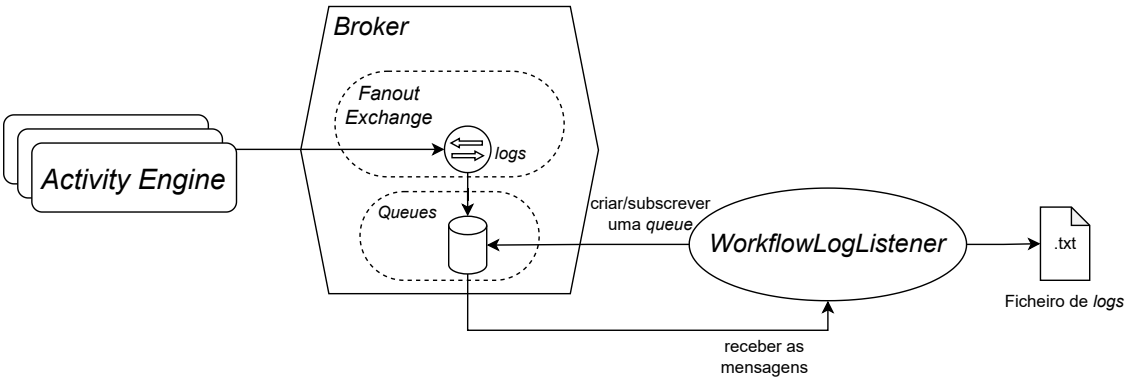


Figura 5.8: Recolha de *Logs* pelo *WorkflowLogListener*

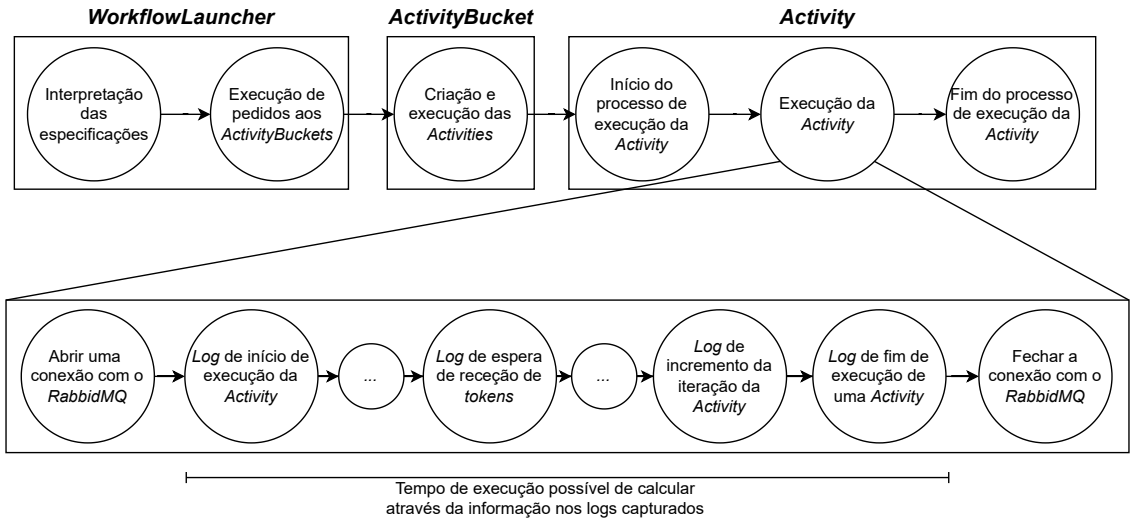


Figura 5.9: Publicação de mensagens de *Log* pelo *Activity Engine*

5.12 Resumo

Neste capitulo foi apresentada a implementação de um protótipo para permitir a experimentação de execução de *workflows* segundo o modelo proposto, e que permitiu retirar as seguintes conclusões:

- A implementação do protótipo foi facilitada pela descrição pormenorizada do

modelo (Capítulo 3) e da arquitetura de suporte apresentada no Capítulo 4 respetivamente;

- A escolha de tecnologias para implementar a arquitetura mostrou-se adequada e flexível;
- A implementação de um mecanismo de recolha de *Logs* foi fundamental no processo de *debugging*, no processo de experimentação e validação de casos concretos do *workflow* apresentado no Capítulo 6.

6

Experimentação e Validação do Modelo

Neste capítulo são apresentados resultados de diferentes avaliações feitas à implementação do modelo de *workflow*. Estas avaliações foram realizadas recorrendo ao desenvolvimento e execução de vários *workflows* onde cada um tem como propósito, avaliar a correção e salientar as diferentes características suportadas pelo modelo de *workflow* apresentado no Capítulo 3, nomeadamente:

- Desenvolvimento de *Tasks* em diferentes linguagens de programação;
- Especificação de padrões de *workflow* como o *Sequence*, *Parallel Split* e *Synchronization*;
- Especificação de *workflows* com múltiplas iterações;
- Distribuição das *Activities* para execução de um *workflow* em múltiplos nós de computação;
- Considerando que o *workflow* tem múltiplas iterações, explorar a especificação de réplicas de uma *Activity*, cuja *Task* tem um tempo de processamento elevado, tirando partido do balanceamento de carga nas sucessivas iterações;
- Avaliação de tempos de execução de *workflow* e de tempos de *overhead* associados à implementação do protótipo;
- Possibilidade de *Tasks* interagirem, durante a sua execução, com serviços externos, nomeadamente serviços com funcionalidades complexas e úteis, disponíveis em fornecedores de *Clouds* públicas;

- Desenvolvimento de *workflows* segundo o modelo *MapReduce*;

Nas secções seguintes são apresentados quatro casos de *workflow*, desenvolvidos e usados experimentalmente com o objetivo de avaliar a operacionalidade do protótipo que implementa o modelo e validar o suporte às características atrás enunciadas. Para cada caso, são apresentados os detalhes essenciais da implementação do *workflow* e a discussão dos resultados obtidos.

6.1 *Workflow* de Operações Aritméticas

Neste primeiro *workflow* pretendeu-se avaliar as características essenciais do modelo, tais como, desenvolvimento de *Tasks* em diferentes linguagens de programação (*Java* e *Python*), especificação de padrões de *workflow* básicos como o *Parallel Split* e *Synchronization* e a especificação de um *workflow* com múltiplas iterações. O *workflow* é constituído por várias *Activities* capazes de realizar cálculos da expressão aritmética descrita na Expressão (6.1). e cujo *workflow* é apresentado na Figura 6.1.

$$(x + y) \cdot (x + y) \tag{6.1}$$

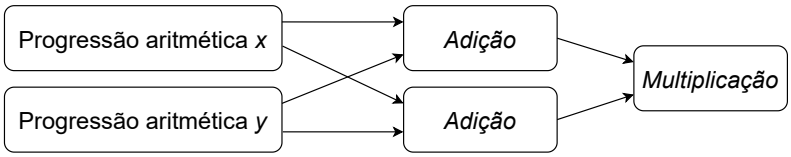


Figura 6.1: Representação abstrata do *workflow* de operações aritméticas

Os valores de x e de y são gerados pela Progressão Aritmética (PA) de n termos como se apresenta na Expressão (6.2), onde a_0 é o primeiro valor da PA e r é a razão. Os valores de x e de y podem ser gerados por duas PA com a_0 e r diferentes. Para o valor de n usou-se o valor da iteração corrente durante a execução do workflow.

$$a_n = a_0 + n \times r, \forall n \in \mathbb{N}_0 \tag{6.2}$$

O *workflow* da Figura 6.1 permite validar o padrão *Parallel Split* na medida em que as *Activities* que geram os valores de x e de y enviam em paralelo dois valores inteiros para as *Activities* de adição que se executam em paralelo. O padrão *Synchronization* é validado na *Activity* de multiplicação, pois esta só se executa após sincronizar a receção de dois valores oriundos das *Activities* de adição.

De forma a poder demonstrar o suporte do modelo ao desenvolvimento de *Tasks* em diferentes linguagens de programação, a *Task* que realiza o cálculo da PA e a *Task* que realiza a operação de adição foram ambas realizadas em *Java*. A *Task* que realiza a operação de multiplicação foi realizada em *Python*.

Na Figura 6.2 apresenta-se o *workflow* com as *Tasks* desenvolvidas. O *workflow* é composto por cinco *Activities*. As *Activities RampUpX* e *RampUpY*, representam as *Activities* de geração de valores das PA (valores x e y da Expressão (6.1)) e utilizam a mesma *Task* em *Java* com o artefacto executável de nome *RampUpTask.jar*. As *Activities Addition1* e *Addition2* representam as *Activities* de adição e utilizam a mesma *Task* em *Java* com o artefacto executável de nome *AdditionTask.jar*. A *Activity Multiplication* representa a *Activity* de multiplicação e utiliza a *Task* em *Python* com o *script* executável de nome *MultiplicationTask.py*.

Ao longo das sucessivas iterações do *workflow*, a *Activity Multiplication* escreve (*write append*) no final do ficheiro de resultados de nome *Results.txt*, os valores inteiros resultantes do cálculo da Expressão (6.1) para os múltiplos valores de x e de y.

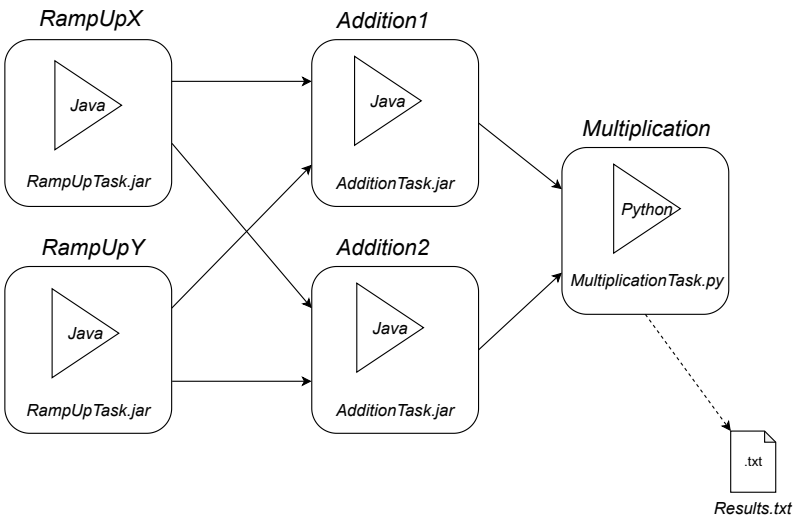


Figura 6.2: Representação abstrata do *workflow* de operações aritméticas

De forma a testar o *workflow*, foram realizadas várias experimentações, com várias iterações e diferentes valores de a_0 e r. Na Listagem 6.1 é apresentado o conteúdo do ficheiro *Results.txt* com os resultados de uma experimentação realizada com cinco iterações. Para a *Activity RampUpX* usaram-se os valores $a_0 = 1$ e $r = 1$ que para cinco iterações produziu os valores 1, 2, 3, 4 e 5. Para a *Activity RampUpY* usaram-se os valores $a_0 = 0$ e $r = 2$ que em cinco iterações produziu os valores 0, 2, 4, 6 e 8.

Por exemplo, para a quarta iteração ($n = 3$) temos:

- $x = 1 + 3 \times 1 = 4$;

- $y = 0 + 3 \times 2 = 6$;
- O resultado de cada adição é igual a $4 + 6 = 10$;
- O resultado da multiplicação é $10 \times 10 = 100$.

```
1
16
49
100
169
```

Listagem 6.1: Resultado de execução do *workflow* de operações aritméticas

Os comportamentos das *Tasks* envolvidas nas *Activities* do *workflow* são:

❖***RampUpTask*:**
argumentos: nenhum;
constantes: i) *string* contendo o valor inteiro representando a_0; ii) *string* contendo o valor inteiro representando r;
comportamento: com base nas constantes recebidas a_0 e r e no valor da iteração corrente, produz o resultado da Progressão Aritmética (PA) apresentada na Expressão (6.2);
resultado: *string* com o valor inteiro resultante da Progressão Aritmética (PA).

