MICHAEL PACHECO

Orientador: Reinaldo Silva Fortes

XPERIMENTOR: UM *FRAMEWORK* PARA O GERENCIAMENTO DE EXECUÇÃO DE EXPERIMENTOS COMPUTACIONAIS

Ouro Preto

Julho de 2019

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

XPERIMENTOR: UM *FRAMEWORK* PARA O GERENCIAMENTO DE EXECUÇÃO DE EXPERIMENTOS COMPUTACIONAIS

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

MICHAEL PACHECO

Ouro Preto
Julho de 2019

D541x Dias, Michael Douglas Pacheco Gonçalves .
Xperimentor [manuscrito]: um framework para o gerenciamento de execução de experimentos computacionais / Michael Douglas Pacheco Gonçalves Dias. - 2019.

viii,55ff.: il.: color.

Orientador: Prof. MSc. Reinaldo Silva Fortes.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação.

1. Framework (Arquivo de computador). 2. Gerenciamento de configurações de software. 3. Sistemas operacionais distribuídos (Computadores). I. Fortes, Reinaldo Silva. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.75-047.42

Catalogação: ficha.sisbin@ufop.edu.br

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO

Xperimentor: Um *framework* para o gerenciamento de execução de experimentos computacionais

MICHAEL PACHECO

Monografia defendida e aprovada pela banca examinadora constituída por:

M.Sc. REINALDO SILVA FORTES – Orientador
Universidade Federal de Ouro Preto

Dra. AMANDA SÁVIO NASCIMENTO E SILVA
Universidade Federal de Ouro Preto

Dr. JOUBERT DE CASTRO LIMA
Universidade Federal de Ouro Preto

Ouro Preto, Julho de 2019

Resumo

A experimentação computacional exerce um importante papel nos dias atuais, sem a qual torna-se impossível a validação de certas hipóteses, principalmente das que necessitam de simulações para serem comprovadas. Desde o surgimento dos computadores digitais, matemáticos e cientistas têm utilizado essa técnica para modelar, executar e validar seus experimentos. No entanto, essas podem ser tarefas não-triviais fazendo com que a construção e a conclusão do experimento demandem mais tempo que o necessário. Este trabalho apresenta o *Xperimentor*, um *framework* gerenciador de experimentos computacionais de propósito geral desenvolvido em Python e JavaScript, que simplifica e agiliza a construção e execução de experimentos computacionais de forma paralela e distribuída. O *framework* foi projetado visando simplificar a construção de experimentos computacionais que envolvam a integração de aplicações desenvolvidas em múltiplas linguagens de programação. Como estudo de caso será utilizado um experimento de Sistema de Recomendação Híbrido que envolve diversas etapas de processamento e integra aplicações desenvolvidas em Python e Java. Através dos resultados obtidos nos experimentos foi possível validar que o *framework* é capaz de gerenciar experimentos complexos de forma paralela e distribuída.

Abstract

Computational experimentation plays an important role in nowadays, without which it becomes impossible to validate certain hypotheses, especially those that require simulations to be proved. Since the emergence of digital computers, mathematicians and scientists have been using this technique to model, execute and validate their experiments. However, these can be non-trivial tasks, making the construction and completion of the experiment take longer than necessary. This paper presents Xperimentor, a general purpose computational experiment management framework developed with Python and JavaScript, that simplifies and speed up the construction and execution of experiments in a parallel and distributed way. The framework was designed to simplify the construction of computational experiments that involves integration of applications developed in multiple programming languages. To validate the framework, a Hybrid Recommendation System experiment will be shown as a case study. That experiment involves several processing steps and uses applications developed with Java and Python. Through the results obtained in the experiments it was possible to validate that the framework is able to manage complex experiments in a parallel and distributed way.

Dedico este trabalho a Deus e aos meus familiares que me proporcionaram uma excelente educação e sabiamente me aconselharam nos momentos difíceis.

Agradecimentos

Primeiramente agradeço a Deus por me agraciar com a oportunidade de cursar este excelente curso e aprimorar meus conhecimentos ao longo destes anos.

Agradeço a todos meus familiares que me incentivaram e me mantiveram motivado durante todo o curso.

Agradeço também ao meu orientador, Reinaldo Silva Fortes, pela sua paciência e atenção. Sem ele este trabalho não existiria.

Um agradecimento especial aos meus falecidos pai (Carlos Alberto Gonçalves Dias), e avô (Newton Ranulfo Pacheco), que, infelizmente, não puderam presenciar o fruto de suas grandiosas personalidade e sabedoria.

Sou grato à Universidade Federal de Ouro Preto pelo cuidado, apoio e dedicação para com seus estudantes.

Agradeço à todos os membros do Departamento de Computação por sua efetividade e profissionalidade.

Agradeço a todos os meus professores por me proporcionarem a chance de me desenvolver tanto intelectualmente quanto pessoalmente através de seus conhecimentos e suas experiências. Aqui, um agradecimento especial ao professor Joubert que sempre nos motivou e nos fez "pensar fora da caixa" com suas aulas dinâmicas e elétricas (com direito à ligeiras pausas para ensinamentos de vida).

E por fim, porém não menos importante, agradeço ao iMobilis - Laboratório de Computação Móvel, que permitiu meu desenvolvimento extraclasse e minha manutenção na instituição através da minha bolsa de pesquisa. Nesse laboratório tive a chance de conhecer, trabalhar e trocar experiências com pessoas excepcionais.

Sumário

1 Introdução **1**

1.1 Justificativa 2

1.2 Objetivos Geral e Específicos 2

1.3 Definição do Problema 3

1.4 Organização do Trabalho 3

2 Revisão da Literatura **4**

2.1 Trabalhos Relacionados 4

2.1.1 FutureGrid 4

2.1.2 PRECIP 5

2.1.3 Comparação entre os *Frameworks* 8

2.2 Fundamentação Teórica 10

2.2.1 Contêineres 10

2.2.2 Kubernetes 12

3 Desenvolvimento **15**

3.1 Visão geral 15

3.2 Design do software 17

3.2.1 Backend - Task Executor 17

3.2.2 Frontend - Xperimentor 18

3.3 Princípios básicos de funcionamento 18

3.4 Fluxo de construção de um experimento 21

3.4.1 Representação de tarefas 22

3.4.2 Assinalamento de dependências 22

3.4.3 Configuração do cluster Kubernetes 23

3.4.4 Execução do experimento 27

4 Experimentos Computacionais **33**

4.1 Contextualização 33

4.2 Preparação do Experimento 34

4.3 Construção do Experimento 36
4.4 Análise e Discussão dos Resultados 37

5 Conclusão **39**

6 Apêndices **40**
.1 Documento de configuração para o gerador de tarefas 40

Referências Bibliográficas **43**

Lista de Figuras

2.1 Estrutura de dados *FutureGrid* 6
2.2 Arquitetura do *PRECIP*. 8
2.3 Visão geral de um *node Kubernetes* 13

3.1 Arco representando uma dependência entre duas tarefas. 16
3.2 Arquitetura do *framework* 16
3.3 Editor de código embutido. 19
3.4 Painel de visualização do experimento 19
3.5 Botões de interação da aplicação. 19
3.6 Diagrama de execução de tarefas 20
3.7 Visualização do grafo de dependências do exemplo 23
3.8 Painel de criação de *clusters* 25
3.9 Painel de configuração de *clusters* 25
3.10 Painel de conexão com o *cluster* 25
3.11 *Cloud Shell* 26
3.12 Listando serviços implantados 26
3.13 Exemplo de uma mensagem de erro de um experimento configurado indevidamente. 27
3.14 Exemplo de falha na execução de uma tarefa. 27
3.15 Exemplo de notificação de falha da execução de uma tarefa. 28
3.16 Painel de visualização do status de uma tarefa 28
3.17 Exemplo de falha não crítica 29
3.18 Marcando uma tarefa como bem sucedida 30
3.19 Visualização de uma tarefa forçada como bem sucedida 31
3.20 Estados de uma tarefa 32

4.1 Visão geral do processo de recomendação. 34
4.2 Grafo final do experimento de recomendação híbrida. 37

Lista de Algoritmos

1 Exemplo de compilação de um executável 21

Capítulo 1

Introdução

A experimentação é um dos tópicos de maior relevância para a ciência. Trabalhos científicos empíricos devem passar por experimentos para que seus resultados possam ser validados [7]. Entretanto, desenvolver experimentos em algumas subáreas específicas da ciência pode ser uma tarefa complexa [3]. Mesmo com o advento dos computadores os quais podem e têm sido utilizados para conduzir experimentos, a tarefa de experimentar pode ser demasiadamente manual.

A Ciência da Computação é uma área que se enquadra nesse perfil. Alguns experimentos requerem uma modelagem complexa e precisam ser divididos em tarefas. Algumas dessas tarefas não possuem dependências de outras e podem ser executadas de forma paralela, ao passo que outras podem envolver dependências resultando assim em uma execução sequencial. Lidar com dependências e paralelização são atividades essenciais para se projetar um experimento eficiente e, dependendo do tamanho deste, elas podem se tornar complexas de serem projetadas e programadas [6].

Outro problema resultante do trabalho manual de se fazer um experimento é a perda de tempo e recursos computacionais. Por exemplo: um experimento está dividido em diversas tarefas. Cada tarefa é um processo a ser executado pelo sistema operacional e possui um conjunto de parâmetros de entrada e deve produzir uma saída. Algumas dessas tarefas possuem parâmetros de entrada relativos à saídas de outras, *i.e*, elas possuem dependências. Se uma tarefa T falhar durante sua execução, todas as outras tarefas que dependem da finalização bem sucedida de T serão interrompidas até que o erro seja corrigido e o processo continue. Dependendo do tipo de experimento, o monitoramento do processo torna-se complexo, pois esse pode durar dias, ou até mesmo semanas. Pode levar muito tempo até que o erro seja detectado e corrigido, tempo este que poderia ter sido utilizado em processamento.

Alguns experimentos precisam ser replicados em diferentes contextos, onde parâmetros são variados a fim de se obter dados comparativos tais como variância, desvio padrão e intervalo de confiança. Obter essas métricas também pode ser uma tarefa muito complexa e monótona para ser feita manualmente. As chances de se cometer um erro são altas e podem comprometer

todo o experimento.

Estudos sobre *frameworks* de experimentação computacional têm sido discutidos com foco nas tecnologias e infraestruturas a serem utilizadas para executar tais experimentos [1, 8]. Esta abordagem, por tratar principalmente de aspectos tecnológicos, resulta em *frameworks* de complexa configuração e utilização. Essa complexidade desvia o foco da construção e execução do experimento para questões técnicas, como aprender e utilizar *software* de linha de comando para configurar e executar o experimento.

A seguir, na Seção 1.1, será apresentada a justificativa do desenvolvimento do *framework* proposto nesse trabalho. A Seção 1.2 explicita os objetivos almejados. A Seção 1.3 diz respeito à definição do problema. Por fim, a Seção 1.4 apresenta a organização do trabalho.

1.1 Justificativa

Como já explicitado anteriormente, no atual estado da arte, não há um sistema gerenciador de experimentos que possui como foco a modelagem e condução do experimento propriamente dito. Os *frameworks* existentes possuem como diferencial a infraestrutura do sistema de experimentação, tais como *clusters* de máquinas, controles de acesso e portabilidade [1, 8]. No entanto, nenhum deles leva em conta o planejamento e a construção do experimento. Ambos necessitam que os experimentos já estejam previamente configurados e armazenados em máquinas virtuais. Portanto, a construção deve ser feita de forma manual. Isso traz algumas consequências negativas como uma utilização precária dos recursos computacionais, por exemplo. Para um experimento de grande porte é necessário que o experimentador tenha o total controle da condução do mesmo. Porém muitas das vezes as variáveis são demasiadamente complexas para serem controladas manualmente. Dessa forma essa prática se torna inviável surgindo assim a necessidade de uma automatização de todo o processo. As chances de que um *software* bem projetado cometa um equívoco é menor que a de um ser humano. Além disso, um experimento executado de forma automatizada pode otimizar a utilização dos recursos computacionais economizando assim tempo, energia e dinheiro. A existência de um *framework* de experimentação computacional também pode permitir uma fácil replicação de um experimento bem como uma expansão do mesmo.

