Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

AcoSched - um Escalonador para o Ambiente de Nuvem Federada ZooNimbus

Gabriel Silva Souza de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientadora
Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo

Coorientador
Prof. MsC. Edward Ribeiro de Oliveira

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo (Orientadora) — CIC/UnB
Prof.ª Dr.ª Maria Emília Machado Telles Walter — CIC/UnB
Prof.ª Dr.ª Maristela Terto de Holanda — CIC/UnB

CIP — Catalogação Internacional na Publicação

Oliveira, Gabriel Silva Souza de.

AcoSched - um Escalonador para o Ambiente de Nuvem Federada ZooNimbus / Gabriel Silva Souza de Oliveira. Brasília : UnB, 2013.
129 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. federação, 2. computação em nuvem, 3. escalonador, 4. colônia de formigas, 5. BioNimbus, 6. ZooNimbus

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

AcoSched - um Escalonador para o Ambiente de Nuvem Federada ZooNimbus

Gabriel Silva Souza de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo (Orientadora)
CIC/UnB

Prof.ª Dr.ª Maria Emília Machado Telles Walter
CIC/UnB

Prof.ª Dr.ª Maristela Terto de Holanda
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 27 de agosto de 2013

Dedicatória

Dedico aos meus pais, com todo o amor.

Agradecimentos

Agradeço, primeiramente, aos meus pais, que dedicaram suas vidas para a formação do meu caráter e da minha educação, e me apoiaram nas horas felizes como também nas horas difíceis enfrentadas ao longo da minha graduação, e a minha irmã, pelo afeto e carinho. Agradeço a minha noiva, pelo amor, compreensão e auxílio durante essa jornada de estudos. Agradeço também a minha orientadora, Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo, pela paciência, atenção e dedicação ao me ajudar e guiar na execução deste trabalho.

Por fim, e, principalmente, agradeço ao meu Coorientador, Edward Ribeiro de Oliveira, pela dedicação, atenção e fundamental apoio acadêmico empregado na realização deste trabalho, e aos meus colegas, Breno Rodrigues Moura e Deric Lima Bacelar, pelo companheirismo e motivação para enfrentar novos desafios.

Resumo

Escalonamento de tarefas é difícil no ambiente de nuvem federada, uma vez que existem muitos provedores de nuvens com capacidades distintas que devem ser consideradas. Em Bioinformática, muitas ferramentas e base de dados necessitam de grandes recursos para processamento e armazenamento de enormes quantidades de dados que são fornecidos por instituições separadas fisicamente. Este trabalho trata o problema de escalonamento de tarefas no ZooNimbus, uma estrutura de nuvem federada para aplicações de bioinformática. Foi proposto um algoritmo de escalonamento baseado no *Load Balancing Ant Colony* (LBACO), chamado de AcoSched, para realizar uma distribuição eficiente que encontre o melhor recurso para executar uma tarefa requisitada. Experimentos foram desenvolvidos com dados biológicos reais executados no ZooNimbus, formado por algumas provedores de nuvem da Amazon EC2 e da UnB.

Palavras-chave: federação, computação em nuvem, escalonador, colônia de formigas, BioNimbus, ZooNimbus

Abstract

Task scheduling is difficult in federated cloud environments, since there are many cloud providers with distinct capabilities that should be addressed. In bioinformatics, many tools and databases requiring large resources for processing and storing enormous amounts of data are provided by physically separate institutions. This work treats the problem of task scheduling in ZooNimbus, a federated cloud infrastructure for bioinformatics applications. A scheduling algorithm based on Load Balancing Ant Colony (LBACO), called AcoSched, was propose to perform an efficient distribution for finding the best resources to execute each required task. Experiments was developed with real biological data executing on ZooNimbus, formed by some cloud providers executing in Amazon EC2 and UnB. The obtained results show that AcoSched makes a significant improvement in the makespan time of bioinformatics applications executing in ZooNimbus, when compared to the DynamicAHP algorithm.

Keywords: federation, cloud computing, scheduling, ant colony, ZooNimbus, BioNimbus

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em um cenário de constantes avanços tecnológicos, incentivado pela demanda do mercado por novidades e por melhores soluções, surgi a Computação em Nuvem a qual disponibiliza um ambiente robusto em serviços computacionais [1]. Em meio às possibilidades que este ambiente de computação distribuída proporciona, encontrar a melhor definição para computação em nuvem não é um trabalho simples, pois não há um consenso para a definição de nuvem [22]. Algumas definições são bem próximas das definições de outros sistemas distribuídos, que chegam a ser confundidas com o conceito de Computação em Grid [17].

Foster [17] definiu nuvem como sendo um recurso que utiliza o paradigma de sistemas distribuídos para obter economia com a escalabilidade de serviços, que são entregues para o cliente de acordo com sua demanda, por meio de acesso remoto, via Internet. Desta forma, o cliente tem acesso a um serviço de sua escolha pela Internet, consumindo-o de acordo com sua necessidade. O serviço é disponibilizado de forma gratuita ou paga, sendo o pagamento proporcional ao seu uso.

Nuvem computacional pode, então, ser exemplificada como um modelo de sistema distribuído que permite o acesso a vários recursos, plataformas, meios de armazenamento, de processamento, de serviços e etc, diferenciando-se de outros modelos computacionais em algumas características, tais como a escalabilidade, a virtualização e o tipo de nuvem, que pode ser privada, pública ou híbrida [30, 38].

Por outro lado, muitas instituições de ensino realizam pesquisas que necessitam de um grande poder de processamento, e nem todas contam com muitos recursos computacionais disponíveis. A área de Bioinformática é uma das áreas que necessitam de infraestrutura robusta para realizar pesquisas e testes, como a manipulação das sequências biológicas presente no DNA e até mesmo simular a interação das proteínas [23].

Nesse cenário, computação em nuvem é uma possível e real solução para essa necessidade. Contudo, a demanda de serviços na área de Bioinformática é grande e o serviço de computação em nuvem pode ser insuficiente, pois pode alcançar um limite físico de recursos disponíveis para uso [1]. Para expandir essa limitação física dos recursos em uma nuvem, surge o conceito de Federação de Nuvem, que é a integração de diversas nuvens em uma única plataforma distribuída, expandindo, assim, a limitação computacional de uma única nuvem e ampliando as possibilidades de nuvem computacional ao solucionar a possível problemática de escassez de recursos computacionais de uma nuvem.

Dessa forma, como dito anteriormente, nuvem federada é um conceito no qual seu ambiente é formado por várias nuvens computacionais que disponibilizam os serviços de forma transparente e com uma possibilidade de expansão maior do que aquela permitida ao contratar apenas uma única nuvem.

No ambiente de federação, diversos provedores de nuvens irão cooperar e utilizar os recursos de outros provedores de nuvens, gerando economia de escala, uso eficiente de seus recursos e aumento do seu conjunto de recursos disponíveis. Tal mecanismo de federação gera uma série de desafios que devem ser solucionados, tais como o controle dos acessos entre as redes de nuvens diferentes, o controle da segurança entre os recursos, a heterogeniedade entre os serviços disponibilizados pelos provedores de nuvens e etc.

Trabalhos anteriores, voltados a área de bioinformática, já foram desenvolvidos abordando o conceito de nuvem federada, como é o caso da plataforma BioNimbus [25, 16, 35]. Assim, utilizando-se dessa plataforma e do cenário disponibilizado por ela, é importante destacar que uma boa forma de usar todos os benefícios proporcionados neste ambiente de computação, ao realizar a execução de tarefas, pode ser obtido por meio de uma política de escalonamento dinâmica. Tal dinamismo, otimiza o uso dos recursos, reduzindo o tempo de processamento e aumentando o número de processos atendidos em um mesmo espaço de tempo.

Todavia, escalonamento de tarefas em nuvem é um problema NP-Difícil [30], e alguns escalonadores tem sido desenvolvidos com o objetivo de otimizar essa tarefa no ambiente de nuvem, como é o caso do algoritmo DynamicAHP [25], implementado na primeira versão da plataforma BioNimbus.

Na tentativa de contribuir com o problema de escalonamento de tarefas em nuvem, alguns trabalhos na literatura [1, 24, 37] têm utilizado políticas de escalonamento baseadas em heurísticas. As heurísticas mais conhecidas e utilizadas são: colônia de formigas [27], a qual simula o real funcionamento de uma colônia de formigas ao realizar a busca por alimentos; e a heurística do algoritmo genético [37], que é baseado nas operações genéticas reais.

1.1 Objetivos

Deve dizer que este trabalho tem como objetivo geral propor um novo escalonador para a plataforma de nuvem federada BioNimbus, que reduza o tempo total de execução das tarefas quando comparado ao algoritmo implementado anteriormente, e, consequentemente, otimize a utilização dos recursos que formam a federação. Para cumprir o objetivo geral citado, este trabalho tem os seguintes objetivos específicos:

- Apresentar uma nova versão da plataforma de nuvem federada BioNimbus, que aumente sua flexibilidade, elasticidade e facilidade de manutenção.
- Executar teste para comparar o desempenho obtido com o algoritmo proposto e o algoritmo implementado inicialmente no BioNimbus.
- Realizar teste com dados reais de Bioinformática.

1.2 Estrutura do Trabalho

O presente trabalho está dividido, em mais cinco capítulos. O Capítulo 2 aborda de maneira sucinta os conceitos de nuvens computacionais, os seus tipos os seus serviços. O Capítulo 3 apresenta o BioNimbus, mostrando sua proposta, sua estrutura e organização, como também sua política de escalonamento utilizada inicialmente. O Capítulo 4 apresenta as propostas de modificações para o ambiente BioNimbus que foram necessárias para o funcionamento da política de escalonamento proposta por este trabalho e resultarão na nova versão do BioNimbus, chamada de ZooNimbus. O Capítulo 5 apresenta o algoritmo de escalonamento proposto, chamado de AcoSched. O Capítulo 6 mostra as análises feitas com relação ao desempenho do algoritmo proposto e o desempenho do algoritmo de escalonamento implementado na versão inicial do BioNimbus. Para finalizar este trabalho, o Capítulo 7 apresenta a conclusão e descreve algumas possibilidades para trabalhos futuros.

Capítulo 2

Computação em Nuvem

Este capítulo faz, inicialmente, considerações sobre nuvem, sua estrutura e seus tipos. Em seguida, descreve alguns serviços disponibilizados pelas nuvens computacionais, e cita também empresas que fornecem alguns dos serviços a serem descritos. Por fim, são apresentados o conceito de nuvem federada e as suas principais características.

2.1 Nuvens Computacionais

As definições de computação em nuvem podem ser confundidas com outras definições de sistemas distribuídos, como, por exemplo, a definição de *grid*, onde Tanenbaum e Steen [33] destacam que uma questão fundamental em um sistema de computação em *grid* é que recursos de diferentes organizações são reunidos para permitir a colaboração de um grupo de pessoas ou instituições. Esses autores ainda pontuam que tal colaboração é realizada sob a forma de uma organização virtual. A semelhança se deve ao fato de que a origem da computação em nuvem é, justamente, a evolução da computação em *grid* [17]. Essa evolução é basicamente devido à mudança no conceito dos serviços oferecidos por *grid*.

Em *grid* tem-se a infraestrutura como principal produto disponibilizado, já em computação em nuvem, infraestrutura é um dos serviços providos, de forma que as possibilidades de utilização dos recursos são aumentadas e, assim, diversos serviços são oferecidos, como, por exemplo, armazenamento, processamento e softwares. Os recursos são disponibilizados de forma mais abstrata e transparente em computação em nuvem, permitindo convenientemente o acesso à recursos disponibilizados via Internet, e que podem ser rapidamente providos e liberados [38].

O modelo de computação em nuvem é caracterizado por três pontos principais que o distingue de outros modelos de sistemas distribuídos, os quais são: o pagamento sob demanda, a computação elástica e a virtualização. Para uma empresa, o pagamento sob demanda para utilização de um recurso computacional significa redução de custos diante da necessidade anterior da aquisição desses recursos para suprir sua demanda de infraestrutura tecnológica, por exemplo. A segunda, a computação elástica, ocorre quando os recursos computacionais disponibilizados são reutilizáveis e realocados para diversos clientes, suportando assim, alta escalabilidade. Por último, a virtualização, uma tecnologia de abstração do acoplamento entre hardware e sistemas operacionais, onde recursos são virtualizados, solucionando muitos problemas que podem ocorrer pela heterogenei-

dade computacional dos recursos. A virtualização permite que ambientes computacionais sejam dinamicamente criados, expandidos e movidos [30], sendo característica essencial, assim como as demais citadas para um ambiente computacional distribuído.

Outra característica importante que deve estar presente em uma nuvem computacional é a tolerância a falha, que é a capacidade de se restabelecer após uma falha, não permitindo erro ou perda de processamento. No caso de falhas, haverá uma instância capaz de assumir o processamento sem interrupção, mecanismo chamado de *failover* [30]. Algumas falhas de sistemas de nuvem relatadas por Lumb *et. al* [30] demonstram uma demora significativa na restauração do sistema após ocorrer um falha, a maior delas foi do sistema da *Microsoft*, o *Microsoft Azure*, o qual gerou uma interrupção de 22 horas nos dias 13 e 14 de Março de 2008 [30]. E no caso do ambiente de computação em nuvem, tempo é dinheiro, e essa interrupção é prejudicial e inaceitável.

2.1.1 Tipos de Nuvens

Nuvem computacional pode ser dividida basicamente em três tipos, os quais são: nuvem privada, nuvem pública e nuvem híbrida. Nas nuvens privadas os dados e processos são gerenciados dentro da organização com menor restrição de largura de banda, exposição da segurança e requisitos legais que o uso de serviços de nuvens pública podem conter [30]. É a nuvem que atende a uma determinada empresa sem que os recursos da nuvem sejam disponibilizado para outras empresas e, geralmente, os recursos requisitados são adquiridos pela empresa.

Por outro lado, nuvem pública descreve computação em nuvem no sentido tradicional, onde os recursos são fornecidos de acordo com a demanda, acessados pela Internet por meio de aplicações/serviços *web* a partir de um provedor de recursos [30], sem a necessidade de aquisição da infraestrutura. Nelas, geralmente, o uso dos recursos são comercializados pelas empresas, ou seja, pelos provedores que disponibilizam os serviços. Além dessa forma de pagamento, terá mais segurança dos seus dados na nuvem privada, como no caso, de o governo federal que não deveria utilizar nuvem pública para rodar a folha de pagamento mensal de todos os servidores públicos do país, pois estaria expondo os dados dos funcionários ao transportá-los para uma nuvem pública.

Por último, nuvem híbrida consiste na nuvem formada de múltiplos provedores de recursos. Ela utiliza tanto recursos de nuvens públicas como recursos de nuvens privadas. Neste cenário, utilizando-se o mesmo exemplo dado acima, onde o governo federal, tem-se que processar a folha de pagamento de seus funcionários mensalmente, seria vantajoso realizar a aquisição de recursos para criar uma nuvem privada e realizar suas demandas em determinados dias do mês, e nos demais disponibilizar sua infraestrutura de nuvem como nuvem pública, provendo o serviço a terceiros. Ao realizar a utilização dos recursos como nuvem privada e disponibilizá-los para serem acessados como provedor de recursos, nuvem pública, essa nuvem seria uma nuvem híbrida.

2.1.2 Serviços em Nuvem

Em todos os ambientes nos quais computação em nuvem pode ser aplicada (acadêmico, científico, empresarial e comercial), a divisão de seus serviços em categorias é necessária

e útil. Estes serviços se dividem em diversos tipos, três deles são mais conhecidos e adotados, e serão abordados neste trabalho.

Infraestrutura como Serviço, *IaaS*, é o tipo mais conhecido de computação em nuvem. A infraestrutura é disponibilizada como um serviço que o usuário acessa de qualquer parte, e escolhe a característica da infraestrutura de acordo com sua necessidade. Uma das suas vantagens é o uso de tecnologias mais recentes presentes nas nuvens [30]. *IaaS* disponibiliza software, hardware e equipamentos, podendo aumentar a escala ou diminuir de acordo com a necessidade da aplicação.

O serviço de armazenamento provido pela *IaaS* é fornecido, por exemplo, pela *Amazon Elastic Compute Cloud*, chamado de *AmazonEC2* [26]; pelo *Dropbox* [9]; pelo *Google*, denominado de *Google Drive* [18]; pela *Microsoft Azure* [28] e diversos outros provedores de nuvem.

Para ilustrar o custo do serviço *IaaS*, será analisada a taxa cobrada pela *AmazonEC2* para o uso de seu serviço, chamado de Instâncias *On Demand*, no qual o pagamento é proporcional a quantidade de horas utilizadas e a configuração da instância. Assim, considerando o uso de cinco instâncias, com o sistema operacional Linux, o tipo de instância *High-CPU Extra-Large*, durante 30 horas em um mês, e sem monitoramento foi gerado um custo de 99 dólares calculados por meio da calculadora disponibilizada pela *Amazon*, a *Simple Monthly Calculator* [1].

O segundo tipo de serviço, Plataforma como Serviço, *PaaS*, oferece um ambiente de alto nível para desenvolvedores criarem, testarem e manipularem aplicações. A maior diferença entre *IaaS* e *PaaS* é a flexibilidade, pois quando a carga do serviço é aumentada o servido automaticamente aumenta a carga de recursos disponíveis, e o *IaaS* não disponibiliza essa extensão automática [7]. O modelo *PaaS* provê uma plataforma poderosa de desenvolvimento, onde o desenvolvedor não precisa se preocupar com as licenças de software, com a capacidade do ambiente de desenvolvimento, com o aumento no uso da infraestrutura para implementação da aplicação.

O *Microsoft Azure* [28] e o *Google App Engine* [19] são exemplos de empresas que fornecem *PaaS*. O *Google App Engine* da Google é um exemplo de *PaaS*, no qual criação e hospedagem de aplicativos *web* são permitidos, não existindo preços predefinidos. O desenvolvedor controla os recursos disponíveis para sua aplicação e o cálculo para pagamento é realizado de acordo com o uso, onde as medidas para tal são baseadas nos recursos que a aplicação utilizar, como armazenamento e largura de banda que são medidos em *gigabytes*. Existe um nível gratuito de uso dos recursos com 500 MB de armazenamento, CPU e largura de banda suficientes para suportar um aplicativo eficiente que ofereça cerca de cinco milhões de visualizações de página por mês gratuitamente. Neste caso, são pagos apenas o uso dos recursos que ultrapassarem esses limites [19].

Por outro lado, Software como Serviço, *SaaS*, disponibiliza aplicações para acesso e utilização dos usuários pela Internet sem que seja necessária a sua instalação. Geralmente, o *Saas* é utilizado para softwares que possuem uma instância da aplicação com banco de dados que é acessada por múltiplos usuários simultaneamente [30], pois a elasticidade da nuvem garante acesso a todos os usuários, mesmo com um número grande e crescente de usuários acessando simultaneamente o software. Assim, o usuário também pode acessar um software pela Internet sem ter que adquirir sua licença, e também sem a necessidade de adquirir um hardware melhor para a sua execução. Algumas empresas que disponibilizam

[1] http://calculator.s3.amazonaws.com/calc5.html

esse serviço são *Salesforce.com* [32] e a *Oracle Cloud* [4]. A *Oracle Cloud* [4] oferece o uso de diversos aplicativos como o *Oracle Store* [4], o *My Oracle Support* [4], o *Eventos de Marketing* [4] e o *Oracle PartnerNetwork* [4].

Os serviços são acessados da mesma forma de qualquer lugar do mundo, sem a necessidade de manipulação dos dados, e nem o transporte dos recursos. O uso pode ser intensificado e o processamento aumentado sem a aquisição de novas máquinas. Logo, o custo é proporcional ao uso, e os recursos estão sempre disponíveis de acordo com a necessidade. Além disso, seu poder pode ser considerado sem limites. Essas são algumas das características do modelo de computação em nuvem que estão sendo exploradas comercialmente, permitindo grandes avanços na computação em escala global, e a evolução desse ambiente de computação é apresentada na Seção 2.2.

2.2 Nuvem Federada

A elasticidade, característica importante do ambiente de nuvem computacional, permite rapidamente o aumento ou a diminuição dos recursos utilizados em uma nuvem. Essa característica permite que o pagamento proporcional ao uso dos serviços presente no ambiente da nuvem computacional seja mais eficiente. Além de garantir uma maior confiabilidade e qualidade no sistema, pois não haverá falhas ou diminuição no desempenho quando os recursos não forem suficientes e seja, então, necessária uma alteração automática na sua escala. O aumento automático da escala dos recursos pode levar a uma ideia de recursos ilimitados.