❖***AdditionTask*:**
argumentos: lista de *strings* contendo dois valores inteiros correspondentes aos operandos da adição;
constantes: nenhuma;
comportamento: realiza a adição dos valores inteiros recebidos nos argumentos;
resultado: *string* contendo o valor inteiro resultante da adição;

❖***MultiplicationTask*:**
argumentos: lista de *strings* contendo dois valores inteiros correspondentes aos fatores da multiplicação;
constantes: *string* contendo o nome do ficheiro para acrescentar o valor resultante da multiplicação;
comportamento: realiza a multiplicação dos dois valores inteiros recebidos nos argumentos e escreve o valor resultante da multiplicação no ficheiro cujo nome é definido como constante;
resultado: nenhum.

Após o desenvolvimento dos artefactos executáveis das várias *Tasks*, foi necessário criar uma imagem *Docker* de suporte à execução dos *containers* das *Activities* do *workflow*. Na Listagem 6.2 apresenta-se o ficheiro *Dockerfile* onde é possível observar a

flexibilidade do protótipo para permitir colocar numa única imagem todas as *Tasks* de um determinado *workflow*. De notar que seria possível definir imagens diferentes para cada *Activity*. No entanto, ter os executáveis das várias *Tasks* na mesma imagem, permite uma mais fácil gestão com um único *Dockerfile* e uma única imagem para o contexto de execução dos *containers* de suporte à execução das várias *Activities* de um *workflow*.

Dado que a imagem base de nome *activity-base-image* já tem suporte para *Java* (para executar o processo *Activity Engine*), para suportar a execução da *Task* desenvolvida em *Python* é necessário adicionar a instalação do ambiente de execução de *scripts Python* (*RUN apk add python3*).

```
1 FROM my-namespace/activity-base-image:latest
2 RUN apk add python3
3 RUN mkdir /var/tasks
4 ENV TASK_EXECUTABLE_DIRECTORY="/var/tasks"
5 COPY RampUpTask.jar /var/tasks/RampUpTask.jar
6 COPY AdditionTask.jar /var/tasks/AdditionTask.jar
7 COPY MultiplicationTask.py /var/tasks/MultiplicationTask.py
```

Listagem 6.2: *Dockerfile* de criação do *workflow* de operações aritméticas

A experimentação de execução do *workflow* foi realizada num único nó computacional, contendo todos os elementos de infraestrutura necessários, ou seja, o *Broker*, o *Activity Bucket*, e as *Activities* do *workflow*. Este nó computacional foi concretizado utilizando uma instância de uma máquina virtual (VM) criada na plataforma de serviços *Cloud*, *Google Cloud Platform* (GCP). O tipo da VM utilizada foi o *e2-standard-4* com 4 vCPU e uma memória de 16GB.

Esta experimentação permitiu validar a flexibilidade do modelo em relação ao suporte de execução de *Tasks* desenvolvidas em diferentes linguagens de programação, neste caso *Java* e *Python*.

Tal como foi adicionado o ambiente de execução de *scripts Python*, poderia ter sido instalado qualquer outro ambiente de execução, podendo-se assim afirmar que o modelo suporta a execução de *Tasks* desenvolvidas em qualquer linguagem de programação.

Esta experiência também salienta a possibilidade de execução de diferentes *Tasks* contidas na mesma imagem *Docker*, necessitando apenas, na especificação do *workflow*, da indicação do comando de execução com o nome do ficheiro executável. Para além da facilidade de manutenção, esta flexibilidade permite também, por exemplo, um programador copiar para a imagem várias versões da mesma *Task* com diferentes algoritmos. Assim é possível executar o *workflow* várias vezes de forma a averiguar qual o

melhor algoritmo, mudando apenas na especificação do *workflow* o nome do executável da *Task* nas diferentes execuções do *workflow*.

Embora este *workflow* seja muito simples do ponto de vista funcional, foi extremamente importante em todo o processo de refinamento e validação do protótipo desenvolvido.

6.2 *Workflow* de Longa Duração

Podemos considerar um *workflow* de longa duração em duas vertentes, ou é um *workflow* com um número muito elevado de iterações, por exemplo um *workflow* em permanente execução com infinitas iterações, ou um *workflow*, que embora possa ter poucas iterações, tem *Tasks* com tempos de processamento elevados.

Nesta experimentação foi desenvolvido o *workflow* que simula um *workflow* de longa duração com uma *Task* (*LongRunningTask*) que simula um tempo de processamento de *n* segundos, que permitiu avaliar os seguintes pontos do modelo:

- Especificação do padrão de *workflow* básico *Sequence*;
- Avaliação de tempos de execução do *workflow* e dos tempos de *overhead* decorrentes da inicialização e execução de *containers*, bem como das interações entre as *Activities* através do *Broker*;
- Considerando que um *workflow* tem múltiplas iterações, tirar partido de balanceamento de carga nas sucessivas iterações, explorando a especificação de réplicas de uma *Activity*, cuja *Task* tem um tempo de processamento elevado;
- Distribuição de *Activities* por múltiplos nós computacionais.

Na Figura 6.3 apresenta-se um *workflow* que consiste na sequência de três *Activities* de nome *A*, *B* e *C*. Cada *Activity* tem o seguinte comportamento: concatenar o resultado da *Activity* anterior, com o nome da própria *Activity* e o número de iteração corrente do *workflow*. No caso da *Activity C*, esta escreve os seus resultados num ficheiro de nome *Results.txt*.

Como exemplo consideremos a execução do *workflow* ao longo de 3 iterações: a *Task* da *Activity* A produz os resultados, *A(0)*, *A(1)* e *A(2)*, a *Task* da *Activity B* produz os resultados *A(0)B(0)*, *A(1)B(1)*, *A(2)B(2)*, e a *Activity C* produz os resultados *A(0)B(0)C(0)*, *A(1)B(1)C(1)* e *A(2)B(2)C(2)*. A *Activity C* escreve os resultados da sua execução no ficheiro de resultados (*Result.txt*) como se apresenta na Listagem 6.3.

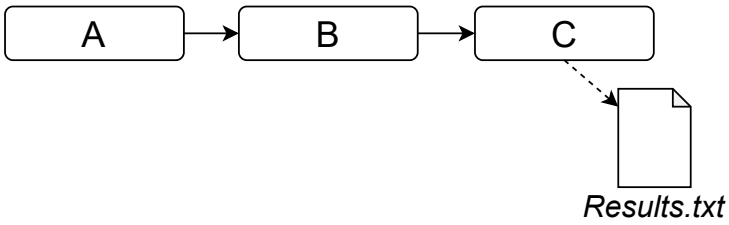


Figura 6.3: *Workflow* com padrão *Sequence* das *Activities A*, *B* e *C*

```
A(0)B(0)C(0)
A(1)B(1)C(1)
A(2)B(2)C(2)
```

Listagem 6.3: Resultados de execução da *Activity C* com 3 iterações

A forma de medição dos tempos de execução do *workflow* tira partido da informação de *Logs* realizados pelos vários *Activity Engine* durante a execução das *Activities*, como se indica no extrato da Listagem 6.4.

```
[2023-09-09 18:38:52.990|wkfName:LongRunningWorkflow|activityName:B|curIter
   ↪ :0]-Activity started
...
[2023-09-09 18:38:53.064|wkfName:LongRunningWorkflow|activityName:A|curIter
   ↪ :0]-Activity started
...
[2023-09-09 18:38:53.202|wkfName:LongRunningWorkflow|activityName:C|curIter
   ↪ :0]-Activity started
...
[2023-09-09 18:38:53.457|wkfName:LongRunningWorkflow|activityName:C|curIter
   ↪ :0]-Waiting for tokens...
...
[2023-09-09 18:38:53.765|wkfName:LongRunningWorkflow|activityName:A|curIter
   ↪ :0]-End current iteration.
...
[2023-09-09 18:38:55.185|wkfName:LongRunningWorkflow|activityName:C|curIter
   ↪ :0]-End current iteration.
...
[2023-09-09 18:38:56.070|wkfName:LongRunningWorkflow|activityName:C|curIter
   ↪ :2]-End Activity execution
```

Listagem 6.4: *Logs* de execução do *workflow* da Figura 6.3

Pressupondo que os vários relógios dos *containers* têm desvios negligenciáveis entre si, as informações de *Log* também permitem demonstrar a característica de controlo descentralizado de execução das várias tarefas de um *workflow*, pois como se pode

observar nos *Logs* da Listagem 6.4, a *Activity B* iniciou-se no *timestamp* "*18:38:52.990*", antes da *Activity A* no *timestamp* "*18:38:53.064*".

A partir dos *Logs* de execução de qualquer *workflow* é possível obter o tempo de execução do *workflow* subtraindo o *timestamp* de terminação da última *Activity* na última iteração pelo *timestamp* de início da primeira *Activity* na primeira iteração.

Para o caso do *workflow* da Figura 6.3 cujos extrato dos *Logs* se apresenta na Listagem 6.4, podemos verificar que o tempo total de execução do *workflow* é de $(18 : 38 : 56.070) - (18 : 38 : 53.064) = 3,006$ segundos.

6.2.1 Avaliação de Tempos de *Overhead*

Uma maneira prática de medir o *overhead* de execução do *workflow* é especificar para todas as *Activities* um tempo de execução da *Task* com o valor de zero segundos como apresentado na Figura 6.4. Teoricamente com esta especificação, o tempo de execução do *workflow* seria de zero segundos. No entanto, são expectáveis tempos de execução maiores que zero, decorrentes da inicialização e execução de *containers*, bem como das interações entre as *Activities* através do *Broker*.

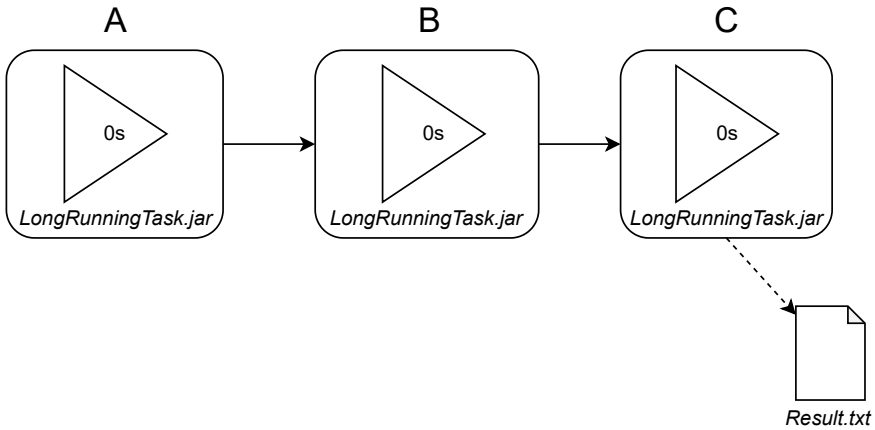


Figura 6.4: *Workflow* com tempo de execução teórico de 0 segundos

Como apresentado na Figura 6.5, a execução do *workflow* foi realizada num único nó computacional, contendo as três *Activities* constituintes do *workflow*, para além dos elementos de infraestrutura necessários, ou seja, o *Broker* e o *Activity Bucket*.

O nó computacional foi concretizado utilizando uma instância de uma máquina virtual (VM) criada na plataforma *Google Cloud Platform* (GCP). O tipo da VM utilizada foi o *e2-standard-4* com 4 vCPU e uma memória de 16GB.