1.2 Objetivos Geral e Específicos

Em vista dos problemas citados anteriormente, o propósito deste trabalho é desenvolver um *framework* que propicie ao experimentador uma forma simples e padronizada de construir, organizar e gerenciar seus experimentos. O *framework* será responsável por orquestrar de forma automática todas as etapas de um experimento desde sua construção à análise dos resultados. Ele receberá como entrada as tarefas juntamente com suas dependências, e coordenará a execução do experimento de forma automática e distribuída. A ideia do *framework*

é que ele seja facilmente configurável através de uma interface gráfica e delegue o mínimo de tarefas ao usuário. Para utilizar o *framework* proposto neste trabalho, o experimentador deverá fornecer as tarefas com suas respectivas entradas e dependências. Em posse dessas informações, o *framework* ficará responsável pela condução do experimento. Quaisquer falhas de execução serão devidamente reportadas ao experimentador para que o mesmo possa corrigi-las e dar continuidade ao processo como um todo.

Os objetivos específicos deste trabalho incluem:

- Proporcionar uma estrutura simples, concisa, flexível e automatizada para a construir experimentos computacionais.
- Permitir correções de erros ocorridos durante a execução das tarefas sem a necessidade de reiniciar ou interromper a execução dos experimentos.
- Maximizar a utilização dos recursos computacionais disponíveis.
- Minimizar o tempo necessário para a finalização de um experimento.

1.3 Definição do Problema

Dados um conjunto de tarefas T e um conjunto de dependências D, deseja-se que ambas tarefas sejam executadas em um ambiente distribuído de forma que suas dependências sejam respeitadas.

O problema consiste em gerenciar a execução das tarefas, criando subconjuntos que possam ser executados em paralelo e sequenciando aqueles que possuam dependências entre si. Visando reduzir o tempo de execução, as tarefas devem ser delegadas para as máquinas disponíveis, levando em conta suas necessidades específicas por recursos de *hardware* e outros requisitos não funcionais.

1.4 Organização do Trabalho

O restante do trabalho se encontra organizado da seguinte forma: O Capítulo 2 introduz e discute sobre os trabalhos relacionados e suas semelhanças e diferenças com o *framework* desenvolvido nesta Monografia bem como os conceitos e tecnologias utilizadas no desenvolvimento do *Xperimentor*. O Capítulo 3 apresenta a modelagem e arquitetura do *Xperimentor* juntamente com um exemplo ilustrativo. O Capítulo 4 apresenta um experimento desenvolvido com o *Xperimentor* utilizando como estudo caso um experimento de Sistema de Recomendação Híbrido. Por fim, o Capítulo 5 apresenta e discute os resultados do experimento realizado no Capítulo 4.

Capítulo 2

Revisão da Literatura

Este capítulo apresenta os trabalhos relacionados encontrados na literatura e os métodos e conceitos utilizados para o desenvolvimento do *Xperimentor*, respectivamente nas Seções 2.1 e 2.2.

2.1 Trabalhos Relacionados

Nesta seção serão apresentados os dois *frameworks* de gerenciamento de experimentos computacionais de propósito geral encontrados na literatura (*FutureGrid* e *PRECIP*, Seções 2.1.1 e 2.1.2, respectivamente). Para cada um será introduzido o seu funcionamento básico, sua arquitetura e sua infraestrutura. Por fim será discutido as semelhanças e diferenças entre ambos e destes para o *Xperimentor*.

2.1.1 FutureGrid

Laszewski *et al.*, 2010 [8], desenvolveram um *framework* de experimentação denominado *FutureGrid* [1]. Esse *framework* tem como principal objetivo desenvolver uma plataforma de experimentos semelhante à um laboratório virtual que permita ao experimentador modelar ambientes e experimentos que possam ser facilmente compartilhados com a comunidade.

FutureGrid possui uma infraestrutura distribuída privada com suporte à tecnologia de provisionamento dinâmico. Essa tecnologia permite que o sistema seja reconfigurado de acordo com as necessidades do experimento. A reconfiguração é feita por um *software* interno do *framework*. *FutureGrid* foi desenvolvido sobre uma arquitetura complexa que agrega tanto *hardware* quanto *software*. O sistema foi dividido em diversos módulos, dentre eles:

- **FG Interfaces**, responsável por fornecer uma interface de linha de comando que oferece suporte à diversa comunidade de usuários.

[1] `https://portal.futuresystems.org/`

- **FG Stratosphere**, prevê funcionalidades para monitorar os *hardwares* e *softwares* executados nesse módulo. Também provê um sofisticado *framework* gerenciador de experimentos permitindo aos usuários gravar e reproduzir experimentos. Para atingir tais objetivos, experimentos planejados podem ser criados através de um *workflow* a partir do qual imagens de discos possam ser armazenadas e reutilizadas em um momento posterior.
- **FG RAIN - Runtime Adaptable INsertion Connfigurator**, permite aos usuários instanciarem imagens em tempo de execução e executar aplicações sobre estas imagens
- **FG Security**, responsável pela segurança de todo o sistema. Provê métodos de autenticação e autorização.

A fim de padronizar e organizar os experimentos armazenados e executados no *FutureGrid*, foi definida a estrutura de dados presente na Figura 2.1.

Para exemplificar a decisão por essa estrutura de dados os autores apresentaram o seguinte caso de uso: Um professor decidiu utilizar o *FutureGrid* como ferramenta para treinar seus alunos em diversos aspectos de *cloud computing*. Primeiro o professor deve criar um projeto no site do *FutureGrid*. O professor pode desejar acessar os recursos da grade, para isso ele deverá criar uma conta. Agora o professor deseja adicionar um estudante ao projeto, sendo assim um particular aparato pode ser compartilhado toda semana nas quais o estudante executará um experimento como atividade. No entanto, como o professor é muito ocupado para criar as tarefas de programação, um aluno assistente é adicionado para ajudar. Ao final espera-se que cada estudante produza um artigo ou relatório bem como uma apresentação expondo o trabalho feito. Através dessa estrutura de dados o projeto é gerenciado pelo professor. Ele delega a adição de alunos ao seu administrador o qual possui uma lista com os nomes de todos os alunos da classe. O assistente de ensino é adicionado como membro da equipe ao projeto e pode gerar novos experimentos e aparatos referentes aos experimentos em questão. O professor decide usar uma estratégia onde cada semana um novo experimento será conduzido.

O *framework FutureGrid* provê uma plataforma que auxilia na preparação do ambiente e no compartilhamento da mesma para replicações dos experimentos. No entanto, o *framework* não gerencia a experimentação em si. As etapas de definição das precedências das subtarefas do experimento e as estratégias de condução do mesmo não é gerenciada pela ferramenta, ficando assim totalmente inerente ao experimentador.

2.1.2 PRECIP

Inspirado no *FutureGrid*, Azarnoosh *et al.* [1], desenvolveram uma *API python* chamada *PRECIP (Pegasus Repeatable Experiments for the Cloud in Python)* [2]. Essa *API* tem como

[2] `https://pypi.org/project/precip/`

```
ACCOUNTS
    USER+
    FIRSTNAME
    LASTNAME
    USER ID
    ...
PROJECTS
    PROJECT+
        APPLICATION INFORMATION
        DESCRIPTION
        USER ID
        ...
        PROJECT ADMINISTRATORS
            USER ID+
        USERS
            USER ID+
            EXPERIMENT+
                MANAGER
                    USER ID+
                STAFF
                    USER ID+
                APPARATUS+
                    ID
                    RESOURCE+
                EXPERIMENT+
                    APPARATUS: ID
                    DATE
                    TIME
                    JOB+
                    SERVICE+
                PUBLICATIONS
                    ARTICLE+
                    REPORT+
                    TALK+
                    DEMO+
```

Figura 2.1: Estrutura de dados *FutureGrid*

principal objetivo formalizar experimentos da ciência da computação em ambientes distribuídos. Ela provê recursos para simplificar e formalizar a execução de experimentos científicos utilizando *Cloud Computing*, que é a entrega por demanda de recursos computacionais através da internet [3]. A *API PRECIP* provê as seguintes características:

- Fornece uma camada de abstração a fim de oferecer suporte à interoperabilidade entre os serviços disponibilizados na nuvem. O condutor do experimento apenas precisará saber utilizar a *API* para executar experimentos entre múltiplas *clouds*, ou mover um experimento de uma *cloud* para outra.
- Reprodutibilidade através dos *scripts*.
- *Verbose logging*.
- Manipulação básica de provisionamento, chaves *SSH* e grupos de segurança com suporte à tolerância a falhas.

Para tornar o *PRECIP* um produto amplamente utilizável, foram examinados os projetos registrados no *FutureGrid* e vagamente agrupados baseando-se em uma visão de alto nível dos requerimentos destes projetos. Algumas dessas classificações se mostraram semelhantes o bastante para serem utilizadas como direcionadoras para o desenvolvimento do *PRECIP*. Um subconjunto dos padrões obtidos foram:

- Bioinformática.
- *Networking*.
- *MapReduce*.
- Educacional.

O *PRECIP* foi projetado em uma arquitetura de camadas representada como regiões concêntricas (Figura 2.2) onde a interface do usuário se encontra no centro. Em seguida se encontra a camada de experimentação representando os passos envolvidos nos experimentos. Na camada mais externa se encontram grupos de conjuntos de *cloud instances*.

Executar um experimento no *PRECIP* requer 4 passos:

1. Configurar o experimento.
    - Estabelecer uma conexão com o serviço.
    - Registrar um par de chaves *SSH* para conexão interna.
    - Configurar os grupos padrão de segurança.

[3] `https://www.ibm.com/cloud/learn/what-is-cloud-computing`

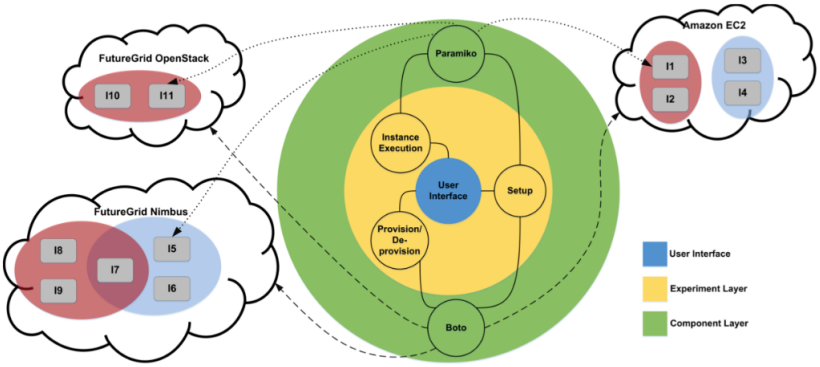

Figura 2.2: Arquitetura do *PRECIP*.

2. Provisionar os recursos.

3. Executar o experimento.

4. Desalocar os recursos provisionados.

Assim como no *FutureGrid*, usuários do *PRECIP* podem utilizar máquinas virtuais de seu interesse para executar seus experimentos. A interação com o sistema é feita via *SSH*. Todavia, o usuário também pode utilizar máquinas virtuais fornecidas pelo serviço de nuvem para executar experimentos mais complexos.

PRECIP gerencia a comunicação com as instâncias *cloud* em um contexto separado do sistema. Para isso são utilizadas as bibliotecas *Boto* [4] e *Paramiko* [5]. Porém o *PRECIP* utiliza de serviços *cloud* pagos como a Amazon, o que não é de interesse para o *framework* sendo proposto nesse trabalho. Além disso, a configuração de um experimento no *PRECIP* é bastante complexa uma vez que o usuário deve possuir conhecimentos de programação em *python* e aprender a *API*. A ideia do *framework* proposto nesse trabalho é que ele seja configurável através de uma interface gráfica e delegue o mínimo de tarefas ao usuário.

2.1.3 Comparação entre os *Frameworks*

Conforme apresentado anteriormente, ambos os *frameworks* analisados e o *Xperimentor* permitem executar experimentos em um ambiente distribuído. No entanto, existem algumas diferenças, conforme já brevemente mencionadas, que são descritas em mais detalhes a seguir:

1. Propósito

   - *FutureGrid*: seus principais propósitos são a reprodutibilidade e o compartilhamento de um experimento através dos quais torna-se possível definir um ambiente

[4] `https://aws.amazon.com/pt/sdk-for-python/`
[5] `https://github.com/paramiko/paramiko`

isolado de tal forma que as reproduções de um experimento possam ser comparadas. Esta comparação é possível pois cada experimento carrega como parte de si o aparato sobre o qual ele será reproduzido. Isso faz com que ele sempre execute sobre a mesma arquitetura de *hardware* e *software*.

