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

Política para Armazenamento de Arquivos no ZooNimbus

Breno Rodrigues Moura
Deric Lima Bacelar

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

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

Coorientador
MsC. Edward de Oliveira Ribeiro

Brasília
2013

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

Coordenador: Prof. Dr. Coordenador Flávio Vidal

Banca examinadora composta por:

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

CIP — Catalogação Internacional na Publicação

Moura, Breno Rodrigues.

Política para Armazenamento de Arquivos no ZooNimbus / Breno Rodrigues Moura, Deric Lima Bacelar. Brasília : UnB, 2013.
137 p. : il. ; 29,5 cm.

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

1. computação em nuvem, 2. armazenamento, 3. federação de nuvens, 4. bioinformática.

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

Política para Armazenamento de Arquivos no ZooNimbus

Breno Rodrigues Moura
Deric Lima Bacelar

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 Emilia Machado Telles Walter
CIC/UnB

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

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

Brasília, 26 de agosto de 2013

Dedicatória

Deric Lima: Dedico este trabalho aos meus pais, que me deram apoio e suporte para seguir os meus sonhos. Dedico também a todos os meus amigos que me deram forças e me motivaram a continuar tentando.

Breno Rodrigues: Dedico este trabalho primeiramente aos meus pais que se preocuparam comigo quando eu passava madrugadas no laboratório, que me apoiaram e me incentivaram a persitir e não desistir dos meus sonhos e trabalhos. Dedico a minha namorada, pela paciência e compreensão durante as tardes e madrugadas as quais estava fazendo este trabalho, dedico também aos meus amigos, que aos quais sem esses não teria atingido os resultados obtidos no trabalho.

Agradecimentos

Agradecemos aos nossos professores e amigos, que nos auxiliaram e nos motivaram para a realização este trabalho. A nossa orientadora, Aletéia Patrícia, pela paciência, dedicação e por nos motivar em seguir em frente com o trabalho nas horas de desespero e agunia por não conseguir avançar no trabalho. Agradecemos ao Gabriel de Sousa, que por trabalhar em um projeto que operou em conjunto com o nosso, nos auxiliou no desenvolvimento do mesmo, formando uma ótima equipe de trabalho. Pelas horas na madrugada e pelas risadas durante o desenvolvimento deste trabalho e pelos desesperos passados juntos. Agradecemos aos nossos pais, que sempre nos falaram para nunca desistir e persistir sempre, para que nosso objetivo fosse alcançado. Agradecemos imensamente ao Edward Ribeiro, que nos mostrou que era possível realizar este trabalho, agradecemos por toda sua ajuda como co-orientador, por conseguir nos motivar nos momentos mais difíceis e por nos mostrar soluções para os problemas que foram encontrados durante a implementação deste projeto, por doar horas do seu tempo, para nos ajudar de forma voluntária e muito prestativa nos momentos de dúvidas, ajuda que sem a qual não atingiríamos os resultados obtidos neste trabalho.

Resumo

Políticas de armazenamento são difíceis de serem implementadas para um ambiente de nuvens federadas, uma vez que existem muitos provedores componentes da federação em nuvem com capacidades de armazenamento distintas que devem ser consideradas. Por outro lado, em Bioinformática, muitas ferramentas e bancos de dados necessitam de grandes volumes de recursos para processarem e armazenarem quantidades enormes de dados, que podem atingir facilmente *terabytes* de tamanho. Este trabalho trata do problema da política de armazenamento no BioNimbus, o qual é uma infraestrutura de nuvens federadas para aplicações de bioinformática. Neste contexto, este trabalho propõe uma política de armazenamento, chamada ZooClouS (*ZooNimbus Cloud Storage*), que se baseia na latência, no custo, no *uptime* e no espaço livre de armazenamento para realizar uma escolha que distribui eficientemente os arquivos para os melhores recursos disponíveis na nuvem federada. Os experimentos foram realizados com dados biológicos reais, os quais foram executados em uma federação de nuvens, instaladas na Amazon EC2, no Windows Azure e na Universidade de Brasília (UnB). Os resultados obtidos mostram que o ZooClouS conseguiu uma melhoria significativa no tempo de *makespan* das aplicações de bioinformática executadas, quando comparado com a política de armazenamento aleatória que estava implementada.

Palavras-chave: computação em nuvem, armazenamento, federação de nuvens, bioinformática.

