UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

UM SISTEMA *WEB* PARA EXECUÇÃO REMOTA DE APLICAÇÕES DE ALTO DESEMPENHO

TRABALHO DE GRADUAÇÃO

Otávio Migliavacca Madalosso

Santa Maria, RS, Brasil

2015

UM SISTEMA *WEB* PARA EXECUÇÃO REMOTA DE APLICAÇÕES DE ALTO DESEMPENHO

Otávio Migliavacca Madalosso

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Profª. Drª. Andrea Schwertner Charão

Santa Maria, RS, Brasil

2015

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

UM SISTEMA *WEB* PARA EXECUÇÃO REMOTA DE APLICAÇÕES DE ALTO DESEMPENHO

elaborado por
Otávio Migliavacca Madalosso

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Andrea Schwertner Charão, Dr[a].
(Presidente/Orientadora)

Benhur De Oliveira Stein, Prof. Dr. (UFSM)

Henrique Pereira, MSc. (CPD - UFSM)

Santa Maria, 08 de Outubro de 2015.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

UM SISTEMA *WEB* PARA EXECUÇÃO REMOTA DE APLICAÇÕES DE ALTO DESEMPENHO

AUTOR: OTÁVIO MIGLIAVACCA MADALOSSO
ORIENTADORA: ANDREA SCHWERTNER CHARÃO
Local da Defesa e Data: Santa Maria, 08 de Outubro de 2015.

Algumas áreas de pesquisa utilizam constantemente algoritmos que demandam alto desempenho dos seus ambientes de execução. Ocasionalmente, surgem algoritmos novos, com diferentes propriedades, que se propõem a resolver um problema de forma mais eficiente e/ou completa. Infelizmente, é comum que esses algoritmos fiquem restritos a ambientes institucionais, limitando muito a sua visibilidade para a comunidade de pesquisa. Este trabalho tem como objetivo criar um portal que permita ao usuário solicitar a execução remota de um algoritmo de acordo com as configurações que o sistema oferecer.

Palavras-chave: Computação de alto desempenho. Programação Web. Framework Django. Execução Remota. Python.

SUMÁRIO

1 INTRODUÇÃO

Algoritmos com grande custo computacional são facilmente encontrados em áreas como meteorologia, biologia e astronomia. Esses algoritmos possuem a característica de consumir um nível elevado de processamento, e consequentemente, os tempos necessários para suas conclusões tendem a ser longos e variam dependendo do ambiente aonde são executados.

Pesquisadores destas áreas podem vir a desenvolver novas implementações de algoritmos utilizados pela comunidade de pesquisa. Essas implementações podem trazer muitos benefícios para outros pesquisadores que necessitam deste tipo de solução. Infelizmente, é comum essas implementações ficarem restritas a ambientes privados, não por questões de licença, mas simplesmente pela ausência de um método prático para disponibilizá-la ao público.

Baseado nessa situação, surge a ideia de desenvolver um portal *web* que permita a execução de algoritmos remotamente, de acordo com as configurações feitas pelo administrador. Desta forma, o usuário seria capaz de utilizar dados próprios para que sejam processados pelos algoritmos, e consiga obter os resultados quando a tarefa for concluída.

Portais como o descrito acima já existem(citar NeSI), porém são desenvolvidos de acordo com a estrutura aonde serão mantidos. A vantagem deste projeto é que será mais genérico, prevendo a sua utilização por diferentes públicos e em diferentes ambientes.

Um algoritmo que se enquadra no propósito do portal e que será utilizado durante o desenvolvimento do mesmo, é a uma versão do algoritmo *Friends-of-Friends*(SURHUD MORE, ????) de complexidade n*log(n) paralelizado através do *framework* OpenMP. Essa variação do algoritmo foi desenvolvida em um projeto de pesquisa vinculado ao INPE (Instituto Nacional de Pesquisas Espaciais) pelo autor deste trabalho.

1.1 Objetivos e Justificativa

1.1.1 Objetivo Geral

O objetivo deste trabalho é criar um modelo de portal *web* que possibilite aos usuários cadastrados no portal executar algoritmos utilizando diferentes dados e disponibilizar o resultado da execução após sua conclusão.

1.1.2 Justificativa

O projeto é capaz de gerar benefícios significativos para a comunidade de pesquisa de diversas áreas, criando um modelo de ambiente que facilite a divulgação e teste de resultados de algoritmos alternativos para resolução de problemas comuns.

Além de servir como modelo, o projeto disponibilizará um algoritmo que se enquadra na categoria alvo do projeto: a versão de complexidade n*log(n) e paralela do *friends-of-friends*.

2 FUNDAMENTOS E REVISÃO DE LITERATURA

Para a produção deste projeto foi necessário fazer um estudo de ferramentas que iriam ser utilizadas para auxiliar na execução do trabalho. A natureza de um portal web, por exemplo, já apresenta um grande número possibilidades de *frameworks web* com o propósito de auxiliarem no desenvolvimento deste portal. Priorizando a velocidade de desenvolvimento e afinidade do autor com a linguagem, foi escolhido o *framework* Django(DJANGO, 2005), para ser utilizado na implementação.