- *PRECIP*: mesmos objetivos do *FutureGrid*.

- *Xperimentor*: seus principais objetivos são permitir ao experimentador gerenciar seus experimentos de forma simples e visual e utilizar um ambiente distribuído para maximizar a performance de um experimento. Diferentemente dos outros *frameworks*, o *Xperimentor* não foi desenvolvido visando a reprodutibilidade e o compartilhamento.

2. Complexidade de uso

   - *FutureGrid*: a infraestrutura do experimento pode ser configurada manualmente pelo experimentador ou de forma automática através de um *software* utilitário de linha de comando. O monitoramento de um experimento também deve ser feito através de utilitários de linha de comando. Estes utilitários são providos pelo componente *FG Interfaces*.

   - *PRECIP*: tanto a configuração quanto a troca de dados de um experimento deve ser feita através de *scripts python*. O experimentador precisa registrar um par de chaves SSH para conexões internas do *framework*. No *PRECIP* o experimentador não é capaz de monitorar um experimento de forma visual. A única maneira que ele possui para monitorar seus experimentos é através de *logs* registrados no *Openstack*.

   - *Xperimentor*: a configuração de um experimento é feita através de um documento *YAML* (*Ain't Markup Language*) [6]. Para que o experimentador possa executar seu experimento de forma distribuída, ele precisa apenas fornecer o endereço IP do *cluster*. O experimentador pode utilizar tanto serviços pagos como *Google Cloud* e *Amazon AWS* como também criar seu próprio *cluster* privado com suas máquinas de preferência. Um *cluster* pré configurado para o *Xperimentor* pode ser utilizado pelo experimentador caso ele assim desejar.

3. Flexibilidade

   - *FutureGrid* e *PRECIP*: ambos necessitam que os experimentos a serem executados sejam empacotados em uma imagem de uma máquina virtual. Dessa forma, qualquer mínima alteração feita em um experimento precisará ser reempacotada em

[6] `https://yaml.org/`

uma nova imagem. Isso faz com que a utilização destes *frameworks* sejam inviáveis para experimentos que precisam ser alterados com frequência.

- Xperimentor: caso um experimento necessite ser alterado, o experimentador precisará apenas alterar as tarefas no arquivo de configuração sem a necessidade de qualquer outra operação. Essa flexibilidade faz com que o *Xperimentor* se sobressaia sobre os demais em experimentos mutáveis.

2.2 Fundamentação Teórica

Esta seção apresenta alguns conceitos e tecnologias que foram utilizados durante o desenvolvimento do *framework*.

Ao refletir sobre a essência do problema de monitorar um experimento foi possível observar que o mesmo poderia ser abstraído para um nível mais genérico. O problema trata-se na verdade de monitorar uma grande quantidade de processos em um cenário de alto grau de paralelização. A partir dessa observação definiu-se que uma arquitetura baseada em sistemas distribuídos proveria uma solução mais viável para o problema. No entanto, ao adotar este tipo de arquitetura é necessário levar em consideração a tolerância à falhas e a comunicação entre as máquinas.

Para solucionar o problema, foi necessário efetuar pesquisas sobre quais seriam as tecnologias atuais mais adequadas que permitiriam elaborar uma solução eficiente e escalável.

Os *softwares Kubernetes* e *Docker Swarm* foram elegidos como as tecnologias mais viáveis para solucionar o problema. No entanto, o *Kubernetes* foi escolhido por possuir uma *API* muito mais robusta que a do *Docker Swarm*, o que permite uma maior flexibilidade de uso e integração com outros *softwares* auxiliares tais como o *Helm*.

A Seção 2.2.1 introduzirá alguns conceitos básicos para que o leitor possa melhor compreender o *Kubernetes* que será apresentado na Seção 2.2.2.

2.2.1 Contêineres

Um contêiner é uma unidade padronizada de *software*. Isso significa que ele traz consigo todo o código fonte de uma aplicação juntamente com todas as suas dependências. Dessa forma, um contêiner se torna independente de um sistema operacional. Isso permite que uma aplicação conteinerizada seja portável entre diferentes sistemas computacionais [2].

Para que uma aplicação possa ser empacotada em um contêiner, é necessário que um *software* especial o faça. O *Docker* é um destes *softwares*, capaz de criar e administrar aplicações conteinerizadas. Para criar um contêiner utilizando o *Docker* é preciso definir uma receita que informará como construir um contêiner. Essa receita é um arquivo textual chamado *Dockerfile* que contém comandos a serem executados pelo *Docker*. O seguinte exemplo ilustra um

Dockerfile para gerar uma imagem de uma aplicação *python* que futuramente será utilizada para instanciar um contêiner.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim
# Set the working directory to /app
WORKDIR /app
# Copy script into the container at /app
ADD app.py /app
# Run app.py when the container launches
CMD ["python", "app.py"]
```

Todo *Dockerfile* precisa declarar uma imagem base sobre a qual a imagem sendo definida será construída. No exemplo acima, a imagem base é a imagem de um interpretador *python*. O comando `WORKDIR` configura qual o diretório de trabalho que o contêiner irá utilizar como raíz. O comando `ADD` adicionará o arquivo *app.py* ao diretório `/app` do contêiner. E por fim, o comando `CMD` faz com que o interpretador *python* execute o *script* `app.py` quando o contêiner estiver pronto para ser executado.

Para construir a imagem basta executar o comando **`docker build -tag=app .`** e para criar um contêiner basta executar **`docker run app`**.

Após executar esses comandos, o *Docker* carregará a aplicação juntamente com suas dependências de forma que o sistema operacional consiga executá-la.

Imagens de contêineres podem ser disponibilizadas publicamente na internet através de repositórios. O *Docker Hub* é um repositório amplamente utilizado que contém diversas imagens disponibilizadas tanto por grandes corporações como por desenvolvedores. Para publicar uma imagem customizada basta executar os seguintes passos:

1. Criar uma conta em `https://hub.docker.com/`
2. Efetuar o *login* no *Docker* através do terminal utilizando o comando **`docker login`**
3. Fornecer uma *tag* para a imagem desejada
   - 3.1. **`docker tag <image-id> <hub-username>:<image-name>:<tag>`**
   - 3.2. **`docker push <hub-username>/<image-name>:<tag>`**

Após ser publicada, a nova imagem poderá ser utilizada por terceiros através do comando **`docker pull <hub-username>/<image-name>:<tag>`**

2.2.2 Kubernetes

Os contêineres oferecem muita praticidade para implantar um *software* em um ambiente de produção. No entanto, quando o número de contêineres cresce, gerenciá-los se torna uma tarefa complexa. O *Kubernetes* é um *software open source* desenvolvido pela *Google* que é capaz de orquestrar contêineres de forma automática em um ambiente distribuído. Ele é capaz de escalonar uma aplicação em tempo real para atender a demanda de acessos. Por exemplo: imagine que há um contêiner de um servidor HTTP em execução e em determinado momento a quantidade de requisições que chega para esse servidor cresça de forma abrupta. O *Kubernetes* irá detectar esse pico e iniciará outros contêineres em outras máquinas do *cluster* para equilibrar os acessos em cada contêiner. Quando o número de requisições diminuir, o *Kubernetes* irá detectar e desalocar os contêineres que estiverem ociosos.

2.2.2.1 Arquitetura básica

O *Kubernetes* possui três conceitos básicos: *nodes*, *pods* e *deployments*. Um *node* é uma máquina específica do *cluster* que possui o *Docker* instalado. Todo *cluster* possui um *node* que é o mestre. É ele quem controla o estado do *cluster*. Todo *node* possui um processo chamado *kubelet* executando em *background*. Esse processo é o meio pelo qual os *nodes* de um *cluster* se comunicam. Um *pod* é um conjunto de um ou mais contêineres relativamente conectados que compartilham armazenamento e recursos de rede. *Deployments* são mecanismos pelos quais é possível configurar e inspecionar um *cluster Kubernetes*. A Figura 2.3 ilustra a relação entre estes componentes.

Para implantar uma aplicação em um *cluster Kubernetes*, são necessários os seguintes passos:

1. Conteinerizar a aplicação.
2. Armazenar a imagem do contêiner em um repositório como o *Docker Hub*.
3. Criar um *deployment*.

O código abaixo ilustra um exemplo de um *deployment* do servidor HTTP chamado *nginx*:

```
# nginx-deployment.yaml
%YAML 1.2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
```

```
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
```

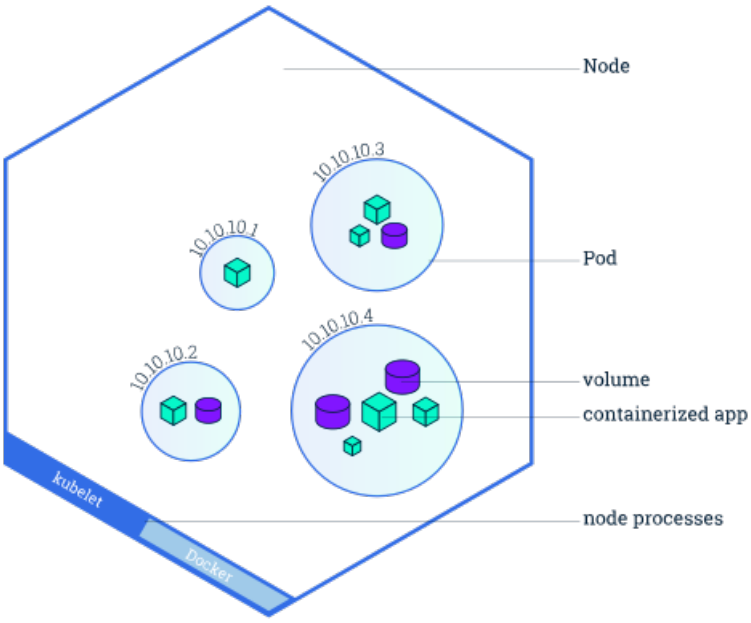

Figura 2.3: Visão geral de um *node Kubernetes*. Esta figura ilustra um *node* isoladamente. Este *node* possui 4 *pods* gerados por um *deployment*, cada qual executando uma aplicação conteinerizada. Cada *pod* possui seu próprio endereço IP interno que pode ser utilizado para comunicação com outros *pods*. Alguns dos *pods* possuem também um volume para armazenamento de dados. Esse volume é montado dentro do contêiner como um diretório. Este volume pode ser tanto um armazenamento físico como também um armazenamento virtual provido por um servidor *NFS*, por exemplo.