Porém, a elasticidade não disponibiliza recursos ilimitados, haverá momentos onde estes recursos ficaram escassos dentro da nuvem e o provedor da nuvem realizará a compra de novos recursos para que sejam atendidas as demandas dos clientes. Realizar a compra de novos recursos quando houver grandes requisições por uma nuvem não é uma boa prática, pois essa demanda pode ser apenas um pico de requisição de recursos e, consequentemente, poderia ocasionar prejuízos já que esses eventos podem ser esporádicos no ambiente da nuvem.

Celesti *et al.* [1] prevê uma evolução para computação em nuvem que soluciona esse problema. A evolução proposta consiste em três etapas que tendem a elevar o compartilhamento de recursos entre as nuvens, suprindo a necessidade de aquisição de recursos desnecessários e aumentando a elasticidade da nuvem.

Segundo Celesti *et al.* [1] computação em nuvem iniciou na fase *"Monolithic"*, na qual os serviços de nuvens estão baseados em arquiteturas proprietárias e ilhas de nuvens, que disponibilizam serviços. Nesse nível não há o compartilhamento de recursos entre as nuvens, mas sim, ilhas de serviços. Na próximo fase, fase dois, chamada de *"Vertical Supply Chain"*, as nuvens ainda são ilhas proprietárias, mas a utilização de outros provedores de recursos já começa a ocorrer, iniciando uma certa interação entre as nuvens. Por fim, a terceira, a chamada *"Horizontal Federation"*, os diversos provedores de nuvens irão cooperar e utilizar os recursos de outros provedores, gerando economia de escala, uso eficiente de seus recursos e aumento de sua capacidade [1]. Desta forma, é permitido que um cliente utilize, por exemplo, recursos de várias nuvens diferentes por meio de um provedor de forma abstrata.

A Figura 2.1 ilustra o cenário de uma nuvem federada, destacando os estágios da evolução e o resultado da formação de nuvem federada utilizando a nuvem local, *home*,

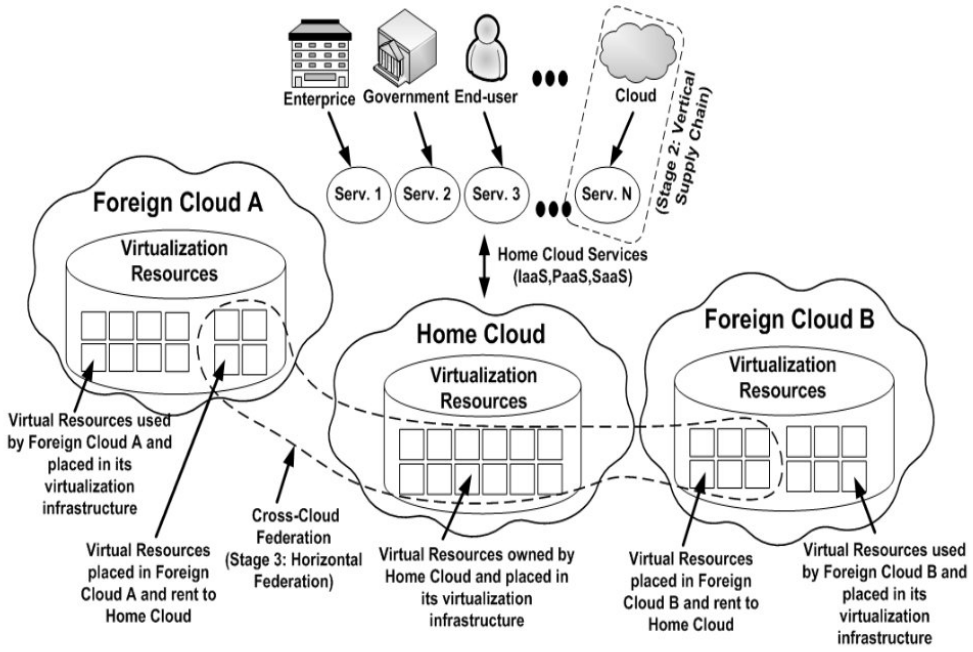

Figura 2.1: Cenário de Nuvem Federada [1].

onde o usuário está conectado a outras duas nuvens A e B, *foreign cloud*, que representam as nuvens utilizadas. A distinção de nuvens nesses dois tipos, *home cloud* e *foreign cloud* é importante para o ambiente federado, pois define claramente a forma de utilização das nuvens, onde a *home cloud* é a nuvem provedora que não é mais capaz de instanciar máquinas virtualizadas, pois sua estrutura se encontra saturada. Assim, esses recursos são então disponibilizados pelas *foreigns clouds* [1]. Os recursos disponíveis por essas três nuvens irão formar a nuvem federada, na qual serão realizadas as tarefas requeridas pelo usuário de forma transparente, assim como ocorre em uma nuvem.

Este trabalho foi desenvolvido utilizando o conceito do terceiro nível de nuvem por meio de uma plataforma de nuvem federada, funcionando em diversos recursos e outros provedores de nuvem computacional. Essa plataforma, já desenvolvido, é chamado de BioNimbus [35], e será detalhado no próximo capítulo.

Capítulo 3

BioNimbus

Este capítulo tem o objetivo, inicialmente, de apresentar a plataforma para federação de nuvens BioNimbus. Em seguida, são detalhados seus serviços e o seu funcionamento básico. Ainda neste capítulo, será apresentado o algoritmo de escalonamento utilizado originalmente no BioNimbus, destacando seu funcionamento e seu desempenho.

3.1 Visão Geral

Integrar e controlar diferentes provedores de infraestrutura, de maneira transparente, flexível e tolerante a falhas, com grande capacidade de processamento e armazenamento é a proposta do BioNimbus para federação de nuvem [35]. O BioNimbus disponibiliza um cenário classificado como a terceira etapa de evolução de nuvem computacional[1]. Proporcionando, dessa forma, um ambiente de recursos supostamente ilimitados para o usuário executar sua aplicação. O usuário visualiza funcionalidades como simples serviços de nuvem, independente de qual provedor de recurso, está sendo utilizado, podendo ser formada uma arquitetura sobre nuvens privadas, públicas ou híbridas.

O BioNimbus foi desenvolvido com a preocupação de deixar as funcionalidades do sistema bem definidas e divididas, para quando novos provedores de recursos participarem da nuvem suas inclusões fossem simples e eficiente. A comunicação dos componentes da nuvem formada pelo BioNimbus é realizada pelo protocolo *Peer-to-Peer* e a integração é flexível, suportando diferentes tipos de infraestruturas com a utilização de *plug-ins*. Os *plug-ins* coletam informações sobre os recursos computacionais disponíveis, e quais são as ferramentas de bioinformática que estão disponíveis para serem utilizadas pelos usuários da nuvem. As aplicações são executadas por meio de interfaces *web*, linhas de comando e outras formas de integração mais complexas [35].

Os *plug-ins* realizam serviços de fundamental importância na arquitetura do BioNimbus, são as interfaces de comunicação entre os provedores e os demais componentes da arquitetura, e também para a comunicação entre os outros provedores [35]. O mapeamento das requisições vindas dos componentes da arquitetura para as ações correspondentes a serem realizadas na infraestrutura dos recursos de cada provedor é realizado pelo *plug-in*. É importante ressaltar que cada infraestrutura tem uma implementação diferente do *plug-in*. Assim sendo, a plataforma BioNimbus utiliza troca de mensagens para realizar o controle e a integração com o usuário. Para isso, esses serviços são realizados por uma rede *Peer-to-Peer*, na qual cada componente envia suas requisições como pares de men-

sagens, de requisição e de resposta, definidas para cada tipo de solicitação prevista na arquitetura. Os tipos de requisição são divididos em três diferentes tipos: informações sobre a infraestrutura do provedor de serviço, gerenciamento de tarefas e transferências de arquivos.

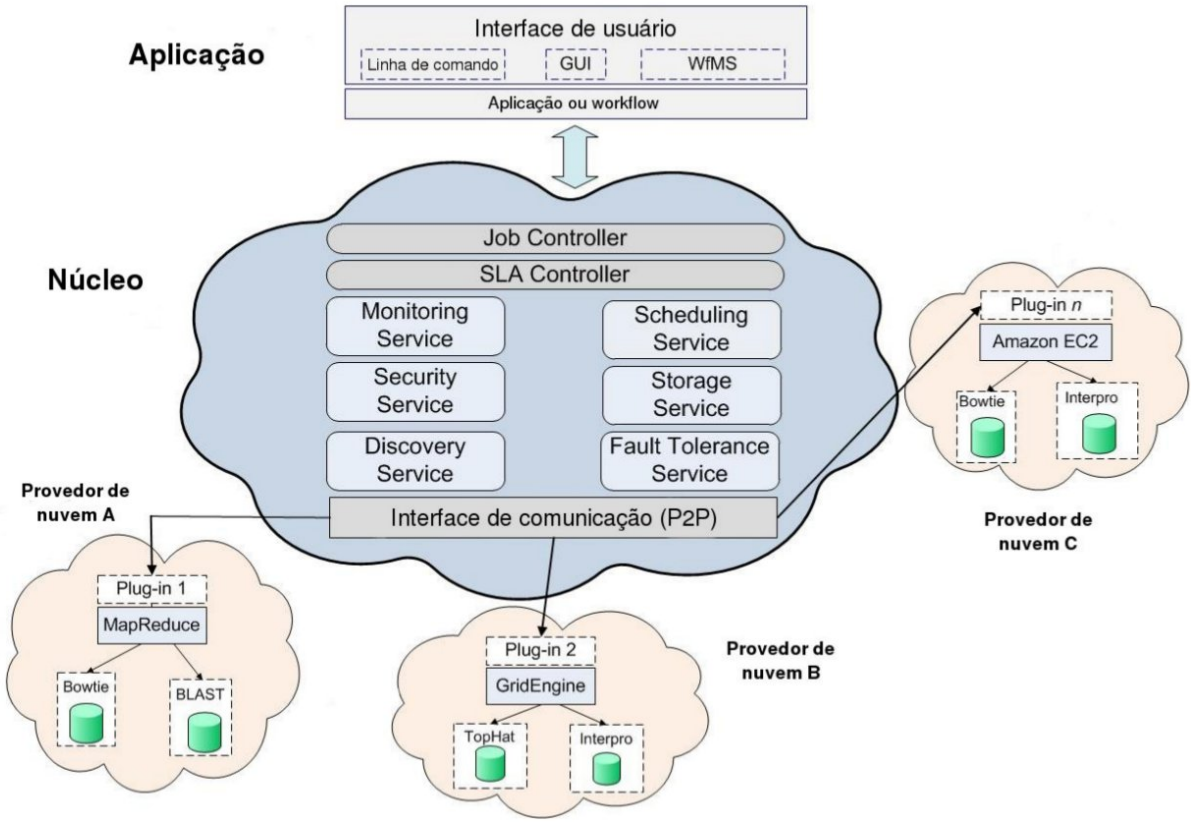

Figura 3.1: Arquitetura do Ambiente de Federação BioNimbus [35].

Este capítulo descreve os serviços de controle utilizados pelo BioNimbus. Contudo, este trabalho está diretamente relacionado à forma de implementação e funcionamento de quatro dos seis serviços ilustrados na Figura 3.1, os quais são o *Monitoring Service*, o *Discovery Service*, o *Storage Service* e o *Scheduling Service*. Estes serviços estão concentrados no núcleo do BioNimbus, onde cada um deles tem uma determinada função na nuvem federada [25].

Algumas propostas de mudanças serão necessárias em alguns serviços para que eles se adaptem e proporcionem um melhor funcionamento no escalonamento de uma tarefa, sem que haja modificação da proposta inicial dos serviços, e sim melhora no desempenho geral da arquitetura.

3.2 Camada de Aplicação

Os serviços providos pela camada de aplicação, exibida na Figura 3.1, estão relacionados à integração do usuário com a nuvem federada. Estes serviços são responsáveis pela coleta das ações que os usuários desejam realizar na nuvem federada, e encaminhamento

das mesmas para o núcleo da nuvem em formato de mensagem. Os serviços precisam fazer parte da rede P2P para que seja possível o envio das mensagens para o núcleo da arquitetura [35]. Estas mensagens estão previstas pelos serviços controladores do núcleo informando-lhes quais tarefas realizar, como listar os arquivos armazenados na federação, fazer um *upload* de arquivo, ou submeter um *job* para execução. Os serviços da aplicação podem ser realizados por meio de linha de comando, interface gráfica (GUI) e *Workflows Management Systems - WfMS*.

3.3 Camada de Núcleo

Na Figura 3.1 também é possível observar o núcleo do BioNimbus e os seus principais serviços, propostos por Saldanha [35], que serão brevemente descritos nesta seção. O núcleo é o responsável por gerenciar a federação de nuvens computacionais. Entre suas funções estão o descobrimento de provedores e recursos, o escalonamento, o acompanhamento das execuções de tarefas, o armazenamento de arquivos, o serviço de segurança, o serviço de tolerância a falhas, etc.

3.3.1 *Job Controller*

O *Job Controller* faz a ligação entre o núcleo da arquitetura e a camada de interação com o usuário. Ele realiza o controle de acesso do usuário à federação para realizar pedidos de execução, e também gerencia os pedidos dos usuários mantendo o controle por usuário e, assim, armazenando os seus dados para consultas posteriores. Na verificação de credenciais dos usuários necessária para o gerenciamento, o controlador acessa o *Security Service*.

3.3.2 *SLA Controller*

O *Service-level Agreement* (SLA), uma acordo de nível de serviço que tem por objetivo a garantia da qualidade do serviço esperada, é responsável pela implementação do chamado ciclo de vida de SLA. No acordo são estabelecidos alguns parâmetros informados pelo usuário, que são identificados pelos chamados *templates*, os quais representam, entre outras coisas, os parâmetros de QoS (*Quality-of-Service*) que um usuário negociou com a arquitetura. Os dados preenchidos pelo usuário na camada de interação descreve os requisitos funcionais dos recursos que deverá ser utilizado pelo usuário. O *SLA Controller* irá então verificar se os requisitos do usuário são suportados pela nuvem federada no instante da requisição. A informação do acordo definido com o usuário é armazenada na sessão do usuário controlada pelo *Job Controller* após a finalização da negociação, devendo ser monitorada para que os requisitos acordados sejam cumpridos.

3.3.3 *Monitoring Service*

Este serviço controla a nuvem federada monitorando as aplicações e os *jobs* para a execução. Ao receber o pedido de execução de um *job*, vindo do *Job Controller*, o serviço de monitoramento identifica se o *job* está disponível em algum provedor de serviço, e envia o pedido para o *Scheduling Service*, o qual monitora o pedido garantindo que todos os

pedidos sejam devidamente atendidos e executados. Mensagens periódicas são enviadas para os recursos para saber sobre a execução das tarefas correspondentes aos pedidos, *jobs*, além de informar ao *Job Controller* sobre o sucesso ou falha da execução de um *job*. Outras mensagens relacionadas com o acompanhamento do estado do *job* são respondidas pelo *Monitoring Service*.

Dessa forma, é por meio das mensagens periódicas enviadas para os recursos que buscam informações sobre o estado dos *jobs*, que o *Monitoring Service* é capaz de verificar se uma execução é finalizada com sucesso, e sem violar os parâmetros de SLA acordados com o usuário.

3.3.4 *Security Service*

O *Security Service* é responsável pelas políticas de cada provedor de serviço que garante independência entre eles. Isso é um dos principais requisitos do *Security Service*, como também criar contextos de segurança entre usuários e provedores de serviço e estabelecer políticas de controle de acesso para um novo provedor na rede P2P da arquitetura [35]. O contexto de segurança divide seus requisitos em três: autenticação, autorização e confidencialidade. Na autenticação é utilizado o protocolo *Single Sign-On* (SSO)[3] para não haver a necessidade de uma autenticação centralizada de usuários, evitando impor limites para a escalabilidade e a dependência entre processos. A autorização é implementada por meio de uma lista de controle de acesso (ACL) que é fornecida para cada novo provedor de serviço, e informa a permissão para um determinado recurso, o que permite que cada provedor tenha o controle de seus recursos. E, por fim, a confidencialidade, que está na parte de sigilo nas trocas de mensagens entre os membros da nuvem federada, porém, a arquitetura do BioNimbus não exige a utilização de sigilo, deixando que cada provedor forneça seu certificado.

3.3.5 *Discovery Service*

O *Discovery Service* identifica as nuvens que formarão a nuvem federada e quais informações ele precisa saber para informar os parâmetros que serão passados para o *plug-in*, além da capacidade dos recursos fornecidos pelos provedores de serviço. O *Discovery Service* envia, então, mensagens de requisição para os recursos pela rede *Peer-to-Peer*, e aguarda as respostas com as informações da infraestrutura e os recursos disponíveis. Todo o provedor que estiver na rede *Peer-to-Peer* pode responder as mensagens de requisição do *Discovery Service*, mas para integrar a rede, primeiramente, ele deve verificar sua permissão com o serviço de segurança, o *Security Service*. O *Discovery Service* mantém uma estrutura de dados contendo os dados fornecidos com as respostas dos recursos, logo a cada nova resposta a estrutura é atualizada ou incrementada, removendo inclusive aqueles provedores que não fazem mais parte da federação.

3.3.6 *Scheduling Service*

O *Scheduling Service* recebe os pedidos de execução de tarefas e distribui dinamicamente as instâncias desses pedidos, chamados de *jobs*, entre os provedores de serviços da federação, mantendo um registro das execuções escalonadas, e controlando a carga

em cada provedor, redistribuindo, assim, as execuções quando os recursos estão sobrecarregados. Para realizar a distribuição e a redistribuição dos pedidos, é realizado o escalonamento segundo uma política já configurada que determina em qual recurso ele irá ser executado. A política de execução recebe a lista de *jobs* para execução e retorna um mapeamento de *jobs* com seus respectivos recursos computacionais disponíveis. Além de receber e escalonar os *jobs*, o *Scheduling Service* monitora a execução de cada *job* para verificar se ele esta há muito tempo aguardando para ser executado e, se estiver, ele cancela o pedido de execução e realiza um novo escalonamento do *job*.

3.3.7 *Storage Service*

O *Storage Service* coordena a estratégia de armazenamento dos arquivos a serem consumidos ou produzidos pelas tarefas executadas. Ele deve decidir sobre o que deve ser feito com os arquivos, sua distribuição e sua replicação. Decisões de qual o local que será feito o *upload* do arquivo são baseadas nos dados repassados pelo *Discovery Service*, o qual possibilita a utilização de diferentes estratégias de armazenamento.

3.3.8 *Fault Tolerance Service*

O *Fault Tolerance Service* é responsável pela garantia da disponibilidade dos demais serviços da arquitetura. Essa garantia é feita ao manter um registro dos pontos da rede P2P em que os demais serviços estão executando, para que seja possível o monitoramento da execução por meio de mensagens em curtos intervalos. Desta forma, se um ponto não responde à requisição, é iniciado uma eleição para determinar onde será iniciado uma nova instância do serviço, garantindo que o serviço não fique indisponível.

3.4 Camada de Infraestrutura

A camada de infraestrutura é formada por todos os recursos computacionais que são providos pelas nuvens que formam a federação. Esses recursos irão executar o BioNimbus e por meio dos *plug-ins* eles são reconhecidos e conectados na rede P2P formada pelo núcleo do BioNimbus. Logo, todos os recursos irão conhecer os demais por meio da interface criada pelos *plug-ins*, que disponibilizará também os dados de cada máquina.

3.4.1 Integração dos Serviços

A integração desses serviços na arquitetura é feita por meio de uma interface que pode ser usada para adicionar novas e não previstas funcionalidades na arquitetura, sem impactar em seu funcionamento. Esta interface exige que sejam implementadas funcionalidades à arquitetura como, por exemplo, a sinalização de eventos pela rede P2P [35]. Tem-se ainda as funcionalidades de interação com o usuário que é realizada pelos serviços da camada de Aplicação. Toda a comunicação entre os serviços controladores de diferentes recursos e as aplicações de interação com o usuário é realizado por meio de uma rede P2P.

O funcionamento da arquitetura BioNimbus depende dos vários serviços analisados neste capítulo. Para os objetivos deste trabalho, o serviço de *Scheduling Service* deve ser estudado detalhadamente. No *Scheduling Service* tem-se a implementação de um

algoritmo de escalonamento, o DynamicAHP [25], e a próxima seção será voltada para a análise do seu funcionamento e para a identificação de suas peculiaridades, vantagens e desvantagens.