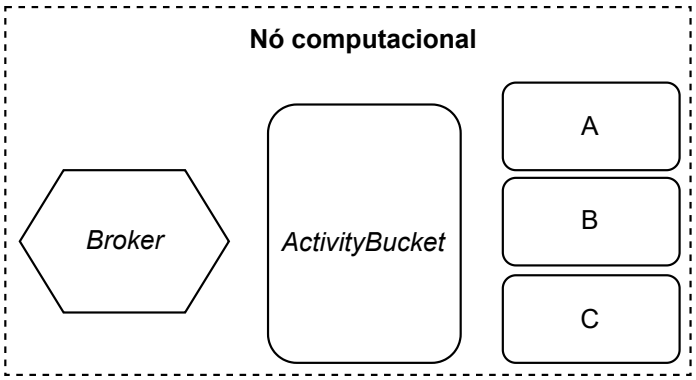


Figura 6.5: *Workflow* em execução num único nó computacional

O *workflow* foi executado com 10 iterações. Na Tabela 6.1 apresentam-se os tempos de execução do *workflow* nas sucessivas iterações.

O tempo decorrido de execução em segundos, representa o tempo de execução do *workflow* após cada iteração, isto é, a subtração do *timestamp* do *Log* "*End current iteration*" da *Activity C* com o *timestamp* do *Log* "*Activity started*" da *Activity A*. Após a primeira iteração, o tempo decorrido é de 2,226 segundos e após a segunda iteração é de 3,115 segundos e assim sucessivamente como se apresenta na Tabela 6.1.

O tempo médio de execução por cada iteração, foi calculado com base no tempo decorrido de execução a dividir pelo número de iteração corrente.

Tabela 6.1: Tempos de execução (segundos) do *workflow* com 10 iterações

Número de iteração corrente	**Tempo decorrido (seg.)**	**Tempo médio de execução a cada iteração (seg.)**
1	2,226	2,226
2	3,115	1,558
3	3,923	1,308
4	4,636	1,159
5	5,553	1,111
6	6,355	1,059
7	7,288	1,041
8	8,057	1,007
9	8,581	0,953
10	9,002	0,900

Dado que as *Tasks* têm tempos de processamento de zero segundos, os tempos médios de execução em cada iteração permitem-nos tirar conclusões sobre os *overheads* associados à inicialização da infraestrutura de suporte à execução e das interações entre as

várias *Activities*. Como se observa na Tabela 6.1, para um elevado número de iterações, os tempos de *overhead* vão-se diluindo ao longo da execução do *workflow* Nesta experimentação, para 10 iterações, o valor de execução médio por iteração, que representa um *overhead*, é inferior a um segundo (0,9 segundos), o que é negligenciável em cenários de *workflow* contendo *Tasks* tempos de execução elevados.

6.2.2 Avaliação e Otimização de um *Workflow* com *Tasks* de Tempo de Execução Elevado

Com o intuito de poder avaliar a especificação de réplicas para a *Activity B* e o balanceamento de carga nas sucessivas iterações, o *workflow* da Figura 6.3 foi executado múltiplas vezes variando o número de réplicas da *Activity B*, como apresentado na Figura 6.6.

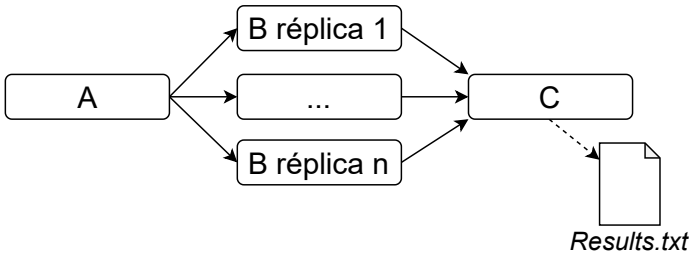


Figura 6.6: *Workflow* com balanceamento de carga entre réplicas da *Activity B*

Relativamente ao comportamento do *workflow*, a concatenação do nome da *Activity* com o valor da iteração corrente, permite-nos averiguar a correção da execução do *workflow* envolvendo múltiplas réplicas.

No caso do *workflow* ser executado com 5 iterações e a *Activity B* tenha 2 réplicas, uma das réplicas vai processar os *Tokens* marcados com as iterações 0, 2 e 4 e a outra réplica os *Tokens* marcados com as iterações 1 e 3.

Para averiguar que o *workflow* se executou corretamente tirando partido de réplicas, basta observar a informação do ficheiro de resultados gerado pela *Task* da *Activity C* e verificar se o nome da réplica da *Activity B* é o esperado para cada iteração, como se apresenta na Listagem 6.5.

```
A(0)B_replica_1_of_2(0)C(0)
A(1)B_replica_2_of_2(1)C(1)
A(2)B_replica_1_of_2(2)C(2)
A(3)B_replica_2_of_2(3)C(3)
A(4)B_replica_1_of_2(4)C(4)
```

Listagem 6.5: Resultado de execução com 5 iterações e 2 réplicas na *Activity B*

De forma a efetuar uma apreciação do desempenho da implementação do modelo ao executar *Long Running Workflows* com *Tasks* com tempo de duração elevado, foi especificado um tempo de duração das *Tasks* das *Activities A* e *C* com o valor de zero segundos, e para a *Activity B* foi especificado o valor de dez segundos como apresentado na Figura 6.7.

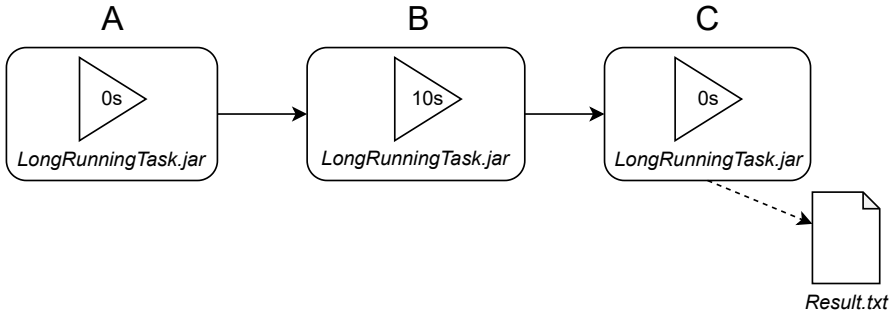


Figura 6.7: *Workflow* onde a *Activity B* tem um longo tempo de execução

Teoricamente, o tempo de execução do *workflow* com uma única iteração seria de 10 segundos. Para 10 iterações, o *workflow* teria um tempo de execução de 100 segundos. Num cenário ideal, recorrendo à especificação de réplicas na *Activity B* como apresentado na Figura 6.8, poderíamos afirmar que para *n* iterações com *n* réplicas, o tempo global de execução do *workflow* seria igual ao tempo de execução de uma única iteração do *workflow*. Por exemplo, para 10 iterações e 10 réplicas, o tempo de execução do *workflow* seria aproximadamente de 10 segundos.

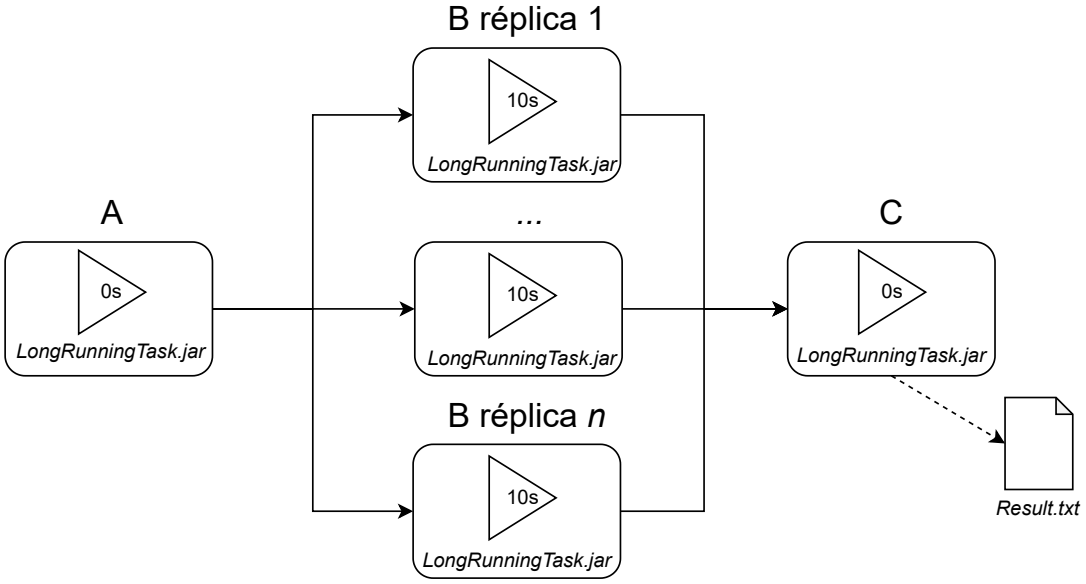


Figura 6.8: *Workflow* com *n* réplicas na *Activity B*

Numa primeira experimentação considerou-se a execução do *workflow* num único nó computacional, o mesmo utilizado para a avaliação de *overheads* (Subsecção 6.2.1), isto é, todas as *Activities*, o *Broker* e o *Activity Bucket* executam-se na mesma máquina virtual.

O *workflow* foi executado com três especificações diferentes, variando o número de iterações para 10, 20 e 30. Para cada especificação do número de iterações, o *workflow* foi executado com os números de réplicas 1, 2, 5 e 10 da *Activity* B. Os resultados das execuções estão representados na Tabela 6.2 a que corresponde o gráfico na Figura 6.9.

Tabela 6.2: Tempos de execução (segundos) do *workflow* com réplicas da *Activity B*

			Nº de iterações		
			10	**20**	**30**
Nº de réplicas	**1**	**Total**	107,079	214,107	321,650
		Overhead	7,079	14,107	21,650
	2	**Total**	56,190	110,763	165,467
		Overhead	6,190	10,763	15,467
	5	**Total**	25,596	50.494	73,436
		Overhead	5,596	10,494	13,436
	10	**Total**	21,351	31,468	44,770
		Overhead	11,351	11,468	14,77

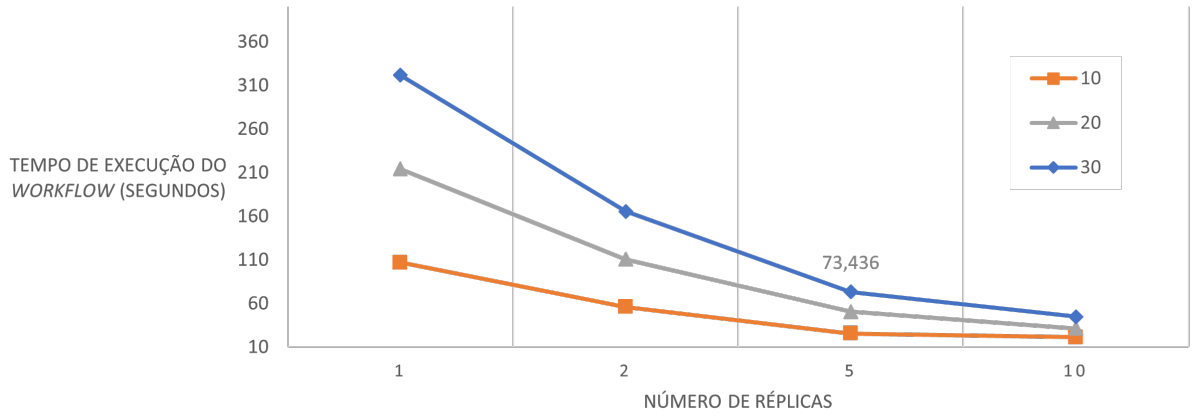


Figura 6.9: Evolução dos tempos de execução do *workflow* em função do número de réplicas

Na Tabela 6.2 e também no gráfico da Figura 6.9, podemos analisar os tempos de execução do *workflow* e de *overheads* associados para cada experimentação, permitindo concluir que é possível reduzir o tempo de execução do *workflow* recorrendo à utilização de réplicas em *Tasks* com tempos de execução longos. No entanto, apenas se observaram melhorias significativas até à utilização de cinco réplicas, observando-se

melhorias menos acentuadas entre as execuções de 5 a 10 réplicas, que pode ser justificado fazendo um estudo do *speedup* para a execução do *workflow*.