```
    - containerPort: 80
```

Esse arquivo `YAML` representa um *deployment* que instrui o *Kubernetes* a criar um *pod* com a imagem do *nginx* e replicá-la em 3 *nodes*. Para aplicar o *deployment* basta executar o comando **`kubectl apply -f nginx-deployment.yaml`**.

Após executar este comando, o *Kubernetes* criará 3 *pods* distintos cada qual com seu contêiner do *nginx*. Porém, os *pods* estarão visíveis apenas na rede interna do *cluster*. É preciso expor os *pods* como serviços para que eles possam ser acessados externamente. Para isto basta executar o comando **`kubectl expose deployment/nginx`**. Agora é possível acessar o *nginx* através do IP do serviço recém criado.

Toda vez que um serviço for acessado, o *Kubernetes* utilizará um componente de *software* chamado *load balancer*. Esse componente é responsável por balancear a carga de acesso entre os *pods* para que os *nodes* não fiquem sobrecarregados.

Capítulo 3

Desenvolvimento

Neste capítulo serão apresentadas as decisões de projetos tomadas durante o desenvolvimento do *framework*. A Seção 3.1 apresenta uma visão geral da arquitetura de *software* e *hardware*. A Seção 3.2 apresenta uma visão um pouco mais detalhada sobre os módulos do *framework* e como eles se relacionam. A Seção 3.3 apresenta o princípio básico de funcionamento e por fim, a Seção 3.4 apresenta o fluxo de construção de um experimento através de um exemplo.

3.1 Visão geral

O problema foi modelado da seguinte forma:

Seja $G = (V, A)$ um dígrafo acíclico representando as tarefas e dependências de um experimento, onde:

1. $V = \{(id, P, D)\}$ é um conjunto vértices representando as tarefas. Cada tarefa possui:
   - id: um identificador único.
   - P: uma cadeia de caracteres contendo o comando que será executado pelo sistema operacional.
   - D: um conjunto de identificadores representando as dependências.
2. $A = \{(t_i, t_j) \mid t_i, t_j \in V;\ id_i \in D_j\}$ é um conjunto não vazio de arcos representando pares de tarefas em que t_i representa uma dependência para t_j, conforme ilustrado na Figura 3.1.

Sendo assim, a arquitetura básica do *framework* se encontra na Figura 3.2. Para criar um experimento o experimentador deve fornecer um arquivo de configuração contendo as definições das tarefas e suas dependências. A partir deste arquivo, o *framework* constrói um grafo de dependências que será utilizado para determinar a ordem de execução destas tarefas.

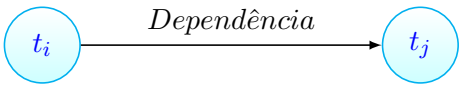

Figura 3.1: Arco representando uma dependência entre duas tarefas.

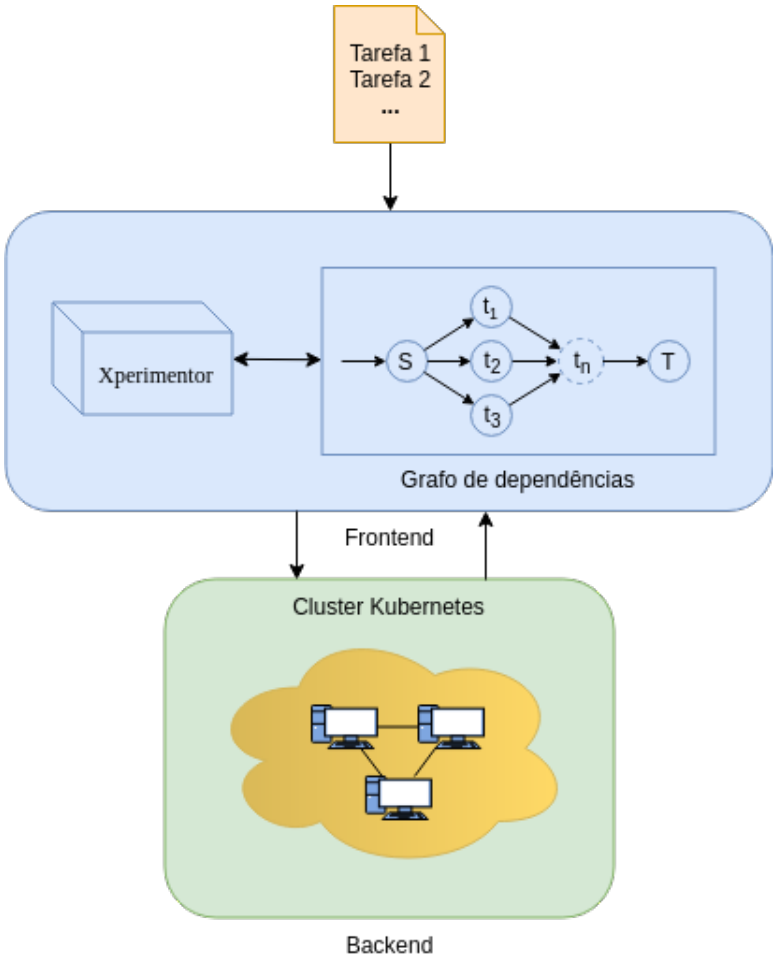

Figura 3.2: Arquitetura do *framework*. Esta figura representa os dois módulos que compõem o *framework*. O módulo representado em cor azul ilustra o *frontend*, que é responsável por gerenciar a execução de um experimento e fornecer uma interface de uso ao experimentador. Este módulo recebe como entrada um arquivo contendo definições de tarefas e dependências. O módulo em cor verde ilustra o *backend* implantado em um *cluster Kubernetes* onde as tarefas do experimento serão executadas.

Quando uma tarefa for escalonada para execução ela será redirecionada para uma máquina do *cluster* onde ela será executada.

3.2 Design do software

Para que o *Xperimentor* pudesse usufruir das funcionalidades do *Kubernetes* de forma eficiente, o seu componente responsável pela execução dos processos precisaria ser projetado como um microsserviço *stateless*. Sendo assim, o *Xperimentor* foi dividido em dois projetos:

1. Backend: microsserviço responsável por executar um único processo.
2. Frontend: responsável por coordenar a execução destes processos e fornecer uma interface de uso ao experimentador.

Ambos abordados a seguir.

3.2.1 Backend - Task Executor

O *backend* consiste de um servidor HTTP desenvolvido em *Python* que trata requisições para executar processos. Esta aplicação está conteinerizada e implantada em um *cluster Kubernetes* onde cada máquina do *cluster* possui uma réplica do *Task Executor* sendo executada como um serviço. Dessa forma, todos os *nodes* serão capazes de executar tarefas. Se em determinado momento a quantidade de tarefas crescer abruptamente, o *Kubernetes* irá detectar este crescimento e escalar a aplicação para que mais contêineres sejam criados a fim de atender a demanda. Caso um contêiner venha a falhar, o *Kubernetes* certificar-se-á de que um novo seja instanciado para substituir o falho.

Toda tarefa de um experimento sendo executado no *Xperimentor* será direcionada para essa aplicação para que ela possa ser executada. Ao receber uma requisição, o *Task Executor* iniciará um processo e registrará todo fluxo produzido nos canais de saída padrão. Ao fim de um processo, o *Task Executor* irá retornar o código de finalização do processo juntamente com as informações produzidas nos canais de saída padrão. Esses dados serão utilizados para determinar se um processo foi executado com sucesso ou não. O *Xperimentor* considera um processo como bem-sucedido se e somente se seu canal de saída padrão de erro for vazio e o código de retorno do processo for igual a zero.

Eis um exemplo de uma requisição HTTP suportada pelo *Task Executor*:

```
curl -X POST https://0.0.0.0:5050/run \
-H 'Content-Type: application/json;charset=UTF-8' \
-d '{ "id": "sayHello", "command": "echo hello" }'
```

Essa requisição contém uma tarefa identificada como *sayHello* cujo comando exibe a mensagem *hello* na saída padrão.

A seguinte resposta será retornada por essa requisição:

```
{
"returnCode": 0,
"stderr": "",
"stdout": "hello\n"
}
```


O campo *returnCode* igual a zero indica que o processo foi executado com sucesso. O campo *stderr* com valor vazio, indica que nenhuma mensagem de erro foi exibida. E por fim, o campo *stdout* indica que a mensagem *hello* foi exibida na saída padrão.

3.2.2 Frontend - Xperimentor

Este projeto consiste em uma aplicação WEB desenvolvida em *JavaScript*. Esta é a aplicação principal do *framework*, responsável por construir e gerenciar a execução de um experimento.

O projeto possui uma única página com um editor de código embutido e um painel de visualização onde o experimentador pode observar o status do experimento.

A configuração deve ser feita através de um documento YAML. Este documento deve conter todos os dados necessários para que o *framework* seja capaz de construir e executar o experimento. Nele o experimentador definirá quais são as tarefas a serem executadas e quais são as dependências de cada uma delas.

As Figuras 3.3, 3.4 e 3.5 demonstram a interface visual do sistema. A Figura 3.3 ilustra o editor de texto que é utilizado para configurar o experimento. A Figura 3.4 ilustra o painel pelo qual é possível monitorar o status de um experimento. A Figura 3.5 ilustra os botões principais da aplicação através dos quais é possível carregar, validar, construir e executar um experimento, respetivamente.

3.3 Princípios básicos de funcionamento

O diagrama da Figura 3.6 ilustra o fluxo de execução de um experimento.

```
%YAML 1.2
---
clusterIP: '35.247.204.254'
tasks:
  - id: clean
    command: rm -rf ~/build/*

  - id: compile_util
    command: gcc -Wall -c ~/build/util.c -o ~/build/util.o
    deps: [clean]

  - id: compile_main
    command: gcc -Wall -c ~/build/main.c -o ~/build/main.o
    deps: [clean]

  - id: mklib
    command: ar -cvq ~/build/libpstr.a ~/build/*.o
    deps: [compile_util, compile_main]

  - id: makeapp
    command: gcc ~/build/*.o -o ~/build/application -L ~/build -lpstr
    deps: [mklib]

  - id: execute
    command: ~/build/application
```

Figura 3.3: Editor de código embutido. Utilizado para configurar o experimento.

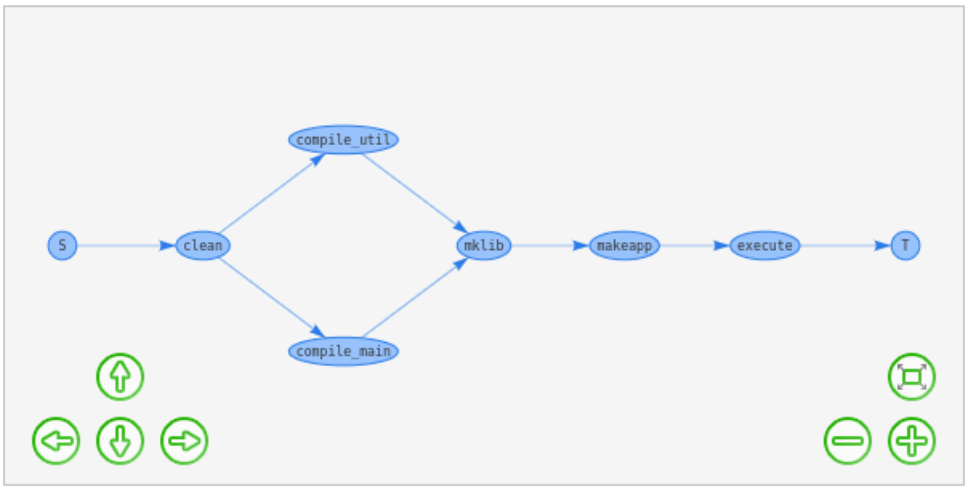

Figura 3.4: Painel de visualização do experimento. Painel dinâmico utilizado para visualizar e organizar o experimento.

Figura 3.5: Botões de interação da aplicação.

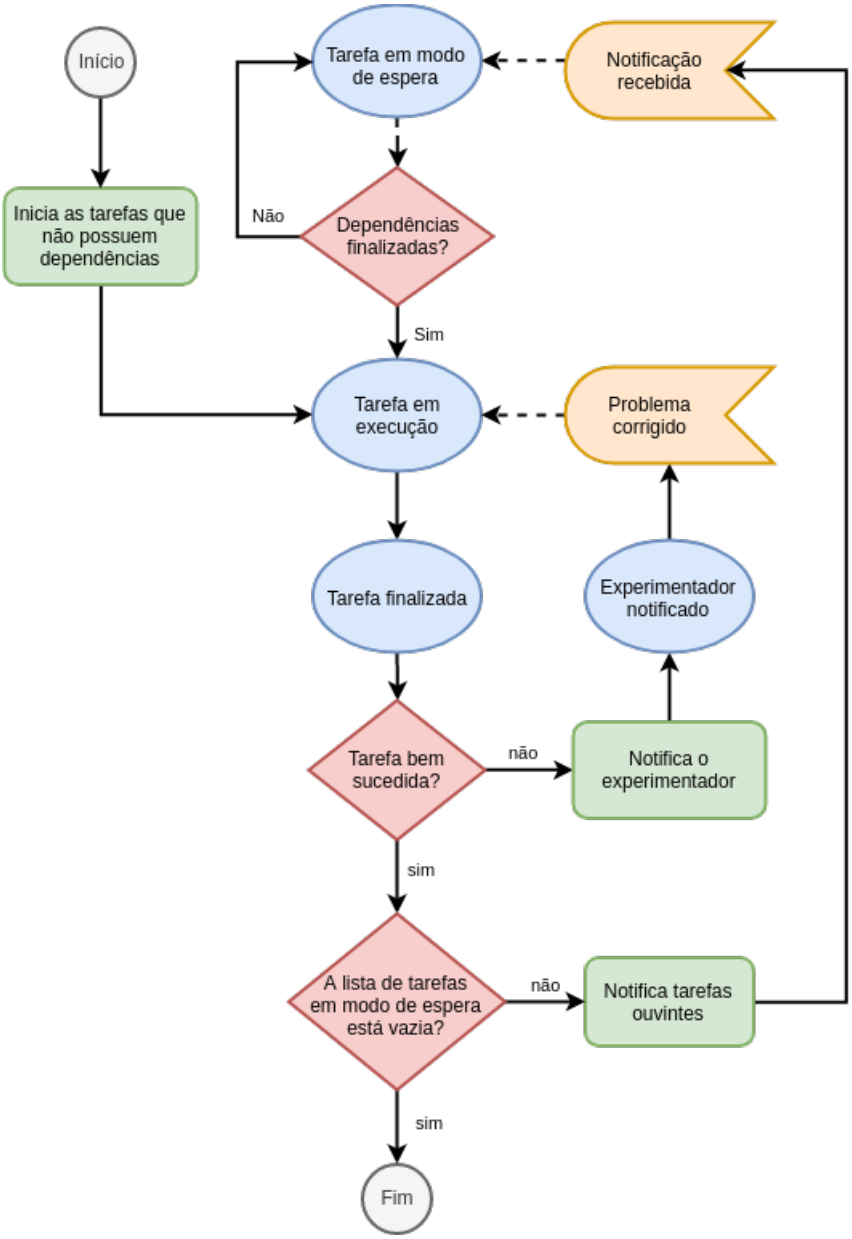

Figura 3.6: Diagrama de execução de tarefas. Este diagrama ilustra de forma sucinta o algoritmo utilizado pelo *framework* para executar as tarefas de um experimento. As formas em verde representam ações executadas pelo *Xperimentor*. As formas em azul representam estados de tarefas e do *framework*. As formas em laranja representam estados que são ativados de forma assíncrona. As setas tracejadas representam mudanças de estados provocadas por eventos assíncronos.

Inicialmente, o *framework* colocará todas as tarefas em modo de espera e iniciará todas as que não possuem dependências. Quando uma tarefa for escalonada para execução ela será redirecionada para o *backend* onde de fato ela será executada.

Quando o *framework* detectar que uma dependência D de uma tarefa T finalizou, ele removerá D da lista de dependências de T e verificará se a mesma se encontra vazia. Caso assim esteja, T será escalonada para execução, caso contrário ela irá aguardar até que suas outras dependências finalizem. O experimento termina quando todas as tarefas finalizarem de forma bem-sucedida.

3.4 Fluxo de construção de um experimento

São necessários três passos para se construir um experimento e cada um deles será apresentado a seguir:

1. Representação de tarefas.
2. Assinalamento de dependências.
3. Configuração do *cluster Kubernetes*.
4. Execução do experimento.

Para exemplificar o fluxo de criação de um experimento será adotado o seguinte problema de compilação de um executável em um ambiente *Linux*:

1. Efetuar o *download* dos seguintes códigos fonte:
   - `http://www.example.com/util.c`
   - `http://www.example.com/math.c`
   - `http://www.example.com/main.c`
2. Compilá-los em arquivos objetos separados
3. Criar uma biblioteca estática `libmath.a` com os arquivos `util.o` e `math.o`
4. Criar um executável `App` com o arquivo objeto `main.o` e a biblioteca `math.a`

As tarefas deste problema são descritas no Algoritmo 1.

Algoritmo 1: Exemplo de compilação de um executável