3.5 O Algoritmo DynamicAHP

O algoritmo proposto por Borges [25], chamado de DynamicAHP, é classificado como um escalonador global, dinâmico e não distribuído, segundo a proposta taxonômica de Casavant e Kuhl [34], a qual é ilustrada de forma resumida na Figura 3.2 que demonstra alguns níveis da classificação.

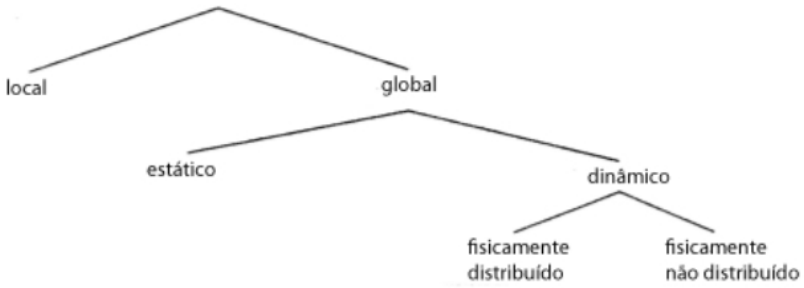

Figura 3.2: Classificação dos Algoritmos de Escalonamento [25].

Um algoritmo pode ser global ou local. Se for local considera somente um processador e se for global são considerados vários locais para executar a tarefa. No nível abaixo o algoritmo é classificado como estático ou dinâmico. Para um algoritmo estático as informações para realizar o escalonamento são carregadas no sistema antes da política ser iniciada. Já o escalonador dinâmico utiliza poucas informações sobre os recursos ao ser carregado no sistema, obtendo-as de forma mais detalhada e atualizada ao longo da rotina de escalonamento. E para um escalonador dinâmico tem-se o nível de classificação onde o escalonador é fisicamente distribuído ou fisicamente não distribuído. O fisicamente não distribuído executa somente em um sistema central e o fisicamente distribuído executa em vários recursos.

O algoritmo DynamicAHP tem como base o método *Analytic Hierarchical Process - AHP*, um método que considera fatores subjetivos para auxiliar os seres humanos na tomada de decisões. Considerando a semelhança na tomada de decisões com um algoritmo de escalonamento, Borges [25] propôs a utilização dessa estratégia de maneira adaptada para auxiliar o algoritmo de escalonamento.

O método AHP facilita o processo de tomada de decisões, pois organiza as percepções, sentimentos, julgamentos e memórias de uma forma capaz de mostrar as forças influentes em uma decisão [25]. Para realizar a tomada de decisão é necessário modelar o problema dividindo-o em problemas menores e estruturando-o em uma árvore de hierarquia de forma que o problema se concentra na raiz da árvore, e os seus nós são divisões do problema em problemas menores. Os problemas menores podem também ser divididos em outros subproblemas. Com a hierarquização e divisão do problema é, então, definido o grau de

relevância geral de um problema representado pela sua prioridade. O cálculo de relevância é feito por meio de uma matriz de comparação.

O DynamicAHP é capaz de tomar decisões de escalonamento sem ter nenhum conhecimento prévio sobre as características da tarefa, e também é capaz de fazer um balanceamento de carga entre os provedores, conforme novos provedores são integrados à federação de nuvem. A ideia deste escalonador é mapear os provedores da nuvem e as tarefas que precisam ser escalonadas, definindo em qual provedor cada tarefa deve ser executada considerando que para a definição da prioridade dos problema não é utilizado a intuição e sim cálculos determinísticos [25].

3.5.1 Implementação e Integração do DynamicAHP

O algoritmo foi implementado com a finalidade de determinar qual recurso vai executar qual tarefa, a partir dos recursos disponíveis e das tarefas a serem executadas. Para obter as informações referentes aos recursos e as tarefas, o DynamicAHP utiliza os valores passados pela comunicação entre os *plug-ins* pertencente a cada recurso que forma a federação.

Para escalonar uma tarefa é necessário obter os recursos disponíveis, estes recursos são armazenados na lista "nós disponíveis", e então é determinado qual é o melhor recurso para executar a tarefa por meio do algoritmo DynamicAHP. No final é retornado o melhor recurso disponível, segundo a prioridade global dos recursos analisados.

O funcionamento do BioNimbus se inicia com a interação do usuário por meio da interface, onde o usuário informa o *workflow* que deseja executar. As tarefas do *workflow* escolhido são enviadas para o controlador de tarefas que confere a validade das mesmas. Se ocorrer erro ao final da execução de alguma tarefa, o gerenciador de tarefas informa ao usuário. Mas, se a tarefa for válida, ela é enviada ao serviço de monitoramento que armazena essa tarefa em uma lista de tarefas pendentes. Esse serviço é responsável por informar ao escalonador que existem tarefas pendentes aguardando por escalonamento e também pela verificação do *status* da tarefa, colocando-a novamente na lista de tarefas pendentes caso ocorra algum erro durante a execução. Ao receber a informação do serviço de monitoramento de que existem tarefas pendentes o serviço de escalonamento é iniciado utilizando então a política de escalonamento implementada.

Tabela 3.1: Interface para ser Implementada por uma Política de Escalonamento no BioNimbus.

abstract shedPolicy
-Map<String, Plugin> cloudMap;
+setCloudMap(Map<String,PluginInfo> cloudMap); +getCloudMap(); +SchedPolicy getInstance(Map<String, PluginInfo> cloudMap); +Map<JobInfo, PluginInfo> shedule(List<JobInfo> jobInfos);

A interface da política de escalonamento do BioNimbus está descrita na Tabela 3.1, na qual é possível observar que a política de escalonamento tem acesso ao *cloudMap*. No

cloudMap os *PlugInfo* estão mapeados por *String*, e cada *PlugInfo* contém informações do recurso. O método *getInstance* retorna a política definida no arquivo de configuração.

Para realizar o escalonamento, o método *schedule* recebe uma lista de *JobInfos* que contém as informações da tarefa, e retorna um mapa de *JobInfo* com *PluginInfo* informando em qual recurso a tarefa deve executar. Com as listas de tarefas e recursos o serviço de escalonamento comunica-se com o provedor da nuvem do recurso para iniciar a execução da tarefa. O serviço de monitoramento é informado para que ele possa verificar o *status* de cada tarefa, e quando a execução terminar de executar ele irá informar ao controlador de tarefas que se comunicou com a interface do usuário informando que a tarefa foi concluída.

3.5.2 Experimento do DynamicAHP no BioNimbus

Para demonstrar o desempenho do algoritmo DynamicAHP na estrutura do BioNimbus, Borges[25] realizou testes na estrutura do BioNimbus utilizando a sua política. Para comparação utilizou também a política de escalonamento do algoritmo *Round Robin* [25]. Estes testes e o ambiente onde eles foram realizados serão descritos nesta seção.

O BioNimbus foi composto por três provedores de nuvem, formado por três *clusters* de 12, 8 e 4 núcleos, com aproximadamente 2,4 terabytes de armazenamento. O armazenamento foi implementado utilizando *Hadoop Distributed File System* (HDFS)[8]. O BioNimbus foi instalado em todos os *clusters* rodando um *plug-in* que permitia a comunicação com a infraestrutura do *Hadoop* e os serviços do BioNimbus que estavam executando no *cluster* com maior capacidade de processamento.

Para a execução das tarefas na infraestrutura do BioNimbus foi instalada uma ferramenta de bioinformática usada para fazer mapeamento genético, que foi oferecida como uma aplicação pelo recurso participante da federação. Os arquivos de entrada foram divididos em três arquivos de 100, 200 e 300 MB, onde cada um tinha um grupo de 20 tarefas. As tarefas foram enviadas para o BioNimbus em grupo de 5 tarefas, com intervalos de um minuto entre os grupos para simular várias submissões feitas por usuários diferentes em momentos distintos.

O algoritmo *Round Robin* implementado com adaptações ao BioNimbus funciona de forma que as tarefas são divididas pelos recursos que possam executá-las e, assim, são atribuídas sequencialmente entre os recursos, sem nenhuma análise sobre o recurso ou tarefa.

Com a Figura 3.3 é possível notar o resultado do teste da execução do algoritmo *Round Robin* e do DynamicAHP, o qual rodando no mesmo ambiente, as mesmas tarefas obtiveram desempenhos diferentes. É possível verificar que o tempo de execução no algoritmo *Round Robin* foi de aproximadamente 6 minutos, maior do que o tempo de execução do DynamicAHP. Também é possível verificar que o DynamicAHP escalonou uma quantidade de tarefas maior para as nuvens com maior poder de processamento, pois entre as características que influenciam na política do seu escalonamento, o processamento é o que sofre maior mudança de valores. Logo, a nuvem com o maior poder de processamento, *cluster* de 12 núcleos, recebeu uma quantidade maior de tarefas do que as outras duas nuvens, e a nuvem com o *cluster* de 8 núcleos recebeu uma quantidade maior de tarefas do que a nuvem com *cluster* de 4 núcleos. O que não acontece nos resultados do algoritmo de *Round Robin*, visto que ele não utiliza nenhuma análise de dados dos recursos.

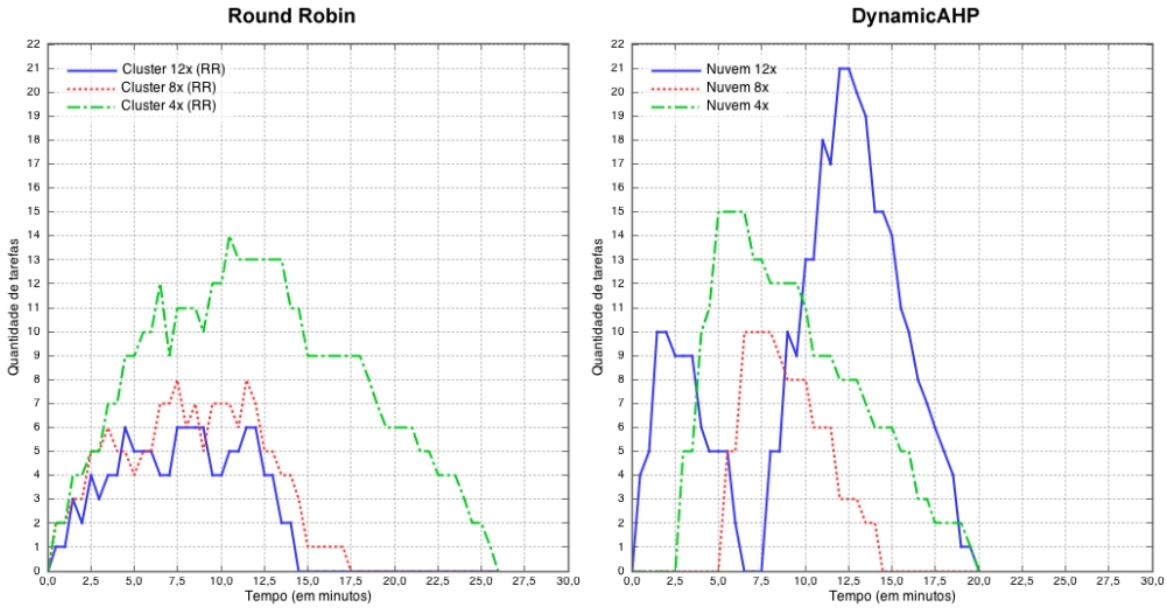

Figura 3.3: Comparação de Tempo de Execução e Quantidade de Tarefas para Ambos os Algoritmos [25].

3.5.3 Vantagens e Desvantagens

O algoritmo DynamicAHP obteve um resultado melhor do que o algoritmo *Round Robin* de acordo com a Figura 3.3, o que é um bom desempenho em um ambiente de nuvem, pois quanto menos tempo um recurso é consumido, menor será o gasto, caso a nuvem utilizada seja uma nuvem paga. Essa vantagem do algoritmo DynamicAHP foi obtida pelo fato do algoritmo escalonar mais tarefas para o recurso com maior poder de processamento, visto que este recurso foi mais utilizado do que os demais, sem comprometer ou utilizar de forma errada, os outros recursos.

A utilização dos recursos disponíveis foi mais eficiente com o escalonamento realizado pelo DynamicAHP, permitindo assim um desempenho final melhor do que o obtido pelo *Round Robin*.

Porém, ao observar a Figura 3.4, onde o tempo de execução das tarefas são exibidos individualmente, verifica-se que existem tarefas onde o tempo de execução utilizando o DynamicAHP é bem superior ao tempo de execução do *Round Robin*, como nas tarefas de 16 à 20. Esse tempo mais alto é devido, segundo Borges [25], a latência da infraestrutura com menor poder de processamento, a nuvem de *cluster* com 4 núcleos, que era menor do que as demais e, portanto, a mais vantajosa. Isso não foi suficiente para melhorar o tempo de execução já que a quantidade de processadores foi mais relevante que a latência, nesse caso. Logo, ao analisar os dados dos recursos, o DynamicAHP deveria ter considerado o fato de que uma latência menor nem sempre irá compensar uma maior capacidade de processamento e, consequentemente, não irá otimizar o desempenho final.

Desta forma, para compensar esse fator, a inclusão de pesos diferentes para as características dos recursos no momento de calcular a sua prioridade global seria uma forma de equilibrar o desempenho do algoritmo e, consequentemente, diminuir o tempo total de execução das tarefas. Assim, ao se analisar as três características, latência, confiabilidade

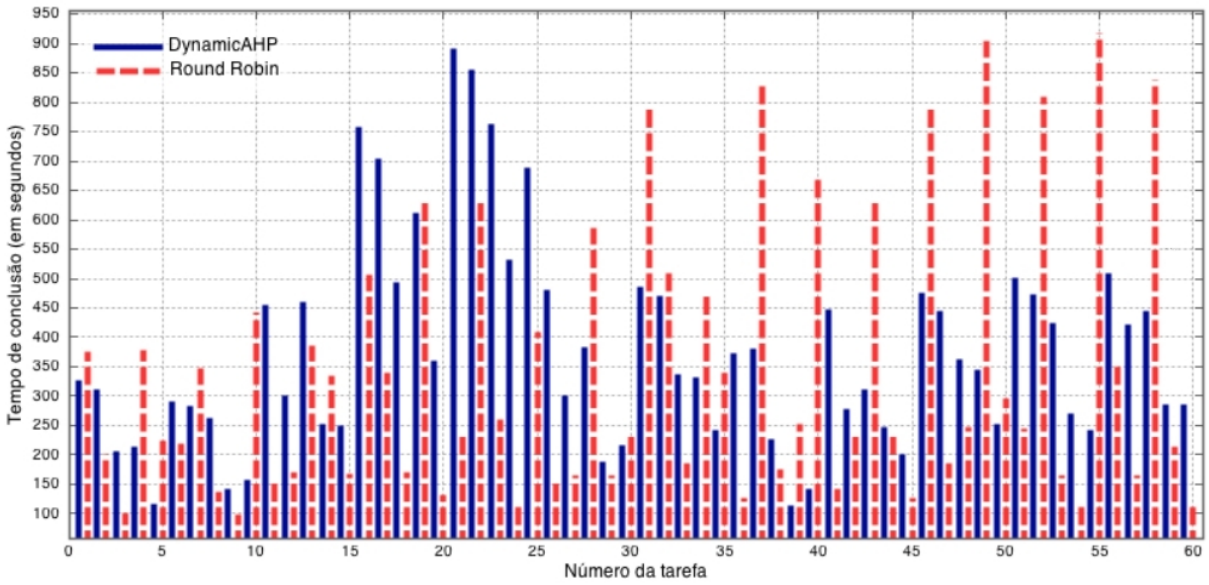

Figura 3.4: Comparação de Tempo de Execução e Quantidade de Tarefas para Ambos os Algoritmos [25].

e processamento, que atualmente tem os mesmos pesos, seria definido um maior peso ao processamento, modificando então o valor da prioridade global.

Além disso, ao analisar todas as tarefas individualmente, a partir da Figura 3.4, será possível verificar que da tarefa 1 até a tarefa 10, o tempo de execução entre os algoritmos é melhor em 5 delas, utilizando a política do *Round Robin* e também será melhor nas outras 5 tarefas se for utilizado a política do DynamicAHP. Já nas tarefas de 11 até 20, o algoritmo *Round Robin* levará vantagem em relação ao DynamicAHP, pois 8 de 10 tarefas obtiveram um tempo menor de execução. O que também ocorre com as tarefas de 21 até 30. Realizando então a mesma análise até a última tarefa, será obtido um total de 37 tarefas executadas com um tempo menor pelo algoritmo *Round Robin* contra 23 tarefas executadas em um tempo menor pelo DynamicAHP, o que demonstra que a política do DynamicAHP, apesar de ter obtido um tempo total de execução final menor do que o *Round Robin*, devido ao melhor aproveitamento dos recursos em determindados momentos, não foi muito eficiente ao analisar as tarefas individualmente e, portanto, poderia ter obtido um tempo ainda menor.

O motivo do alto tempo de execução obtido pelo *Round Robin*, 25min37s para rodar todas as 60 tarefas contra 19min36s do DynamicAHP [25], pode ser analisado verificando-se o ambiente em que as tarefas foram executadas na nuvem.

O algoritmo DynamicAHP foi a primeira política de escalonamento implementada no BioNimbus, o que resultou na experimentação e melhor observação da demanda e requisição de variáveis que devem ser consideradas ao realizar o escalonamento. Como já foi apontado antes, a definição de pesos para realizar o escalonamento iria modificar o valor da prioridade global, e possivelmente melhorar o desempenho de acordo com a demanda do usuário. Além disso, alguns pontos ainda podem ser considerados ao realizar o escalonamento sem sobrecarregar ou prejudicar o tempo necessário para o escalonamento, levando a melhora no desempenho e no comprimento de requisitos dos clientes. Alguns

pontos que poderiam ser considerados pelo DynamicAHP e são considerados pelo algoritmo proposto por este trabalho, que dizem respeito a demanda do cliente e a possível melhora do desempenho geral do escalonador. Outros fatores são a melhora no balanceamento de carga do ambiente da federação, distribuindo melhor as tarefas de forma mais dinâmica e igualitária entre os recursos considerando suas capacidades computacionais; a consideração do tipo de recurso sobre o qual o cliente quer executar sua tarefa, poupando custos finais; a consideração da capacidade da memória primária disponível, otimizando o tempo de execução das tarefas; e considerando a localidade dos arquivos de entrada, evitando possíveis cópias desnecessárias desses arquivos entre os recursos da federação.

O próximo capítulo apresenta as mudanças realizadas na plataforma BioNimbus, a fim de torná-la mas dinâmica, eficiente, escalável e adaptável à implementação de novos serviços, como por exemplo, um novo algoritmo de escalonamento de tarefas.

Capítulo 4

Modificação do BioNimbus

Este capítulo descreve os serviços e funcionalidades agregadas à nova versão do BioNimbus, onde o presente trabalho foi implementado. O capítulo inicia descrevendo sobre as tecnologias que foram utilizadas na nova versão, as modificações nos serviços do BioNimbus e o impacto em sua arquitetura.

4.1 Modificação da Estrutura do BioNimbus

O BioNimbus sofreu modificações nos serviços implementados a partir da sua primeira versão, proposta por Saldanha [35]. Essas modificações foram provocadas a partir da necessidade de se alterar o serviço de comunicação P2P por um serviço de comunicação mais escalável. É importante ressaltar que as propostas e os objetivos buscados por Saldanha [35], tais como a integração e o controle de diferentes provedores de infraestrutura, uma arquitetura flexível e tolerante a falhas, com grande capacidade de processamento e armazenamento foram mantidas e otimizadas. Dessa forma, a nova versão do BioNimbus continua sendo uma plataforma que constrói uma federação em nuvem, e que executa serviços de bioinformática com recursos heterogêneos, públicos ou privados.

Assim como o BioNimbus, existem outros sistemas, baseados na troca de mensagens entre processos [33], que formam um sistema distribuído e possibilitam a comunicação entre aplicações remotas. Assim, outras formas de integração e comunicação entre aplicações distribuídas foram propostas, tais como a Chamada de Procedimento Remoto (RPC) que modifica a forma de comunicação e programação desses sistemas, aumentando a transparência no acesso às aplicações remotas. A ideia principal ao se propor a comunicação por meio de RPC era permitir que programas chamassem procedimentos localizados em outras máquinas [33] de maneira transparente. Essas chamadas de procedimento podem ser divididas em grupos de chamadas do servidor e do cliente.