Para o cálculo de *speedup* utilizou-se a Expressão (6.3):

$$Speedup = (tempo\ de\ execução\ para\ 1\ réplica)/(tempo\ de\ execução\ para\ n\ réplicas) \quad (6.3)$$

Assim, os valores de *speedup* são:

- 1,90 para duas réplicas, onde o valor ideal teórico seria 2;
- 4,18 para cinco réplicas, onde o valor ideal teórico seria 5;
- 5,01 para dez réplicas, onde o valor ideal teórico seria 10.

Estes valores de *speedup* confirmam o estipulado pela lei de *Amdahl* [49] que estabelece que para diferentes configurações de execução em paralelo, neste caso a *Activity B*, o *speedup* é limitado pelas partes não paralelizáveis, neste caso a *Activity A* e a *Activity C*, bem como a sobrecarga dos recursos computacionais num único nó.

Nos resultados da experimentação da Tabela 6.2 podemos observar que para a execução do *workflow* com 10 iterações e uma única réplica, o tempo de *overhead* é de 7,079 segundos, o que é comparável com o valor de 9,01 segundos obtido na experimentação de cálculo de *overhead* onde não existiam réplicas e a *Activity B* tinha um tempo de execução de zero segundos. Esta diferença pode ser justificada pela sobreposição em paralelo dos tempos de execução de todas as iterações da *Activity A*, enquanto a *Activity B* ainda está a executar uma única iteração com o tempo de processamento de 10 segundos.

A segunda experimentação, para as mesmas condições de número de iterações e o número réplicas, consistiu em usar 4 nós computacionais na plataforma de serviços *Cloud*, *Google Cloud Platform* (GCP) com o tipo da máquina virtual (VM) *e2-standard-4* com 4 vCPU e uma memória de 16GB. Um dos nós foi apenas dedicado à execução do *Broker*, e os restantes três nós dedicados à execução dos *Activity Buckets* e das *Activities*, incluindo as réplicas da *Activity B*.

O *workflow* foi novamente executado com três variações do número de iterações: 10, 20 e 30 e para cada variação do número de iterações, com réplicas da *Activity B* de 1, 2, 5 e 10. Os resultados das execuções estão representados na Tabela 6.3 e no gráfico da Figura 6.10.

Tabela 6.3: Tempos de execução (segundos) do *workflow* com réplicas da *Activity B*

			Nº de iterações		
			10	**20**	**30**
Nº de réplicas	**1**	**Total**	104,846	208,781	313,025
		Overhead	4,846	8,781	13,025
	2	**Total**	53,262	105,726	157,458
		Overhead	3,262	5,726	7,458
	5	**Total**	23,763	44,900	66,248
		Overhead	3,763	4,900	6,248
	10	**Total**	16,528	26,831	38,221
		Overhead	6,528	6,831	8,221

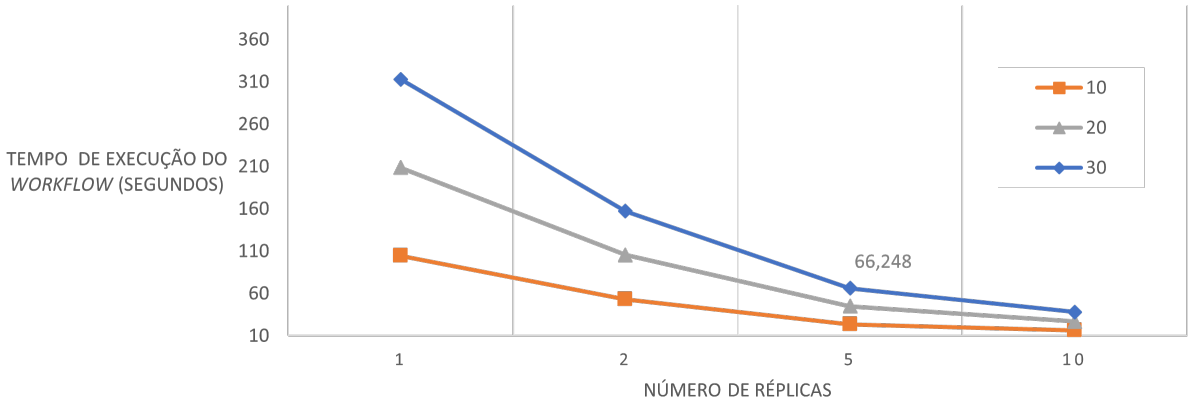


Figura 6.10: Evolução dos tempos de execução do *workflow* em função do número de réplicas

Comparando a primeira experimentação (apenas num nó computacional), com esta experimentação, onde são utilizados quatro nós de computação, podemos observar que os tempos de execução são inferiores. Por exemplo, para 30 iterações e 5 réplicas, temos um tempo de execução de 66,248 segundos quando na experimentação anterior obtivemos um tempo de execução de 73,436 segundos.

Podemos afirmar que a execução do *workflow* beneficiou com o aumento do número de nós computacionais. Apesar desta melhoria, tal como nos resultados anteriores, notamos que a partir das execuções com cinco réplicas, não existem melhorias tão acentuadas para um maior número de réplicas. Podemos novamente confirmar esta afirmação fazendo um estudo do *speedup* para a execução com dez iterações e com duas, cinco e dez réplicas. Os valores de *speedup* são:

- 1,96 para duas réplicas, onde o valor ideal teórico seria 2;
- 4,41 para cinco réplicas, onde o valor ideal teórico seria 5;

- 6,34 para dez réplicas, onde o valor ideal teórico seria 10.

Podemos observar que não houve diferenças significativas nos valores de *speedup* para duas e cinco réplicas, tendo sido calculado um valor de 1,90 e 4,18 no primeiro lote de experimentação. No entanto, a utilização de vários nós computacionais permitiu uma melhoria no valor de *speedup* para dez réplicas, passando de 5.01 na primeira experimentação para o *speedup* de 6.34, que podemos justificar como sendo consequência de cada nó ter menos sobrecarga de processamento permitindo uma aceleração das partes não paralelizáveis do *workflow*, neste caso a *Activity A* e *Activity C*.

De seguida, descreve-se o desenvolvimento de uma *Task* de nome *LongRunningTask* envolvida nas experimentações. Esta *Task* é genérica para poder ser utilizada nas *Activities* A, B e C, como se indica a seguir:
argumentos: *string* com um valor de texto a ser concatenado no resultado da *Task*;
constantes:i) Valor entre 0 e *n* representando o número de segundos em que a *Task* cessa temporariamente a execução, simulando tempos de execução elevados e ii) *string* com o nome de um ficheiro a ser escrito com o resultado da *Task* (só aplicável ao caso da *Activity C*);
comportamento: concatenar as diversas *strings*: valor recebido como argumento; nome da *Activity* e o número da iteração corrente. Se existir a constante com o nome de um ficheiro para escrita de resultados, a concatenação de *strings* é acrescentada nesse ficheiro (*write append*);
resultado: *string* contendo o resultado da concatenação.

Para desenvolver a experimentação do *workflow*, foi criada uma imagem *Docker* cujo ficheiro *Dockerfile* se apresenta na Listagem 6.2.

```
1 FROM my-namespace/activity-base-image:latest
2 RUN mkdir /var/task
3 ENV TASK_EXECUTABLE_DIRECTORY="/var/task"
4 COPY  LongRunningWorkflow.jar /var/task/LongRunningWorkflowTask.jar
```

Listagem 6.6: *Dockerfile* de criação do *workflow* de longa duração

Como ilustração de uma especificação completa de um *workflow*, apresenta-se no Anexo A, em formato JSON, um exemplo usado nesta experimentação.

6.3 *Workflow* de Processamento de Imagens

De forma a poder avaliar um *workflow* que interaja com serviços externos durante a sua execução, foi desenvolvido o *workflow* apresentado na Figura 6.11, que tem como

objetivo processar um conjunto de imagens com formato *jpeg* ou outro, existentes no sistema de ficheiros distribuídos (diretoria */data/sharedDirectory* associada à *workingDirectory* dos *containers*).

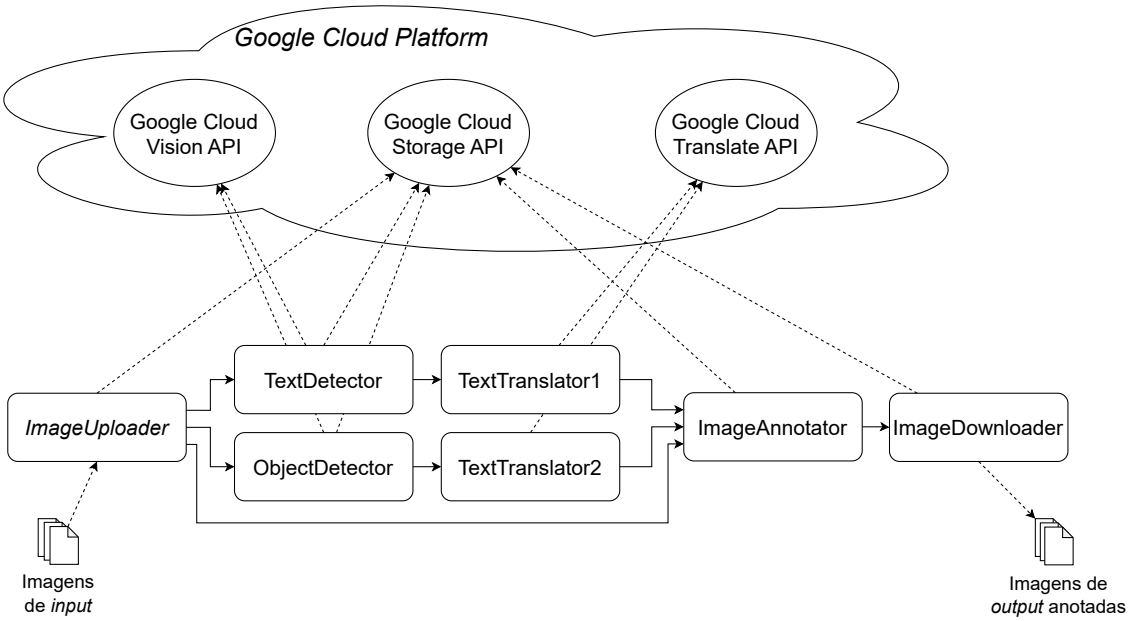


Figura 6.11: *Workflow* para deteção de objectos e textos em imagens

Em cada iteração o *workflow* processa uma imagem, pelo que, o número de iterações é igual ao número de imagens a serem processadas. O *workflow* deteta objetos (*Activity ObjectDetector*) e textos (*Activity TextDetector*) que existam nas imagens. No final (*Activity ImageAnnotator*), são produzidas novas imagens, com o formato *jpeg*, marcadas com um polígono delimitador de área e a identificação num determinado idioma, por exemplo português ou inglês, do nome dos objetos e dos textos encontrados.

As *Activities* do *workflow* para encontrar objetos e texto na imagem recorrem ao serviço *Vision API* da *Google Cloud Platform* (GCP).