```
1 wget http://www.example.com/util.c
2 wget http://www.example.com/math.c
3 wget http://www.example.com/main.c
4 gcc -c util.c math.c
5 ar rcs libmath.a util.o math.o
6 gcc -c main.c -o main.o
7 gcc main.o -lmath -o App
```

3.4.1 Representação de tarefas

Para construir um experimento dentro do *Xperimentor* é necessário definir quais serão as tarefas a serem executadas. Essa definição é feita através de um documento *YAML* onde o experimentador listará cada uma das tarefas.

Toda tarefa possui obrigatoriamente um identificador único e um comando que será executado por ela. O seguinte trecho de um documento *YAML* apresenta uma lista com as tarefas presentes no Algoritmo 1.

```
tasks:
  - id: util_file
    command: "wget -P ~/build http://www.example.com/util.c"
  - id: math_file
    command: "wget -P ~/build http://www.example.com/math.c"
  - id: main_file
    command: "wget -P ~/build http://www.example.com/main.c"
  - id: compile_main
    command: "gcc -c main.c -o main.o"
  - id: util_math
    command: "gcc -c util.c math.c"
  - id: libmath
    command: "ar rcs libmath.a util.o math.o"
  - id: compile_all
    command: "gcc main.o -lmath -o App"
```

3.4.2 Assinalamento de dependências

Algumas tarefas devem aguardar a finalização de outras para que elas possam iniciar. Isso gera uma dependência que deve ser explicitada no momento da declaração de cada tarefa. O assinalamento de dependências é feito através do campo *deps*. As tarefas do seguinte código são as mesmas do anterior, porém com suas dependências assinaladas.

```
tasks:
  - id: util_file
    command: "wget -P ~/build http://www.example.com/util.c"
  - id: math_file
    command: "wget -P ~/build http://www.example.com/math.c"
  - id: main_file
    command: "wget -P ~/build http://www.example.com/main.c"
  - id: compile_main
```

```
    command: "gcc -c main.c -o main.o"
    deps: [main_file]
  - id: util_math
    command: "gcc -c util.c math.c"
    deps: [util_file, math_file]
  - id: libmath
    command: "ar rcs libmath.a util.o math.o"
    deps: [util_math]
  - id: compile_all
    command: "gcc main.o -lmath -o App"
    deps: [libmath, compile_main]
```

A Figura 3.7 ilustra o grafo gerado pelo *Xperimentor* para representar as tarefas e suas dependências.

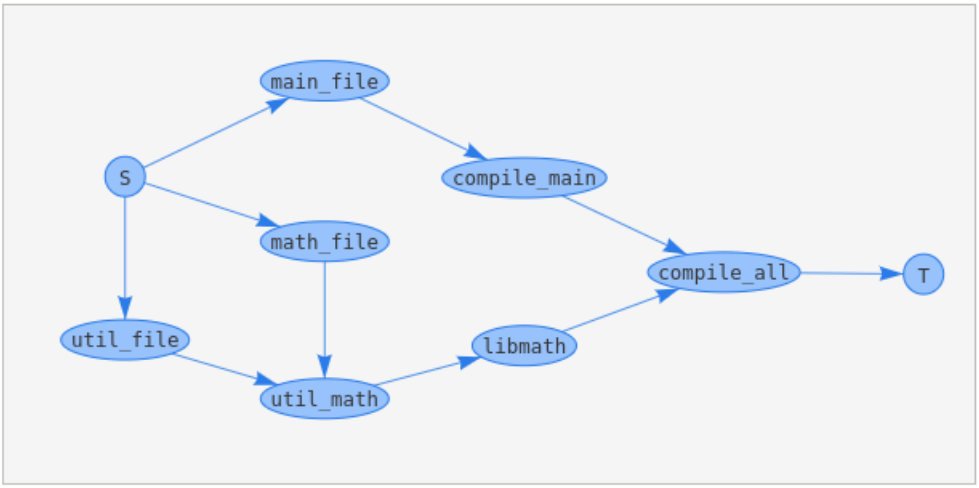

Figura 3.7: Visualização do grafo de dependências do exemplo. Observe que os nós S e T são adicionados automaticamente pelo *framework*.

3.4.3 Configuração do cluster Kubernetes

Para que as tarefas possam ser executadas é necessário que o experimentador informe o endereço de um servidor HTTP que esteja executando a aplicação do *backend*, o *Task Executor*, apresentado na Seção 3.2.1. Caso o experimento seja pequeno e não necessite de várias máquinas para executá-lo, o experimentador pode simplesmente baixar e executar o *Task Executor* localmente em sua máquina. Por padrão ele será executado na porta 5050 e no caminho `/run`. Para que o *framework* possa acessá-lo basta inserir o seguinte comando no arquivo de configuração:

```
clusterEndpoint: 'https://0.0.0.0:5050/run'
```

Existem duas soluções principais para se criar um *cluster Kubernetes*. A primeira e mais simples é contratar um serviço privado como o *Kubernetes Engine* da Google, ou o *Elastic Kubernetes Service - EKS* da Amazon. A segunda é configurar um *cluster* utilizando as máquinas em posse do experimentador.

A seguir será apresentado o processo para se configurar um *cluster* utilizando o período de testes do *Google Cloud Platform*. Este é um serviço pago, porém o experimentador pode configurar seu próprio *cluster* seguindo os passos apresentados em `https://gist.github.com/Pacheco95/b8c400b119338aae12054080d7e4739c`.

O primeiro passo é criar uma conta `https://cloud.google.com`. Após criar a conta deve-se acessar `https://console.cloud.google.com` e ativar o período de avaliação gratuita. Após a ativação será possível criar o *cluster* através do link `https://console.cloud.google.com/kubernetes`. Ao acessar este link, um painel será exibido conforme a Figura 3.8. Para criar o *cluster* basta clicar no botão **Criar cluster**.

No painel que será aberto, o experimentador poderá renomear o *cluster* para um valor desejado conforme a Figura 3.9. Em seguida basta clicar no botão **Criar** e aguardar.

Após a finalização da construção do *cluster* será necessário efetuar o *deploy* do *Task Executor* e do *Xperimentor*. Para isso basta clicar no botão **Conectar** e em seguida clicar no botão **Executar no Cloud Shell** conforme a Figura 3.10.

Após efetuar esses passos, um terminal virtual será exibido ao experimentador conforme ilustrado na Figura 3.11. Através desse terminal é possível efetuar *deployments* e outras operações que envolvem o *Kubernetes*.

Para efetuar os *deployments* basta executar os seguintes comandos:

```
curl https://gist.githubusercontent.com/Pacheco95/\
    5356a16d81c290658fe255cd1437246a/raw | bash
curl https://gist.githubusercontent.com/Pacheco95/\
    9ccb3dd4daf120a3eb7f6d21a6da4f27
```

Após executar estes comandos, o *cluster* já estará apto a ser utilizado. Por fim, deve-se executar o comando `kubectl get services` para obter os IPs externos do *Task Executor* e do *Xperimentor* conforme ilustrado na Figura 3.12. O IP do *Xperimentor* será utilizado para acessar o *frontend* do *framework* e o IP do *Task Executor* deverá ser fornecido no arquivo de configuração do experimento através do campo `ClusterEndpoint`.

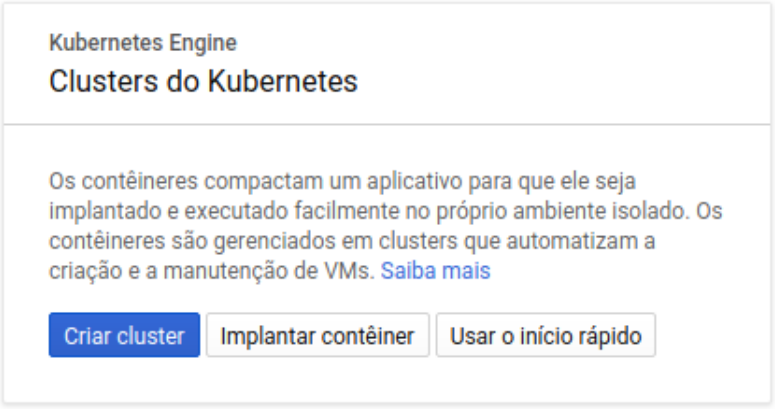

Figura 3.8: Painel de criação de *clusters*. Através deste painel, o experimentador criará um *cluster* de máquinas pré configuradas com o *Kubernetes*.

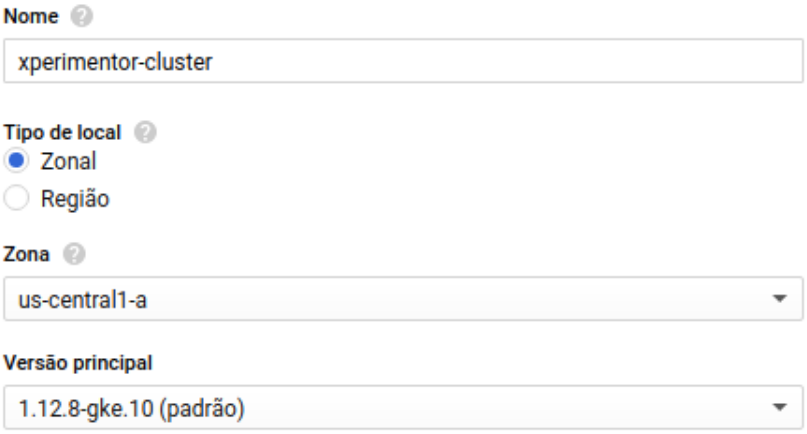

Figura 3.9: Painel de configuração de *clusters*. Através deste painel, o experimentador poderá configurar o *cluster* da maneira desejada.

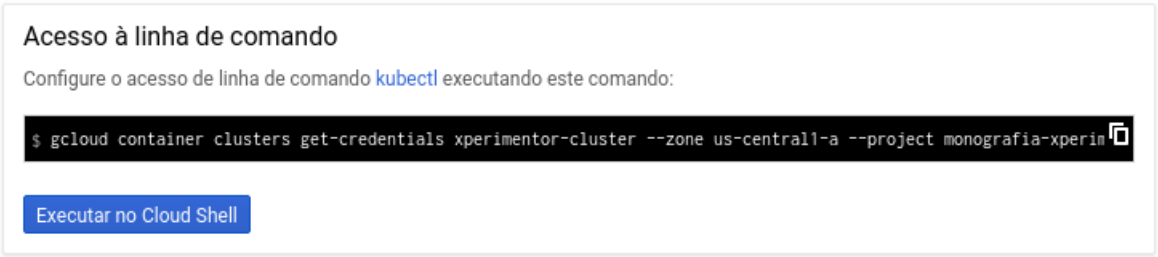

Figura 3.10: Painel de conexão com o *cluster*. Através deste painel é possível conectar-se às máquinas do *cluster* através de um terminal virtual

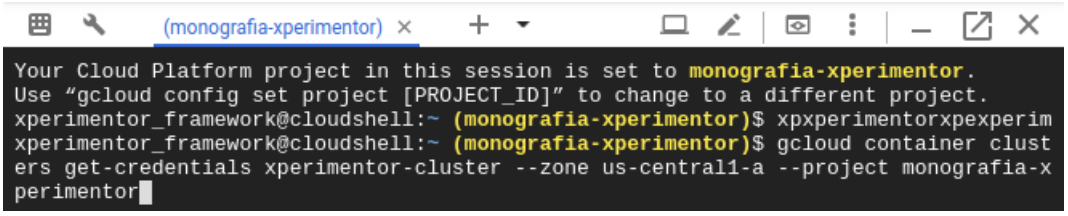

Figura 3.11: *Cloud Shell*. Através deste terminal o experimentador poderá efetuar o *deploy* de aplicações no *cluster* além de outras operações.

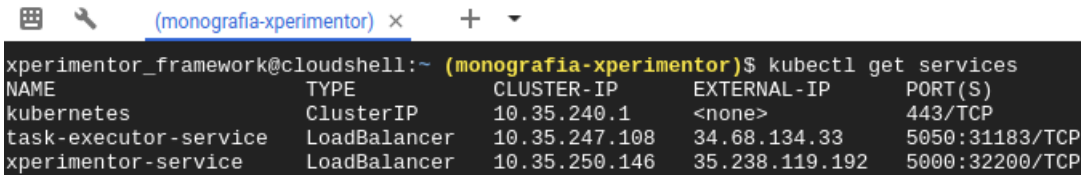

Figura 3.12: Listando serviços implantados. Através deste comando, é possível obter informações sobre todos os serviços implantados pelo *Kubernetes*

3.4.4 Execução do experimento

Após ser devidamente configurado, o experimento estará apto a ser executado. Para executá-lo basta pressionar o botão *build* para que o grafo de dependências seja construído e em seguida pressionar o botão *execute*. Se o experimento estiver configurado erroneamente, uma mensagem de erro será exibida, como pode ser visto na Figura 3.13. Caso contrário o experimento se inicia.

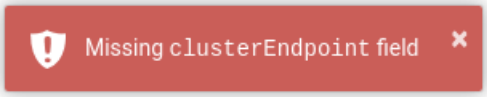

Figura 3.13: Exemplo de uma mensagem de erro de um experimento configurado indevidamente.

Se por ventura uma tarefa falhar, o *framework* notificará o experimentador para que o mesmo possa identificar a causa da falha e dar continuidade ao processo.

A Figura 3.14 ilustra um exemplo em que duas tarefas falharam porque os arquivos necessários por elas não existiam.

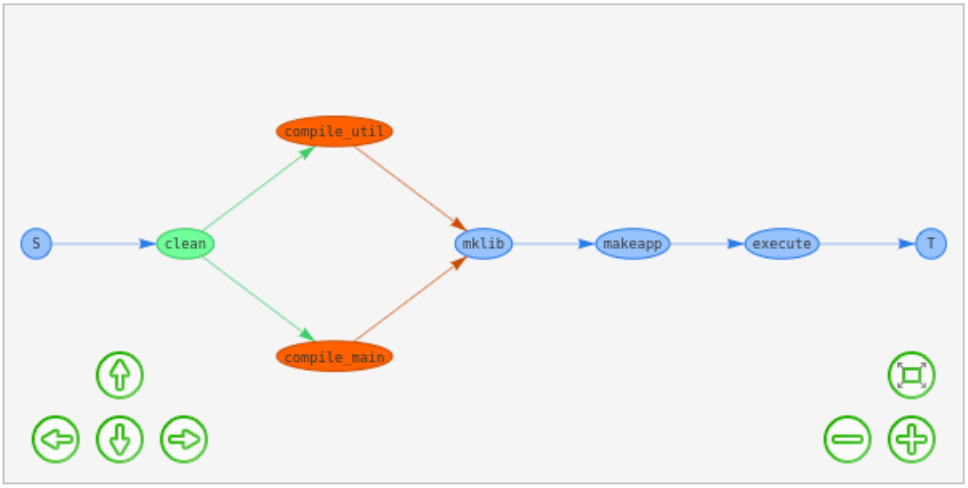

Figura 3.14: Exemplo de falha na execução de uma tarefa.

O *framework* irá notificar o experimentador conforme ilustrado na Figura 3.15. A partir dessa notificação o experimentador poderá clicar na tarefa que falhou e diagnosticar qual foi o problema ocorrido. A Figura 3.16 ilustra as informações a respeito da execução da tarefa `compile_util`.

Quando uma tarefa finalizar com um código diferente de zero ou então produzir uma mensagem na saída de erro padrão, o *Xperimentor* informará que a tarefa foi finalizada

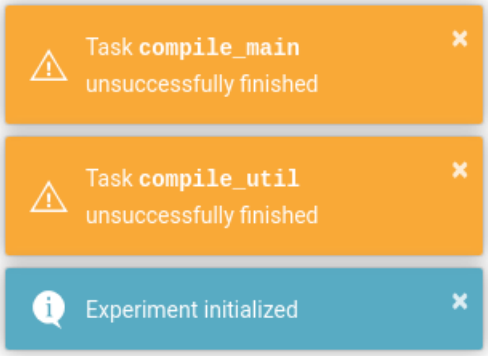

Figura 3.15: Exemplo de notificação de falha da execução de uma tarefa.

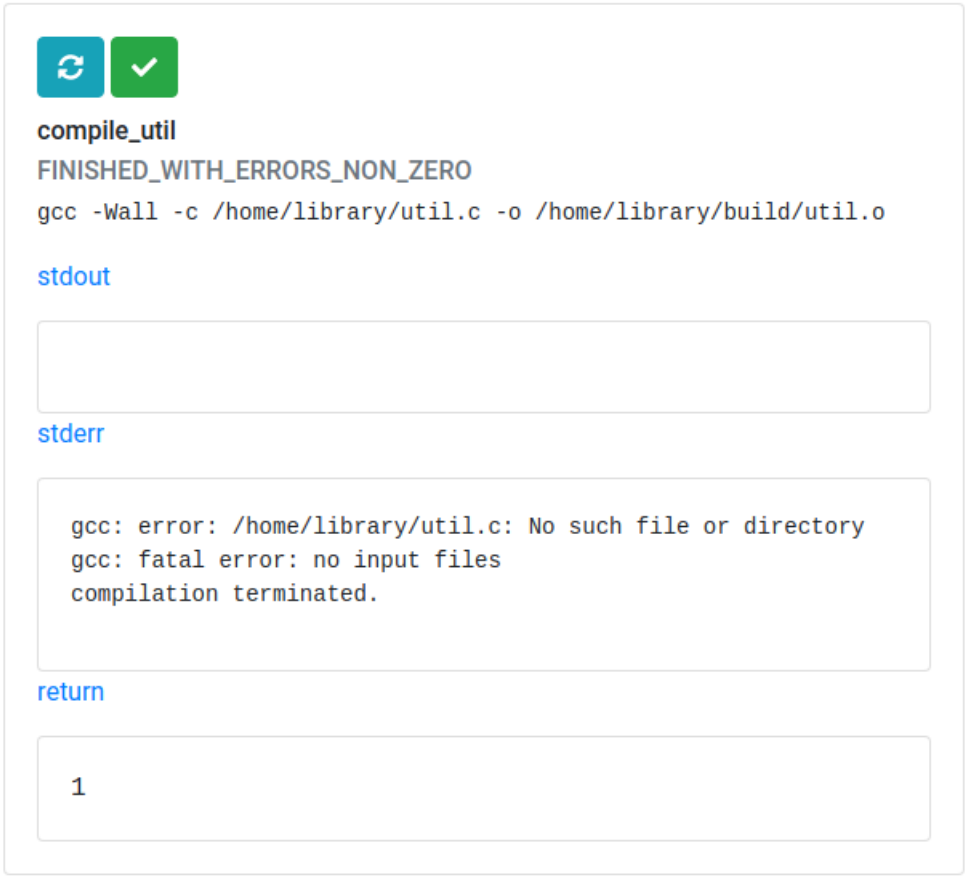

Figura 3.16: Painel de visualização do status de uma tarefa. Quando o erro for corrigido, o experimentador poderá clicar no botão azul com o símbolo de setas circulares para que a tarefa seja re-executada.

de forma mal sucedida. No entanto, podem haver casos em que o erro ocorrido não seja prejudicial à execução da tarefa. Nesse caso, o experimentador pode instruir ao *Xperimentor*

que ele aceite a tarefa como bem sucedida. Para isto, basta clicar na tarefa para abrir o painel ilustrado na Figura 3.16 e então clicar no botão verde com o símbolo de *check*. Ao executar essa ação o *Xperimentor* iniciará todas as tarefas que estejam aguardando apenas pela tarefa recém aceita. A Figura 3.17 ilustra o caso em que a tarefa *clean* falhou devido ao fato de que o *framework* não tinha permissão para deletar o diretório `build/`. Como esse não é um erro crítico, o experimentador pode clicar na tarefa *clean* e em seguida, no painel de monitoramento da tarefa, clicar no botão verde *accept* conforme demonstrado na Figura 3.18. Ao executar essa ação, o *Xperimentor* marcará a tarefa *clean* na cor roxa simbolizando que ela foi forçada como bem sucedida e em seguida iniciará as tarefas *compile_util* e *compile_main* dando continuidade à execução do experimento. O resultado dessa operação pode ser visto na Figura 3.19.

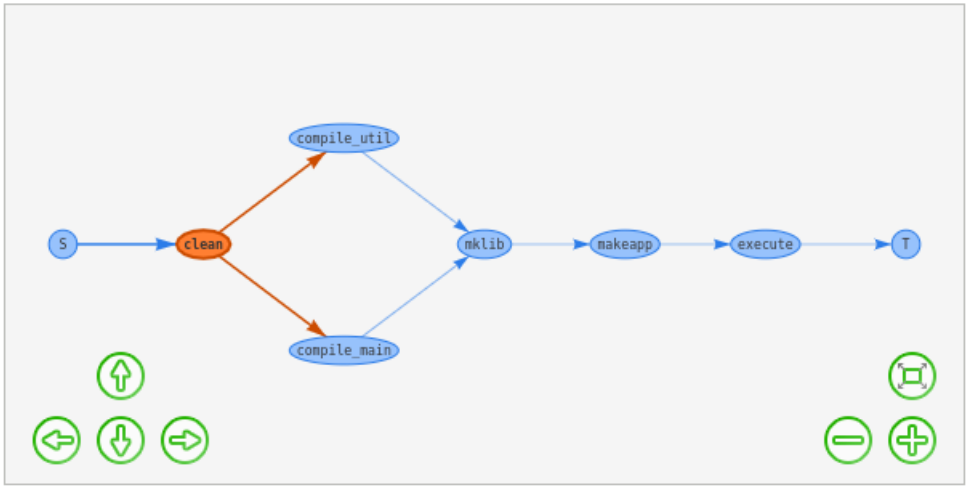

Figura 3.17: Exemplo de falha não crítica. Neste exemplo, a tarefa *clean* falhou devido à erros de permissão mas este erro não impede que o experimento possa continuar.

Toda tarefa possui um estado que varia durante a execução do experimento. Existem 8 estados ao total, sendo eles:

- **WAITING**
  Status inicial de toda tarefa.
- **RUNNING**
  Sinaliza uma tarefa em execução.
- **SUCCESSFULLY_FINISHED**
  Sinaliza uma tarefa finalizada de forma bem sucedida.
- **FINISHED_WITH_ERRORS**
  Sinaliza que uma tarefa finalizou com o fluxo de erro padrão não vazio.

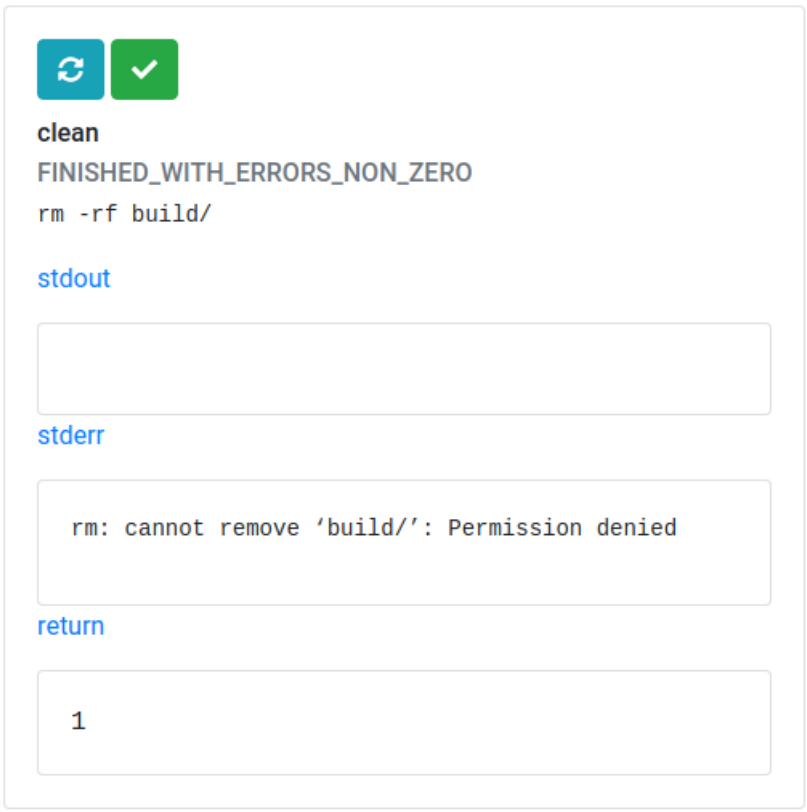

Figura 3.18: Marcando uma tarefa como bem sucedida. Neste exemplo, a tarefa *clean* falhou devido à erros de permissão mas este erro não impede que o experimento possa continuar. Sendo assim, o experimentador pode instruir ao *Xperimentor* que ele aceite a tarefa como bem sucedida e dê continuidade ao experimento. Para isto, basta clicar no botão verde.

- **FINISHED_WITH_NON_ZERO**
  Sinaliza que uma tarefa finalizou com código diferente de zero.
- **FINISHED_WITH_ERRORS_NON_ZERO**
  Sinaliza que uma tarefa finalizou com o fluxo de erro padrão não vazio e com código diferente de zero.
- **FAILED**
  Sinaliza que uma tarefa não pôde ser executada.
- **FORCED_SUCCESSFULLY_FINISHED**
  Sinaliza que uma tarefa foi marcada como bem sucedida.

A Figura 3.20 ilustra esses estados em forma de uma máquina de estados finita.

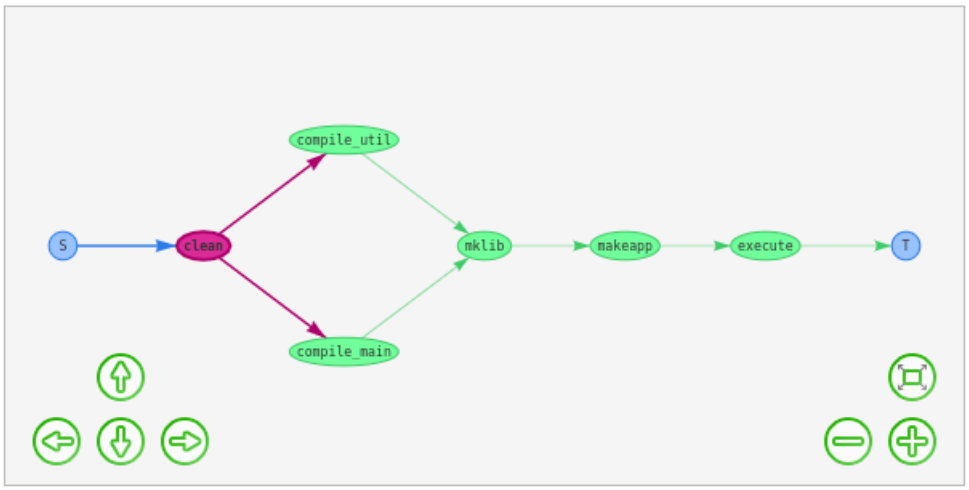

Figura 3.19: Visualização de uma tarefa forçada como bem sucedida. Ao forçar uma tarefa como bem sucedida, o *Xperimentor* irá renderizá-la com a cor roxa para que o experimentador possa identificar que ela falhou mas o erro foi ignorado. Uma vez que a tarefa *clean* foi marcada como bem sucedida, o *Xperimentor* dará início às tarefas *compile_util* e *compile_main*.