Abstract

Storage policy is difficult in federated cloud environments, since there are many cloud providers with distinct storage capabilities that should be addressed. In bioinformatics, many tools and databases requiring large resources for processing and storing enormous amounts of data, that can easily achieve terabytes of size, are provided by physically separate institutions. This work treats the problem of storage policy in ZooNimbus, a federated cloud infrastructure for bioinformatics applications. We propose a storage policy, named ZooClouS (ZooNimbus Cloud Storage), that is based on the latency, on cost, on uptime and on free size to perform an efficient choice to distribute the files to the best resources available in the federated cloud to execute each required task. We developed experiments with real biological data executing on ZooNimbus, formed by some cloud providers executing in Amazon EC2 and UnB. The obtained results show that ZooClouS makes a significant improvement in the makespan time of bioinformatics applications executing in ZooNimbus, when compared to the random algorithm.

Keywords: cloud computing, storage, cloud federation, bioinformatic.

Sumário

1 Introdução **1**
- 1.1 Motivação . . . 2
- 1.2 Problema . . . 2
- 1.3 Objetivos . . . 3
  - 1.3.1 Principal . . . 3
  - 1.3.2 Específicos . . . 3
- 1.4 Estrutura do Trabalho . . . 3

2 Computação em Nuvem **5**
- 2.1 Conceitos Básicos . . . 5
  - 2.1.1 Características de uma Nuvem . . . 6
  - 2.1.2 Arquitetura de uma Nuvem . . . 7
  - 2.1.3 Tipos de Nuvens . . . 9
  - 2.1.4 Comparação entre os Principais Provedores de Nuvem . . . 11
- 2.2 Federação de Nuvens . . . 12
- 2.3 Armazenamento em Nuvem . . . 14
  - 2.3.1 *Data-storage-as-a-Service (DaaS)* . . . 15
- 2.4 Considerações Finais . . . 16

3 Do BioNimbus para o ZooNimbus **17**
- 3.1 BioNimbus . . . 17
  - 3.1.1 Arquitetura do BioNimbus . . . 18
  - 3.1.2 Classes do BioNimbus . . . 22
  - 3.1.3 Serviço de Armazenamento no BioNimbus . . . 22
- 3.2 Apache ZooKeeper . . . 24
  - 3.2.1 *Znodes* Persistentes e *Znodes* Efêmeros . . . 26
  - 3.2.2 *Watchers* . . . 26
  - 3.2.3 Garantias . . . 27
  - 3.2.4 API Simples . . . 27
- 3.3 Apache Avro . . . 27
  - 3.3.1 Esboços . . . 28
  - 3.3.2 Comparando o Avro com outros Sistemas . . . 29
- 3.4 Integração com o Apache ZooKeeper . . . 29
- 3.5 Integração com o Apache Avro . . . 30
- 3.6 A Nova Versão do BioNimbus - ZooNimbus . . . 31
- 3.7 Considerações Finais . . . 33

4 A Política de Armazenamento ZooClouS **34**
4.1 Mudanças nos outros Serviços do BioNimbus . . . 34
4.1.1 *Discovery Service* . . . 35
4.1.2 *Monitoring Service* . . . 36
4.1.3 *Scheduling Service* . . . 37
4.1.4 *Fault Tolerance Service* . . . 38
4.2 *Storage Service* . . . 39
4.2.1 *Upload* de Arquivos . . . 40
4.2.2 Replicação de Arquivos . . . 41
4.2.3 Tolerância a Falhas . . . 42
4.2.4 *Download* de arquivos . . . 44
4.2.5 Listagem de Arquivos . . . 45
4.2.6 O Algoritmo A* . . . 46
4.2.7 ZooNimbus Cloud Storage - ZooClouS . . . 48
4.3 Considerações Finais . . . 49

5 Estudo de Caso **51**
5.1 Tempo de Decisão e Transferência . . . 51
5.1.1 Ambiente de Execução . . . 51
5.1.2 Escolha do Melhor Servidor para os Arquivos . . . 52
5.2 Execução de Tarefas . . . 53
5.2.1 Ambiente de Execução . . . 53
5.2.2 Resultados Obtidos . . . 54
5.3 Considerações Finais . . . 55

6 Conclusão e Trabalhos Futuros **56**

Referências **57**

Lista de Figuras

2.1 Relação Entre os Atores Envolvidos em uma Nuvem [39]. 7
2.2 Os Atores e as Camadas de uma Plataforma em Nuvem [39]. 8
2.3 Tipos de Nuvens (adaptado de Vmware [40]). 10
2.4 As Fases da Computação em Nuvem (segundo White E. et al. [41]). 13
2.5 Evolução do armazenamento de dados na nuvens (adaptado de J. Wu [42]). 15