Para auxiliar na coordenação e organização do sistema distribuído, foi utilizado um serviço para aplicações distribuídas da Fundação Apache [13], o Zookeeper [15]. O Zookeeper tem a proposta de fornecer um ambiente de fácil controle para auxiliar no armazenamento de dados de tamanho pequeno que são utilizados para gerenciar um sistema distribuído. Esse serviço foi intensamente utilizado na nova implementação do BioNimbus, e principalmente na implementação do algoritmo de escalonamento proposto neste trabalho, o qual será descrito no capítulo seguinte. A estrutura básica do Zookeeper será descrita na Seção 4.2.

4.2 Zookeeper Apache

O Zookeeper [15] é um serviço de coordenação de aplicações distribuídas, criado pela Apache [13] para ser de fácil programação e manipulação. Ele utiliza um modelo de dados parecido com a estrutura de árvores de diretórios, e possui uma implementação de serviços para a sincronização, a manutenção e a configuração de grupos e nomenclaturas. Tem a finalidade de reduzir a dificuldade de implementar os serviços dos sistemas distribuídos, que podem levar a uma alta complexidade, a uma grande dificuldade de gestão, de coordenação e de manutenção.

4.2.1 Arquitetura Básica do Zookeeper

O serviço Zookeeper tem o objetivo de facilitar a implementação de serviços de coordenação em aplicações distribuídas, utilizando espaços de nomes hierárquicos compartilhados que são organizados de forma parecida com um sistema padrão de arquivos. Esses espaços de nomes são chamados de *znodes* e se assemelham a arquivos de diretórios que são mantidos em memória quando o serviço está em execução. Um exemplo ilustrado de *znode* pode ser visto na Figura 4.1. Os dados armazenados no Zookeeper podem estar presentes em cada *znode* como um conjunto de bytes que são tratados pela aplicação como uma variável do tipo *string*, tendo como finalidade coordenar serviços remotos, auxiliando na construção e na manutenção de um sistema distribuído, porém seu tamanho máximo permitido por *znode*, para armazenamento de dados, é de 1 megabyte (MB) [36]. Os *znodes* são identificados pelo seu caminho na estrutura, como o *znode /app/p_1* da Figura 4.1

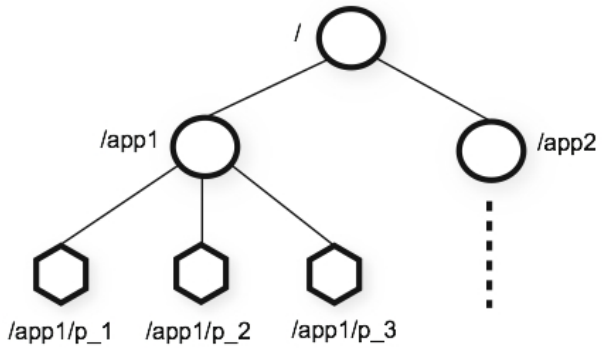

Figura 4.1: Exemplo de uma Estrutura Zookeeper [15].

O armazenamento dos valores no *znode* deve ser voltado para informações de uso geral de controle e coordenação, tais como metadados, caminhos, endereços, dados de configuração, etc.

O ZooKeeper permite que processos distribuídos se coordenem por meio dos *znodes* [15]. Pelo seu alto desempenho, ele pode ser utilizado em sistemas distribuídos de grandes dimensões e, assim como os processos distribuídos que ele coordena, o próprio ZooKeeper tem a finalidade de funcionar de forma distribuída, e seus servidores serem replicados ao longo de sua intensa utilização.

Os *clusters* formados pelos servidores Zookeeper tem como finalidade disponibilizar o serviço Zookeeper da mesma forma que seria fornecido se existisse apenas um servidor

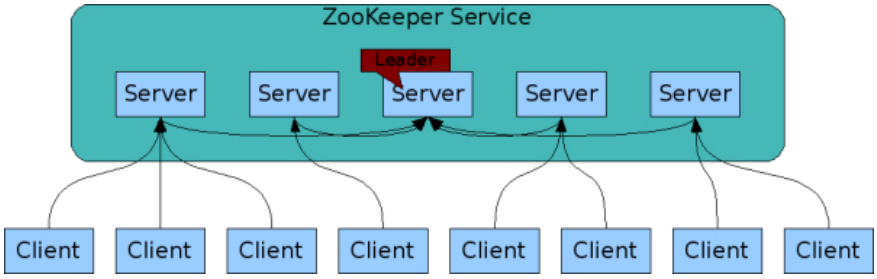

Figura 4.2: *Cluster* de Servidores Zookeeper [15].

Zookeeper. A Figura 4.2 demonstra um exemplo de servidores do Zookeeper formando um *cluster* para promover o serviço Zookeeper. Nesses *clusters*, em todos os servidores Zookeeper é mantida a mesma estrutura e informações.

Cada novo *znode* criado no servidor Zookeeper representa um novo diretório na estrutura de dados similar à uma árvore, com o diretório raiz e seus *znodes* filhos. É possível criar vários filhos dentro dos *znodes* pais, utilizando-se *znode* do tipo persistente, capazes de conter filhos e continuar a existir no servidor Zookeeper após a queda do sistema. O outro tipo de *znode* existente no Zookeeper é o *znode* efêmero, cujas características são o fato de não poder conter *znodes* filhos, e uma vez criados são automaticamente deletados quando o cliente Zookeeper responsável pela sua criação for desconectado do servidor. Em um sistema distribuído, por exemplo, cada novo nó participante da rede pode se registrar na rede ao criar uma pasta efêmera no servidor Zookeeper que realiza sua coordenação. Ao ficar indisponível, a pasta efêmera que o representava no servidor Zookeeper irá ser automaticamente apagada, e o sistema de arquivo distribuído pode ser informado da mudança na rede [15].

O Zookeeper suporta o conceito de *watchers* [15] (veja a Figura 4.3) que funciona como uma espécie de observador de modificações, similar ao conceito proposto pelo padrão de projeto *Observer* [10], enviando alertas a quem solictar. Esses *watchers* podem ser adicionados nos *znodes* para observar alguma mudança de dados ou de novos *znodes*. O *watcher* é disparado uma única vez ao detectar alguma alteração no *znode* onde ele foi adicionado. Para receber novos alertas daquele mesmo *znode*, é necessário realizar a inclusão de novos *watchers*. Cada *znode* pode conter mais de um *watcher*, que quando observa alguma alteração dispara o alerta a todos os clientes do serviço que solicitaram a notificação. Na Figura 4.3, o Zookeeper verifica que o Recurso A está indisponível e envia o alerta para os demais recurso sobre sua indisponibilidade.

Existem três tipos de alterações que são informadas pelos *watchers*, os quais são a mudança de dados, a criação de um *znode* e a exclusão de um znode. Para cada uma delas é necessário adicionar um *watcher*.

Com a finalidade de prover e garantir uma interface simples de utilização, o Zookeeper permite apenas sete tipos de operações em sua estrutura. As operações são de criação, exclusão e verificação de existência de *znodes*, recuperação e envio de dados para os *znodes*, a recuperação da lista de filhos e a sincronização dos dados.

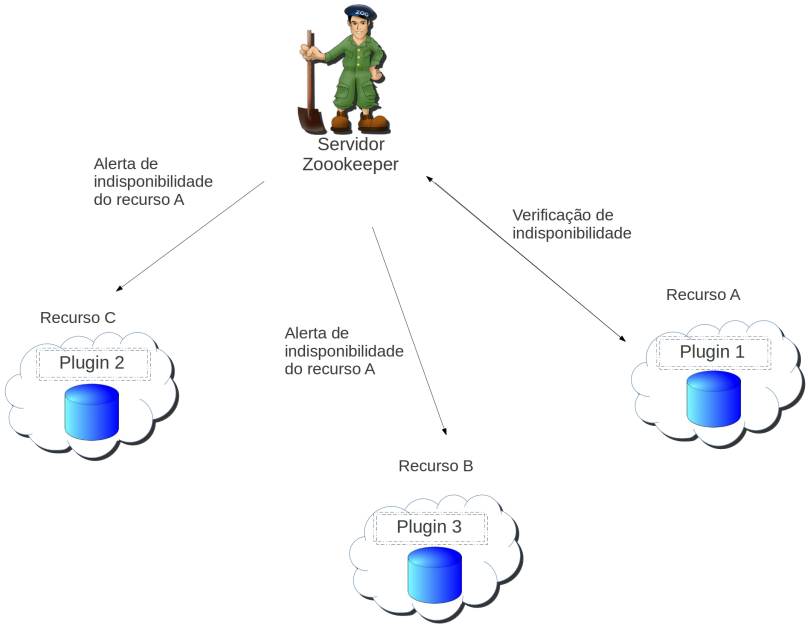

Figura 4.3: Exemplo de Disparo de um Alerta feito pelo ZooKeeper.

4.2.2 Estrutura Zookeeper no Ambiente BioNimbus

A estrutura do Zookeeper utilizada no BioNimbus foi criada com objetivo de facilitar o controle dos recursos participante da federação, e o funcionamento dos serviços que formam o sistema. A estrutura segue uma organização em que cada recurso que utiliza o sistema é visualizado como um *znode*, que é criado ao estar disponível e apagado quando indisponível. O tipo de *znode* mais adequado para realizar essa representação dos *plug-ins* é o do tipo efêmero, que indicaria sua disponibilidade. Porém, como a utilização de *znode* efêmero não poderia permitir a criação de *znodes* filhos que foram utilizados pelos serviços do BioNimbus para manter dados e informações de controle e coordenação, foi necessário criar um *znode* persistente representando cada novo recurso. Para controlar a disponibilidade do recurso foi realizada a criação de um *znode* efêmero filho do *znode plug-in*, e continuou-se permitindo que outros *znodes*, no mesmo nível hierárquico, fossem criados. Os outros *znode* filhos foram criados para armazenar e manter dados dos serviços utilizados na arquitetura do BioNimbus, como o *znode* do serviço de escalonamento e de armazenamento.

A estrutura do Zookeeper, para o BioNimbus, é ilustrada na Figura 4.4, onde cada *plug-in* da federação é representado por um *znode*, *peer_IdPlugin*, que possui seu conjunto de *znodes* filhos que contem informações referentes ao seu funcionamento. O *znode* filho do *peer_IdPlugin*, *scheduling*, é o *znode* mais utilizado pelo algoritmo proposto por este trabalho, pois contém os dados do escalonamento, das tarefas que aguardam a execução e das tarefas em execução. O *znode scheduling* armazena também os dados utilizados pelo escalonador e os demais *znodes* filhos. Quando uma tarefa é submetida pelo usuário no BioNimbus, e como o BioNimbus ainda não sabe em qual recurso ela vai executar antes do escalonamento, seus metadados são armazenados no *znode jobs* do servidor

Zookeeper. Somente após o seu escalonamento ela é encaminhada para o *peer_ IdPlugin*, correspondente ao recurso eleito pelo escalonador.

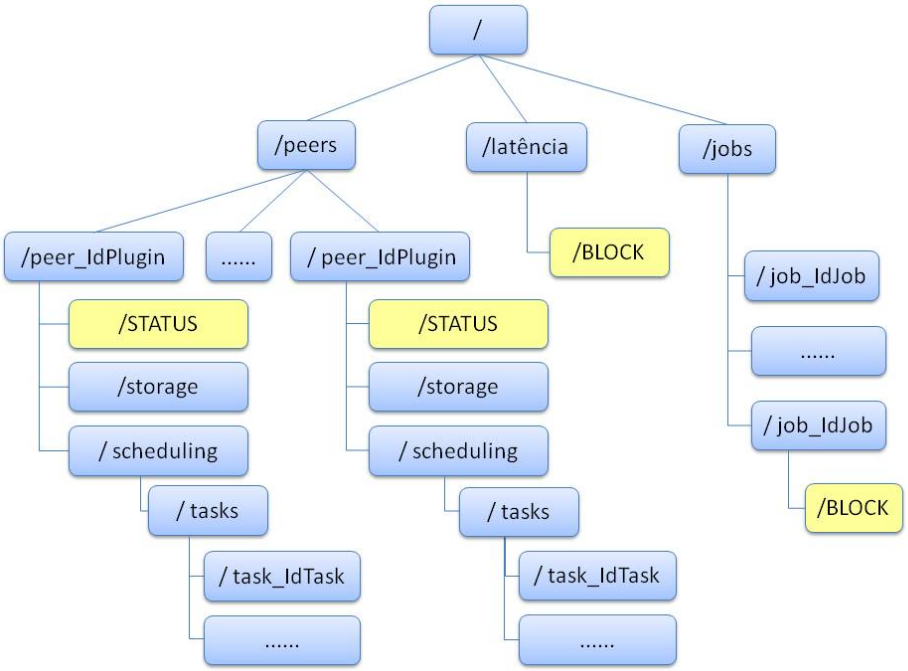

Figura 4.4: Estrutura dos *znodes* do Servidor Zookeeper para o BioNimbus.

Assim, a tarefa escalonada é um *znode* do tipo persistente, *task_ IdTask*, filho do *znode tasks*, localizado hierárquicamente como filho do *znode peer_ IdPlugin*, assim como a tarefa em execução, diferenciando-se apenas pelo seu estado, armazenado como dado no próprio *znode*.

4.3 Apache Avro

O Apache Avro [12] é um sistema de RPC e de serialização de dados desenvolvido pela fundação Apache [13]. O Avro é um exemplo de uma longa lista de sistemas RPC livres e comerciais desenvolvidos nas últimas decadas. Como exemplos de sistemas anteriores tem-se o sistema Thrift [14], criado pelo Facebook [11] e, atualmente, mantido como um projeto da Fundação Apache [13]; e o Protocol Buffers [20] criado pela *Google*. Um característica importante desses sistemas de RPC modernos é o suporte a mais de um protocolo de transporte de dados em rede [12], e a mais de um formato de serialização de dados. O Avro por exemplo, suporta o formato de dados binários e JSON [21], e os protocolos HTTP e um protocolo binário próprio [12].

A diferença entre o Avro e os seus antecessores é o fato dele ter sido pensado para trabalhar com grande volumes de dados. O Avro possui algumas características [12] definidas pela Apache, tais como uma rica estrutura de dados com tipos primitivos (*int*, *boolean*, *array*, etc), um formato de dados compacto, rápido e binário, um arquivo que contém os dados persistentes, o uso da tecnologia RPC e uma simples integração com linguagens dinâmicas.

Tabela 4.1: Exemplo de um Esquema Utilizado na Plataforma.

```
record   PluginFile {
         string id;
         string path;
         string name;
         long size;
         array<string> pluginId;
     }
```

O uso de esquemas (veja a Tabela 4.1) também está presente na estrutura do Avro, permitindo uma serialização rápida e pequena, já que a escrita e a leitura são facilitadas por esses esquemas, e uma maior portabilidade e evolução dos esquemas utilizados pelo Avro. Os esquemas presentes no Avro são armazenado juntamente com os dados persistidos em arquivos, podendo serem mais tarde lidos com facilidade por qualquer programa, o que reflete também no uso do RPC, onde cliente e servidor tem o mesmo esquema e podem resolver as correspondências entre os campos de dados divergentes de forma simples. O Avro possui seus esquemas definidos com o formato JSON [21], que está presente na arquitetura BioNimbus, e auxilia na implementação de linguagens que possuem sua biblioteca.

O Avro foi escolhido dentre as demais tecnologias que promovem a comunicação remota, para compor a nova estrutura do BioNimbus, pela sua flexibilidade na implementação e facilidade de manipulação.

4.4 Nova Versão do BioNimbus - ZooNimbus

Com as mudanças na implementação das tecnologias utilizadas na arquitetura do BioNimbus, uma nova versão foi criada e chamada de ZooNimbus, que faz referência ao Zookeeper incorporado ao projeto. Essa nova versão não modifica as definições, os objetivos e os serviços do BioNimbus (como observado na Figura 4.5) mantendo a proposta inicial do Saldanha [35]. As mudanças foram na comunicação entre os recursos da federação, na coordenação e no controle dos serviços, que resultaram em uma plataforma para federação mais eficiente, de simples programação, fácil manutenção e, consequentemente, maior confiabilidade.

Todos os serviços sofreram alterações na implementação, agregando, por exemplo, uma maior tolerância a falhas, já que o controlador do ZooNimbus, o Zookeeper, mantém os dados de configuração e de utilização comuns aos serviços que são constantemente acessados e atualizados, permitindo assim a criação de uma estrutura de recuperação.

Para a utilização do Zookeeper na coordenação e no controle das atividades do ZooNimbus foi necessária a implementação de um método responsável pelo tratamento das notificações em todos os módulos de serviço. Assim, todas as mudanças e atualizações que ocorrem no servidor Zookeeper e são responsáveis pela comunicação dos processos, são corretamente identificadas e utilizadas de forma personalizada em cada módulo, modificando, então, a interface de implementação dos serviços, acrescentando dois novos métodos, que podem ser visualizados na Tabela 4.2.

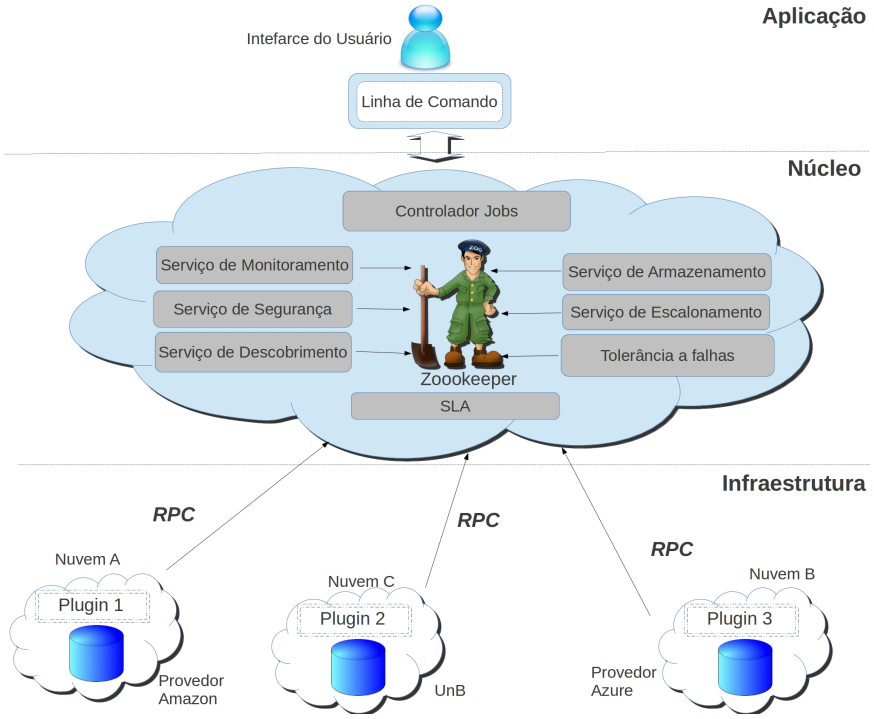

Figura 4.5: Estrutura do ZooNimbus.

Tabela 4.2: Novos Método da Interface Service Implementados pelos Serviços no ZooNimbus.

Service
+verifyPlugins();
+event(WatchedEvent eventType);

O método *verifyPlugins* (veja a Tabela 4.2) é um método público que realiza a atualização de todos os recursos disponíveis na nuvem federada ao receber um alerta informando sobre alguma alteração no número de recursos.

Desta forma, sempre que algum *znode* é alterado e existe um *watcher* responsável pelo seu monitoramento, um alerta é criado e enviado para a classe Java que faz a implementação da Interface *Watcher* no projeto. Essa classe implementada não realiza o tratamento da notificação, ela envia essa notificação para o serviço que fez o pedido de criação do *watcher*, e ele então recebe esse alerta no novo método *event()* da interface *Service*, que pode ser visualizado pela Tabela 4.2. Cada *watcher* pode ser de um tipo diferente, e essa diferença juntamente com a definição do que ela representa em um determinado serviço é o que representa o sentido daquela notificação. Por exemplo, no módulo de descoberta, quando um *watcher* do tipo exclusão, que foi adicionado no *znode STATUS* do *peer_ IdPlugin* é disparado pelo Zookeeper, significa que o recurso está indisponível e o módulo deve deixar esse recurso em modo de espera para ser apagado da estrutura do servidor Zookeeper, até que os demais módulos tenham feito a recuperação de seus dados contidos no servidor referentes a este recurso. Os diferentes *status* do ZooNimbus podem ser visualizados na Figura 4.6.