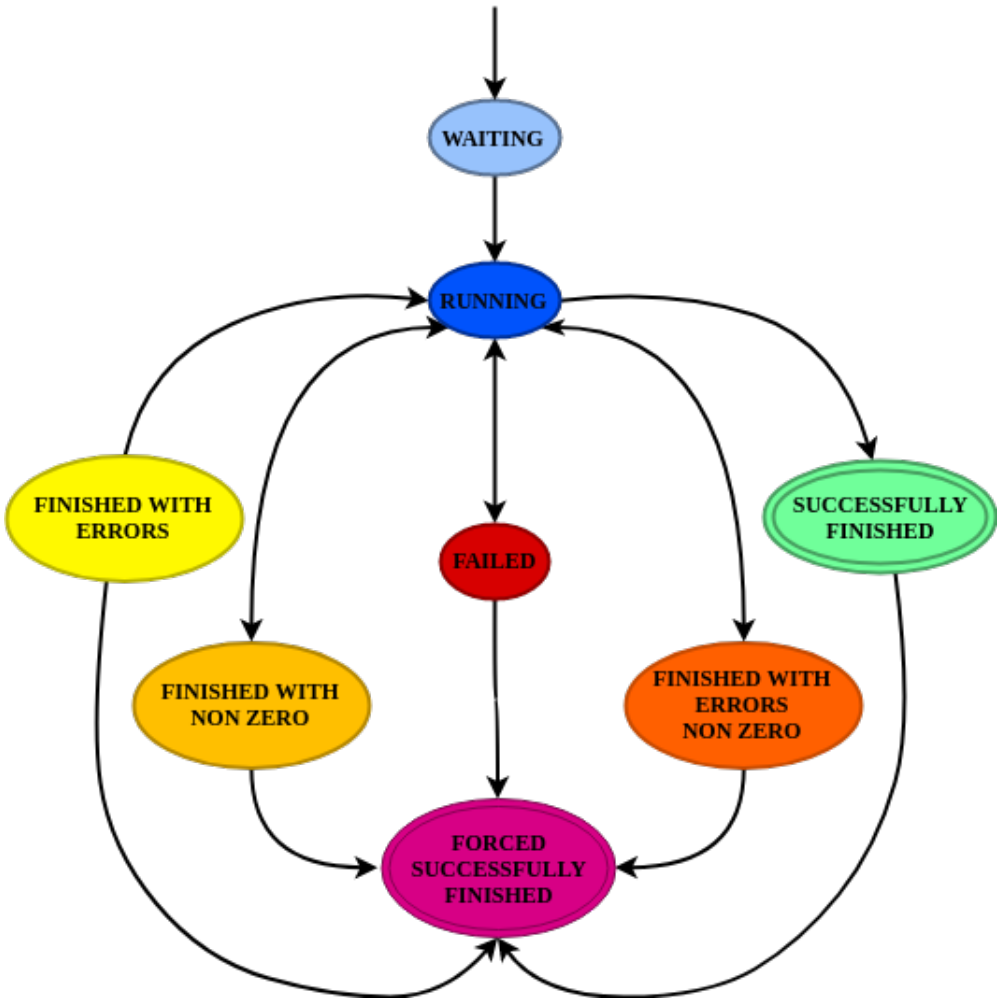

Figura 3.20: Estados de uma tarefa. Esta figura ilustra o autômato finito determinista que controla os estados de uma tarefa. A descrição das transições foram omitidas a fim de simplificar a visualização.

Capítulo 4

Experimentos Computacionais

Nesta seção será apresentado como estudo de caso a construção de um experimento de Sistema de Recomendação Híbrido. A Seção 4.1 introduz o conceito básico de Sistemas de Recomendação contextualizando os experimentos conduzidos. A Seção 4.2 aborda a preparação do experimento e as ferramentas utilizadas. A Seção 4.3 demonstra os passos executados para construção do experimento. Por fim, a Seção 4.4 apresenta e discute os resultados obtidos.

4.1 Contextualização

Sistemas de Recomendação são técnicas baseadas em aprendizado de máquina utilizadas para recomendar itens aos usuários de acordo com seus perfis de consumo. Neste trabalho, serão abordadas duas das principais categorias de recomendação: a filtragem colaborativa e a híbrida. A filtragem colaborativa consiste na aplicação de algoritmos que recomendam aos consumidores itens que pessoas com perfis semelhantes de consumo preferiram no passado. Para fazer estas recomendações os algoritmos se baseiam em uma matriz $R = MxN$ de *ratings* onde $R_{m,n}$ representa o *rating* dado pelo consumidor m para o item n. Através dessa matriz os algoritmos podem fazer predições dos *ratings* que os usuários forneceriam para itens que ainda não foram classificados por eles. A hibridização é a aplicação conjunta de diversas técnicas para produzir resultados mais precisos. Neste trabalho, a hibridização foi feita utilizando-se diversos algoritmos de filtragem colaborativa. Esta hibridização produz recomendações mais relevantes através da ponderação dos algoritmos de filtragem colaborativa, por meio de técnicas de regressão linear, dando uma maior prioridade aos algoritmos mais relevantes para um determinado usuário [4, 5].

Alguns algoritmos de filtragem colaborativa são sensíveis à estrutura dos dados presentes na matriz de *ratings*, fazendo com que sua performance oscile para diferentes estruturas. A fim de minimizar essas diferenças serão utilizadas métricas de caracterização dos dados extraídas da matriz de *ratings*. Essas métricas são denominadas *metafeatures* e através delas é possível aprimorar a ponderação dos algoritmos de filtragem colaborativa realizada pelos

algoritmos híbridos para que ao final de todo o processo sejam geradas recomendações ainda mais precisas e relevantes.

4.2 Preparação do Experimento

O Sistema de Recomendação Híbrido abordado foi construído seguindo-se um fluxo básico de cinco etapas, sendo elas:

1. Calcular as *metafeatures*. Esta etapa será efetuada por um programa desenvolvido em Java denominado *MetaFeatureCalculator*.

2. Executar os algoritmos de filtragem colaborativa. Esta etapa será efetuada por um programa desenvolvido em *Python* denominado *PredictionCF*.

3. Executar os algoritmos híbridos com os resultados dos cálculos das *metafeatures* e das filtragens colaborativas. Esta etapa será efetuada por um programa desenvolvido em *Python* denominado *PredictionWHF*.

4. Avaliar os resultados das filtragens para determinar o quão precisas e relevantes foram as predições. Esta etapa será efetuada por um programa desenvolvido em Java denominado *EvaluatorCF*.

5. Avaliar os resultados dos algoritmos híbridos para determinar o quão precisos e relevantes foram as predições. Esta etapa será efetuada por um programa desenvolvido em Java denominado *EvaluatorWHF*.

6. Efetuar cálculos estatísticos sobre os resultados e comparar a eficácia dos algoritmos de predição. Esta etapa será efetuada por um programa desenvolvido em Java denominado *CalculateStats*.

A Figura 4.1 ilustra a relação entre estas etapas.

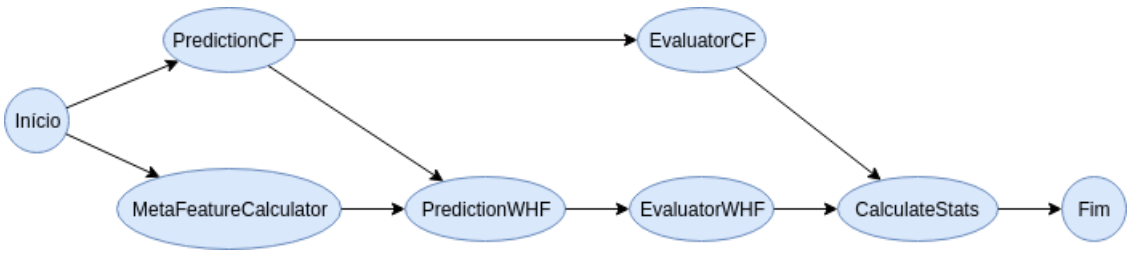

Figura 4.1: Visão geral do processo de recomendação.

Para que o experimento seja validado será feita a análise de três bases de dados. Sobre cada base será aplicada uma validação cruzada de 5 *folds* a fim de se obter métricas estatísticas para a validação.

Os recursos utilizados no experimento estão descritos a seguir:

- Bancos de dados
  - Bookcrossing
  - Jester
  - ML1M
- *Metafeatures*
  - PCR
  - PR
  - GINI
  - PEARSON
  - POMEAN
  - SD
- Algoritmos de Filtragem Colaborativa
  - Sigmoid
  - Biased
  - MF
  - Uknn
  - Iknn
  - SVD
  - Latent
  - Factor
  - BiPolar
  - SO
- Algoritmos híbridos
  - STREAM
  - FWLS

  - HS

- Medidas estatísticas
  - Média
  - Intervalo de confiança

A explicação de cada recurso foge ao escopo deste trabalho. O leitor encontrará informações detalhadas sobre ambos em [4, 5].

4.3 Construção do Experimento

O experimento sendo abordado neste capítulo possui uma característica combinatória que resultará em um número elevado de combinações. Cada combinação gerada deverá ser transcrita em uma tarefa exequível pelo *Xperimentor*.

Todo o fluxo apresentado na Figura 4.1 deverá ser replicado para diferentes combinações de bancos de dados, *folds*, *metafeatures* e algoritmos. Entretanto, alguns algoritmos e métricas não serão utilizados em algumas bases por não serem aplicáveis.

Como o número de combinações geradas é grande, torna-se inviável para o experimentador criá-las manualmente. Para contornar este problema foi desenvolvido um programa que gera estas combinações. Este programa recebe um arquivo YAML contendo definições de processos e receitas que instruem como gerar tarefas através destes processos. Cada processo presente neste arquivo deve possuir um identificador, um comando, uma lista de argumentos de entrada, uma lista de argumentos de saída e um padrão para gerar um arquivo de *log*. As listas de argumentos de entrada e saída serão utilizadas para identificar dependências entre os processos criados. Dessa forma, se um processo X produzir A como saída e outro processo Y receber A como entrada, o gerador fará com que toda tarefa gerada através de Y seja dependente das tarefas geradas através de X.

O seguinte trecho define um processo através do qual serão geradas as tarefas do *MetafeatureCalculator*.