O *workflow* tem inicialmente a *Activity ImageUploader* e no final a *ImageDownloader* que são responsáveis por transferir imagens entre a *workingDirectory* dos respetivos *containers* e o serviço de armazenamento de ficheiros *Cloud Storage* da *Google Cloud Platform* (GCP).

Para testar o *workflow* foram realizadas várias experimentações com múltiplas imagens que contêm objetos e/ou texto. Na Figura 6.12 ilustram-se do lado esquerdo os exemplos de imagens antes do processamento e do lado direito as imagens resultantes da execução do *workflow* com as deteções traduzidas para o idioma inglês, bem como os respetivos polígonos delimitadores de área.

Figura 6.12: Imagens antes e depois de serem processadas pelo workflow

De seguida descreve-se os pressupostos e comportamento das *Tasks* envolvidas nas múltiplas *Activities*:

❖***ImageUploaderTask***, utilizada pela *Activity ImageUploader*:
argumentos: nenhum;
constantes: *string* com um *path* relativo à *workingDirectory* do *workflow*, com as imagens a serem processadas, por exemplo "*/input/image_%.jpg*";
comportamento: substitui o símbolo % do *path*, com o número de iteração corrente de forma a obter o nome da imagem de *input*. Por exemplo, para a terceira iteração o ficheiro terá o *path* "*/input/image_3.jpg*". É gerado um *Google Cloud Storage URI* (GSURI)

e depois o ficheiro identificado com este GSURI é enviado para o serviço *Cloud Storage* através da API *google-cloud-storage*;
resultado: *string* contendo o GSURI gerado que será utilizado nas *Activities* seguintes para identificar a imagem já armazenada no *Google Cloud Storage*.

❖***TextDetectorTask***, utilizada pela *Activity TextDetector*:
argumentos: *string* contento um GSURI de uma imagem;
constantes: nenhuma;
comportamento: com o GSURI recebido como argumento, realiza um pedido à API *google-cloud-vision* para encontrar texto na imagem. Transforma o resultado do pedido à API numa lista de anotações. Cada anotação contém as seguintes informações: i) uma *string* com o texto detetado; ii) um valor decimal representando a percentagem de confiança na deteção do texto; iii) lista de vértices, representando o polígono delimitador onde foi encontrado o texto na imagem;
resultado: *string* em formato JSON com a serialização da lista de anotações.

❖***ObjectDetectorTask***, utilizada pela *Activity ObjectDetector*:
argumentos: *string* contento um GSURI de uma imagem;
constantes: nenhuma;
comportamento: com o GSURI recebido como argumento, realiza um pedido à API do *google-cloud-vision* para encontrar objetos na imagem. Transforma o resultado do pedido à API numa lista de anotações. Cada anotação contém as seguintes informações: i) uma string com o nome do objeto detetado; ii) um valor decimal representando a percentagem de confiança na deteção do objeto; iii) lista de vértices, representando o polígono delimitador onde foi encontrado o objeto na imagem;
resultado: *string* em formato JSON com a serialização da lista de anotações.

❖***TextTranslatorTask***, utilizada pelas *Activities TexTranslator1* e *TexTranslator2*:
argumentos: *string* em formato JSON com a serialização de uma lista de anotações. Cada anotação contém uma *string* (*label*) que é o texto detetado ou o nome de um objeto detetado;
constantes: *string* contendo o código do idioma para o qual será feita a tradução;
comportamento: recorrendo à API do *google-cloud-translate*, por cada anotação, seja de texto ou o nome de um objeto, é feito um pedido de tradução do *label* para o idioma cujo código é recebido como constante;
resultado: *string* em formato JSON com a serialização da lista de anotações devidamente traduzidas.

❖***ImageAnnotatorTask***, utilizada pela *Activity ImageAnnotator*:
argumentos: i) GSURI da imagem existente no *Google Cloud Storage*; ii) *string* em formato JSON com a serialização da lista de anotações de texto detetado e já traduzido; iii) *string* em formato JSON com a serialização da lista de anotações de objetos detetados com o nome dos mesmos já traduzido;
constantes: nenhuma;
comportamento: por cada deteção, é desenhando um polígono em torno de cada objeto ou de cada texto, marcados com um identificador no idioma pretendido. A nova imagem anotada é armazenada no *Google Cloud Storage* com o nome da imagem inicial concatenado com o prefixo "*annotated_*";
resultado: string contendo o novo GSURI do Google Cloud Storage.

❖***ImageDownloaderTask***, utilizada pela *Activity ImageDownloader*:
argumentos: *string* contendo o GSURI de uma imagem anotada;
constantes: *string* com um *path* relativo à *workingDirectory* do *workflow* com as imagens de *output*, por exemplo "*/output/annotated_image_%.jpg*";
comportamento: substitui o símbolo % do *path*, com o número de iteração corrente de forma a obter o *path* de *output* para a imagem. Por exemplo, para a terceira iteração o ficheiro terá o *path* "*/output/annotated_image_ 3.jpg*". A partir do *Google Cloud Storage* realiza o *download* da imagem, cujo GSURI é recebido por argumento, para um ficheiro de acordo com o *path* indicado como constante;
resultado: nenhum;

Para desenvolver a experimentação deste *workflow*, foi criada uma imagem *Docker* cujo ficheiro *Dockerfile* se apresenta na Listagem 6.7.

```
1  FROM my-namespace/activity-base-image:latest
2  #install letter font in order to annotate the images
3  RUN apk --no-cache add msttcorefonts-installer fontconfig \
4      && update-ms-fonts && fc-cache -f
5  RUN mkdir /var/task
6  ENV TASK_EXECUTABLE_DIRECTORY="/var/tasks"
7  ENV GOOGLE_APPLICATION_CREDENTIALS="/var/workingDirectory/gcpCredentials/
       ↪ gcp_key.json"
8  COPY  ImageUploaderTask.jar /var/tasks/ImageUploaderTask.jar
9  COPY  TextDetectorTask.jar /var/tasks/TextDetectorTask.jar
10 COPY  ObjectDetectorTask.jar /var/tasks/ObjectDetectorTask.jar
11 COPY  TextTranslationTask.jar /var/tasks/TextTranslationTask.jar
12 COPY  ImageAnnotatorTask.jar /var/tasks/ImageAnnotatorTask.jar
13 COPY  ImageDownloaderTask.jar /var/tasks/ImageDownloaderTask.jar
```

Listagem 6.7: *Dockerfile* de criação do *workflow* de processamento de imagens

A imagem vai servir de base para executar qualquer uma das *Activities* especificadas. Para ilustrar a possibilidade de instalar *packages* necessários à execução de uma *Task*, considerou-se neste caso que a *Task ImageAnnotatorTask* recorre a fontes não instaladas na imagem base. Assim é necessário instalar um *package* com as fontes como se ilustra na Listagem 6.7. Dado que as *Tasks* recorrem aos serviços da GCP (*Storage*, *Translate* e *Vision API*), é também necessário definir a variável de ambiente *GOOGLE_APPLICATION_CREDENTIALS* para indicar o ficheiro JSON que contém as credenciais de autorização para acesso aos serviços.

A experimentação foi realizada em quatro nós computacionais. Um dos nós foi dedicado apenas para a execução do *Broker*, e os restantes três nós com as *Activities* do *workflow*. Os nós computacionais foram concretizados utilizando instâncias de máquinas virtuais (VM) criadas na plataforma de serviços *Cloud*, GCP, com o tipo *e2-standard-4* com 4 vCPU e uma memória de 16GB.

Esta experimentação permitiu validar a generalidade do modelo no desenvolvimento de *Tasks* com recurso a serviços externos na *Cloud*. Para além disso este exemplo permitiu testar e validar a flexibilidade da estrutura composta de *Tokens* trocados entre *Activities*, através do *Broker*. Como se descreveu na passagem da lista de anotações entre *Activities*.

Embora fosse sedutor experimentar com um maior número de imagens, os custos monetários associados à utilização de mais nós computacionais, bem como das chamadas à *Google Vision API*, limitaram a experimentação do *workflow* para uma maior escala.

6.4 *Workflow* de Mineração de Texto

De forma a poder avaliar a possibilidade de desenvolver um *workflow* segundo o modelo *MapReduce* foi desenvolvido o *workflow* de mineração de texto apresentado na Figura 6.13. Este *workflow* com uma única iteração, processa o ficheiro de texto *input.txt* para contar a frequência de palavras nele existentes. Considera-se uma palavra sempre em minúsculas, separada por espaços, como uma sequência de caracteres alfanuméricos.

A primeira *Activity* (*Splitter*) vai dividir as linhas do ficheiro *Input.txt* numa quantidade de partes (*splits*) correspondente ao número de *Activities Mapper*, neste caso 5. Cada *Activity Mapper* recebe do *Splitter*, um intervalo de linhas a processar no ficheiro *Input.txt*. Cada palavra encontrada é copiada para um ficheiro intermédio (por exemplo *Mapper1.txt*) como um par *(<palavra>, 1)*. Por exemplo, a palavra "*casa*" gera uma nova linha no ficheiro intermédio com o par *(casa, 1)*. O *Mapper* não executa contagens

nem agrupa palavras, podendo o ficheiro intermédio conter várias linhas com pares repetidos.

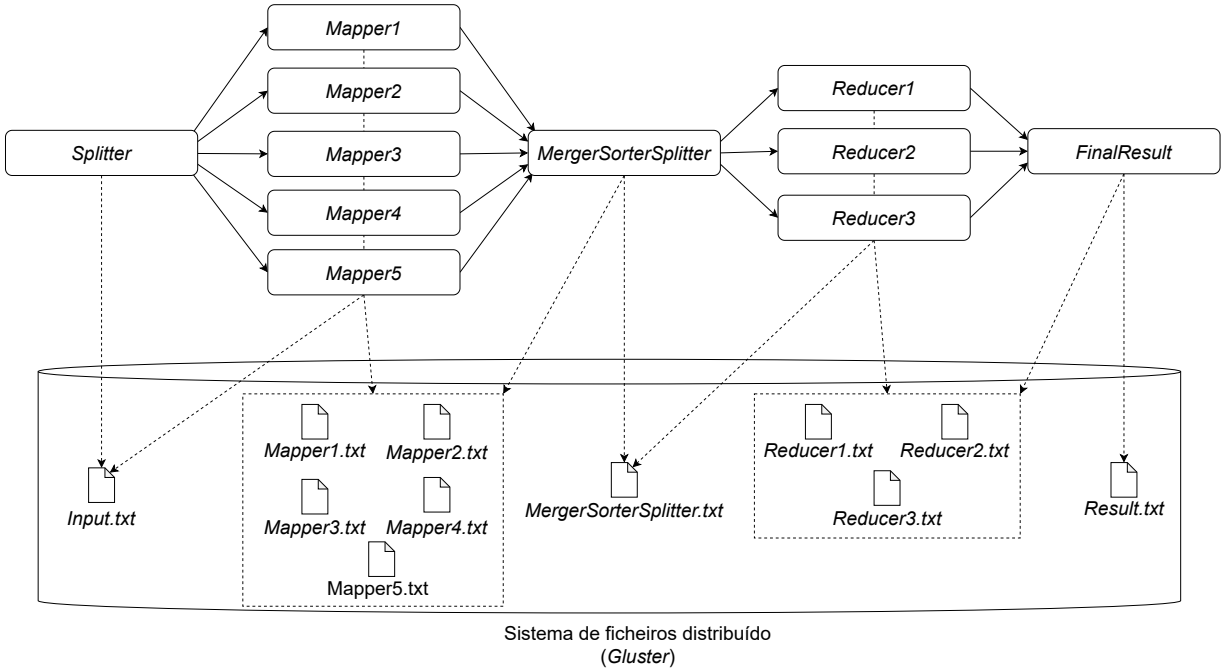


Figura 6.13: *Workflow* de mineração de texto utilizando o modelo *MapReduce*

A *Activity MergerSorterSplitter* faz a junção dos pares existentes em todos os ficheiros intermédios gerados pelas *Activities Mapper*, gerando um novo ficheiro intermédio com os pares ordenados alfabeticamente, e cria partições alfabéticas em função do número de *Activities Reducer*, neste caso 3.