Também é necessário o uso de um gerenciador de tarefas para lidar com as tarefas que serão geradas através do portal. Como os algoritmos que serão utilizados neste portal podem demandar alto custo computacional, é necessário que o ambiente aonde o portal será mantido não seja o mesmo aonde as tarefas serão executadas. Assim, impossibilitando a concorrência entre a execução algoritmo e do processo que mantém o portal. Seria inapropriado o portal aguardar a conclusão de uma tarefa ser concluída para só então retornar uma resposta ao usuário que a solicitou, para resolver isso, foi escolhido o gerenciador Celery para realizar o controle dessas execuções.

2.1 Framework Django

Django(DJANGO, 2005) é um *framework* escrito na linguagem *Python* para o desenvolvimento de aplicações *web* que encoraja o desenvolvimento ágil, em alto nível e com design pragmático. Desde julho de 2005 é um um projeto de código aberto publicado sob a licença BSD.

O principal objetivo do *framework* é facilitar a criação de *websites* dirigidos a banco de dados e se relaciona muito com a política de *DRY (Don't Repeat Yourself)* e com o conceito de aplicações modulares (ou plugáveis). Dentre as características desejadas para realização do projeto, o Django pareceu adequado por apresentar suporte a tecnologias que favorecem o desenvolvimento ágil do projeto, como mapeamento objeto-relacional e um modelo de desenvolvimento em camadas *(Model-View-Template)*.

2.2 Celery Task Queue

Celery(CELERY, 2009) é um gerenciador de tarefas assíncronas baseado em troca de mensagens entre a aplicação que irá criar tarefas, a lista de tarefas pendentes e os *workers* (processos que irão executar as tarefas). O fluxo de trabalho do Celery pode ser dividido nos estágios de requisição, execução e retorno de tarefas, a requisição é feita através do portal, pelo usuário, e enfileirada na lista de tarefas pendentes. Enquanto houverem tarefas pendentes, elas serão atribuídas aos *workers* que fazem parte do sistema, que irão executar e retornar ao portal o resultado da execução.

Além de resolver o problema de controle de tarefas, essa aplicação também traz um outro benefício ao projeto que é a possibilidade de expandir o número de *workers* que irão tratar das tarefas, ou seja, conforme a demanda de execuções aumente, é prático configurar outra máquina para fazer o papel de mais um *worker*.

2.3 Redis

Redis[1] é uma ferramenta de armazenamento e comunicação de dados mantida sob a licença BSD (open-source). Ele será utilizado neste projeto como *broker* (ferramenta para fazer a troca de mensagens) entre as máquinas que vão agir como *workers* e a máquina que irá manter a aplicação disponível aos usuários (Django server).

Além do Redis, há somente uma outra opção estável de *broker* para ser utilizado em conjunto com o Celery: o RabbitMQ. Dentre essas duas opções, o Redis foi escolhido por apresentar mais informação na documentação e mais participação da comunidade desenvolvedora que utiliza Celery.

2.4 Friends of Friends

O *Friends-of-Friends* é um algoritmo utilizado para manipular e analisar grandes quantidades de dados produzidos por simulações da área da astronomia, mais especificamente em tópicos como a distribuição de matéria escura em grande escala, a formação de halos de matéria escura, e a formação e evolução de galáxias e aglomerados. Essas simulações tem um papel fundamental no estudo desses assuntos.(BERTSCHINGER, 1998; G. EFSTATHIOU M. DAVIS,

[1] http://redis.io/

1985)

Na 15ª edição do ERAD-RS[2], foram publicados resultados de execuções de uma nova implementação do *Friends-of-Friends*(OTAVIO M. MADALOSSO, 2015) originados de um projeto de pesquisa cujo objetivo era reduzir a complexidade do algoritmo e paralelizá-lo para que obtivesse uma diminuição de seu tempo de execução.

O algoritmo funciona utilizando uma entrada de dados composta por posições de *n* corpos celestes que devem ser agrupados de acordo com um dado raio. Quando dois corpos estão posicionados a uma distância menor do que a do raio informado, eles pertencem a um mesmo grupo e qualquer outro corpo que estiver a uma distância menor ou igual ao raio de qualquer integrante de um grupo, também pertence ao grupo. O resultado esperado do algoritmo é informar quais corpos estão relacionados entre sí seguindo essa regra.[3]

Uma característica significativa desse algoritmo é o grande volume de dados que compõem o arquivo de entrada, durante o desenvolvimento do algoritmo, foi utilizado um arquivo de entrada com informação de 317 mil corpos celestes. Como esse arquivo precisa ser enviado a partir do usuário para o sistema, é necessário que seja determinado um limite do tamanho do arquivo e um prazo de validade para o qual esse arquivo continuará disponível no sistema após sua utilização. Caso contrário é inevitável que, conforme o portal seja utilizado, os recursos de memória do ambiente destinados a manter os arquivos sejam comprometidos.

2.5 Trabalhos Relacionados

Os portais de eResearch por exemplo, focam em usuários pesquisadores, que demandam alta capacidade computacional para realizarem suas pesquisas, além de acesso a certos conjuntos de dados e aplicações ligadas a suas áreas de pesquisa. Todos esses recursos são disponibilizados para o usuário através de complexas uniões de diferentes recursos, gerenciados pelas instituições que mantém o serviço e entregues para o usuário final de forma transparente, facilitando a utilização pelo mesmo.