```
- id: MetaFeatureCalculator
  command: "java -jar MetricCalculator.jar {DB} {Fold} {MF} 60 0"
  in: [DB, Fold, MF]
  out: [calcMF]
  log: "MetaFeatureCalculator-{DB}-{Fold}-{MF}.out"
```

Para que sejam geradas as tarefas desse processo, receitas devem ser fornecidas. Uma receita contém uma lista de valores para cada parâmetro de um processo.

Segue abaixo um exemplo de uma receita para o processo *MetaFeatureCalculator*. O arquivo completo se encontra no Apêndice .1.

```
- id: RecipeExample
  uses:
      DB: ["Bookcrossing"]
      Fold: ["F1", "F2"]
      MF: ["PCR", "PR"]
```

A partir dessa receita, o gerador fará um produto cartesiano dos parâmetros passados e irá gerar as seguintes combinações:

- `java -jar MetricCalculator.jar Bookcrossing F1 PCR 60 0 > MetaFeatureCalculator-Bookcrossing-F1-PCR.out"`
- `java -jar MetricCalculator.jar Bookcrossing F1 PR 60 0 > MetaFeatureCalculator-Bookcrossing-F1-PR.out"`
- `java -jar MetricCalculator.jar Bookcrossing F2 PCR 60 0 > MetaFeatureCalculator-Bookcrossing-F2-PCR.out"`
- `java -jar MetricCalculator.jar Bookcrossing F2 PR 60 0 > MetaFeatureCalculator-Bookcrossing-F2-PR.out"`

Para este experimento foram geradas 389 tarefas que resultaram no grafo presente na Figura 4.2.

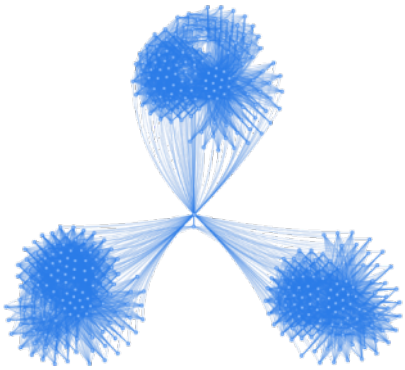

Figura 4.2: Grafo final do experimento de recomendação híbrida.

4.4 Análise e Discussão dos Resultados

Através dos resultados obtidos pela execução do experimento conclui-se que o *framework* é capaz de distribuir todas as tarefas e executá-las corretamente respeitando as dependências. Após a codificação do experimento não foi necessário nenhum tipo de intervenção humana,

indicando que os principais objetivos deste trabalho foram alcançados. Para um experimento deste porte seriam necessárias várias horas de experimentação supervisionada que resultaria em baixo grau de paralelismo. Sem o *framework*, o experimentador deveria monitorar a máquina durante todo o experimento esperando que uma tarefa terminasse para que a próxima fosse executada. Entretanto, o *Xperimentor* ainda não possui um sistema de armazenamento persistente. Os dados que são acessados por uma tarefa devem ser replicados nas máquinas do *cluster* de forma manual pelo experimentador. Uma solução relativamente simples para esse problema seria utilizar uma das máquinas como sistema de armazenamento. Tal máquina deve conter um diretório que seja visível às demais através do protocolo *Network File System* (NFS). Dessa forma, é possível implantar um serviço no *Kubernetes* que ficará responsável exclusivamente por fornecer uma interface de acesso às outras máquinas do *cluster*. Para que uma máquina possa acessar o servidor NFS, é necessário que o diretório exposto seja montado dentro dos contêineres como o diretório de trabalho atual.

Capítulo 5

Conclusão

Neste trabalho foi apresentado o *Xperimentor*, um *framework* para o gerenciamento de execução de experimentos computacionais.

Através dos resultados obtidos nos experimentos foi possível concluir que o *Xperimentor* é capaz de executar experimentos complexos que produzem grafos de dependências muito densos. No entanto, algumas funcionalidades ainda não foram implementadas e ficarão como atividades da continuação deste trabalho sendo elas:

- Integrar o *Xperimentor* com o banco de dados não-relacional *Mongo DB* para que o experimentador possa armazenar seus experimentos. Com a adoção dessa estratégia de armazenamento persistente, o experimentador poderá pausar um experimento e continuá-lo posteriormente.
- Construir uma interface gráfica que permita a construção e edição de um experimento através de interações *drag and drop*.
- Permitir ao experimentador agrupar tarefas relacionadas em um único nó para que a visualização do experimento seja mais clara.
- Desenvolver um *software* que auxilie o experimentador a configurar um *cluster Kubernetes* pessoal.

Capítulo 6

Apêndices

.1 Documento de configuração para o gerador de tarefas

```
# Documento utilizado para gerar as tarefas do experimento
# apresentado na Seção 4.3
processes:
  - id: MetaFeatureCalculator
    command: "java -jar MetaFeatureCalculator.jar {DB} {Fold} {MF} 60 0"
    in: [DB, Fold, MF]
    out: [calcMF]
    log: "MetaFeatureCalculator-{DB}-{Fold}-{MF}.out"

  - id: predictionCF
    command: "python -u PredictionCF.py {DB} {Alg} {Fold} 60 0"
    in: [DB, Alg, Fold]
    out: [execAlg]
    log: "Prediction-CF-{DB}-{Alg}-{Fold}.out"

  - id: Hybrid
    command: "python -u PredictionWHF.py {DB} {HF} {Fold} 60 0"
    in: [DB, calcMF, execAlg, HF, Fold]
    out: [execHF]
    log: "../Predictions-WHF-{DB}-{HF}-{Fold}.out"

  - id: EvaluatorCF
    command: "java -jar EvaluatorCF.jar {DB} {Fold} {Eval} CF 60 0"
    in: [DB, Fold, execAlg]
    out: [Eval]
```

```
    log: "EvaluatorCF-{DB}-{Fold}-{Eval}-{execAlg}.out"

  - id: EvaluatorWHF
    command: "java -jar EvaluatorWHF.jar {DB} {Fold} {Eval} WHF 60 0"
    in: [DB, Fold, execHF]
    out: [Eval]
    log: "EvaluatorWHF-{DB}-{Fold}-{Eval}-{HF}.out"

  - id: CalculateStatistics
    command: "java -jar CalculateStats.jar {DB} ALL {Eval} {Stats} 60 0"
    in: [DB, Eval]
    out: [Stats]
    log: "CalculateStats-{DB}-{Eval}-{Stats}.out"

recipeDefaults:
  Fold: ["F1234-5", "F1235-4", "F1245-3", "F1345-2", "F2345-1"]
  HF: ["STREAM", "FWLS", "HR"]
  execAlg: ["ALL"]
  calcMF: ["ALL"]
  execHF: ["STREAM", "FWLS", "HR"]
  Eval: ["RMSE", "F1", "EPC", "EILD"]
  Stats: ["mean", "IC"]

recipes:
  - id: ExBC
    uses:
      DB: ["Bookcrossing"]
      MF: ["PCR", "PR", "GINI", "PEARSON", "PQMEAN", "SD"]
      Alg: ["Sigmoid", "Biased", "MF", "Uknn",
            "SVD", "Latent", "Factor", "BiPolar", "SO"]
  - id: ExJE
    uses:
      DB: ["Jester"]
      MF: ["PCR", "PR", "PEARSON", "PQMEAN"]
      Alg: ["Biased", "MF", "Iknn", "SVD", "Factor", "BiPolar", "SO"]
  - id: ExML
    uses:
      DB: ["ML1M"]
```

```
MF: ["PCR", "GINI", "PQMEAN", "SD"]
Alg: ["Sigmoid", "Biased", "MF", "Uknn", "Iknn",
      "SVD", "Latent", "Factor", "BiPolar", "SO"]
```

Referências Bibliográficas

[1] S. Azarnoosh, M. Rynge, G. Juve, E. Deelman, M. Niec, M. Malawski, and R. F. Da Silva. Introducing precip: an api for managing repeatable experiments in the cloud. In *2013 IEEE 5th international conference on cloud computing technology and science (CloudCom)*, pages 19–26. IEEE, 2013.

[2] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333, May 2017.

[3] G. Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.

[4] R. S. Fortes, A. R. R. de Freitas, and M. A. Gonçalves. A multicriteria evaluation of hybrid recommender systems: On the usefulness of input data characteristics. In *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 623–633. INSTICC, SciTePress, 2017.

[5] R. S. Fortes, A. Lacerda, A. Freitas, C. Bruckner, D. Coelho, and M. Gonçalves. User-oriented objective prioritization for meta-featured multi-objective recommender systems. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*, UMAP '18, pages 311–316, New York, NY, USA, 2018. ACM.

[6] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 322–331. IEEE, 2013.

[7] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[8] G. von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Vockler, R. J. Figueiredo, J. Fortes, and K. Keahey. Design of the fu-

turegrid experiment management framework. In *2010 Gateway Computing Environments Workshop (GCE 2010)(GCE)*, volume 00, pages 1–10, Nov. 2011.