Cada *Reducer* vai processar o ficheiro gerado pela *Activity MergerSorterSplitter* e dentro da partição alfabética que recebeu como argumento, faz a redução, isto é, a contagem de pares da mesma palavra, escrevendo um par *(<palavra>, <contador>)* num novo ficheiro intermédio (por exemplo *Reducer1.txt*). A *Activity FinalResult* faz a junção dos ficheiros gerados pelas *Activities Reducer* num ficheiro final (*Result.txt*) e depois ordena este ficheiro por ordem decrescente da frequência das palavras.

Para testar a execução do *workflow* foi utilizado um ficheiro *input.txt* com 3600 linhas, contendo a história "*Alice no país das maravilhas*" no idioma inglês.

Através do ficheiro de resultados intermédio gerado pela *workflow*Task da *Activity MergerSorterSplitter*, verificou-se que no total foram detetados 26561 pares.

Na Listagem 6.8, apresenta-se o resultado de execução do *workflow* com um excerto do ficheiro *Result.txt*.

```
(the, 1632)
```

```
(and, 845)
...
(alice, 386)
...
(don't, 61)
...
(turtle,57)
...
(rabbit,43)
...
(zigzag,1)
```

Listagem 6.8: Extrato do ficheiro *Result.txt* resultante da execução do *workflow*

Este *workflow*, composto por 11 *Activities*, necessitou do desenvolvimento das seguintes *Tasks*:

❖***SplitTask***, utilizada pela *Activity Splitter*:
argumentos: nenhum;
constantes: nome do ficheiro a processar e um número *n*, representando o número de partes (*splits*) para dividir o ficheiro;
comportamento: conta o número de linhas do ficheiro a processar e divide em *n splits*;
resultado: lista de *splits*, em que cada um é descrito como uma *string* com o formato *alice_in_wonderland.txt,1,721*, isto é, o nome do ficheiro a processar, linha inicial e linha final.

❖***MapTask***, utilizada pelas 5 *Activities Mapper*:
argumentos: *string* com o *split* de linhas do ficheiro a ser processado, por exemplo, *alice_in_wonderland.txt,1,721*;
constantes: nenhuma;
comportamento: Percorre o ficheiro de *input* dentro do intervalo indicado, e por cada palavra encontrada escreve o par *(<palavra>,1)* num ficheiro intermédio nomeado com o nome da *Activity* com a extensão "*.txt*". Por exemplo, se a *Activity Mapper1*, encontrou a linha "*Era uma vez*", vai ser criado um novo ficheiro de nome *Mapper1.txt* com os pares *(Era, 1)*, *(uma, 1)* e *(vez, 1)*, com um par em cada linha do ficheiro;
resultado: *string* contendo o nome do ficheiro onde foram guardados os pares, por exemplo *Mapper1.txt*.

❖***MergeSortSplitTask***, utilizada pela *Activity MergerSortedSplitter*:
argumentos: lista de nomes de ficheiros onde existem os pares gerados pelos *Mappers*;
constantes: número inteiro *n* representando o número de partições alfabéticas que é em função do número de *Reducers*;

comportamento: realiza a junção dos pares contidos em todos os ficheiros intermédios produzidos pelos *Mappers*. Num novo ficheiro intermédio com o nome *MergerSorterSplitter.txt* são escritos os pares por ordem alfabética. Por fim cria *n* partições alfabéticas como um intervalo entre dois caracteres. Por exemplo entre *f* e *m*;
resultado: lista de intervalos representada como uma *string* no formato: nome do ficheiro criado; letra inicial do intervalo e letra final do intervalo, por exemplo, *MergerSorterSplitter.txt,f,m*.

❖***ReduceTask***, utilizada pelas 3 *Activities Reducer*:
argumentos: *string* com: nome do ficheiro intermédio criado pela *Activity MergerSortedSplitter* (*MergerSorterSplitter.txt*); letra inicial do intervalo e letra final do intervalo. Por exemplo, *MergerSorterSplitter.txt,f,m*;
constantes: nenhuma;
comportamento: conta os pares da mesma palavra existentes na partição alfabética do ficheiro indicado nos argumentos, criando um novo par *(<palavra>, <contador>)* que representa a frequência da *<palavra>* no ficheiro inicial, escrevendo esse par num novo ficheiro nomeado com o nome da *Activity* e com a extensão "*.txt*", por exemplo, *Reducer1.txt*. Por exemplo, se no ficheiro *MergerSorterSplitter.txt* existirem os pares *(alice, 1)*, *(alice, 1)* e *(alice, 1)*, é escrito o par *(alice, 3)* no ficheiro *Reducer1.txt*;
resultado: *string* contendo o nome do novo ficheiro criado, por exemplo *Reduce1.txt*.

❖***FinalResultTask***, utilizada pela *Activity FinalResult*:
argumentos: lista de *strings* contendo os nomes dos ficheiros produzidos pelas *Activities Reducer*;
constantes: *string* com o nome do ficheiro para serem escritos os resultados finais (*Result.txt*);
comportamento: realiza a junção de todos os ficheiros indicados em argumentos, e cria um novo ficheiro cujo nome é indicado em constante (*Result.txt*). O conteúdo do ficheiro é ordenado de forma decrescente pela frequência de cada palavra;
resultado: nenhum.

Para criar a imagem *Docker* de suporte à execução de todas as *Activities* do *workflow*, indica-se na Listagem 6.9 o ficheiro *Dockerfile*.

A experimentação foi realizada em quatro nós computacionais. Um dos nós foi dedicado apenas para a execução do *Broker*, e os restantes três nós para execução dos *Activity Bucket* e as *Activities* do *workflow*. Cada nó computacional foi concretizado com uma máquina virtual (VM) instânciada na plataforma *Google Cloud Platform* (GCP), com o tipo *e2-standard-4* com 4 vCPU e uma memória de 16GB.

```
1 FROM my-namespace/activity-base-image:latest
```

```
2 RUN mkdir /var/tasks
3 ENV TASK_EXECUTABLE_DIRECTORY="/var/tasks"
4 COPY  SplitTask.jar /var/tasks/SplitTask.jar
5 COPY  MapTask.jar /var/tasks/MapTask.jar
6 COPY  MergeSortSplitTask.jar /var/tasks/MergeSortSplitTask.jar
7 COPY  ReduceTask.jar /var/tasks/ReduceTask.jar
8 COPY  FinalResultTask.jar /var/tasks/FinalResultTask.jar
```

Listagem 6.9: *Dockerfile* de criação do *workflow* de mineração de texto

Esta experimentação permitiu validar a flexibilidade do modelo na especificação e execução de *workflows* segundo o modelo de *MapReduce*. Permitiu também avaliar, de forma mais exaustiva, o funcionamento da execução do *workflows* em conjunto com o sistema de ficheiros distribuídos (*Gluster*) que permite partilhar ficheiros entre os nós computacionais, nomeadamente os ficheiros intermédios criados pelas diferentes *Activities* do *workflow*.

Esta experimentação permitiu validar com sucesso a estratégia de indicar nos *Tokens* únicamente os nomes de ficheiros intermédios existentes no sistema de ficheiros distribuídos, mostrando eficiência em casos de partilha de grandes volumes de dados entre as diferentes *Activities*.

O *workflow* da Figura 6.13 podia ter sido desenvolvido com múltiplas iterações para processar mais do que um ficheiro, adotando uma estratégia idêntica à experimentação com o *workflow* do processamento de imagens descrita na Secção 6.3, mas por questões de simplicidade e de mais fácil explicação de resultado, a execução foi limitada apenas ao processamento de um ficheiro.

6.5 Resumo

Neste capítulo foram feitas diversas experimentações com a execução de vários *workflows* de forma a avaliar as características do modelo proposto e do protótipo implementado, como se indica a seguir:

- Suporte de padrões de *workflow* básicos como *Sequence*, *Parallel Split* e *Synchronization*, na especificação de *workflows* segundo o modelo proposto;
- Grande flexibilidade no desenvolvimento de *Tasks* em diferentes linguagens de programação;

- Agrupamento de várias *Tasks* na mesma imagem *Docker*, podendo na especificação do *workflow* definir o nome do executável da *Task* e o seu comando de execução;
- Especificação e execução de *workflows* com múltiplas iterações;
- Tempos de *overhead* bastante aceitáveis para cenários reais;
- Melhoria dos tempos de execução de um *workflow*, especificando réplicas de *Activities* de longa duração, para balanceamento de carga nas sucessivas iterações;
- Execução de *Activities* em diferentes nós computacionais, tirando partido de processamento paralelo e distribuído;
- Desenvolvimento de *Tasks* com recurso a serviços externos, por exemplo serviços disponibilizados em *Clouds* públicas;
- Flexibilidade na estrutura dos *Tokens* entre *Activities*, permitindo a utilização de valores simples, ou de estruturas compostas no formato JSON;
- Especificação e execução de *workflows* segundo o modelo *MapReduce*;
- Transmição de grande volume de dados entre *Activities*, através de um sistema de ficheiros distribuídos (*Gluster*).

7

Conclusões

Neste capítulo apresentam-se as conclusões finais sobre a realização do Trabalho Final de Mestrado (TFM), que propõe um modelo de execução de *workflows* em infraestruturas de computação distribuídas suportadas em máquinas virtuais e *containers*. Adicionalmente é proposta uma arquitetura que suportou a implementação de um protótipo funcional e operacional que permitiu a experimentação de casos concretos de aplicações moduladas segundo o paradigma de *workflow*.

Na Secção 7.1 resumem-se as conclusões da relevância do trabalho realizado relativamente à atualidade e importância do tópico em que se enquadra.

Na Secção 7.2 resumem-se as principais características do modelo proposto.

Na Secção 7.3 resumem-se as principais características da arquitetura de suporte à implementação do modelo.

Na Secção 7.4 resumem-se as principais características do protótipo de experimentação desenvolvido.

Na Secção 7.5 resumem-se as conclusões da experimentação e validação com casos concretos de aplicações moduladas como *workflows*.

Na Secção 7.6 apresentam-se alguns aspetos considerados importantes para trabalho futuro.

7.1 Relevância do Trabalho Realizado

A realização de pesquisa e estudo dos trabalhos relacionados salientou que o tópico deste trabalho se mantém ainda bastante relevante e que existem ainda questões em aberto no tema de desenvolvimento e execução de *workflows*, pelo que o trabalho teve como objetivo a resposta a algumas destas questões. Com forte inspiração no modelo AWARD [22], que propôs algumas soluções para as questões em aberto, o trabalho realizado recorre a tecnologias mais recentes, tais como, a especificação de um *workflow* utilizando uma linguagem de representação de dados mais atual como o JSON, *middlewares* seguindo o modelo *publish/subscribe* e o suporte à virtualização através de *containers*. Tendo estes aspetos em consideração, foi concetualizado um modelo de *workflow*, uma arquitetura de suporte à implementação desse modelo e realizado um protótipo de experimentação que permitiu executar aplicações concretas, moduladas como *workflows*.