3.1 BioNimbus uma Arquitetura de Federação de Nuvens Computacionais para Aplicações de Bioinformática (adaptado de [34]). 18
3.2 Modelo de Serviço do Apache Zookeeper[14]. 25
3.3 Hierarquia de ZNodes no ZooKeeper. 26
3.4 Exemplo de Arquivo *.avdl* Utilizado pelo Avro no ZooNimbus. 28
3.5 Exemplo de Conexão entre o ZooNimbus e o ZooKeeper. 30
3.6 Chamadas RPC Passo-a-Passo no ZooNimbus. 31
3.7 Arquitetura de Federação em Nuvens ZooNimbus. 32
3.8 Estrutura do ZooKeeper com Relação ao *Storage Service*. 33

4.1 Assinatura que Deve ser Implementada ao Utilizar a Interface *SchedPolicy*. 38
4.2 Sequência de Ações que Ocorrem Durante o Processo de *Upload* 40
4.3 Sequência de Ações que Ocorrem Durante o Processo de Replicação dos Arquivos. 42
4.4 Sequência de Ações que Ocorrem Durante o Processo de Tolerância a Falhas. 43
4.5 Sequência de Ações que Ocorrem Durante o Processo de *Download*. 44
4.6 Sequência de Ações que Ocorrem Durante o Processo de Listagem dos Arquivos. 45
4.7 Algoritmo A* Otimizado para Armazenamento em Nuvens 47

5.1 Destino de Arquivo Enviado ao ZooNimbus com a Implementação da ZooClouS, e sem a Implementação da ZooClouS. 52
5.2 Tempo Médio de Transferência de Arquivo com o ZooClouS e de Forma Aleatória. 53
5.3 Tempo de Execução de Tarefas com o ZooNimbus. 55

Lista de Tabelas

2.1 Comparação entre provedores de computação em nuvens (adaptado de G. Magalhães [26]). 11

4.1 Interface Implementada pelos Serviços no BioNimbus. 35
4.2 Nova Interface Implementada pelos Serviços no ZooNimbus. 35

5.1 Nomes e Tamanhos dos Arquivos de Entrada das Tarefas. 54

Capítulo 1

Introdução

Em um cenário no qual a maior parte do processamento computacional é realizado por servidores locais e *data centers* proprietários, onde o poder de processamento computacional nem sempre é utilizado por completo, paga-se por energia, manutenção e tecnologias que podem não ser usadas em sua totalidade. Neste contexto, surgiu a ideia de que, em vez de pagar por uma estrutura proprietária, de manutenção cara, que às vezes fica ociosa, fosse utilizada uma estrutura que usasse os seus recursos de acordo com a necessidade de cada usuário. Assim, os recursos seriam alocados de acordo com a demanda de utilização necessária. Essa é a chamada computação sob demanda, cuja a grande vantagem é que encerra o cenário onde o usuário é obrigado a manter e pagar por uma infraestrutura que não é usada em sua totalidade sempre.

Nesse cenário, surge a plataforma de computação em nuvem, a qual busca reduzir o custo computacional, aumentar a confiabilidade e a flexibilidade para transformar computadores de tal forma que todo o tratamento dos dados e aplicações são vistos como um serviço [3]. O objetivo é que a computação em nuvem disponibilize um serviço totalmente sob demanda, em que para todos os processamentos solicitados, sejam alocados recursos de acordo com a necessidade, e que esse processo seja totalmente dinâmico, oferecendo poder computacional somente na medida necessária.

Atualmente, é possível encontrar alguns serviços de nuvens públicas por meio de empresas como a Amazon [25], a Microsoft [6] e a Google [16]. Em sistemas de computação em nuvem, diferentes fundamentos estão presentes, tais como a virtualização, a escalabilidade, a interoperabilidade, a qualidade de serviço (QoS), os mecanismos sobre falhas e os modelos de entrega de nuvens (privadas, públicas e híbridas). Os três principais tipos de serviços de nuvens disponibilizados são os de software (SaaS), de infraestrutura (IaaS) e de plataforma (PaaS) como serviços.