Desta forma, a modificação do estado de disponibilidade do servidor ZooNimbus é representado Figura 4.6, ele ocorre do estado do recurso de A, quando o servidor está disponível para o estado B, quando o servidor fica indisponível, e por fim para o estado C, onde o servidor está esperando que seja feito todas as recuperações para ele poder ser apagado da estrutura do servidor Zookeeper.

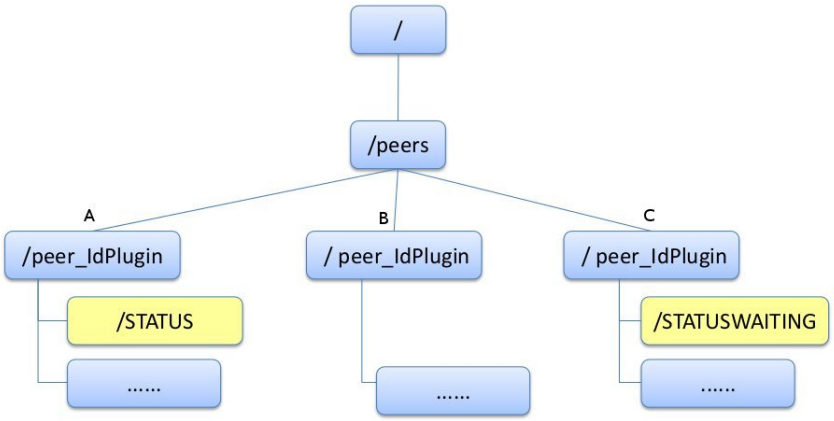

Figura 4.6: Possíveis Estados de um Servidor ZooNimbus.

As mudanças nos serviços relativos a sua forma de funcionamento e integração ao Zookeeper, estão descritas de forma mais detalhada nas próximas seções.

4.4.1 *Discovery Service*

No módulo *Discovery Service* houve a substituição das mensagens *broadcast* da rede P2P pelo controle feito via Zookeeper. Assim sendo, não há mais a necessidade de pesquisar na rede P2P formada pelo BioNimbus para a descoberta da existência e características de novos nós participantes, pois esses dados são lançados no servidor Zookeeper, assim que o novo recurso iniciar sua execução do ZooNimbus. O novo *znode* criado para o novo nó e os seus *znodes* filhos contém as informações que são utilizadas pela plataforma ZooNimbus para gerenciar a federação e para que os módulos funcionem de forma correta. Após o lançamento dos dados, o serviço de descoberta executa uma rotina de reconhecimento dos dados lançados por outros recursos que já iniciaram a execução do ZooNimbus, conhecendo as informações dos demais nós. Assim, sempre que for necessário verificar quais recursos estão disponíveis, pode ser realizada uma leitura de seus dados no Zookeeper, além da utilização de *watchers* para notificar as mudanças em sua estrutura e no seu estado.

Como notado, com a utilização desses *watchers*, não há a necessidade do serviço de descoberta realizar constantes verificações de indisponibilidade dos recursos, provendo uma maior consistência, já que os recursos contidos no servidor *znode* não irão ficar indisponíveis sem que os demais nós da federação sejam avisados. A utilização da classe *PluginInfo* se manteve, e os dados dessa classe, contendo informações do *plug-in*, é ar-

mazenada em cada nó *znode peer_ IdPlugin*, que representa o recurso, podendo ser facilmente acessada por qualquer outro recurso.

O mecanismo de recuperação dos metadados dos servidores ZooNimbus armazenados nos servidores ZooKeeper, geram diferentes estados para a estrutura dos *znodes* no servidor ZooKeeper. Esses estado podem ser visualizados na Figura 4.6. O nó *znode STATUS*, mostrado também na Figura 4.4, que representa a disponibilidade do recurso, é verificado sempre que necessário e existirá até que o recurso fique indisponível. Ao ficar indisponível, o *znode STATUS* é, automaticamente, apagado pelo próprio Zookeeper, propriedade dos *znodes* efêmeros, e então é enviado um alerta àqueles que solicitaram o aviso, e um novo *znode, STATUSWAITING*, é criado permitindo aos demais serviços a recuperação das informações necessárias contidas nos *znodes* daquele servidor ZooNimbus antes que sua estrutura seja apagada pelo módulo de monitoramento, o *Monitoring Service*.

Desta forma, todos os módulos podem solicitar a notificação de indisponibilidade dos recursos utilizando-se um *watcher* no *znode*, e realizar a criação do *znode STATUSWAITING*, iniciando o processo de recuperação das informações do recurso. Assim, para manter o controle e a consistência da tolerância a falha, antes de iniciar a rotina de recuperação dos dados, é verificado se já existe o *znode STATUSWAITING*, e se a recuperação já foi realizada por outro recurso.

Essa nova implementação do serviço de descoberta na federação em nuvem, continua permitindo a escalabilidade do sistema, a utilização de recursos estáticos e dinâmicos, e a flexibilidade com os diversos tipos de recursos.

4.4.2 *Monitoring Service*

O módulo de monitoramento modifica, em partes, sua proposta inicial de pedido de execução e acompanhamento do estado dos *jobs* que foram enviados para execução. Seu principal acompanhamento agora é referente aos recursos que formam a federação em nuvem. Esse acompanhamento é feito de forma integrada ao Zookeeper, com a utilização de *watchers* e tratamento do recebimento de alertas dos mesmos, realizando assim um trabalho de monitoramento livre de envio e solicitação de recebimento de mensagem via rede P2P, evitando consumo de banda e possíveis erros com o não recebimento de dados. Dessa forma, as solicitações e os envios de dados são todos direcionados ao servidor Zookeeper.

O módulo de monitoramento permite ainda que os dados principais utilizados pelos módulos de cada servidor ZooNimbus, armazenados na estrutura do Zookeeper, possam ser recuperados para possíveis rescontruções ou garantias de execuções de serviços solicitados ao ZooNimbus, como foi brevemente descrito na seção anterior.

Quando este módulo recebe o aviso do Zookeeper de que o *znode STATUS* foi excluído, ele verifica se o *znode STATUSWAITING* existe, e se não existir, realiza a sua criação, permitindo aos demais módulos a recuperação dos dados desse servidor ZooNimbus. Como não é possível garantir a ordem cronológica do envio dos avisos realizados pelo Zookeeper, todos os módulos que solicitaram o aviso de indisponibilidade dos demais servidores, podem realizar a criação do *znode STATUSWAITING*, ao receberem a notificação de indisponibilidade para iniciarem suas rotinas de recuperação dos dados.

Assim sendo, ao checar a existência do *znode STATUSWAITING*, o *Monitoring Service* irá verificar se todos os demais módulos já realizaram a recuperação dos dados por meio

da análise das informações contidos em forma de *string* no *STATUSWAITING*. Caso seja verificado que todos os módulos já tenham realizado a recuperação, a estrutura referente ao *peer_ IdPlugin* indisponível será apagada. E caso seja verificado que os módulos ainda não realizaram suas recuperações de dados, o módulo *Monitoring Service* irá aguardar até que seja feita a recuperação, para então, excluir os *znode*.

4.4.3 *Storage Service*

O *Storage Service*, responsável pela gerência e controle dos arquivos existentes na arquitetura, agora implementa uma política de armazenamento para que o uso dos recursos possa ser feito de maneira mais eficiente e estratégica, avaliando o melhor local de armazenamento para cada novo arquivo enviado ou gerado nos servidores ZooNimbus. Para realizar a avaliação de cada recurso com a finalidade de quantificá-los e classificá-los, é realizado o cálculo de custo de armazenamento para cada arquivo. Esse cálculo de custo é feito toda vez que um novo arquivo é armazenado no ZooNimbus, seja por meio de um *upload*, ou por meio da saída de alguma tarefa que foi executada no ambiente.

Com a finalidade de garantir a existência de uma arquivo na federação, foi criado um mecanismo de replicação dos arquivos pertencentes ao ZooNimbus que é executado sempre que um novo arquivo passa a existir na federação, e gera um número de duas cópias para cada arquivo. Estes arquivos, obrigatoriamente, devem ser armazenados em dois recursos distintos na federação.

As modificações desse módulo e a política de armazenamento presente no ZooNimbus foi implementada pelo trabalho do Bacelar e Moura [31].

4.4.4 *Fault Tolerance Service*

Este serviço está presente na nova implementação de todos os demais serviços, o que favorece o seu desempenho e garante um maior sucesso na garantia da tolerância a falhas.

No *Storage Service*, a tolerância a falhas está presente quando o mesmo recebe o alerta sobre a indisponibilidade de um recurso e cria um *znode STATUSWAITING*, sinalizando essa indisponibilidade para os demais módulos, e realizando a rotina de recuperação do arquivos que continham nesse recurso. Como os arquivos são armazenados de forma duplicada nos servidores ZooNimbus, o módulo de armazenamento irá identificar qual o servidor que contém esses arquivos e realizar a duplicação dos mesmos em outro servidor ZooNimbus.

No *Scheduling Service* está presente quando ele também recebe o alerta informando sobre a indisponibilidade do recurso e inicia a recupeção de todos as tarefas que foram escalonados para este recurso, e ainda não foram executadas ou estavam em execução, de acordo com os metadados armazenado no Zookeeper, realocando-os para o *znode jobs*, onde irão aguardar um novo escalonamento.

No *Monitoring Service* a tolerância a falha ocorre quando recebe o alerta que informa sobre a indisponibilidade do recurso, e inicia o procedimento para verificar se os demais módulos já realizaram a recuperação de todos os dados armazenados no servidor Zookeeper. O *Monitoring Service* também é responsável pela verificação do não escalonamento de algum *job* ou não execução de alguma tarefa já escalonada, e pela verificação

da inclusão de algum arquivo nos recursos que o módulo de armazenamento não tenha reconhecido.

Toda essa recuperação é possível porque todos os recursos da federação conhecem os demais, e mantém um *watcher* nos seus respectivos *znodes STATUS*.

4.4.5 *Scheduling Service*

Este serviço é responsável pela execução de uma tarefa enviada para o ZooNimbus, garantindo que todas as etapas necessárias para o escalonamento e execução funcionem de forma correta. Ele acompanha uma tarefa, antes, durante e após a execução da mesma tarefa. A forma de realizar o serviço de escalonamento foi completamente refeita baseada nas novas tecnologias utilizadas, Zookeeper Apache [15] e Avro Apache [12]. A comunicação entre os recursos para verificar o estado das tarefas em execução ou para o escalonamento é agora implementada com a utilização do Zookeeper, deixando de ser direcionada *peer* a *peer* na rede. Os procedimentos são realizados quando é necessário informar o estado, a atualização e a inclusão das tarefas. Esses dados são enviados e recuperado diretamente no servidor Zookeeper.

Quando um cliente está conectado a um servidor ZooNimbus, ou seja, conectado a qualquer recurso que participe da federação em nuvem, e solicita a execução de uma tarefa, o servidor será o responsável por receber essa solicitação e lançá-la no *znode jobs* do servidor Zookeeper, juntamente com todas as informações referentes ao *job*, necessárias para realizar a sua execução. Ao realizar esse lançamento, o módulo de escalonamento, que está constantemente rodando e aguardando uma notificação, é alertado sobre o novo *job* a ser executado. O método responsável pelo tratamento de notificações do módulo de escalonamento irá recuperar todas as informações lançadas no Zookeeper e encaminhá-las para que a rotina de escalonamento seja realizada. Para que esse novo modelo de implementação do módulo de escalonamento possa ser bem utilizado pelas políticas de escalonamento, elas devem implementar o método visualizado na Figura 4.7, que retorna um mapa das tarefas enviadas para o escalonamento com o recurso eleito pelo escalonador para executar a tarefa. Para a chamada desse método, deve ser informado quais as tarefas para escalonar e também a conexão utilizada com o ZooKeeper.

```
Map<JobInfo, PluginInfo> schedule(Collection<JobInfo> jobInfos, ZooKeeperService zk);
```

Figura 4.7: Assinatura que Deve Ser Implementada ao Utilizar a Interface *SchedPolicy*.

Após a realização da rotina e da política de escalonamento, um recurso é eleito para executar o *job*, e essa informação é disponibilizada por meio do Zookeeper. Um *znode task_IdJob* é criado dentro do *znode task* na estrutura do recurso eleito para o escalonamento e o *znode job_IdJob*, que pertencia ao *znode jobs* é apagado. Essa criação de um novo *znode* gera o disparo do *watcher* adicionado no *znode tasks* pelo módulo de escalonamento que irá tratar esse alerta como um pedido de execução da tarefa. O método *event()*, mostrado na Tabela 4.2, irá realizar o tratamento do alerta recebido e encaminhará o pedido de execução da tarefa. Quando a execução é finalizada, o método que realizou sua execução irá modificar o estado da tarefa no ZooKeeper, de tarefa executada "Executando"para "Executada". Ao perceber a mudança, o ZooKeeper novamente irá informar

ao *Scheduling Service*, por meio de um *watcher*, do novo estado da tarefa e a rotina de conclusão da tarefa irá ser iniciada após esse alerta.

As tarefas de bioinformática executadas no ambiente da federação, necessitam de arquivos de entrada que contém os dados que serão processados pelos serviços de bioinformática, e geram um arquivo de saída. Esses arquivos devem existir em algum recurso da federação antes de ser solicitado a execução do serviço, caso esse arquivo não exista em nenhum servidor ZooNimbus, é informado ao cliente que solicitou a execução. O cliente deve então realizar o *upload* do arquivo, serviço provido de forma transparente pelo módulo de armazenamento. O módulo de armazenamento irá então executar uma política que seleciona para qual recurso da federação o arquivo de entrada será enviado. Considerando que haverá momento onde o módulo de armazenamento irá enviar o arquivo de entrada para um recurso diferente daquele que o módulo de escalonamento pode solicitar a execução do serviço, ambos os módulos devem realizar algum procedimento que minimize esse conflito, o qual pode aumentar o tempo de execução de uma tarefa.

O procedimento realizado para este cenário, onde é necessário a transferência de arquivos, consiste na utilização de um parâmetro comum que é comum aos módulos de serviços para eleger um recurso. Esse parâmetro é a latência, a qual é calculada a partir do local que contém o arquivo, para cada um dos demais recursos da federação. A latência é utilizada pela política de escalonamento ao calcular o poder computacional de um servidor ZooNimbus, podendo alterar o valor da probabilidade de sua escolha do recurso.

Desta forma, ao realizar o escalonamento e definir qual recurso irá executar a tarefa, o módulo de escalonamento realiza a verificação da existência do arquivo de entrada nesse recurso eleito, e caso o arquivo não exista, é realizada a solicitação de transferência do arquivo para aquele recurso.

Ao finalizar a execução da tarefa, o usuário poderá então realizar o *download* do arquivo de saída independente da localidade deste arquivo.

A política de armazenamento proposta para esse trabalho e implementada no ambiente ZooNimbus será descrita na próxima seção.

Capítulo 5

Algoritmo de Escalonamento - AcoSched

Este capítulo apresenta o algoritmo de escalonamento proposto neste trabalho, o chamado AcoSched. Inicialmente, faz-se uma descrição sobre o algoritmo base utilizado na criação do AcoSched. Em seguida, são apresentados a descrição das mudanças, as adaptações e as demais considerações feitas para que o algoritmo proposto funcione de maneira satisfatória no ambiente do ZooNimbus.

5.1 Algoritmo LBACO

Um bom escalonador deve adaptar sua estratégia de acordo com o ambiente e os tipos de tarefas escalonadas [24]. Esta adaptação está presente, de acordo com Li *et al.* [24], no algoritmo *Ant Colony Optimization* (ACO) [27], um algoritmo de escalonamento dinâmico. Pelas suas características e propostas, o ACO é classificado por Li *et al.* [24] como um algoritmo apropriado para uma ambiente de nuvem. Ele é um algoritmo de busca aleatória que imita o comportamento de uma colônia de formigas real em busca de alimentos utilizando feromônios[1] para traçar os caminhos.

Em uma colônia de formigas real, cada caminho traçado por uma formiga em busca de alimento é sinalizado com o seu feromônio, e ao encontrar um alimento a formiga retorna a colônia levando uma certa quantidade do alimento encontrado, e aumentando a intensidade do seu caminho percorrido ao liberar mais feromônio. O feromônio é depositado com uma intensidade proporcional a quantidade e a qualidade do alimento encontrado, criando um percurso para guiar as outras formigas até a fonte de alimento. Essa característica de uma colônia de formigas real é explorada artificialmente pelo algoritmo ACO [27]. Li *et al.* [24] apresentaram o algoritmo *Load Balancing Ant Colony Optimization* (LBACO), baseado na proposta inicial do algoritmo ACO de Dorigo *et al.* [27], para encontrar o melhor recurso disponível para escalonar uma tarefa em uma nuvem dinâmica.

O LBACO minimiza o *makespan*, tempo de execução total de uma tarefa, e também equilibra a carga interna do sistema. Para isso, quando uma formiga encontra uma trilha de feromônio, a probabilidade da formiga escolher essa trilha é proporcional a intensidade

[1] feromônio: substância biologicamente muito ativa, secretada por insetos e mamíferos, com funções de atração sexual, demarcação de trilhas ou comunicação entre indivíduos [2].

do feromônio da trilha. A cada nova formiga que decidir pela mesma trilha, a intensidade do feromônio irá aumentar pois, cada formiga irá depositar mais feromônio. Desta forma, um caminho ideal irá ser formado, encontrando um bom recurso para escalonar a tarefa em um ambiente de nuvem, que leva em consideração também o dinamismo da nuvem que pode alterar constantemente a quantidade e qualidade dos recursos, e modificar o melhor percurso, a cada nova iteração do algoritmo.

5.1.1 Funcionamento do LBACO

Para analisar melhor o funcionamento do LBACO, será feita uma breve descrição do funcionamento básico do ACO, apontando as fórmulas para sua implementação. Assim, o algoritmo ACO inicia seu funcionamento com posicionamento das formigas, no instante zero, em diferentes pontos com a intensidade inicial do feromônio com valor de τ entre os pontos i e j. Após o início da procura, o feromônio de cada recurso é dado pela Fórmula 5.1:

$$p_{ij}^{k}(t) = \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}(t)]^{\beta}}{\Sigma[\tau_{ik}(t)]^{\alpha}[\eta_{ik}(t)]^{\beta}} \tag{5.1}$$

Onde n é o valor da heurística definida por, $n = 1/d$, sendo d a distância dos pontos i até j. Os valores de α e β são parâmetros de controle, relativos ao feromônio e ao valor da heurística, e t é o tempo em que é calculado a probabilidade.

Após calculado a probabilidade de seleção daquele recurso, o valor do feromônio é globalmente atualizado. O algoritmo ACO funciona basicamento com um *loop* executando até encontrar o melhor caminho traçado pelas formigas, o qual é o caminho mais curto.

Como dito anteriormente, no LBACO o escalonamento de tarefas é baseado no funcionamento do ACO e para uma nova tarefa escalonada, o resultado do escalonamento anterior é considerado. Assim, a carga da tarefa é considerada em um novo escalonamento para que seja realizado o balanceamento de carga. O feromônio inicial de cada recurso ou Máquina Virtual (VM) é dado pela Fórmula 5.2 no instante t = 0.

$$\tau_{ij}(0) = num\_processadores * frequencia\_processador + largura\_banda\_VM \tag{5.2}$$

O valor do número de processadores é utilizado de forma dinâmica e com uma aproximação do uso real e atual da CPU de uma VM, ou seja, quando é obtido o número de processador para aplicar na Fórmula 5.1, é obtido o valor do número de processadores disponíveis em cada recurso para aquele intervalo de tempo em que é realizado o cálculo da capacidade computacional. Isso reduz a sua capacidade computacional, e aumenta o balanceamento de carga de trabalho, pois, consequentemente, sua probabilidade de escolha dada pela Fórmula 5.2, será proporcional ao uso de seus processadores.