7.2 Modelo de *Workflow* Proposto

O modelo proposto foi concetualizado de forma descomprometida de qualquer tecnologia e em síntese tem as seguintes características:

- Considerando um *workflow* como um grafo, no modelo cada vértice é designado como uma *Activity* genérica com *Input Ports* e *Output Ports*, executando uma *Task* de acordo com o domínio da aplicação. Os arcos entre vértices são suportados por um mecanismo de comunicação designado por *Channel* que transporta *Tokens* entre duas *Activities*. Um *Token* é um objeto que contém dados compostos por um ou mais campos, num formato de acordo com as especificidades da aplicação modelada como *workflow* e informações de controlo;

- O modelo não limita a execução de *Activities* a nenhum ambiente computacional específico, permitindo-as serem executadas em qualquer tipo de infraestrutura distribuída baseada em tecnologias de virtualização, tais como máquinas virtuais e *containers*. Esta flexibilidade de execução de *Activities* permite que estas sejam executadas em múltiplos nós de computação heterogéneos em termos de capacidades de processamento e de memória;

- O modelo garante um controlo de execução de *workflows* descentralizado. Este objetivo é cumprido através da definição do controlo de uma *Activity* como sendo

autónomo, permitindo às *Activities* executarem-se automaticamente quando receberem *Tokens* em todos os seus *Input Ports*;

- O fluxo de *Tokens* entre as *Activities* de um *workflow* é suportado através de um mecanismo intermediário de comunicação (*Broker*), seguindo o modelo *publish/-subscribe*, permitindo assim a comunicação entre *Activities* de forma indireta e assíncrona;
- O conteúdo de um *Token* é transparente ao modelo, isto é, depende apenas das especificidades da aplicação a ser concretizada como *workflow*;
- O modelo permite a especificação de *workflows* com padrões de *workflow* básicos;
- O modelo permite a execução de *workflows* com múltiplas ou infinitas iterações;
- O modelo permite a utilização de réplicas de *Activities* com longos tempos de execução, suportando assim balanceamento de carga entre essas réplicas ao longo das sucessivas iterações;
- A especificação de *workflows* e o desenvolvimento aberto de *Tasks* é desacoplado de aplicações concretas de um determinado domínio das ciências ou de linguagens de programação, permitindo ao programador focar-se na implementação dos algoritmos de cada *Task* associada a cada *Activity*, sem ter de conhecer outros detalhes do modelo de *workflow*;
- A especificação de um *workflow* segundo o modelo é facilmente concretizável através de linguagens de representação de dados, por exemplo, *JavaScript Object Notation* (JSON).

A flexibilidade do modelo proposto permitiu a definição de uma arquitetura concetual de referência com vista à realização de implementações concretas.

7.3 Arquitetura de Suporte à Implementação do Modelo

A arquitetura concetual de suporte à implementação do modelo foi definida com o propósito de facilitar a implementação, sem compromissos tecnológicos, de um sistema de execução de *workflows*. De seguida enumeram-se as características mais importantes da arquitetura:

- O controlo de execução de uma *Activity* (*Activity Engine*) permite a sua execução de forma autónoma;

- O contrato de ativação de uma *Task* estabelece os critérios a ter em consideração durante a sua execução;
- O mecanismo de comunicação assíncrona entre as *Activities* (*Broker*) segue o modelo *publish/subscribe* suportado por *queues*, permitindo uma execução autónoma das *Activities* de um *workflow*;
- O contexto de execução de uma *Activity* é um *container* instanciado a partir de imagens individualizadas (*custom*) que suportam diferentes ambientes de desenvolvimento e execução da *Task* associada.

A arquitetura pressupõe a existência de uma infraestrutura com múltiplos nós computacionais de suporte à execução de *workflows*, flexível para poder ser configurada através de máquinas virtuais disponibilizadas em qualquer plataforma *Cloud* seja privada ou pública.

Para flexibilizar o processamento distribuído das *Activities* de um *workflow*, em cada nó existe um componente interlocutor, designado *Activity Bucket*, de apoio ao lançamento e monitorização de *Activities* e que expõe uma REST API. O *Activity Bucket* flexibiliza a execução e controlo em cada nó computacional das *Activities* como *containers*. Adicionalmente a REST API do *Activity Bucket* facilita o desenvolvimento de ferramentas de lançamento de *workflows* focadas na distribuição de *Activities* pelos diferentes nós computacionais tal como o componente *WorkflowLauncher* implementado no protótipo de experimentação.

Esta arquitetura deixa em aberto uma vasta possibilidade de escolhas tecnológicas para realizar sistemas de execução de aplicações concretas moduladas como *workflow* segundo o modelo proposto.

7.4 Implementação do Protótipo

A implementação do protótipo de experimentação foi facilitada pela descrição pormenorizada do modelo proposto e arquitetura de suporte, deixando unicamente em aberto as escolhas tecnológicas para a implementação de um protótipo de experimentação.

Para permitir o processo de desenvolvimento de *software* dos diferentes componentes da arquitetura de suporte ao modelo de *workflow*, considerou-se uma infraestrutura computacional constituída por um conjunto de máquinas virtuais (VM) instanciadas na *Google Cloud Platform* (GCP) através do programa *Education Grants*. Cada VM tem

o sistema operativo *Ubuntu*, com o ambiente *Java OpenJDK* 11 [44] e o ambiente de execução *Docker* [37] para suportar a execução de *containers*.

Para suportar a partilha de ficheiros, cada máquina virtual foi configurada com o sistema de ficheiros distribuídos *GlusterFS* [43], que permite a replicação total de ficheiros em todas as VM.

Dado que cada VM tem um endereço IP interno da rede da GCP e um endereço IP público, acessível via SSH em qualquer computador, esta infraestrutura permitiu que o protótipo se executasse num *cluster* virtual de nós computacionais.

Em cada nó computacional, executa-se um processo *daemon* designado *ActivityBucket* que tem duas importantes funcionalidades: i) disponibiliza uma interface REST API para a partir da especificação das *Activities* do *workflow*, poder executá-las e monitorizá-las dentro do nó computacional; ii) lança em execução e monitoriza os *Docker containers* onde se executam as *Activities*.

A REST API do *ActivityBucket* permitiu o desenvolvimento de uma aplicação de nome *WorkflowLauncher* que a partir de uma linguagem de especificação de *workflows* em formato JSON suporta a experimentação com aplicações concretas moduladas como *workflows*. No entanto, é possível que qualquer outro programador desenvolva ferramentas de apoio à especificação e execução de *workflows*.

Cada *Docker container* encapsula a execução de uma *Activity* tendo como imagem base a componente *Activity Engine* que implementa a máquina de estados que gere o ciclo de vida da *Activity*, isto é, a estratégia de iterações que lhe foi atribuída, a obtenção de *Tokens* dos *Input Ports*, a execução da *Task* associada à *Activity* e em função dos resultados da *Task* produz *Tokens* nos *Output Ports*.

Como mecanismo de intermediação (*Broker*) de *Tokens* entre *Activities* utilizou-se o *middleware RabbitMQ* [39] que também suportou a implementação de um mecanismo de recolha de *Logs*, não idealizado nem no modelo nem na arquitetura. Este mecanismo de *Logs* mostrou-se fundamental no processo de *debugging*, no processo de experimentação e na validação de casos concretos de *workflow*.

Em síntese, a implementação do protótipo permitiu avaliar a correção do modelo e a correspondente arquitetura concetual de suporte.

7.5 Experimentação e Validação

Para validar o modelo de *workflow* proposto e o protótipo implementado segundo a arquitetura concetual, definiram-se os seguintes objetivos:

1. Validação da correção dos resultados de execução de um *workflow* com múltiplas iterações, com *Tasks* de *Activities* desenvolvidas em múltiplas linguagens de programação e com os padrões de *workflow* básicos, tais como *Sequence*, *Parallel Split* e *Synchronization*;

2. Avaliação dos tempos de *overhead* e melhoria dos tempos de execução de um *workflow*, especificando réplicas das *Activities* que tenham *Tasks* com tempos de processamento elevados, proporcionando balanceamento de carga nas sucessivas iterações;

3. Demonstração da independência do modelo face a requisitos funcionais a implementar nas *Tasks* das *Activities*;

4. Avaliação da especificação de *workflows* referenciados noutros trabalhos, por exemplo *MapReduce*, para validar a escala em termos de número de *Activities* bem como a passagem de volume de dados de grande dimensão entre as *Activities* utilizando o sistema de ficheiros distribuídos (*GlusterFS*).

Para o objetivo 1, foi desenvolvido e executado um *workflow* simples que a partir dos números gerados por progressões aritméticas, calcula uma expressão envolvendo adições e multiplicações. Os resultados permitem concluir a correta operacionalidade do protótipo.

Para o objetivo 2, foi desenvolvido e executado um *workflow* como uma sequência de três *Activities A*, *B* e *C* que executam uma *Task* genérica parametrizada com uma constante que simula um tempo de execução em segundos. A experimentação com este *workflow* permite concluir que os tempos de *overhead* são negligenciáveis. Utilizando réplicas na *Activity B* com um tempo de execução de 10 segundos, foi possível concluir a redução drástica do tempo global de execução do *workflow* para um elevado número de iterações e um número variável de réplicas.

Para o objetivo 3, foi desenvolvido e executado um *workflow* com múltiplas *Activities* cujas *Tasks* tiram partido de serviços disponibilizados na *Google Cloud Platform*, nomeadamente armazenamento de ficheiros (*Cloud Storage*), processamento de ficheiros de imagem (*Vision API*) e de tradução de texto (*Google Translate*). A experimentação permite concluir que o modelo é flexível no desenvolvimento da imagem do *Docker container* que suporta a execução de uma *Activity* e na independência face ao desenvolvimento de *Tasks* em qualquer domínio de aplicação.

Para o objetivo 4, foi desenvolvido e executado um *workflow* com 11 *Activities* que implementa o modelo *MapReduce* para contabilizar a ocorrência de palavras em ficheiros

de texto. A experimentação permitiu concluir a operacionalidade do protótipo nomeadamente a partilha de ficheiros contendo resultados intermédios entre as diferentes *Activities*.

Em síntese é possível considerar o total sucesso do Trabalho Final de Mestrado.

7.6 Trabalho Futuro

Após a realização deste Trabalho Final de Mestrado, incluindo as múltiplas experimentações com o protótipo implementado, é possível indicar de seguida alguns pontos como contributo para possíveis trabalhos futuros:

- Criar uma aplicação com uma interface gráfica para apoiar os utilizadores a especificar, executar e monitorizar *workflows* de forma amigável e intuitiva;
- Estender a REST API dos *Activity Buckets*, expondo a meta-informação da capacidade de processamento de cada nó computacional, permitindo a possibilidade de serem acrescentadas na especificação de um *workflow*, requisitos computacionais para cada *Activity*. Com esta extensão, tanto da especificação de um *workflow* como da REST API dos *Activity Buckets*, seria também interessante o desenvolvimento de um mecanismo de distribuição de *Activities* de um *workflow* com recurso a algoritmos mais sofisticados, possivelmente até tirando partido de inteligência artificial;
- Criar bibliotecas em *Java* e/ou noutras linguagens de programação, ferramentas de apoio ao desenvolvimento de uma *Task*, permitindo aos programadores terem uma curva de aprendizagem menor na compreensão do contrato de uma *Task* e também potenciar a reutilização de código;
- Criar uma imagem *Docker* de apoio ao desenvolvimento de *workflows*, contendo uma *Task* que serve como *proxy* ou adaptador de executáveis que não seguem o contrato de uma *Task*, de forma a poder especificar um *workflow* em que as *Tasks* poderiam ser quaisquer aplicações legadas ativadas na linha de comandos;
- Acrescentar ao modelo, arquitetura e respetiva implementação requisitos de segurança nos pilares de confidencialidade e integridade de forma a poder suportar cenários onde o sigilo ou mesmo a propriedade intelectual sejam de grande importância;

- Por último fica o desafio que em colaboração com cientistas de outras áreas da ciência e com acesso a infraestruturas de virtualização com mais nós computacionais, realizar experimentações envolvendo *workflows* em maior escala em termos de *Activities* e volume de dados.