Como a quantidade de dados a ser processada aumenta cada vez mais no mundo da computação, seja em processos de aplicações ou no próprio armazenamento de dados, surge a necessidade de ter um poder de processamento e armazenamento cada vez maior, que possa atender pedidos em um tempo hábil, de modo automático e dinâmico. Porém, uma nuvem possui recursos limitados, e por conta disso surgiu a ideia de se integrar nuvens. O objetivo é aumentar os recursos disponíveis, pois se um recurso de uma nuvem se esgotar, uma outra nuvem pode utilizar os seus recursos caso estejam disponíveis. Dessa ideia emergiu o conceito de Federação de Nuvens.

As federações de nuvens são conjuntos de nuvens que possuem todos os seus recursos

gerenciados por meio de uma interface conectada a todas elas, de forma que se uma nuvem não tiver recursos para determinado processamento, uma outra nuvem que esteja com recursos disponíveis no momento possa realizar este processamento.

Nesse cenário, Hugo et al. [33] propuseram uma arquitetura de federação de nuvens chamada BioNimbus, simples e dinâmica, que pode executar diferentes aplicações dentro da federação. Atualmente, ela vem sendo utilizada para a execução de *workflows* de bioinformática, mas nada impede que seja usada para outras aplicações. Dentre os objetivos do BioNimbus estão integrar e controlar diferentes provedores de infraestrutura, oferecendo um serviço flexível e tolerante a falhas. Neste trabalho, fizemos várias modificações na estrutura original do BioNimbus, em especial, foi realizada a integração do ZooKeeper Apache [14] e do Avro Apache [11]. Como consequência, este passou a ser chamado de ZooNimbus, embora continue a prover uma plataforma que disponibiliza uma federação em nuvem que executa serviços de bioinformática com recursos heterogêneos, públicos ou privados.

Todavia, como o ZooNimbus não tinha uma política de armazenamento de dados definida, o objetivo deste trabalho é propor uma política de armazenamento para a plataforma de federação ZooNimbus, de tal forma que o *makespan* das tarefas executadas no ambiente federado seja minimizado.

1.1 Motivação

A federação de nuvens tem se mostrado uma plataforma capaz de integrar diferentes infraestruturas, proporcionando uma maior flexibilidade na escolha de provedores e uma visão na qual a quantidade de armazenamento e processamento passe a ideia para o usuário de que esses recursos são ilimitados, porém não existe uma padronização para a tecnologia de integração com outras nuvens.

Uma arquitetura de federação de nuvens computacionais híbridas para execução de *workflows* de bioinformática, denominada BioNimbus, foi implementada para tentar alcançar essa padronização, mas essa arquitetura não possui uma política para o armazenamento de dados, utilizando, uma forma aleatória de alocação de dados. Isto faz com que as aplicações executadas no BioNimbus tenham um desempenho baixo.

Neste cenário, a motivação deste trabalho é melhorar o tempo de execução total de aplicações executadas no ambiente de nuvens federadas BioNimbus, por meio de uma escolha correta do melhor recurso para armazenar arquivos de entrada e saída vindos destas aplicações.

1.2 Problema

Atualmente, não existe uma política de armazenamento eficiente no BioNimbus. Assim, os arquivos são armazenados em qualquer recurso no qual o cliente se conecta de forma aleatória. Como consequência, o ZooNimbus:

- Não possui política de armazenamento de dados implementada;
- Não realiza operações com os seus dados, tais como redundância, replicação, tolerância a falhas, backups, etc;

- Os dados armazenados não são gerenciados pelo módulo de armazenamento.

1.3 Objetivos

1.3.1 Principal

Implementar uma política de armazenamento de dados na arquitetura de federação em nuvens ZooNimbus, para que os dados armazenados na federação sejam tratados de forma a otimizar o desempenho na execução das tarefas e na plataforma de federação em nuvens.

1.3.2 Específicos

Para cumprir o objetivo principal, este trabalho tem os seguintes objetivos específicos:

- Realizar um estudo de caso com os arquivos do BioNimbus para testar o desempenho no armazenamento dos arquivos na nuvem;
- Utilizar o Apache Avro e Apache Zookeeper para implementar uma política de armazenamento para os arquivos do BioNimbus;
- Implementar uma política de armazenamento no BioNimbus;
- Discutir os resultados obtidos com a política de armazenamento definida para o BioNimbus.