A seguir segue uma lista de alguns desses portais:

- New Zealand eScience - NeSI
- The National e-Science Centre - NeSC

[2] Escola Regional de Alto Desempenho do Rio Grande do Sul
[3] Algoritmo Friends of Friends https://gclusters11.wikispaces.com/SBCS+1.1+FOF+Masses

- Monash eResearch Centre

Existem também portais que a interação que o usuário faz com o sistema é a produção de código fonte em determinada linguagem. O objetivo desses portais varia entre o ensino de novas técnicas de programação, ensino das linguagens e desafios de desempenho de algoritmos. Para as situações de ensino, o sistema pode instruir o usuário como resolver determinado problema e solicitar que o mesmo resolva algum problema semelhante, provando que aprendeu o conteúdo proposto. Para desafios de algoritmos, o usuário envia a sua solução proposta para o sistema executar, e após a execução, recebe uma avaliação que é publicada em um rank, promovendo competição entre os usuários. Exemplos de portais que seguem essa metodologia:

- HackerRank
- CodeinGame
- Codecademy

Outro tipo de portal é o Algorithmia[4], uma plataforma que serve de intermediário entre desenvolvedores e utilizadores de algoritmos, o Algorithmia permite aos usuários cadastrados disponibilizar e/ou utilizar algoritmos de outros usuários. Um desenvolvedor que deseja utilizar algum dos algoritmos disponíveis no Algorithmia em um projeto próprio, adiciona uma chamada do algoritmo selecionado e paga de acordo com a utilização e os preços estabelecidos pelo site.

[4] https://algorithmia.com

3 DESENVOLVIMENTO

A metodologia do projeto concentrou-se em dar prioridade aos requisitos críticos ao projeto. Os requisitos foram determinados com base no *story telling* (Figura 3.1) criado no início do projeto e dela foram extraidos os requisitos e funcionalidades do sistema, gerando o backlog do produto que será apresentado a seguir.

Através desse *story telling*, foi desenvolvido o *backlog* do produto, criando funcionalidades baseado na estória e atribuindo a elas diferentes pesos de acordo com sua importância para o sistema.

Foram utilizadas algumas aplicações extensíveis ao Django que poderiam trazer ganhos ao projeto em questão de agilidade de desenvolvimento. Foram elas a *django-registration-redux* [5], para registro de usuário, e a *django-crispy-forms*[6], que efetua validação e renderização de formulários.

3.1 Funcionalidades

As funcionalidades do sistema podem ser divididas até agora em 3 grupos distintos (ver Figura 3.3).

3.1.1 Usuário Anônimo

Um usuário anônimo possui a permissão de acessar áreas de informação a respeito do sistema, de contato com o administrador do sistema, e da área de registro, onde pode solicitar o

[5] http://django-registration-redux.readthedocs.org/
[6] http://django-crispy-forms.readthedocs.org/en/latest/

```
"   O cliente possui um conjunto de algoritmos que deseja
    disponibilizar para os usuários cadastardos no seu
    portal. Para isso, quer um sistema que possa
    disponibilizar e gerenciar algoritmos, usuários e
    tarefas(execuç es  solicitadas pelos usuários).

Os usuários cadastrados devem efetuar o login no sistema, e
    serem direcionados para o ambiente aonde possam
    visualizar as tarefas que requisitou (se houver alguma),
    ou seguir para a tela de solicitação de uma nova tarefa.

A solicitação de uma nova tarefa deve permitir ao usuário
    selecionar o algoritmo que deseja utilizar, fazer o
    upload do arquivo de entrada e solocitar a execução da
    tarefa com as configuraç es  selecionados."
```