Referências

[1] I. Taylor, E. Deelman, D. Gannon, M. Shields et al., *Workflows for e-Science: scientific workflows for grids*. Springer, 2007, vol. 1.

[2] *Workflow Management Coalition*, 1993. URL: wfmc.org (acedido em 20/10/2023).

[3] B. Ludäscher, S. Bowers & T. McPhillips, "Scientific Workflows", em *Encyclopedia of Database Systems*, L. LIU & M. T. ÖZSU, eds. Boston, MA: Springer US, 2009, páginas 2507–2511. DOI: 10.1007/978-0-387-39940-9_1471.

[4] W. Aalst, A. Hofstede & N. Russell, *Workflow Patters Website*, 2011. URL: workflowpatterns.com (acedido em 20/10/2023).

[5] C. Medeiros, G. Vossen & M. Weske, "WASA: A workflow-based architecture to support scientific database applications", em *Database and Expert Systems Applications: 6th International Conference, DEXA'95 London, United Kingdom, September 4–8, 1995 Proceedings 6*, Springer, 1995, páginas 574–583.

[6] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew & E. Deelman, "Producing an infrared multiwavelength galactic plane atlas using montage, pegasus and amazon web services", em *23rd Annual Astronomical Data Analysis Software and Systems, ADASS, Conference*, 2013.

[7] Y. Wang, G. Mehta, R. Mayani, J. Lu, T. Souaiaia, Y. Chen, A. Clark, H. Yoon, L. Wan, O. Evgrafov et al., "RseqFlow: workflows for RNA-Seq data analysis", *Bioinformatics*, vol. 27, n.º 18, páginas 2598–2600, 2011.

[8] B. Berriman, J. Good, A. Laity, J. Jacob, D. Katz, E. Deelman, G Singh & M. Su, "Web-based tools-montage: An astronomical image mosaic engine", 2008.

[9] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. Jordan, C. Kesselman, P. Maechling et al., "Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example", em *2006 Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, IEEE, 2006, páginas 14–14.

[10] D. Brown, P. Brady, A. Dietz, J. Cao, B. Johnson & J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis", *Workflows for e-Science: Scientific workflows for grids*, páginas 39–59, 2007.

[11] S. Majithia, M. Shields, I. Taylor & I. Wang, "Triana: A graphical web service composition and execution toolkit", em *Proceedings. IEEE International Conference on Web Services, 2004.*, IEEE, 2004, páginas 514–521.

[12] V. Curcin & M. Ghanem, "Scientific workflow systems-can one size fit all?", em *2008 Cairo International Biomedical Engineering Conference*, IEEE, 2008, páginas 1–9.

[13] B. Tang, M. Moca, S. Chevalier, H. He & G. Fedak, "Towards mapreduce for desktop grid computing", em *2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, IEEE, 2010, páginas 193–200.

[14] *The Kepler Project*, 2002. URL: kepler-project.org (acedido em 20/10/2023).

[15] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao & Y. Zhao, "Scientific workflow management and the Kepler system", *Concurrency and computation: Practice and experience*, vol. 18, n.º 10, páginas 1039–1065, 2006.

[16] A. Hartman, S. Riddle, T. McPhillips, B. Ludäscher & J. Eisen, "Introducing WATERS: a workflow for the alignment, taxonomy, and ecology of ribosomal sequences", *BMC bioinformatics*, vol. 11, n.º 1, páginas 1–14, 2010.

[17] J. Cummings, A. Pankin, N. Podhosrzki, G. Park, S. Ku, R. Barreto, S. Klasky, C. Chang, H. Strauss, L. Sugiyama et al., "Plasma edge kinetic-MHD modeling in tokamaks using Kepler workflow for code coupling, data management and visualization", *Communications in Computational Physics*, vol. 4, n.º 3, páginas 675–702, 2008.

[18] *Ptolemy II*, 1996. URL: ptolemy.berkeley.edu/ptolemyII/index.htm (acedido em 20/10/2023).

[19] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. Silva, G. Papadimitriou & M. Livny, "The evolution of the pegasus workflow management software", *Computing in Science & Engineering*, vol. 21, n.º 4, páginas 22–36, 2019.

[20] *DAGMan Workflows*, 2002. URL: htcondor.readthedocs.io/en/latest/automated-workflows/index.html (acedido em 20/10/2023).

[21] D. Thain & M. Livny, "Building reliable clients and services", *The Grid: Blueprint for a New Computing Infrastructure*, vol. 2, 2004.

[22] L. Assunção, "A Model for Scientific Workflows with Parallel and Distributed Computing", Available at hdl.handle.net/10362/19975, PhD thesis, Universidade Nova de Lisboa, 2016.

[23] K. Gilles, "The semantics of a simple language for parallel programming", *Information processing*, vol. 74, páginas 471–475, 1974.

[24] N. Carriero & D. Gelernter, "Linda in context", *Communications of the ACM*, vol. 32, n.º 4, páginas 444–458, 1989.

[25] K. M. D. Sweeney & D. Thain, "Efficient Integration of Containers into Scientific Workflows", em *Proceedings of the 9th Workshop on Scientific Cloud Computing*, sér. ScienceCloud'18, Tempe, AZ, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3217880.3217887.

[26] Y. D. Dessalk, N. Nikolov, M. Matskin, A. Soylu & D. Roman, "Scalable Execution of Big Data Workflows Using Software Containers", em *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, sér. MEDES '20, Virtual Event, United Arab Emirates: Association for Computing Machinery, 2020, 76—83. DOI: 10.1145/3415958.3433082.

[27] D. Ghoshal & L. Ramakrishnan, "Programming Abstractions for Managing Workflows on Tiered Storage Systems", *ACM Trans. Storage*, vol. 17, n.º 4, 2021, ISSN: 1553-3077. DOI: 10.1145/3457119.

[28] S. Singhal, A. Sussman, M. Wolf, K. Mehta & J. Y. Choi, "DYFLOW: A Flexible Framework for Orchestrating Scientific Workflows on Supercomputers", em *50th International Conference on Parallel Processing Workshop*, sér. ICPP Workshops '21, Lemont, IL, USA: ACM, 2021. DOI: 10.1145/3458744.3474037.

[29] R. F. d. Silva, L. Pottier, T. Coleman, E. Deelman & H. Casanova, "WorkflowHub: Community Framework for Enabling Scientific Workflow Research and Development", em *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2020, páginas 49–56. DOI: 10.1109/WORKS51914.2020.00012.

[30] M. Abedini Ala & B. Roy, "Facilitating Asynchronous Collaboration in Scientific Workflow Composition Using Provenance", vol. 6, n.º EICS, 2022. DOI: 10.1145/3534520.

[31] N. Freeman, J. Stubbs, R. Cardone & C. Garcia, "Detailed Functional Overview of an API and Workflow Engine for Scientific Research Computing", sér. PEARC '23, Portland, OR, USA: Association for Computing Machinery, 2023, 10–17. DOI: 10.1145/3569951.3593609.

[32] J. Stubbs, R. Cardone, M. Packard & et al, "Tapis: An API Platform for Reproducible, Distributed Computational Research", em *Advances in Information and Communication*, K. Arai, ed., Cham: Springer International Publishing, 2021, páginas 878–900.

[33] *Argo Workflows - The workflow engine for Kubernetes*, 2023. URL: argoproj.github.io/workflows/ (acedido em 20/10/2023).

[34] *Kubernetes*, 2014. URL: kubernetes.io/ (acedido em 20/10/2023).

[35] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[36] F. R, N. M & R. J, *HTTP Semantics*, 1991. URL: datatracker.ietf.org/doc/html/rfc9110.html (acedido em 20/10/2023).

[37] *Docker: Accelerated Container Application Development*, 2013. URL: docker.com (acedido em 20/10/2023).

[38] *IntelliJ IDEA – the Leading Java and Kotlin IDE - JetBrains*, 2001. URL: jetbrains.com/idea/download (acedido em 20/10/2023).

[39] *RabbitMQ*, 2007. URL: rabbitmq.com (acedido em 20/10/2023).

[40] *Docker Hub Container Image Library | App Containerization*, 2019. URL: hub.docker.com (acedido em 20/10/2023).

[41] *Spark Framework*, 2013. URL: sparkjava.com (acedido em 20/10/2023).

[42] *Gson library*, 2008. URL: github.com/google/gson (acedido em 20/10/2023).

[43] *Gluster*, 2009. URL: gluster.org (acedido em 20/10/2023).

[44] *OpenJDK 11*, 2018. URL: openjdk.org/projects/jdk/11 (acedido em 20/10/2023).

[45] *rabbitmq-java-client library*, 2008. URL: github.com/rabbitmq/rabbitmq-java-client (acedido em 20/10/2023).

[46] *docker-java library*, 2014. URL: github.com/docker-java/docker-java (acedido em 20/10/2023).

[47] *State Pattern*. URL: refactoring.guru/design-patterns/state (acedido em 20/10/2023).

[48] M. Fowler, *Service Locator*, 2004. URL: martinfowler.com/articles/injection.html#UsingAServiceLocator (acedido em 20/10/2023).

[49] J. L. Gustafson, "Amdahl's Law", em *Encyclopedia of Parallel Computing*, D. Padua, ed. Boston, MA: Springer US, 2011, páginas 53–60. DOI: 10.1007/978-0-387-09766-4_77.

Ficheiro de Especificação do *Workflow* de Longa Duração

```
{
  "workflowName": "LongRunningWorkflow",
  "workingDirectory": "/longRunningWorkflow",
  "iterations": 10,
  "activities": [
    {
      "activityName": "A",
      "containerImage": "my-namespace/long-running-workflow-activity:
    ↪ latest",
      "task": {
        "executionCommand": "java -jar", "executable": "
    ↪ LongRunningTask.jar",
        "constants": [ "0" ]
      },
      "ports": [{ "name": "OPortA", "type": "OUT", "channel": "IPortB
    ↪ " }],
      "mappingsTaskArguments": [],
      "mappingsTaskResults": [{ "portName": "OPortA", "index": 0 }]
    },
    {
      "activityName": "B",
```

```
19        "containerImage": "my-namespace/long-running-workflow-activity:
    ↪ latest",
20        "replicas": 5,
21        "task": {
22          "executionCommand": "java -jar", "executable": "
    ↪ LongRunningTask.jar",
23          "constants": [ "10" ]
24        },
25        "ports": [
26          { "name": "IPortB", "type": "IN", "channel": "OPortA" },
27          { "name": "OPortB", "type": "OUT", "channel": "IPortC"}
28        ],
29        "mappingsTaskArguments": [{ "portName": "IPortB", "index": 0 }]
    ↪ ,
30        "mappingsTaskResults": [{ "portName": "OPortB", "index": 0 }]
31      },
32      {
33        "activityName": "C",
34        "containerImage": "my-namespace/long-running-workflow-activity:
    ↪ latest",
35        "task": {
36          "executionCommand": "java -jar", "executable": "
    ↪ LongRunningTask.jar",
37          "constants": [ "0", "LongRunningWorkflowResults.txt"]
38        },
39        "ports": [{ "name": "IPortC", "type": "IN", "channel": "OPortB"
    ↪  }],
40        "mappingsTaskArguments": [{ "portName": "IPortC", "index": 0 }]
    ↪ ,
41        "mappingsTaskResults": []
42      }
43    ]
44  }
```

Listagem A.1: Especificação do *workflow* de Longa Duração representado na Subsecção 6.2.2