1.4 Estrutura do Trabalho

Este trabalho está divido em seis capítulos. No Capítulo 2 será apresentado conceitos e caracterísiticas de computação em nuvem, abordando a limitação que a nuvens possuem. E diante dessa questão, surge o conceito de integrar várias nuvens, as chamadas federações de nuvens. Mostrando que as federações integram as nuvens, não importando o seu tipo. Abordando também conceitos e exemplos de um dos temas mais discutidos nas federações de nuvens, o armazenamento em nuvem.

No Capítulo 3, é apresentado a arquitetura de federação de nuvens proposta por Hugo Saldanha [34], o BioNimbus, integrando os programas da Apache, o Apache Zookeeper e o Apache Avro, uma nova versão do BioNimbus surge, o ZooNimbus.

No Capítulo 4 será mostrado as mudanças que ocorreram com a implementação do ZooNimbus em relação aos serviços que o BioNimbus implementava, e por fim é demonstrado o algoritmo utilizado na implementação da política de armazenamento, o ZooClouS, que foi baseado no Algoritmo A* [45], proposto para o ambiente de computação em nuvem em conjunto com as modificações sofridas no serviço de armazenamento.

No Capítulo 5 é descrito os resultados obtidos com a implementação da política, em relação ao tempo de transferência de arquivos, transferindo os arquivos do ZooNimbus de forma aleatória, como estava sendo feito no BioNimbus, e transferindo os dados do ZooNimbus com o ZooClouS, observando também a localidade de onde o arquivo seria armazenado a partir do referencial, cliente. Por último, são apresentados testes que

comprovam que quanto maior a quantidade de recursos disponíveis mais rápido as tarefas serão executadas.

No Capítulo 6 são apresentadas as conclusões obtidas com a realização deste trabalho, assim como os trabalhos futuros, no qual pode ajudar a melhorar a arquitetura de forma exponencial.

Capítulo 2

Computação em Nuvem

O objetivo deste capítulo é apresentar, inicialmente, o paradigma de computação distribuída chamado computação em nuvem, deixando claro todas as suas particularidades e características. Também serão ressaltadas as suas diferenças entre as principais prestadoras desse serviço. Em seguida, na Seção 2.1, serão detalhados os conceitos da computação em nuvem, o surgimento da ideia, o funcionamento da tecnologia, e as suas camadas de abstração. Logo após, tem-se a Seção 2.2, na qual são demonstrados os objetivos da federação em nuvens, sua definição e suas fases. Por último, na Seção 2.3, são apresentados alguns conceitos básicos sobre o armazenamento de dados em uma infraestrutura na nuvem, também é apresentado um modelo de serviço que exemplifica o armazenamento em nuvem, que é entregue sob demanda como um serviço, chamado de *Data-storage-as-a-Service*.

2.1 Conceitos Básicos

Nos anos 60, a computação entrava na terceira geração de computadores, na qual os grandes transistores foram substituídos por circuitos integrados, que possibilitaram um grande aumento de processamento de dados, diminuição do tamanho dos computadores e do consumo de energia elétrica [19]. Com esse aumento no poder computacional e a diminuição no consumo de energia pelos computadores, surgiu a ideia esboçada pelo especialista em inteligência artificial John McCarthy [22], em que a computação fosse vista como uma utilidade pública, como os serviços básicos de água, de energia, de gás e de telefone. Assim, o poder de processamento computacional e de armazenamento poderiam ser disponibilizados sob demanda.

O crescimento em poder computacional e em capacidade de armazenamento alcançados pelos computadores daquela época era cada vez maior, e a quantidade de problemas complexos que estava sendo tratados por cientistas, físicos, biólogos e engenheiros também estavam crescendo cada vez mais, exigindo uma demanda computacional que muitas vezes não era disponibilizada por uma única máquina.

Essa realidade começou a ser transformada a partir da década de 70, quando surgiram os sistemas computacionais distribuídos, que evoluíram naturalmente, substituindo os sistemas centralizados (os chamados *mainframes*) [1].

Na década de 80, motivadas pelo alto custo de aquisição e manutenção de máquinas paralelas, assim como pela dependência estabelecida entre o usuário e o fabricante da

máquina, as comunidades envolvidas com pesquisa em sistemas distribuídos e computação paralela propuseram a utilização de sistemas distribuídos como plataforma de execução paralela. Isso concretizou uma melhor relação custo/benefício para a computação paralela, pois proporcionou menor custo de implantação e maior poder computacional a uma ampla variedade de aplicações [1].