Figura 3.1 – *Storytelling*

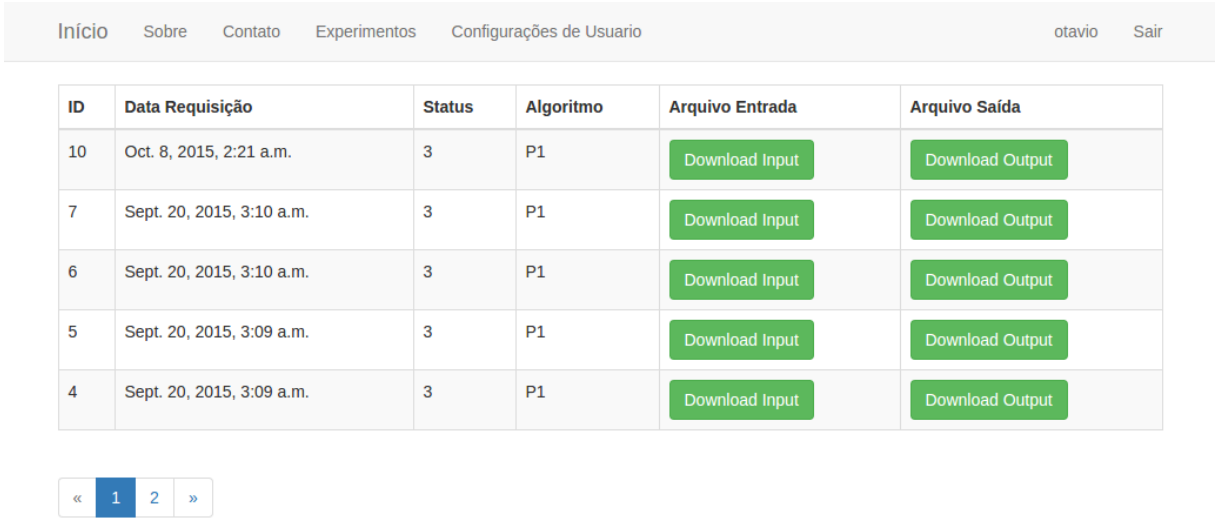

Figura 3.2 – Tela de acompanhamento das execuções.

registro e, seguindo as orientações apresentadas, fazer login como um usuário registrado.

3.1.2 Usuário registrado

O usuário registrado tem permissão de criar experimentos no portal, para isso ele faz o *upload* de um arquivo que será utilizado como entrada no algoritmo selecionado. Além disso o usuário também pode monitorar o estado dos experimentos que ele requisitou e fazer o *download* dos arquivos de cada experimento, tanto o arquivo de entrada, como a saída(se houver) do algoritmo (ver Figura 3.2).

3.1.3 Administrador

O administrador tem as mesmas capacidades do que um usuário registrado e detém privilégios de acesso ao painel de administração do Django. Isso permite que ele faça o cadastro de novos algoritmos no sistema, de novos experimentos padrão e a editar qualquer informação que o sistema detenha no banco de dados.

3.2 Modelos de dados

Definidas as funcionalidades a serem desenvolvidas, foram criados modelos de dados para manter as informações dos usuários, de algoritmos disponíveis pelo sistema, execuções que seriam requisitadas pelos usuários, e execuções pré-definidas.

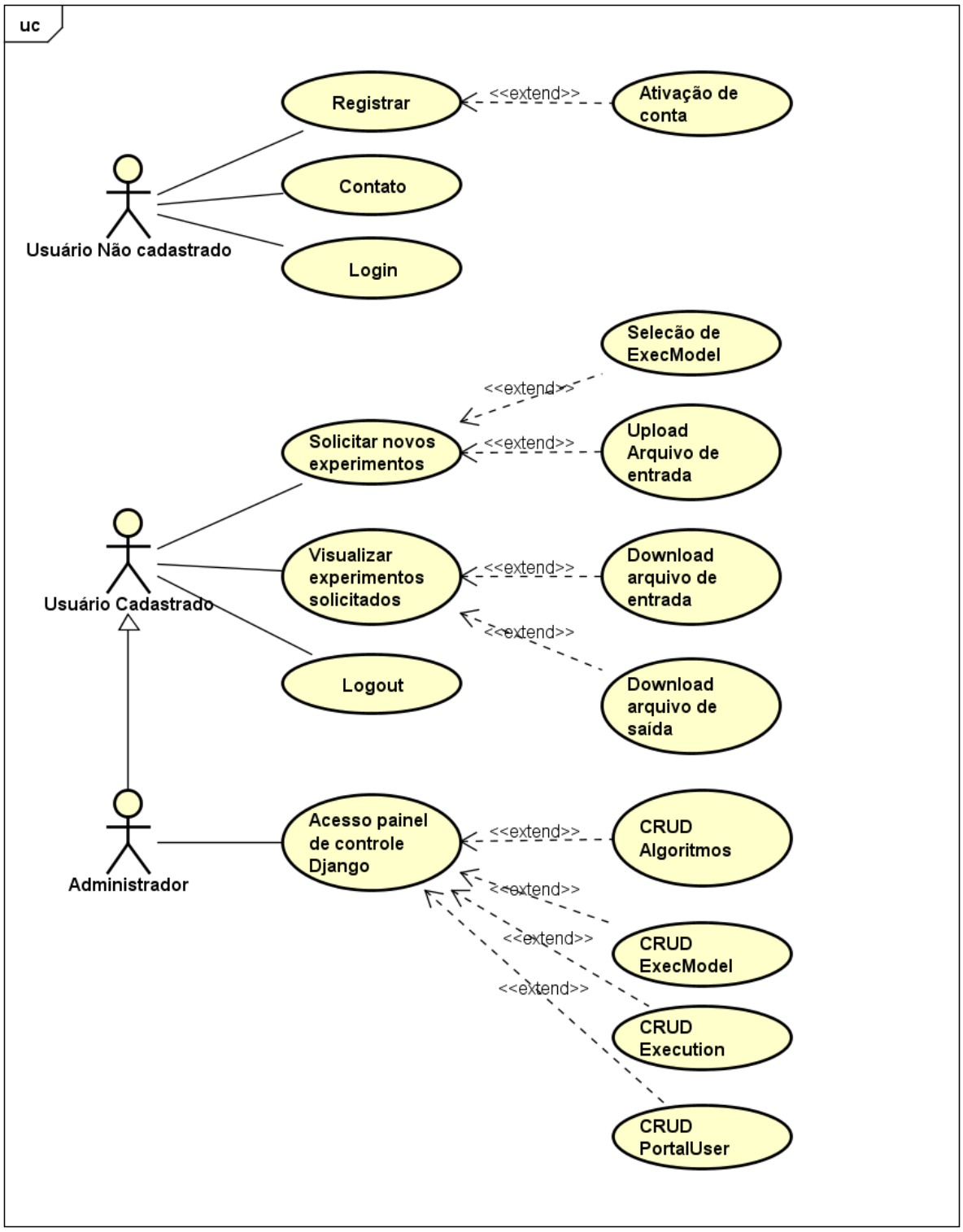

Figura 3.3 – Diagrama de casos de uso do projeto.