A frequência do processador é de Milhões de Instruções Por Segundo (MIPS). A probabilidade de escolha da formiga definida pelo ACO na Fórmula 5.1, é adaptado para o LBACO e dada pela Fórmula 5.3:

$$p_{ij}^{k}(t) = \frac{[\tau_{j}(t)]^{\alpha}[EV_{j}]^{\beta}[LB_{j}]^{\gamma}}{\Sigma[\tau_{k}(t)]^{\alpha}[EV_{k}]^{\beta}[LB_{k}]^{\gamma}} \tag{5.3}$$

Onde EV_j representa a capacidade computacional na VMj dada pela Fórmula 5.4:

$$EV_j = num\_processadores_j * frequencia\_processador_j + largura\_banda\_VM_j \quad (5.4)$$

E também onde LB_j representa o balanceamento de carga da VMj, dado pela Fórmula 5.5, e para um tempo esperado medido pela Fórmula 5.6 :

$$LB_j = 1 - \frac{tempo\_esperado_j - ultimo\_tempo_j}{tempo\_esperado_j + ultimo\_tempo_j} \quad (5.5)$$

$$tempo\_esperado_j = \frac{tamanho\_total\_tarefa}{EV_j} + \frac{tamanho\_arquivo\_entrada}{EV_j} \quad (5.6)$$

Os parâmetros α , β e γ são para controle do peso relativo da trilha do feromônio, da capacidade computacional e do balanceamento de carga das VMs, respectivamente.

Para que o escalonamento dinâmico funcione, com a atualização do feromônio marcando o melhor recurso, é necessário que sua intensidade diminua quando estiver com muita carga, deixando de ser o melhor recurso. Para isso, é utilizado uma equação de atualização do feromônio, dada pela Fórmula 5.7, na qual $\Delta\tau_j = 1/T_{ik}$ quando uma formiga completa suas visitas em todas os recursos, ao final de uma iteração, o feromônio local é atualizado.

$$\tau_j(t+1) = (1-p) * \tau_j(t) + \Delta\tau_j \quad (5.7)$$

Com T_{ik} sendo a menor probabilidade calculada pela formiga na iteração i. Na Fórmula 5.7, p é o coeficiente de decaimento do feromônio.

Quando a formiga completa suas visitas e encontra a melhor solução é utilizado $\Delta\tau_j = D/T_{op}$, onde T_{op} é a melhor solução atual, e D é o coeficiente de encorajamento.

Assim sendo, o algoritmo inicia colocando as formigas aleatóriamente em algumas VMs para que elas possam definir a probabilidade de escolha daquela VM e, então, visitar todas as demais VMs. Após as visitas serem completadas, elas atualizam o valor do feromônio, e se a melhor solução não for encontrada, realizam novamente suas visitas. As iterações das visitas acabam quando a melhor solução é encontrada ou quando o número de iterações máximas é atingido.

Nesse cenário, o escalonamento é realizado objetivando minimizar o tempo de execução das tarefas e manter o balanceamento de carga em todas as VMs. Para controlar o balanceamento de carga é utilizada a Fórmula 5.8:

$$T_i = \frac{tamanho\_total\_tarefa_j}{num\_processadores_j * frequencia\_processador_j} \quad (5.8)$$

Onde $tamanho\_total\_tarefa$ é o tamanho total das tarefas submetidas para a VMj, e para determinar a carga de trabalho das VMs é utilizado o DI (*Degree Imbabalance*), grau de desbalanceamento, dado pela Fórmula 5.9:

$$DI = \frac{T_{max} - T_{min}}{T_{avg}} \quad (5.9)$$

O T_{max} e T_{min} são, respectivamente, os máximos e mínimos de todas as VMs, e o T_{avg} é o tempo médio.

5.1.2 Resultados da Implementação do LBACO

Os resultados dos testes descritos nesta Seção foram obtidos com a implementação do LBACO por Li *et al.* [24] na plataforma *CloudSim* [6], que é uma ferramenta que permite modelagem, simulação e experimento em cima de uma infraestrutura de nuvem.

Os parâmetros de configuração dos valores utilizados pelas fórmulas do LBACO foram apresentados pela Tabela 5.1.

Tabela 5.1: Parâmetros de Configuração do LBACO [24].

Parâmetros	**Valores**
Números de tarefas	100-500
Números de formigas na colônia	8
Número de iterações	50
p	0.01
α	3
β	2
γ	8

Os testes foram realizados comparando as execuções entre os resultados dos algoritmos LBACO, ACO e o *First Come First Served* (FCFS) [33]. O algoritmo FCFS tem como objetivo encontrar o tempo de execução de cada tarefa, e o ACO tem como objetivo minimizar o *makespan* de um conjunto de tarefas qualquer. Por outro lado, o LBACO tem o objetivo de minimizar o *makespan*, e ainda realizar o balanceamento de carga.

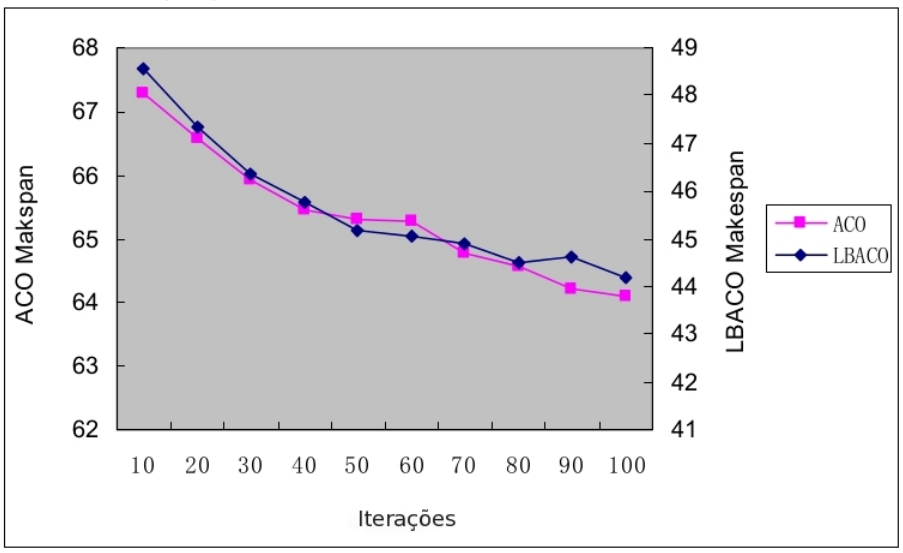

Figura 5.1: Comparação do *Makespan* dos Algoritmos LBACO e ACO [24].

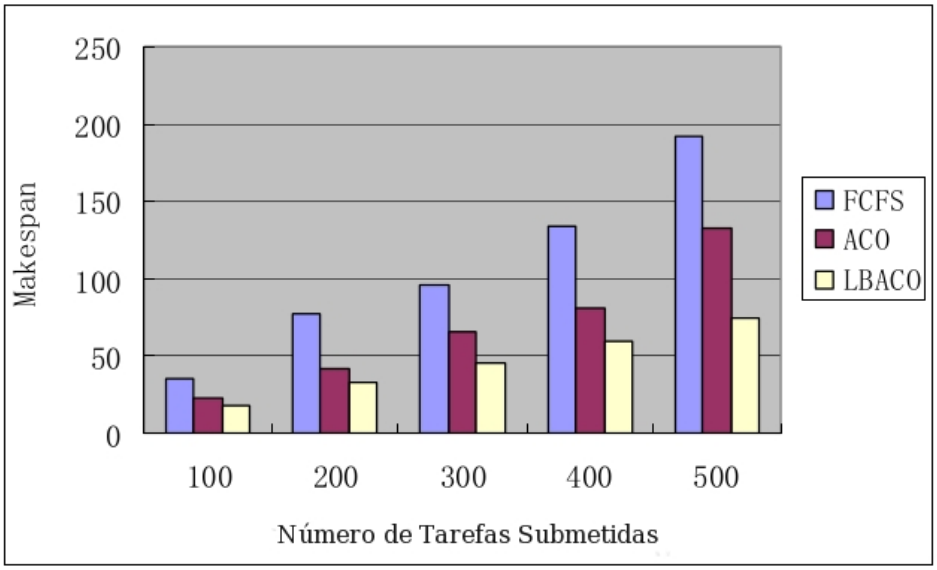

Figura 5.2: Comparação do *Makespan* dos três Algoritmos [24].

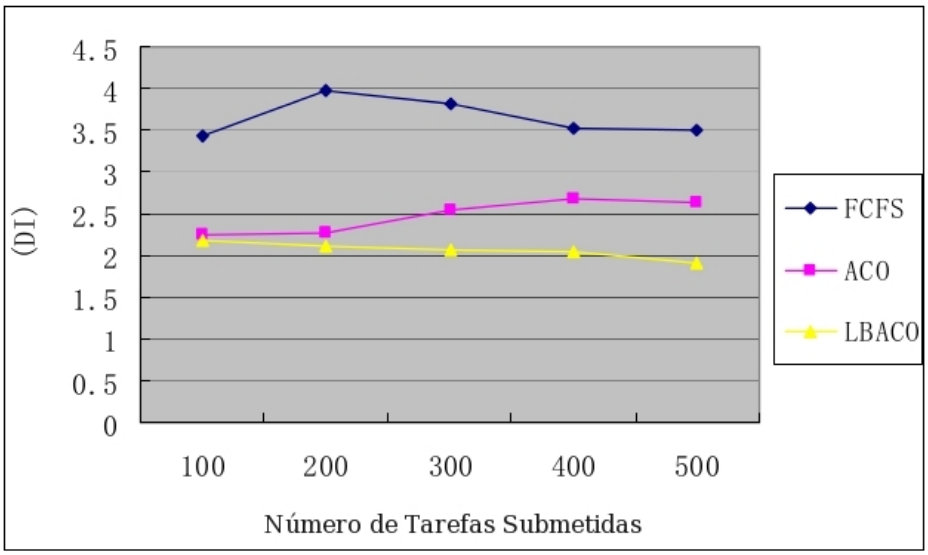

Figura 5.3: Média do DI de cada Algoritmo [24].

A Figura 5.1 mostra a comparação de desempenho dos algoritmos LBACO e ACO para a execução de 300 tarefas, utilizando o tempo disponibilizado pelo CloudSim em milissegundo para cálculo do *makespan*. Na Figura 5.2 tem-se a execução de 100-500 tarefas e a comparação do *makespan* de cada algoritmo. E, por último, tem-se os resultados do DI médio de cada algoritmo apresentados na Figura 5.3.

Como foram obtidos tempos mais elevados de *makespan* no algoritmo LBACO em relação aos tempos obtidos no algoritmo ACO após as 50 iterações (veja Figura 5.1) foi utilizado para os demais testes, o número máximo de 50 iterações. É possível observar que o algoritmo LBACO obteve melhores resultados do que os demais, tanto na minimização do *makespan* como também no balanceamento de carga, o que levou a um melhor tempo

de execução das tarefas.

5.2 Evolução do LBACO

A utilização do LBACO em um ambiente de nuvem computacional obteve bons resultados como foi visto na seção anterior. Sua adaptação para um ambiente de nuvem federada irá acrescentar outros fatores que devem ser analisados ao realizar o escalonamento.

Li *et al.* [24] indicam para trabalhos futuros a utilização de um possível vetor contendo informações sobre o requisito da tarefa que será escalonada. Essa característica de utilização de um vetor que contenha as informações do algoritmo está presente no trabalho proposto e deve ser considerada ao realizar um escalonamento.

O LBACO deve sofrer modificações para que seja adaptado a um ambiente de nuvem federada e não mais apenas para um provedor de nuvens. Tais modificações devem considerar o tipo de tarefa escalonada para que a saída indique o recurso capaz de executar a tarefa em menor tempo possível.

Vários testes foram necessários para adaptar o algoritmo em questão, o LBACO, em uma nuvem federada, implementada pela arquitetura ZooNimbus. Após as modificações de requisitos inicias, a política de escalonamento foi testada para que os parâmetros de configuração necessários para sua execução fossem alterados, manipulando o tempo de resposta do escalonador a fim de obter o melhor tempo.

Uma das alterações na implementação e adaptação do LBACO, por exemplo, foi na distribuição das formigas que não aconteceria de forma totalmente aleatória dentre todas os recursos da federação em nuvem, sendo apenas realizada entre aqueles recursos que são capazes de rodar a tarefa. As demais mudanças realizadas no LBACO para o ambiente do ZooNimbus são descritas na Seção 5.4.

5.3 Algoritmo AcoSched

A política de escalonamento baseada no LBACO e implementada no ambiente ZooNimbus foi intitulada de AcoSched. Para tal implementação foi necessário realizar algumas adaptações na proposta inicial do LBACO, por exemplo, a substituição do valor da largura de banda no cálculo da potência computacional, pelo valor da latência calculada entre os recursos. Ambos os valores são calculados entre o servidor ZooNimbus que contém o arquivo de entrada necessário para a execução da tarefa, e os demais servidores que pertencem a federação em nuvem.

A substituição da largura de banda pela latência se deve ao fato da política de armazenamento de arquivos do ZooNimbus utilizar a latência como um dos fatores de eleição dos recursos para armazenamento dos arquivos, e também pelo fato de que os arquivos de entrada utilizados para realizar a execução de uma tarefa tem tamanhos físicos menores do que o arquivo de saída, o que de forma geral faz com que a largura de banda tenha uma importância menor na capacidade computacional de processamento. Considerar a latência ao invés da largura de banda pelo fato da política de armazenamento utilizar esse parâmetro de comparação é uma das características principais desta nova versão do BioNimbus, pois na sua versão inicial não havia uma política de armazenamento, e o

algoritmo de escalonamento então implementado não utilizava como parâmetro de eleição a localização dos arquivos de entrada, o que levava a uma replicação indiscriminada dos arquivos de entrada pelos recursos e, consequentemente, uma maior demora de execução das tarefas submetidas na nuvem federada.

A escolha de um recurso para execução de uma tarefa feita pela política de escalonamento, pode ser diretamente alterada ao considerar a localidade do arquivo que será utilizado na execução da tarefa, podendo otimizar o tempo de execução de uma tarefa, já que pode ser escolhido um recurso que contenha esse arquivo, mesmo que esse recurso não seja, computacionalmente, o melhor recurso. Portanto, essa deve ser uma importante consideração que deve ser realizada por um algoritmo de escalonamento, e o AcoSched faz essa consideração ao utilizar o valor da latência entre os recursos que contenham o arquivo necessário e os demais recursos. Quanto menor a latência, maior será a probabilidade de escolha de um recurso, e como essa latência é calculada entre o recurso que possue o arquivo para os demais, o recurso que possuir o arquivo será aquele que possuirá a menor latência, e, portanto, o recurso com maior probabilidade de escolha ao considerar esse parâmetro.

Apesar dessa consideração, o resultado do escalonamento não resulta, necessariamente, na eleição do recurso que contenha o arquivo, mas sim na maior probabilidade de escolha desse recurso ou na maior probabilidade de escolha de outro recurso que contenha uma menor latência para envio do arquivo.

Assim, a nova forma de cálculo da capacidade computacional que substituiu a Fórmula 5.2 da Seção 5.1.1, é então descrita pela Fórmula 5.10:

$$\tau_{ij}(0) = num\_processadores * frequencia\_processador - latencia\_VM \tag{5.10}$$

Como pode ser notado, o valor da latência é subtraído da multiplicação do número de processadores, pela frequência do processador. Nesse caso, é realizada a subtração da latência pois a sua relação com a capacidade computacional é inversa, ou seja, quanto maior a latência menor a capacidade computacional, já que quanto maior a latência menor a velocidade na transferência do arquivo na rede. Ao contrário da largura de banda que seria somada ao valor da capacidade computacional, pois quanto maior a largura de banda mais rápido será a transferência.

A fórmula do cálculo da probabilidade de escolha de uma VM, Fórmula 5.3, foi modificada e adicionou mais uma parâmetro de escolha relativo à quantidade de memória principal disponível no recurso, fator que deve ser considerado no ambiente de bioinformática que requer um alto processamento e, consequentemente, uma maior quantidade de memória primária. O cálculo da probabilidade ficou então definido pela Fórmula 5.11:

$$p_{ij}^{k}(t) = \frac{[\tau_j(t)]^{\alpha}[EV_j]^{\beta}[LB_j]^{\gamma}[MR_j]^{\delta}}{\Sigma[\tau_k(t)]^{\alpha}[EV_k]^{\beta}[LB_k]^{\gamma}[MR_k]^{\delta}} \tag{5.11}$$

Onde MR_j se refere à capacidade de memória, dada pelo tamanho do espaço livre em memória, e γ é o seu coeficiente de controle para a VM_j.

A probabilidade de escolha da próxima VM a ser visitada não foi o único parâmetro utilizado para realizar o escalonamento das tarefas. Ao executar a política de escalonamento, é realizado um filtro dos recursos disponíveis para verificar qual deles contém os

serviços e qual deles é público ou privado de acordo com a escolha do cliente que solicitou a execução da tarefa. Para uma escolha de um ambiente de nuvem federada híbrida, não é realizado o filtro de verificação do tipo do recurso.

Os dados utilizados para a execução das fórmulas usadas na política são armazenados e recuperados a partir dos metadados do servidor Zookeeper. Elas são armazenadas na forma de estrutura de dados de vetor, contendo dados da última probabilidade de escolha, feromônio, valores dos coeficientes de controle, tamanho do último *job* executado, tempo da última execução e tamanho total das tarefas executadas no recurso.

Após recuperação dos dados as formigas são aleatóriamente distribuídas nos servidores ZooNimbus, ou seja, é realizada a seleção de alguns recursos em que as formigas irão calcular sua probabilidade e o seu feromônio.

As formigas utilizadas pelo AcoSched são representadas computacionalmente por números aleatórios que variam de acordo com a quantidade de recursos, e identificam quais são os recursos que devem ser selecionados para realizarem os cálculos dos valores utilizados no algoritmo. O seu feromônio corresponde a um dos valores calculados e variam, numericamente, de acordo com a capacidade computacional, balanceamento de carga, latência, memória primária, etc. Desta forma, esses feromônios são referentes a qualidade dos recursos visitado, e varia a cada nova iteração das formigas com os recursos.

O AcoSched considera uma lista de prioridades e fatores para realizar o escalonamento, proporcionando ao usuário o melhor recurso para execução de uma tarefa. Ao prover o melhor recurso, é utilizado alguns tipos de serviços que são disponibilizados por nuvens computacionais, como o serviço de *Iaas* ao selecionar um recurso para execução de uma tarefa, e o serviço *Saas* ao selecionar um recurso que contém a ferramenta necessária para execução da tarefa.

As prioridades consideradas pelo algoritmo são implementadas na ordem: tipo da nuvem solicitada, disponibilidade do tipo do serviço solicitado, localização dos arquivos de entrada, capacidade computacional do recurso, quantidade de memória primária disponível, tamanho dos arquivos de entrada que devem ser utilizados na execução das tarefas, selecionando sempre a tarefa com maior arquivo de entrada, e além do tamanho do arquivo de entrada é selecionado para a execução aquela tarefa que foi enviada a mais tempo, permitindo uma execução justa para todas as tarefas. Para garantir a ordem de algumas prioridades é utilizado a alteração dos valores dos coeficientes de controle, os quais são α, β, γ e δ.

Para garantir a execução de todas as tarefas enviadas para a nuvem federada, e em um menor tempo de execução, os módulos de escalonamento e de monitoramento armazenam os metadados referentes às tarefas no Zookeeper. Esses metadados são utilizados para o monitoramento do balanceamento da nuvem, para a verificação de tarefas pendentes, para encaminhamento ao escalonamento e para a execução, além dos demais mecanismos que coordenam e organizam as execuções de tarefas. Essa integração e sincronização dos metadados com os servidores ZooNimbus garantem ao usuário uma maior consistência e transparência do sistema distribuído.

A transparência do ZooNimbus pode ser verificada, por exemplo, quando um usuário faz a solicitação de execução de uma tarefa no ambiente após verificar a existência do arquivo de entrada necessário na federação. A tarefa é enviada para escalonamento e é executada independente da localização física do arquivo de entrada, ou seja, mesmo que um recurso que não possua o arquivo de entrada necessário para a execução da tarefa seja

eleito para executar a tarefa pela política de escalonamento, o ZooNimbus irá garantir que essa decisão seja respeitada e o recurso eleito execute a tarefa. Para garantir tal execução, o recurso eleito faz a solicitação de transferência daquele arquivo utilizando-se dos metadados armazenados no Zookeeper que identificam a localidade do arquivo. O fim da transferência é aguardado para que a tarefa seja então executada. Todos os procedimentos realizados para atender a solicitação do usuário são realizados, sem que se identifique tais mecanismos.