Na década de 90, a Internet causou uma mudança drástica nos conceitos do mundo da computação, começando com o conceito de computação paralela, mudando para a computação distribuída depois para a computação em grade e, recentemente, a computação em nuvem. Muitas empresas da área de tecnologia da informação entraram nesse novo paradigma realizando implementações na nuvem. A Amazon desempenhou um papel fundamental lançando o seu *Amazon Web Services* (AWS) em 2006. Em seguida, a Google e a IBM também anunciaram planos, investimentos e começaram projetos de pesquisa em nuvem [21], buscando melhorar a utilização de recursos computacionais.

Na plataforma de nuvem, os recursos utilizados variam desde memória, armazenamento, poder de processamento até largura de banda, sendo que todos eles estão disponibilizados por meio da Internet, podendo serem acessados de qualquer lugar do mundo, motivo pelo qual ocorre a associação ao termo "nuvem".

Nesse ambiente, a sua flexibilidade está na alocação dos recursos, que ocorre sob demanda. Do ponto de vista econômico, o usuário paga apenas pela utilização dos recursos na nuvem, sem a necessidade de utilizar os recursos de seu computador ou servidor local. A ideia é que não seja necessário adquirir softwares ou infraestruturas, pois é mais barato pagar apenas quando ocorre uma demanda para a utilização deles. Para empresas isso significa uma grande redução nos custos, podendo atingir uma economia de milhões de dólares [8].

No cenário da computação em nuvem, ocorre uma alternativa na forma de utilização de serviços. Ao contrário do cenário no qual o usuário coloca serviços para serem feitos por um servidor local, ele contrata um provedor para que ele realize todos esses serviços. Estes serviços podem ser de armazenamento de dados, utilização da memória e do poder de processamento da provedora para rodar aplicações, e também infraestrutura para hospedar serviços. Com base neste cenário, a computação em nuvem possui uma série de características na forma de realizar seus serviços.

2.1.1 Características de uma Nuvem

A computação em nuvem representa uma nova maneira de usar os recursos computacionais, pois ela possui os seguintes princípios:

- Diversidade: ocorre a atuação em nuvens públicas, privadas e também em nuvens híbridas, que serão explicadas em detalhes na Subseção 2.1.3;
- Compartilhamento de Infraestrutura: muitos clientes podem compartilhar a mesma infraestrutura, incluindo até mesmo instâncias de um aplicativo;
- Serviços Sob Demanda: seja por número de usuários, transações ou combinação entre vários itens;
- Preços com Base no Uso: a cobrança pelo serviço é feita de acordo com a quantidade e tempo dos recursos utilizados;

- Elasticidade: a capacidade de adicionar ou remover recursos em um servidor de cada vez e com um prazo de execução de minutos ao invés de semanas, permitindo combinar os recursos para melhor atender a carga de trabalho.
- Escalabilidade: a capacidade de gerenciar uma quantidade crescente de trabalho de uma maneira eficaz, e ou a capacidade de ser ampliado para acomodar o crescimento.

A computação em nuvem opera sobre os conceitos de virtualização e computação distribuída, em infraestruturas que podem ser supercomputadores, *clusters* ou grades. A ideia do serviço é semelhante ao de uma rede elétrica, quanto mais se usa, maior é o valor pago pelo uso, o qual é conhecido como modelo *pay-per-use* [4].

Existem várias empresas que já estão disponibilizando serviços de computação em nuvem, por exemplo a Amazon [25], a Microsoft [6] e a Google [16].

2.1.2 Arquitetura de uma Nuvem

A arquitetura de uma nuvem é composta por uma série de elementos, entre eles estão os atores envolvidos no processo e as camadas básicas de abstração da nuvem, onde ocorre a divisão das diferentes funções desempenhadas pelos atores e pelas infraestruturas utilizadas por eles.

A relação entre os atores de uma nuvem pode ser vista na Figura 2.1, no qual encontram-se todos os envolvidos no processo. Os três principais atores são os Provedores de Serviços (*Service Providers*), os Usuários do Serviço e os Provedores de Infraestrutura (*Infrastructure Providers*). Os provedores de infraestrutura disponibilizam um ambiente completo no qual os provedores de serviços colocam a sua aplicação e a deixam disponível para os usuários do serviço [39].