Essas 4 classes compõem o modelo de dados (Figura 3.5) que integram todas as funcionalidades do sistema até o presente estado de desenvolvimento.

3.2.1 Algorithm

Essa classe é responsável por manter os dados referentes ao(s) algoritmo(s) que o sistema disponibiliza, é composto por 3 atributos que representem o nome do algoritmo, a descrição de seu propósito, e o comando que deverá ser utilizado para executá-lo.

3.2.2 Execution

A classe *Execution* mantém todas as informações referentes a um pedido de execução. Ela inclui uma chave estrangeira que referencia o usuário que requisitou-a, a data na qual a requisição foi feita, o status da execução, outra chave estrangeira que referencia qual algoritmo será utilizado, e dois campos para os endereços nos quais devem ser mantidos os dados de entrada e saída à serem usados nesta execução.

3.2.3 PortalUser

Representa o usuário no portal, mantém os dados de cadastro e mais alguns referentes a datas e preferências:

- Username (Utilizado para o login)
- Password
- E-mail
- Nickname (Utilizado para mensagens de contato e alertas pelo sistema)
- Company
- DateRegister (Data de quando foi feito o registro do usuário)
- LastAccess (Data do último acesso do usuário no sistema)
- ResultsPerPage (Preferência de quantos resultados por página o usuário deseja quando for listar os experimentos que requisitou)

```python
from django.db import models
from django.contrib.auth.models import User


class Algorithms(models.Model):
    idAlg = models.AutoField(primary_key=True)
    nameAlg = models.CharField(null=False, blank=False, max_length=100)
    desc = models.CharField(null=True, blank=False, max_length=500)
    command = models.CharField(null=False, blank=False, max_length=100)

    def __unicode__(self):
        return self.nameAlg


class UsuarioFriends(models.Model):
    nickname = models.CharField(
        default='default', max_length=30, blank=False, null=True)
    company = models.CharField(
        default='default', max_length=50, blank=False, null=True)
    usuario = models.OneToOneField(User)
    date_register = models.DateTimeField('date_register', auto_now_add=True)
    last_access = models.DateTimeField('last_access', auto_now=True)
    resultsPerPage = models.IntegerField(default=10)

    def __unicode__(self):
        return self.nickname


def user_directory_path_in(instance, filename):
    return './users/user_{0}/{1}/input'.format(instance.request_by.usuario.id, instance.id)


def user_directory_path_out(instance, filename):
    return './users/user_{0}/{1}/output'.format(instance.request_by.usuario.id, instance.id)


class Execution(models.Model):
    request_by = models.ForeignKey(UsuarioFriends)
    date_requisition = models.DateTimeField(
        'date_requisition', auto_now_add=True)
    status = models.IntegerField(default=1)
    algorithm = models.ForeignKey(Algorithms, null=True, blank=False)
    inputFile = models.FileField(upload_to=user_directory_path_in, null=True)
    outputFile = models.FileField(upload_to=user_directory_path_out, null=True)
    time = models.FloatField(default=-1)

    def __unicode__(self):
        return self.request_by.id
```