5.4 O AcoSched no ZooNimbus

O AcoSched foi adaptado e implementado de forma fortemente integrada ao ZooNimbus. De maneira que, para completar uma rotina de execução com a utilização do AcoSched, o serviço de escalonamento tem que realizar todos os procedimentos necessários para que seja obtido um bom escalonamento de uma tarefa.

A rotina de escalonamento de uma tarefa pode ser melhor compreendida por meio da Figura 5.4. Nessa figura, nota-se que o usuário mantém uma conexão com o ambiente ZooNimbus através da conexão com algum recurso participante da federação. As requisições de tarefas são registradas no servidor Zookeeper e o mesmo é encarregado de enviar o alerta para todos os recursos que irão receber a requisição de escalonamento, e retornar para o servidor ZooNimbus a escolha feita pelo escalonamento. O ZooNimbus irá então informar ao recurso eleito sobre a tarefa a ser executada, que irá executar e retornar um aviso de confirmação de execução após a sua conclusão.

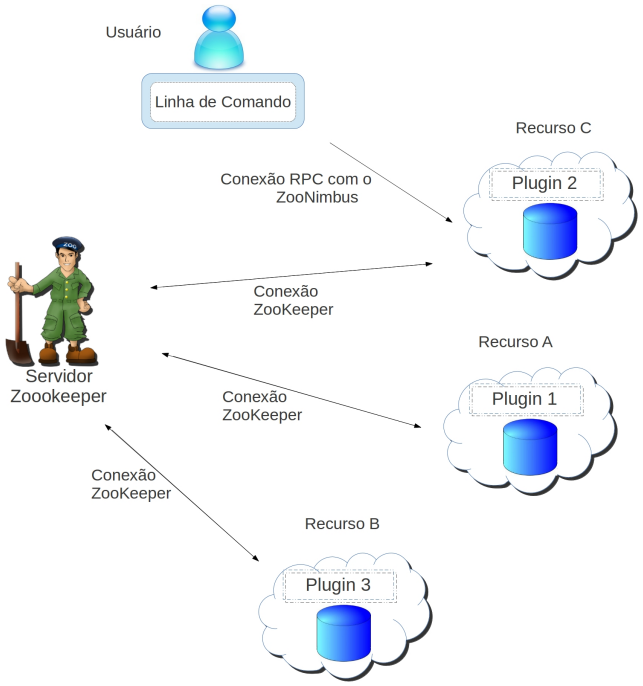

Figura 5.4: Conexões entre Usuário e ZooNimbus para Executar uma Tarefa.

Para realizar o escalonamento de uma tarefa no ZooNimbus, o AcoSched considera algumas características das tarefas, dos recursos, dos valores das latências entre os recur-

sos, dos tipos de nuvens que participam da federação, os serviços disponibilizados pelos recursos, e outras informações que serão abordadas ao longo desta seção.

Assim, diferentes cenários para a execução de uma tarefa podem acontecer quando um *job* é submetido para execução. A rotina geral dessas tarefas, do momento de submissão até o momento que é gerado o arquivo de saída e o processo é concluído, será descrita a seguir, e suas possíveis variações serão apresentas ao longo de sua descrição.

Quando uma tarefa é submetida pelo usuário, seja por um *workflow* ou envio da tarefa pela linha de comando do ZooNimbus, as informações necessárias que devem conter em uma classe do tipo *JobInfo* são coletadas e submetidas ao servidor Zookeeper. Essas informações serão armazenadas no *znode job_IdJob*, criado para representar o *job* no Zookeeper. A identificação do *job* é gerada de forma randômica e será mantida durante todo o tempo de existência daquele *job* no ZooNimbus. Quando o *znode* do *job* é criado como filho do *znode jobs*, ele representa uma tarefa a ser escalonada e sua criação gera um alerta para todos os serviços que adicionaram um observador no *znode jobs*. Se no momento da submissão do *job* ao ZooNimbus existir N recursos na federação, todos eles receberam o alerta do novo *job* e irão tratar essa nova demanda do sistema. Para evitar que mais de um recurso realize o escalonamento de um mesmo *job*, foi necessário criar um semáforo para esse *job*. O semáforo foi criado na forma de um *znode* efêmero, filho do *znode job_IdJob*, chamado de *LOCK*. Assim, o serviço de escalonamento, ao receber um alerta da existência de um *job*, verifica se existe o *znode LOCK*, e se não existir ele cria esse *znode*, que representa o uso do *job* por um recurso, e continua a rotina para escalonar esse *job*. Após essa verificação, apenas um recurso está utilizando o *job* e realizando o seu escalonamento, e esse recurso antes de enviar a tarefa para que a política AcoSched realize sua rotina, irá solicitar o cálculo da latência entre o recurso que contém os arquivos de entrada necessários para executar essa tarefa e as demais.

Para controlar o envido da solicitação de cálculo da latência, pois pode ocorrer mais de uma solicitação de diferentes servidores ZooNimbus, foi criado um mecanismo de permissão e informação aos recursos para que cada latência fosse calculada de forma única para cada arquivo, e pertencendo somente ao recurso que possui o arquivo desejado. Quando uma tarefa é lida no *znode jobs* para ser escalonada, o *Scheduling Service* irá solicitar o cálculo da latência antes de iniciar a rotina da política de escalonamento. É realizada uma solicitação via RPC para o recurso que contenha o arquivo, e este irá calcular a latência para cada recurso pertencente a federação.

Iniciado o AcoSched, é solicitada a lista dos recursos existentes na federação para que seja filtrado os recursos que cumprem as solicitações do usuário. Primeiramente, são filtrados aqueles que pertencem ao tipo de nuvem que o usuário solicitou, pública, híbrida ou privada. Em seguida, é verificado qual desses recursos restantes possuem o serviço solicitado pelo usuário. Com a lista de possíveis recursos que irão executar o *job*, é recuperado os metadados armazenados nos *znodes scheduling* dos recursos com as informações utilizadas pelo AcoSched. Se não exisitr esses dados, significa que é a primeira execução do AcoSched nesse servidor ZooNimbus e, portanto, não existem dados gerados pela política. Essas informações podem significar uma maior probabilidade de escolha do recurso pois contêm o valor do feromônio da execução anterior.

Ao realizar um novo escalonamento, será utilizado os dados das execuções anteriores e a lista dos possíveis recursos disponíveis. As formigas irão novamente iniciar suas visitas para que o feromônio anterior seja atualizado pela Fórmula 5.7 e os demais cálculos

realizados para que sejam geradas novas probabilidades de escolha dos recursos. No AcoSched, diferentemente do conceito praticado por uma colônia de formiga real, que marcam com o feromônio o caminho até o recurso encontrado, a marcação com o feromônio é feita sobre o recurso.

A probabilidade de escolha de cada recurso calculada, baseada na Fórmula 5.3, será utilizada como indicador do recurso que a tarefa deve ser executada após todas as visitas serem finalizadas. Os dados então utilizados para o cálculo do melhor recurso e seus valores de saída serão enviados para armazenamento no *znode sched*, da estrutura do Zookeeper, referente a cada recurso. O algoritmo de escalonamento irá concluir sua rotina de eleição de um recurso e retornar as informações para o módulo *Scheduling Service*. O módulo *Scheduling Service*, por sua vez, irá encaminhar a execução para o recurso eleito e aguardar uma nova solicitação informando a finalização da execução da tarefa. Assim, quando a tarefa é finaliza a política de escalonamento ainda será requisitada para que finalize sua rotina de escalonamento com os dados resultantes da execução, registrando esses dados no servidor Zookeeper para que eles possam ser utilizados como metadados para novos escalonamento, como também para medir o balanceamento de carga do sistema.

Assim, quando uma tarefa é escalonada e está esperando para ser enviada para a execução, ela mantém seu estado como "Pendente". Após o envio para execução, caso a tarefa necessite de algum arquivo que precisa ser transferido para o local onde será executado a tarefa, ela irá receber o estado de "Espera", que sinaliza seu posicionamento no módulo de escalonamento. Mas, se a tarefa pode ser executada ou recebe o arquivo solicitado para execução, é iniciada sua execução e seu estado passará para tarefa em "Execução". Finalizado a execução, seu estado passará para tarefa "Finalizada", podendo ser concluída sua rotina no *Scheduling Service*. Todos os estados das tarefas são atualizados no servidor Zookeeper e na aplicação, auxiliando a execução da tarefa, o acompanhamento e sua possível recuperação, caso seja necessário.

O *Monitoring Service* auxilia também na execução das tarefas, garantindo que as tarefas enviadas para execução sejam executadas e acompanhadas de forma síncrona com o estado de cada uma delas.

Dessa forma, após o usuário submeter uma tarefa na federação, ele irá receber como retorno o número de identificação da tarefa para que sua execução possa ser acompanhada no ZooNimbus, caso a tarefa seja enviada para execução.

5.5 Vantagens do AcoSched

O AcoSched tem algumas características que maximizam os resultados de seu escalonamento. Ele é baseado na heurística de formiga que utiliza probabilidade para realizar a distribuição de tarefas e, consequentemente, o balanceamento de carga. Além disso, ele utiliza dados computacionais mais próximos da realidade.

Além disso, o algoritmo também busca considerar a localidade do arquivo como um fator que deve ser avaliado ao escalonar uma tarefa, pois uma possível transferência de arquivos entre os recursos pode levar a um aumento no tempo total de execução de uma tarefa.

Para otimizar ainda mais o tempo total de execução das tarefas, o AcoSched prioriza escalonar, primeiramente, tarefas que possuem arquivos de entrada de tamanhos maiores dentro de um conjunto de tarefas a serem executadas, visando escalonar para os melhores

recursos as tarefas que irão utilizar maior poder de processamento, assim como o DynamicAHP. Contudo, o AcoSched analisa também o tempo máximo permitido para uma tarefa aguardar o escalonamento, impedindo assim sua demora para início da execução, ou seja, ao enviar um conjunto de tarefas de tamanhos variáveis para execução, o AcoSched irá priorizar aquelas tarefas de tamanhos maiores sem que o escalonamento das tarefas de tamanhos menores seja sempre adiado, pois novas tarefas de tamanhos maiores podem ser constantemente enviadas para execução. Tal consideração irá fazer com que um grupo de tarefas possa ser executado em menor tempo e de forma justa, já que as tarefas que levam um maior tempo de execução serão executadas mais rapidamente nos recursos com melhor poder computacional, e as demais também serão enviadas para execução sem que sejam deixadas sempre para o final. Por outro lado, tarefas que não necessitam de grande capacidade de processamento serão executadas em outros recursos de menor capacidade.

Além disso, ao realizar um escalonamento, a política AcoSched considera as execuções anteriores de cada recurso, utilizando-se desses dados para calcular o tempo aproximado que irá levar para executar uma tarefa nesses recursos. Essa análise é mais um parâmetro que favorece um resultado mais preciso do escalonamento, ou seja, a escolha do melhor recurso disponível.

O AcoSched também possue algumas desvantagens para realizar o escalonamento. Uma das suas desvantagens é o fato de utilizar muitas iterações no seu algoritmo ao escalonar uma tarefa devida a quantidade de formigas que buscam o melhor recurso, e a quantidade de vezes que essas formigas analisam um recurso computacional,l podendo levar a um alto tempo de escalonamento.

O capítulo seguinte irá apresentar os resultados obtidos nos testes realizados com a execução do escalonamento feito pelo algoritmo AcoSched em comparação com os resultados obtidos pelo algoritmo presente na versão inicial do BioNimbus, o DynamicAHP. Ambos os algoritmos irão ser executados no mesmo ambiente, a plataforma de federação de nuvem ZooNimbus, executando o mesmo conjunto de tarefas.

Capítulo 6

Resultados do AcoSched no ZooNimbus

Este capítulo descreve os mecanismos e os resultados obtidos com a utilização da política de escalonamento proposta e implementada no ambiente ZooNimbus. Inicialmente, é realizada a descrição dos ambientes que foram executados os testes. Em seguida, são apresentadas as tarefas executadas no ambiente e os parâmetros de configuração necessários para a utilização da política de escalonamento AcoSched. Ao final deste capítulo são descritos os resultados obtidos com a implementação do AcoSched, comparando-o ao algoritmo implementado na versão anterior, o DynamicAHP.

6.1 Ambiente de Execução

Para verificar se o tempo de resposta e o tempo total de execução das tarefas no ZooNimbus utilizando-se do algoritmo de escalonamento proposto, o AcoSched, foi reduzido e obteve melhor desempenho no ambiente federado, foi realizada uma comparação do desempenho por meio da análise dos tempos totais de execução obtidos com o algoritmo DynamicAHP [25], que foi implementado na primeira versão da plataforma de federação em nuvem. Ambos os algoritmos foram executados na atual versão da plataforma, o ZooNimbus, utilizando-se dos mesmo recursos físicos que formaram a federação em nuvem.

Para formar uma nuvem federada, executando os servidores ZooNimbus, foram utilizados três nuvens.

A primeira nuvem, uma nuvem do tipo privada, formada na rede da Universidade de Brasília (UnB), era composta por 3 computadores com a configuração de 8 GB de memória primária, processador Intel(R) Core(TM) i7-3770 com frequência de 1.6 GHz com oito núcleos e 2 TB de memória secundária.

A segunda nuvem, a nuvem pública da AmazonEC2 [26], era composta por quatro computadores, sendo três deles com a configuração de 15 GB de memória primária, processador Intel(R) Xeon(R) CPU E5-2650 com frequência de 2.0GHz com 4 núcleos, e 8 GB de memória secundária. O outro computador pertencente a esta nuvem pública tinha a configuração de 8 GB memória primária, processador Intel(R) Xeon(R) CPU E5645 com frequência de 2.4GHz com 2 núcleos, e 8 GB de memória secundária.

A terceira nuvem, também pública, provida pelo serviço da Azure [28], era composta por quatro computadores. Nesta nuvem, infelizmente, os testes realizados com sua participação na federação não foram suficientes para obter bons resultados na execução dos servidores ZooNimbus ao utilizar ambos os algoritmos de escalonamentos presentes no

ZooNimbus, o DynamicAHP e o AcoSched. Tal fato ocorreu porque os algoritmos utilizam os valores de latência entre os servidores ZooNimbus para realizar os cálculos do escalonamento e não foi possível realizar a medida dessas latências entre os recursos desta nuvem e os recursos das demais nuvens porque a Azure [28], provedora do serviço de nuvem, faz o bloqueio à utilização de protocolos de rede do tipo ICMP, utilizado para medir a latência através do comando *ping*. Alguns programas foram testados para medir a latência e solucionar tal inviabilidade, porém o tempo de resposta de tais ferramentas foram mais elevados do que o previsto, o que levou a uma excessiva demora para realizar o escolamento, e isso inviabilizou a sua utilização. Porém, neste mesmo ambiante de nuvem, foi possível executar os servidores ZooNimbus em uma rede interna, com os endereços de rede interna, onde o protocolo ICMP não é bloqueado. Essa execução na rede interna possibilitou a regulagem dos valores do parâmetro de configuração do ZooNimbus, descritos na Seção 6.1.2.

O servidor Zookeeper utilizado para a execução das tarefas, inicialmente, estava sendo executado em uma das máquinas que processava também o servidor ZooNimbus. Assim, observou-se que esta máquina recebia os alertas do Zookeeper mais rapidamente do que as demais e também que, ao executar o ambiente ZooNimbus, utilizando-se das nuvens para formar uma federação, foram obtidos resultados com tempos elevados de execução dos próprios serivoderes localizados na nuvem diferente daquela que continha o servidor Zookeeper, além do fato de algumas conexões com o servidor não ficarem estáveis em momentos diversos. Para solucionar essa perda de desempenho devido a localidade do servidor Zookeeper, foi utilizado um *cluster* de servidores Zookeeper formado por máquinas localizadas em ambas as nuvens. O *cluster* foi formado então por um servidor localizado em uma das máquinas pertencentes a nuvem da UnB, e os outros dois servidores localizados em recursos pertencentes a nuvem da Amazon [26].

6.1.1 Ferramentas e Dados de Execução

Para realizar as verificações do tempo de execução total das tarefas enviadas para o ZooNimbus, foi criado um conjunto de tarefas. As tarefas executadas para os testes utilizaram a ferramenta Bowtie [5], uma ferramenta de bioinformática, disponível como serviço nos servidores ZooNimbus.

Nesse cenário, foi criado um grupo de tarefas composto por 10 tarefas. Tais tarefas tinham arquivos de entrada de diferentes tamanhos, variando de 178 até 252 MB cada, disponível em ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/seq/, banco de dados do NCBI [29] . Os arquivos de entrada, que podem ser visualizadas na Tabela 6.1 com os seus respectivos tamanhos, estavam presentes nos servidores ZooNimbus.

As tarefas foram enviadas para execução sequencial, e a medida do tempo de execução total foi feita em milissegundos, onde seu tempo foi calculado entre o envio da primeira tarefa para a execução e o tempo de finalização da última tarefa. Foi enviado para execução o conjunto de tarefas que continham os arquivos de entradas descritos na Tabela 6.1. Para cada arquivo de entrada, havia um par de tarefas, totalizando 10 tarefas para o conjunto de tarefas.

O grupo de tarefas criado foi enviado para execução dez vezes, resultando num total de 100 tarefas escalonadas para cada algoritmo avaliado, o DynamicAHP e o AcoSched.

Tabela 6.1: Nome e Tamanho dos Arquivos de Entrada das Tarefas.

Nome	**Tamanho**
hs_alt_HuRef_chr1.fa	252 MB
hs_alt_HuRef_chr2.fa	247 MB
hs_alt_HuRef_chr3.fa	198 MB
hs_alt_HuRef_chr4.fa	189 MB
hs_alt_HuRef_chr5.fa	178 MB

Apesar destas tarefas terem sido enviadas sequencialmente, o grupo de tarefa era subdividido em pares, onde ao enviar um tarefa para execução, o seu par somente era enviado após a confirmação de sua execução. Tal confirmação era feita através da verificação da existência do arquivo de saída da tarefa no ambiente ZooNimbus.

6.1.2 Parâmetros AcoSched

Para que o AcoSched obtivesse bons resultados, foram necessárias várias modificações nos seus parâmetros de controle ao longo dos testes realizados. Ao modificar os parâmetros, a escolha do escalonador era também modificada. Para que tais mudanças nos parâmetros fossem positivas, gerando menor tempo de execução total das tarefas, e com a finalidade de obedecer as prioridades definidas, ao realizar o escalonamento, reguladas por meio dos coeficientes de controle, os parâmetros ficaram definidos nos valores mostrados na Tabela 6.2.

Tabela 6.2: Parâmetros de Configuração Definidos para o AcoSched.

Parâmetros	**Valores**
Números de tarefas	10
Números de formigas na colônia	2
Número de iterações	10
p	0.08
α	2
β	4
γ	2

Inicialmente, porém, os parâmetros foram fixados nos valores descritos na Tabela 6.3. Tais valores levaram a um pequeno desbalanceamento entre os servidores ZooNimbus, e um crescente valor do resultado obtido para o cálculo da probabilidade de escolha de cada servidor e no valor do feromônio. Foi analisado que o elevado valor da probabilidade era devido ao coeficiente de controle do feromônio, valor de α, e o valor do coeficiente de controle da memória primária, o valor de γ. O número de iterações das formigas também levaram a uma regulagem da saída do escalonamento do algoritmo, que se tornou mais constante, ou seja, sempre que a mesma tarefa era executada sob as mesmas condições, o resultado obtido era mais similar e proporcional ao cenário avaliado.

É possível notar também, pela comparação das duas tabelas citadas, que o valor de p, referente ao coeficiente de decaimento do feromônio, sofreu modificações. Este coeficiente

Tabela 6.3: Parâmetros de Configuração Inicial do AcoSched.

Parâmetros	**Valores**
Números de tarefas	10
Números de formigas na colônia	2
Número de iterações	5
p	0.01
α	3
β	4
γ	3

de decaimento é responsável por determinar qual será o impacto dos valores calculados anteriormente para a atualização do feromônio local e global.

6.2 Resultados Obtidos

Os resultados dos escalonamentos das tarefas realizadas no ZooNimbus, ao executar o algoritmo de escalonamento AcoSched, foram comparados aos resultados obtidos ao executar a política de escalonamento DynamicAHP [25], e podem ser, inicialmente, visualizados pela Figura 6.1. Nesta Figura, que representa o primeiro teste realizado, onde foi executado 40 tarefas, divididas em 4 conjuntos, pode-se verificar que o tempo de execução dos mesmos conjuntos de tarefas escalonadas pelo algoritmo AcoSched é na maioria das vezes menor do que o tempo obtido pelo escalonamento feito com o algoritmo DynamicAHP.