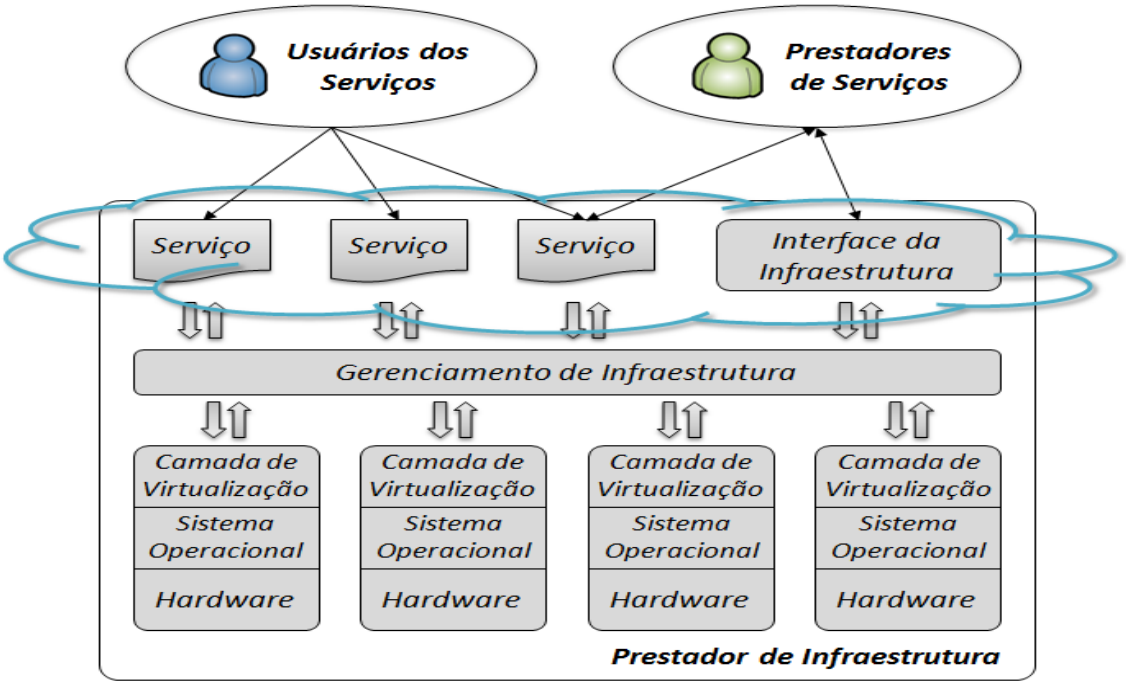

Figura 2.1: Relação Entre os Atores Envolvidos em uma Nuvem [39].

Dessa forma, uma arquitetura de computação em nuvem possui basicamente três camadas as quais são mostradas na Figura 2.2. Nessa figura, encontra-se próximo ao cliente o maior nível de abstração, onde ficam todas as aplicações disponíveis. A camada mais baixa é a camada de infraestrutura, onde estão todos os componentes físicos da nuvem (servidores, *data centers*, roteadores e sistemas de armazenamento). O cliente não necessita adquirir nenhum desses componentes para rodar a sua aplicação, ele precisa somente pagar pelo uso destes recursos de acordo com a sua necessidade, tanto de armazenamento quanto de processamento.

A camada do meio é a camada de plataforma, a qual oferece um ambiente para o provedor de serviço, sem que este precise se preocupar em comprar software e ter que instalá-lo. Por meio desta plataforma eles podem controlar todos os sistemas e ambientes necessários para o software ser desenvolvido, testado, implantado e hospedado na web para os usuários.

A camada de aplicação é onde ocorre a interação com os usuários do serviço. As aplicações na nuvem ficam disponíveis para acesso. Logo, o requisito mais importante para se ter acesso ao sistema é uma conexão com a Internet.

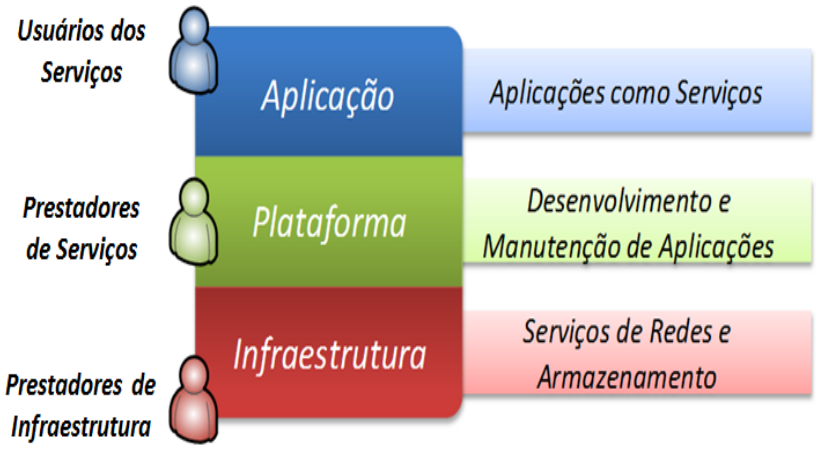

Figura 2.2: Os Atores e as Camadas de uma Plataforma em Nuvem [39].