Figura 3.4 – Models.py

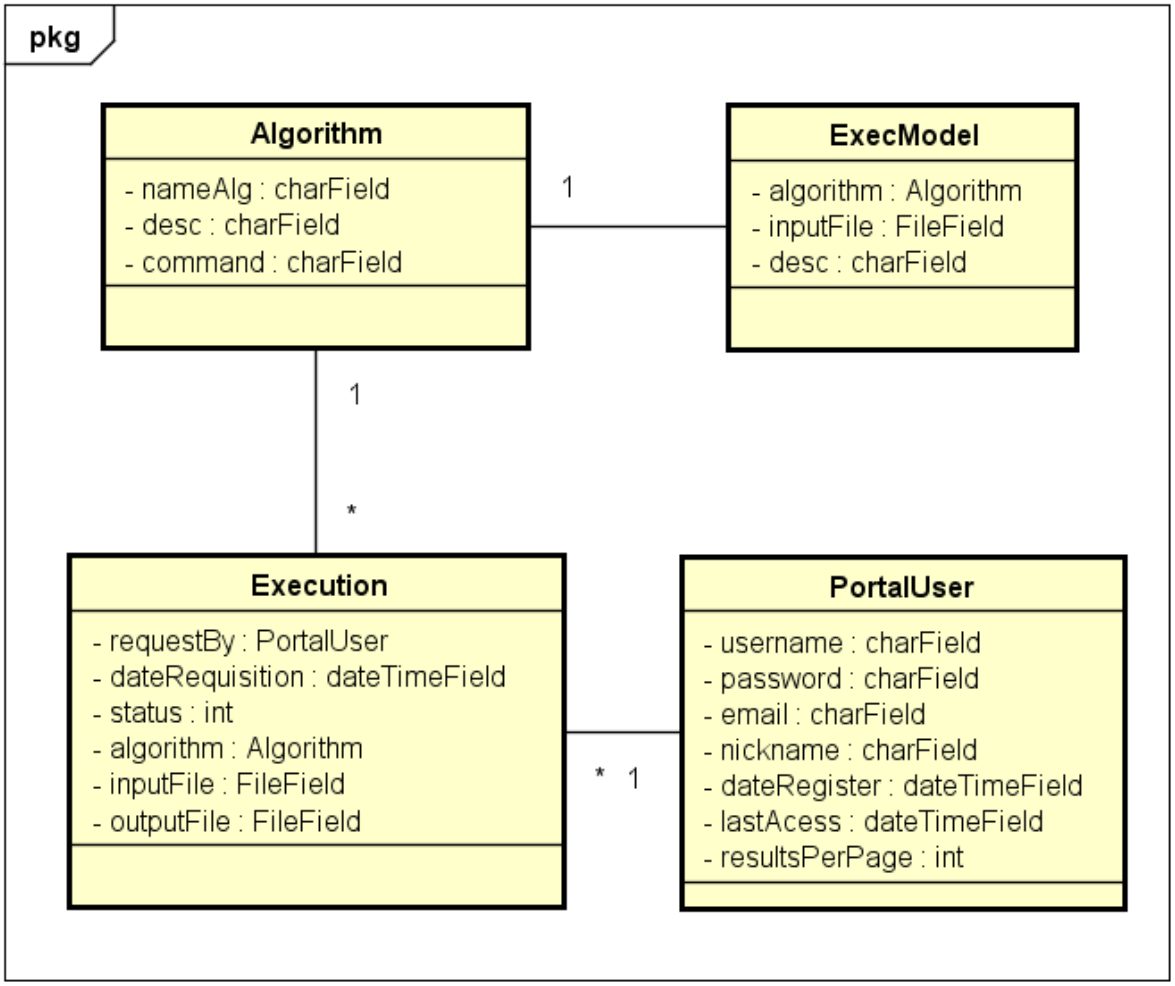

Figura 3.5 – Diagrama de classes do projeto.

3.3 Formulário de Contato

A página de contato do sistema foi desenvolvida utilizando um formulário que exige informações necessárias para identificação e endereço de resposta de quem efetuou o contato. Para isso foi utilizada a aplicação *django-crispy-forms* para obtenção de dados e o envio de e-mail a partir do servidor Django para concluir a função.

3.3.1 Django Crispy Forms

Durante a implementação dessa tela foi instalada a aplicação Crispy. Ela trata da criação de formulários e executa a validação e renderização desses formulários. Quando um usuário preenche incorretamente um formulário, ela trata de renderizar um novo fazendo alterações nos campos preenchidos incorretamente, tornando evidente aonde se encontram os erros de preenchimento.

3.4 Registro de Usuário

Depois disso houve o estudo da logística de registro de usuários, como a aplicação deve ser aberta a qualquer pessoa, é imprescindível que haja uma forma autônoma de registro de novos usuários. No primeiro momento tentou-se implementar a funcionalidade utilizando formulários e verificações desenvolvidas no próprio projeto, mas durante o desenvolvimento foi encontrada uma solução mais rápida e mais adequada, o uso do app *Django-Registration-Redux*.

O *Django-Registration-Redux* é uma aplicação extensível que provê as funcionalidades de registro de usuários para sites que utilizam Django. Essa aplicação já dispõe de templates e formulários para fazer a sua função, também contém um sistema de ativação de contas no qual o usuário que solicitou registro recebe um e-mail no endereço usado no cadastro que contém um link para ativar sua conta, forçando uma verificação de que o e-mail utilizado existe e pertence mesmo ao usuário.

Por padrão ela mantém apenas os dados de e-mail, nome e senha do usuário que solicitou o registro, o que satisfaz os requisitos mínimos para executar sua funcionalidade, porém para o sistema em desenvolvimento, pareceu interessante ter a possibilidade de obter mais dados do usuário em seu cadastro, tais como instituição a qual o usuário pertence (Universidade, empresa, etc.), grau de escolaridade, e outras informações que podem ser úteis posteriormente ao administrador do sistema.