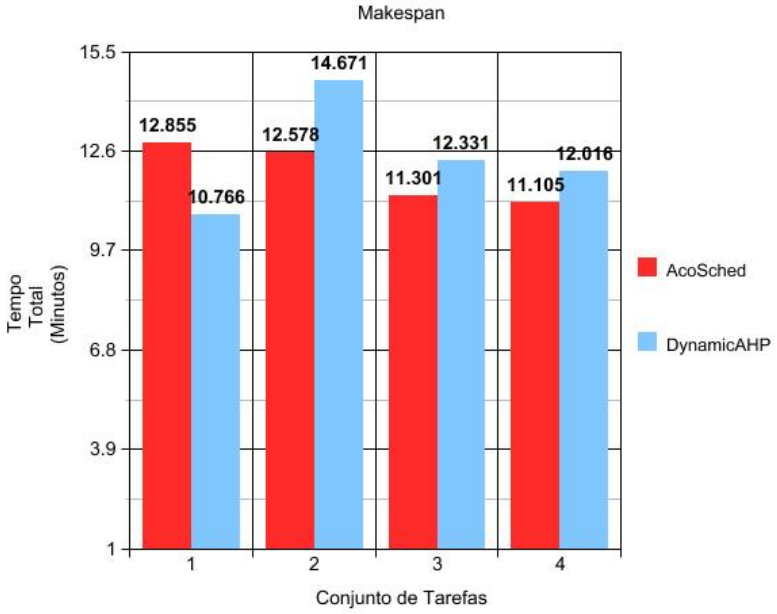

Figura 6.1: *Makespan* do Conjunto de Tarefas Executadas na Nuvem da UnB.

Os resultados mostrados na Figura 6.1 foram obtidos ao executar os servidores ZooNimbus somente na nuvem privada da Universidade de Brasília, levando um tempo médio de execução de 11 minutos e 967 segundos para os conjuntos de tarefas, e um tempo médio de 80,1 milésimos de segundos para o escalonamento de todas as tarefas do conjunto número 4 (veja Figura 6.1) de tarefas executadas, ambos os tempos obtidos ao utilizar a política AcoSched. Por outro lado, para com o DynamicAHP foi obtido um tempo médio de execução de 12 minutos e 446 segundos para cada conjunto de tarefas e um tempo médio de 2,1 milésimos de segundos para o escalonamento das tarefas do conjunto número 4 (veja Figura 6.1).

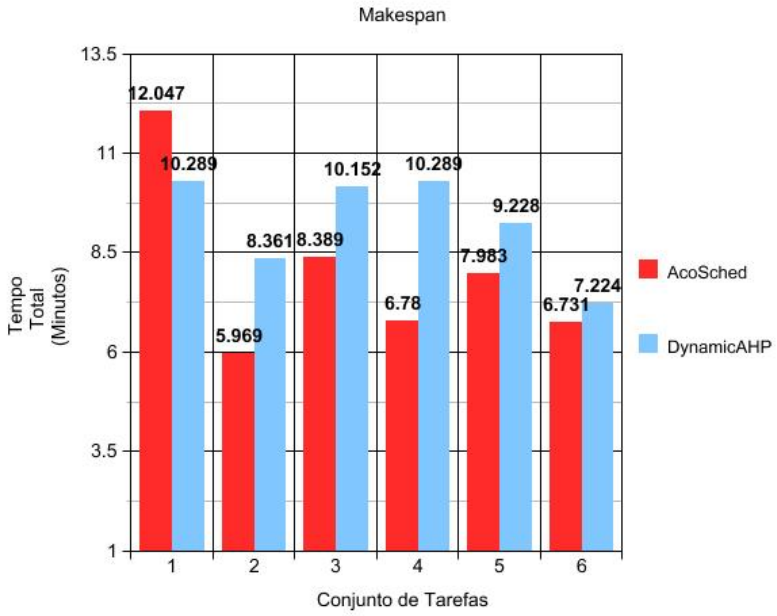

Figura 6.2: *Makespan* do Conjunto de Tarefas Executadas na Federação, com duas nuvens.

Na Figura 6.2, tem-se a analise do tempo total de execução das tarefas no ambiente de nuvem federada ZooNimbus, utilizando-se da nuvem da Universidade de Brasília e da nuvem pública provida pela Amazon [26]. É possível analisar por meio da Figura 6.2 que a diferença do tempo de execução do AcoSched tornou-se maior quando comparado ao DynamicAHP, ao executar o algoritmo na nuvem federada, isso porque ao analisar a saída do escalonamento, é possível verificar que o DynamicAHP realizou uma maior distribuição das tarefas entre os servidores ZooNimbus, utilizando um maior número de servidores diferentes para executar o conjunto de tarefas em relação ao AcoSched. Essa distribuição, porém, não foi a melhor decisão, visto que alguns desses servidores levaram um tempo maior para executar as tarefas em relação aos demais. Ainda observando a Figura 6.2, visualiza-se uma execução feita em menor tempo do que as demais, isso ocorreu porque nessa execução não houve transmissão de arquivos para a execuções de algumas tarefas, o que levou a um tempo total de execução menor.

Ao analisar outros resultados obtidos a partir da Figura 6.3, que mostra os tempos de escalonamentos obtidos, com o conjunto número 4 de tarefas executadas visualizado na Figura 6.1, nota-se que o AcoSched obtém tempos mais elevados para escalonar uma tarefa do que o DynamicAHP, porém esses tempos de escalonamento não foram refletidos no tempo total de execução das tarefas, o que significa que o AcoSched demora mais para

tomar uma decisão ao realizar o escalonamento, porém essa demora é compensada pelo fato da sua decisão resultar em um melhor escalonamento e, consequentemente, um menor tempo total de execução das tarefas.

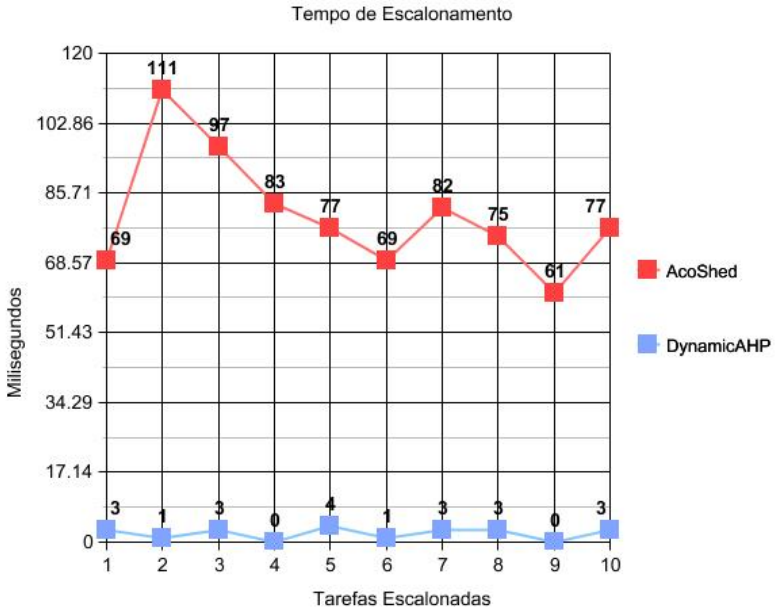

Figura 6.3: Tempo de Escalonamento do Primeiro Conjunto de Tarefas das Políticas Analisadas.

O maior tempo de escalonamento obtido pelo AcoSched é devido ao número de iterações realizadas pelas formigas para efetuar os cálculos das fórmulas descritas no Capítulo 5, como também pela quantidade de fatores analisados pelo AcoSched para realizar o escalonamento. Assim, comparando-se a variação dos tempos obtidos ao executar ambos os algoritmos, é possível notar que o tempo total de execução obtido pelo AcoSched é também mais constante do que aquele obtido pelo DynamicAHP.

Desta forma, ao analisar os resultados mostrados na Figura 6.1, quando o ZooNimbus é executado em somente uma nuvem, com os resultados mostrados pela Figura 6.2, quando o ZooNimbus é executada no ambiente federado, composto por duas nuvens, é possível notar que a variação do tempo total de execução foi reduzido ao aumentarmos o número de recursos disponíveis para a execução das tarefas, porém essa variação de tempo não foi proporcional a quantidade de recursos que foram acrescentados. Tal fato pode ser analisado como resultante da qualidade dos recursos adicionados. Na Figura 6.1 tem-se computadores com uma maior capacidade computacional, além também do fato da latência obtida na rede interna da UnB ser menor do que a obtida entre os recursos da UnB e da Azure.

Além disso, a Figura 6.4 mostra os valores dos feromônios obtidos a cada iteração do algoritmo AcoSched para os recursos analisados ao realizar o escalonamento de uma tarefa no ambiente do ZooNimbus executado na nuvem da UnB. Os valores foram obtidos sempre que o algoritmo AcoSched realizava a atualização do feromônio local. É possível analisar uma maior variação dos valores a cada 2 atualizações de feromônio, equivalente ao número de formigas utilizado no escalonamento. Essa mudança é devida a atualização global

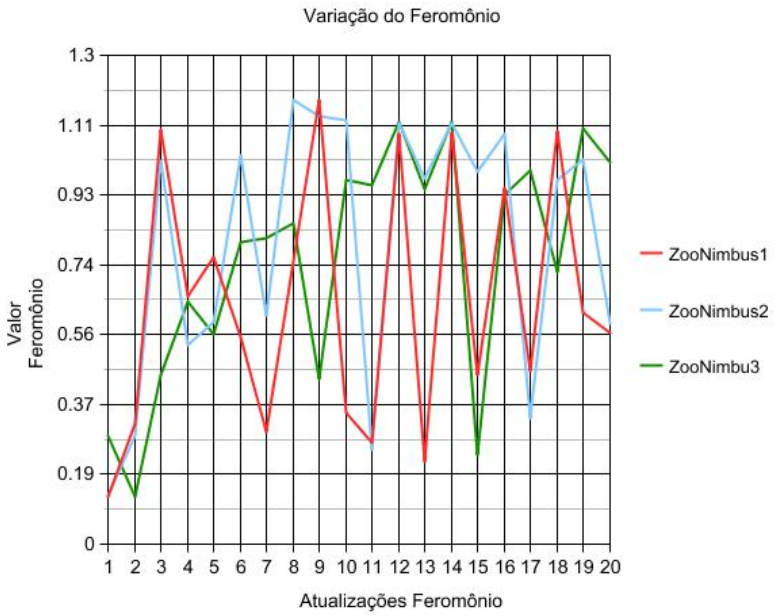

Figura 6.4: Variação dos Feromônios Obtidos ao Realizar o Escalonamento de uma Tarefa.

realizada sempre que as formigas terminam as visitas de todos os recursos participantes da federação em uma iteração do algoritmo.

Capítulo 7

Conclusão e Trabalhos Futuros

Neste trabalho foi proposto um algoritmo dinâmico que realiza o escalonamento de tarefas no ambiente de nuvem federada, objetivando a execução da tarefa nos recursos da nuvem no menor tempo possível. Para tal, foi proposto o algoritmo de escalonamento AcoSched, que utiliza probabilidade para realizar o escalonamento de uma tarefa por meio de uma heurística de colônia de formigas. A finalidade da política de escalonamento proposta é identificar um recurso disponível que seja capaz de executar uma tarefa no menor tempo possível. Assim, é eleito o melhor recurso dentre os disponíveis.

O AcoSched mostrou-se eficiente quando comparado com o algoritmo implementado anteriormente no ambiente de federação em nuvem, reduzindo o tempo total de execução dos conjuntos de tarefas ao considerar mais fatores para realizar o escalonamento. Apesar do aumento no tempo de decisão do escalonamento, o algoritmo demonstrou sua eficiência ao compensá-lo no tempo de execução da tarefa, otimizando, assim, as execuções de tarefas no ambiente.

Para implementar tal algoritmo, foi necessário realizar uma série de alterações no ambiente de nuvem federada, tornando-o mais flexível, dinâmico e de fácil manutenção. As modificações não foram realizadas alterando as propostas iniciais criadas por Saldanha [35], apenas modificando as tecnologias utilizadas para implementação do ambiente e a forma de comunicação entre os recursos da federação. Além das adaptações realizadas no algoritmo base para a política de escalonamento proposta, para que o AcoSched atendesse as necessidades da federação e funcionasse de forma eficiente.

Em relação aos trabalhos futuros, para otimizar ainda mais a eficiência das políticas de escalonamento no ambiente do ZooNimbus, será interessante a implementação de um controle de balanceamento, que seria constantemente executado pelo módulo de monitoramento e ficaria responsável por garantir o balanceamento de carga de trabalho dos recursos da nuvem de forma proporcional ao desempenho de cada recurso. Tal controle seria executado independente do algoritmo de escalonamento utilizado, após o escalonamento das tarefas. Outra proposta é a substituição da latência utilizada para realizar o escalonamento por outra variável relacionada, ou uma nova proposta na forma de cálculo da latência, já que pode haver problemáticas que atrapalhem o cálculo da latência em uma rede, como ocorreu no ambiente da Microsoft Azure.

Ao utilizar o serviço provido pelo Zookeeper, novos desafios e possibilidades surgiram para a implementação e controle dos serviços que formam o núcleo do ZooNimbus. Um desses desafios está relacionado ao acesso dos metadados disponíveis do servidor

Zookeeper, como é o caso dos metadados lançados no servidor quando uma nova tarefa é criada para ser escalonada, que geram concorrência entre os servidores ZooNimbus para realizar o escalonamento destas tarefas. Algumas dessas concorrências foram identificadas e tratadas de forma simples, porém outras concorrências podem surgir e a forma de tratamento atual de algumas delas podem ser otimizadas.

Ao analisar o uso do Zookeeper e o do Avro Apache no ambiente ZooNimbus, uma nova abordagem pode ser obtida, com relação ao papel de coordenação desempenhado pelo Zookeeper. Essa abordagem pode ser mais flexível, por exemplo, ao executar um escalonamento e enviar os metadados referentes a decisão da política para o servidor Zookeeper. Deve ser analisado o desempenho do ZooNimbus ao utilizar o Zookeeper para informar e receber os dados das tarefa escalonadas, pois esse procedimento pode ser substituído por uma chamada de procedimento remoto (RPC) direta ao recurso eleito. Deixando, assim, o recurso eleito encarregado de persistir os metadados no servidor Zookeeper da tarefa que ele irá executar e, portanto, otimizando o tempo total de execução da tarefa.

Referências

[1] Celesti A., Tusa F., Villari M., and Puliafito A. How to enhance cloud architectures to enable cross-federation. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 337 –345, july 2010. vii, 1, 2, 7, 8, 9

[2] Houaiss A., Villar M. de S., and Franco F. M. de M. *Dicionário Houaiss da língua portuguesa*. São Paulo, 1 edition, 2009. 32

[3] Pashalidis A. and Mitchell C. J. A taxonomy of single sign-on systems. In *In Information Security and Privacy, 8th Australasian Conference, ACISP 2003*, pages 249—-264, 2003. 12

[4] Oracle and/or its affiliates. Oracle cloud. https://cloud.oracle.com/mycloud/ f ? p=service:home:0, 2012. Acessado online em 09 de Dezembro de 2012. 7

[5] Langmead B., Trapnell C., Pop M., and Salzberg S. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. In *Genome Biology, 10(3):R25+*, 2009. 45

[6] Wickremasinghe B., Calheiros R.N., and Buyya R. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 446 –452, april 2010. 35

[7] Chih-Yung Chen and Hsiang-Yi Tseng. An exploration of the optimization of excutive scheduling in the cloud computing. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 1316 –1319, march 2012. 6

[8] Borthakur D. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, pages 1–14, 2007,http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf. 16

[9] Dropbox. Dropbox jobs. https://www.dropbox.com/jobs, 2012. Acessado online em 09 de Dezembro de 2012. 6

[10] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition, 1994. 22

[11] Facebook. Facebook. https://www.facebook.com/facebook, 2013. Acessado online em 3 de julho de 2013. 24

[12] The Apache Software Foundation. Apache avro™ 1.7.4 documentation. http://avro.apache.org/docs/current/, 2013. Acessado online em 30 de junho de 2013. 24, 30

[13] The Apache Software Foundation. The apache software foundation. http://www.apache.org/, 2013. Acessado online em 30 de maio de 2013. 20, 21, 24

[14] The Apache Software Foundation. Apache thrift™. http://thrift.apache.org/, 2013. Acessado online em 3 de julho de 2013. 24

[15] The Apache Software Foundation. Apache zookeeper. http://zookeeper.apache.org/, 2013. Acessado online em 30 de maio de 2013. vii, 20, 21, 22, 30

[16] Saldanha H., Ribeiro E., Borges C., Araujo A., Gallon R., Holanda M., Walter M. E., Togawa R., and Setubal J. C. *BioInformatics*. In Tech, 2012. 2

[17] Foster I., Zhao Y., Raicu I., and Lu S. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE 08*, pages 1 –10, nov. 2008. 1, 4

[18] Google Inc. Google drive. https://drive.google.com/. 6

[19] Google Inc. Google app engine. https://developers.google.com/appengine/docs/, 2012. Acessado online em 09 de Dezembro de 2012. 6

[20] Google Inc. Protocol buffers. http://code.google.com/p/protobuf/, 2013. Acessado online em 3 de julho de 2013. 24

[21] Yahoo! Inc. Introducing json. http://www.json.org/, 2013. Acessado online em 03 de julho de 2013. 24, 25

[22] Geelan J. Twenty-one experts define cloud computing. http://cloudcomputing.sys-con.com/node/612375/, 2009. Acessado online em 10 de Novembro de 2012. 1

[23] Karlsson J., Torreno O., Ramet D., Klambauer G., Cano M., and Trelles O. Enabling large-scale bioinformatics data analysis with cloud computing. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 640 –645, july 2012. 1

[24] Li K., Xu G., Zhao G., Dong Y., and Wang D. Cloud task scheduling based on load balancing ant colony optimization. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual*, pages 3 –9, aug. 2011. vii, viii, 2, 32, 35, 36, 37

[25] Borges C. A. L. Escalonamento de tarefas em uma infraestrutura de computação em nuvem federada para aplicações em bioinformática. http://hdl.handle.net/123456789/377, 2012. Departamento de Ciência da Computação, Universidade de Brasília. vii, 2, 10, 14, 15, 16, 17, 18, 44, 47

[26] Amazon Web Services LLC. Amazon elastic compute cloud. http://aws.amazon.com/pt/ec2/, 2012. Acessado online em 12 de Novembro de 2012. 6, 44, 45, 48

[27] Dorigo M. and Blum C. Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344(2-3):243–278, November 2005. 2, 32

[28] Microsoft. Uma plataforma sólida na nuvem para ideias criativas. http://www.windowsazure.com/pt-br/, 2012. Acessado online em 12 de Novembro de 2012. 6, 44, 45

[29] U.S. National Library of Medicine. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/, 2013. 45

[30] Rimal B. P., Eunmi Choi, and Lumb I. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44 –51, aug. 2009. 1, 2, 5, 6

[31] Moura B. R. and Bacelar D. L. Política para armazenamento de arquivos no zoonimbus, 2013. Departamento de Ciência da Computação, Universidade de Brasília. 29

[32] salesforce.com. Salesforce.com. http://www.salesforce.com/br/, 2012. Acessado online em 07 de Dezembro de 2012. 7

[33] Andrew S. Tanenbaum and Maarten V. Steen. *Sistemas Distribuídos : princípios e paradigmas*. São Paulo, 2 edition, 2007. 4, 20, 35

[34] Casavant T.L. and Kuhl J.G. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141 –154, feb 1988. 14

[35] Saldanha H. V. Bionimbus: uma arquitetura de federação de nuvens computacionais híbrida para a execução de workflows de bioinformática. Master's thesis, Departamento de Ciência da Computação, Universidade de Brasília, 2012. vii, 2, 8, 9, 10, 11, 12, 13, 20, 25, 51

[36] Tom White. *Hadoop: The Definitive Guide*. 3 edition, 2012. 21

[37] Wei Y. and Tian L. Research on cloud design resources scheduling based on genetic algorithm. In *Systems and Informatics (ICSAI), 2012 International Conference on*, pages 2651 –2656, may 2012. 2

[38] Yang Z., Yin C., and Liu Y. A cost-based resource scheduling paradigm in cloud computing. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, pages 417 –422, oct. 2011. 1, 4