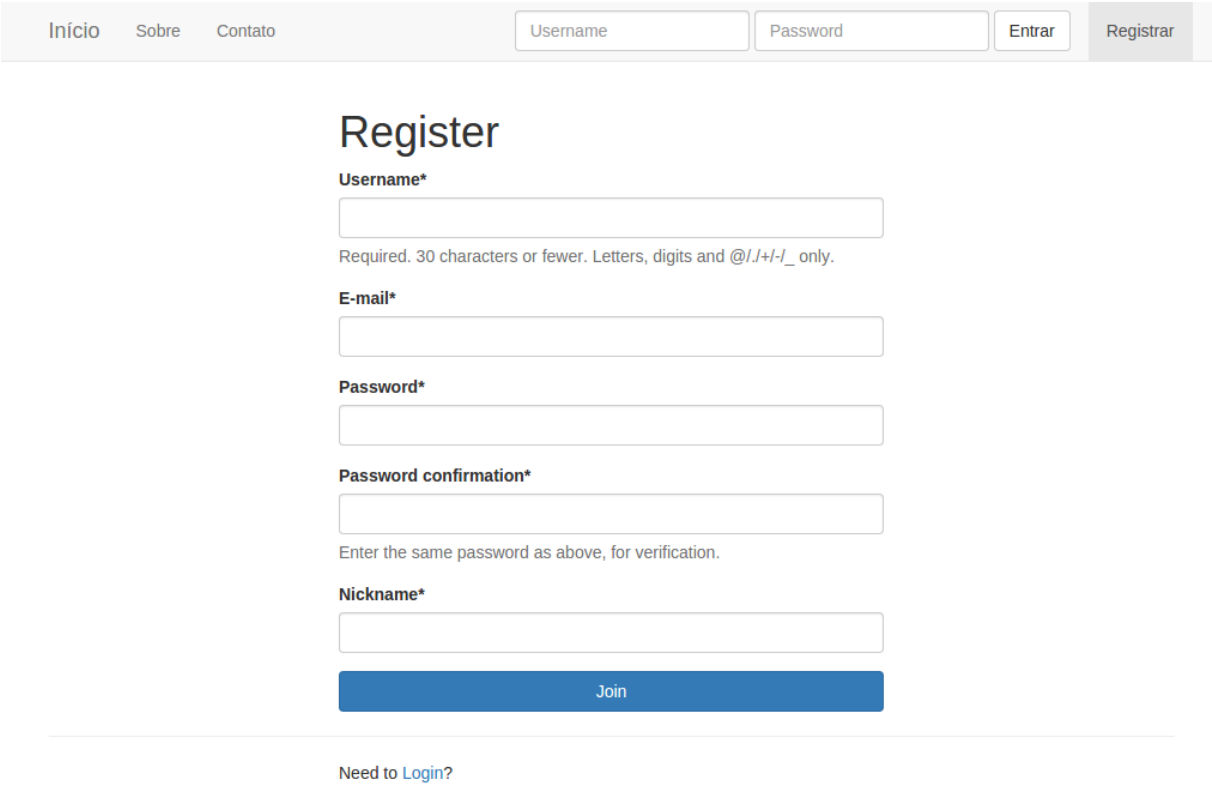

Figura 3.6 – Formulário de Registro de Usuário.

Além disso, essa aplicação não tem por padrão, nenhuma forma de filtro de domínios de endereços de e-mail, o que também pode ser útil ao sistema, caso o administrador queira limitar o acesso do portal aos usuários que detenham um e-mail de um domínio específico.

Para contornar essas questões, foi criado um novo modelo de dados (PortalUser) que estende o modelo de dados da aplicação(Registration User), tornando possível estender as funções de registro para esse modelo novo, de modo que no processo de registro novos campos possam ser requisitados, verificados e validados pelo sistema antes de proceder com a criação do usuário.

3.5 Execução remota de tarefas

A execução remota de tarefas foi um requisito que exigiu outro estudo em busca de ferramentas e técnicas para sua realização. É necessário esclarecer que não é viável que o próprio processo que mantém a aplicação no ar e trata de todos os *requests* realizados por usuários também lide com as execuções dos experimentos solicitados pelos mesmos, pois isso causaria uma lentidão muito grande no sistema.

Para contornar esse problema foram verificadas duas técnicas: a criação de um novo processo que faria a execução do experimento, ou então o uso de alguma aplicação que gerencie filas de tarefas e distribuição das mesmas para demais processos e/ou máquinas.

Compreendida a dimensão e complexidade de criar um sistema para execução de tarefas a partir do zero, optou-se por buscar por aplicações compatíveis com as tecnologias utilizadas do projeto e que pudessem satisfazer a necessidade identificada.

A aplicação escolhida foi o Celery, ele gera filas de execução de tarefas por meio de troca de mensagens. A máquina que mantém o portal, também mantém um processo de execução do broker Redis, que faz o envio e recebimento de mensagens entre o processo que cria novas tarefas, e os workers disponíveis para receber tarefas.

A execução de qualquer algoritmo é feita através da mesma função "RunExperiment"(Figura 3.7). Essa função recebe como argumentos o comando que o worker deve executar e o identificador do arquivo de entrada que deve ser utilizado. Em seguida, o worker faz o download do arquivo de entrada para seu sistema de arquivos local e realiza a execução do algoritmo selecionado com essa entrada recém adquirida. Por fim, cria um formulário POST contendo o arquivo de saída da execução e o envia junto com um request para a máquina que gerou a tarefa, a view para qual o request será mapeado através da url irá salvar o arquivo recebido como a saída da execução e atualizar os dados relativos a aquele experimento.

Essa aplicação permite ao sistema Django criar tarefas que serão executadas pelos chamados *workers*. Os *workers* são processos independentes que devem ser iniciados nas máquinas que irão executar o processamento dos algoritmos e que trocam mensagens com o sistema que solicitou a execução.

No momento, a execução das tarefas criadas pelo sistema estão todas sendo realizadas por um *worker* executado na mesma máquina que mantém o portal, posteriormente espera-se fazer com que isso seja alterado para que a máquina que realiza o processamento seja exclusivamente dedicada à essa tarefa.

3.6 Sistema de arquivos

Como as tarefas realizadas pelo portal exigem dados para serem processados pelo algoritmo, e gera novos dados, foi necessário criar um sistema de arquivos para manter uma ordem na qual seja possível recuperar esses arquivos após o seu processamento. O sistema foi implementado conforme a Figura 3.8 ilustra. Nele, é escolhido um diretório como sendo a raiz de

```python
from models import UsuarioFriends, Execution
from celery.utils.log import get_task_logger
from celery.decorators import task
import requests
import os
import time

logger = get_task_logger(__name__)

@task(name="RunExperiment")
def RunExperiment(execution, ide):
    print("\n Executando o exp %s, algoritmo: %s" % (ide, execution))
    os.system("mkdir " + str(ide))
    os.system("wget http://10.1.4.28:8000/experiments/downloadInputFile?id=" +
              str(ide) + " -O ./" + str(ide) + "/input")
    start = time.time()
    os.system(execution + " " + str(ide) + "/input >" + str(ide) + "/output")
    dur = time.time() - start

    print dur
    files={'file': str("/"+str(ide) + "/output")}
    path = str(str(ide)+"/output")
    print path
    files = {'file': open(path, 'rb')}
    data = {'id':str(ide),'time':dur}
    #   r = requests.post('http://10.1.4.28:8000/about/')
    r = requests.post('http://10.1.4.28:8000/experiments/result', files=files,data=data)
    print (r.status_code, r.reason)
```

Figura 3.7 – Função RunExperiment

todos os arquivos que o portal irá manter referente a dados de entrada e saída dos algoritmos, e esses dados serão mantidos em diretórios nomeados de acordo com o id do experimento ao qual pertencem, e esse diretório, por sua vez, será mantido em um outro diretório nomeado de acordo com o id do usuário que requisitou o experimento.

Após a implementação desse sistema de arquivos e e combinando as funcionalidades com o que já havia sido implementado referente a execução de tarefas, foi possível desenvolver uma página no portal na que o usuário pudesse obter os arquivos de dados referentes a cada experimento solicitado por ele e verificar o estado de cada experimento. Esses estados são definidos de acordo com o andamento do processo de execução, sendo que até o momento, são possíveis 3 estados:

- "Aguardando": Experimento aguarda sua execução por um *worker*.
- "Executando": Experimento está sendo executado, mas ainda não concluiu.
- "Finalizado": Experimento já foi executado e já possui os dados disponíveis para o usuário.

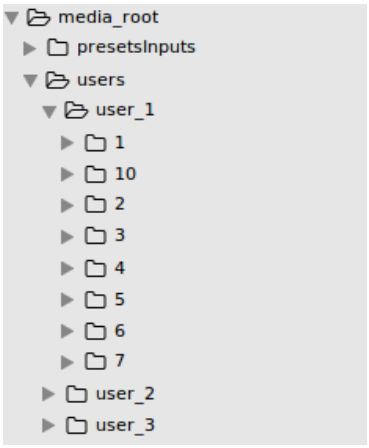

Figura 3.8 – Sistema de Arquivos.

4 PRÓXIMAS ETAPAS

Como continuação do trabalho realizado até o momento, são planejadas as seguintes atividades:

1. fazer a execução das tarefas em uma máquina diferente da que mantém o portal.
2. Implementar sistema de notificação ao usuário para quando execuções forem finalizadas.
3. Implementar nova classe de usuário capaz de obter dados estatísticos do sistema como número de execuções, relatórios do sistema.
4. Adicionar tempo de execução na classe *Execution*.

REFERÊNCIAS

BERTSCHINGER, E. Simulations of structure formation in the universe. **Annu. Rev. Astron. Astrophys 36**, [S.l.], 1998.

CELERY. **Celery**: distributed task queue. http://celery.readthedocs.org/en/latest/, acessado em Outubro de 2015.

DJANGO. **Django Framework**. https://www.djangoproject.com/, acessado em Outubro de 2015.

FORMS, C. **Django-crispy-forms**. 2009.

G. EFSTATHIOU M. DAVIS, S. D. M. W. C. S. F. Numerical techniques for large cosmological N-body simulations. **Astrophysical Journal Supplement Series (ISSN 0067-0049), vol. 57**, [S.l.], 1985.

OTAVIO M. MADALOSSO, A. S. C. Implementação do algoritmo Friends of Friends de complexidade n*logn(n) para classificação de objetos astronômicos. **ERAD-RS XV**, [S.l.], 2015.

SURHUD MORE, G. E. **Friends-of-Friends Masses**. https://gclusters11.wikispaces.com/SBCS+1.1+FOF+Masses, acessado em Novembro de 2015.