Assim sendo, a computação em nuvem classifica todos os seus recursos em serviços, baseados nas camadas de abstração mostradas na Figura 2.2. Abaixo tem-se uma breve descrição dos serviços presentes na computação em nuvem, os quais são Software como Serviço (SaaS), Plataforma como Serviço (PaaS) e Infraestrutura como Serviço (IaaS):

- Software como Serviço (SaaS): aplicações são desenvolvidas especificamente para o ambiente de computação em nuvem, e são fornecidas como serviço por provedores para os usuários. Nesta camada os serviços podem utilizar os ambientes de programação fornecidos pelos provedores da camada PaaS, ou podem utilizar diretamente a infraestrutura fornecida pela camada IaaS. O acesso a essas aplicações é realizado remotamente, podendo ser acessado de qualquer lugar a qualquer momento. Uma cobrança pode ser feita pela utilização do serviço na nuvem. Entre as vantagens da SaaS estão as tarefas de manutenção, operação e suporte que ficam por conta do provedor do serviço [34]. Softwares como Google Docs [17] e Sales Cloud [9] são algumas aplicações que exemplificam esta camada.

- Plataforma como Serviço (PaaS): a plataforma como serviço permite aos usuários utilizar a computação em nuvem para o desenvolvimento de qualquer aplicação utilizando o *kit* de desenvolvimento fornecido pela nuvem. Os usuários não precisam instalar o *kit* na máquina local, ele já estará disponível na nuvem [30], possibilitando assim o desenvolvimento de aplicações para nuvens, que utilizam as APIs fornecidas pelos provedores dos serviços, para executarem aplicações customizadas. Esses serviços disponibilizam uma escalabilidade impressionante a desenvolvedores e empresas que oferecem serviços por meio de aplicações web, e assim, aumentam a utilização de recursos de maneira transparente e quase instantânea, de acordo com a utilização do serviço, sem a necessidade de trabalho com aquisição, instalação e configuração de novo hardware, e nem de nova aplicação [34]. Microsoft Azure [6] e Google AppEngine [16] são algumas das aplicações que exemplificam a camada de serviço PaaS.
- Infraestrutura como Serviço (IaaS): ela fornece um ambiente para uma aplicação composto de hardware e software, baseado no acordo de nível de serviço equivalente ao uso da infraestrutura. O valor da cobrança é calculado pelo uso dos recursos da infraestrutura, que pode ser de tempo de processamento, quantidade de dados armazenados e/ou de largura de banda utilizada. O contratante deste serviço não corre o risco de pagar por uma infraestrutura que fique ociosa por várias semanas ou meses, pois só pagará por ela quando for utilizada. Na infraestrutura como serviço ocorre a oferta de componentes de hardware, *firewalls*, serviços de configuração, máquinas virtuais, etc [2]. A *Amazon Elastic Compute Cloud* (EC2) [25] e o GoogleFS [18] são alguns exemplos de ferramentas que fornecem esta camada de serviço.

2.1.3 Tipos de Nuvens

As nuvens podem ser classificadas em três tipos de implementação, conforme demonstrado na Figura 2.3. A escolha de qual é o tipo mais adequado vai depender da necessidade da aplicação que será implementada. Os tipos de implementação de nuvens são públicas, privadas e híbridas [4].

Figura 2.3: Tipos de Nuvens (adaptado de Vmware [40]).

- Nuvens Públicas: na nuvem pública todos os serviços podem ser acessados por qualquer usuário ou organização. As nuvens públicas possuem a vantagem de terem uma maior escalabilidade de recursos, evitando o inconveniente dos usuários terem que comprar equipamentos para a nuvem caso ocorra uma necessidade temporária de um poder de processamento maior. Assim, geralmente, nuvens públicas são uma boa alternativa quando [30]:
  - Se deseja testar e desenvolver o código do aplicativo;
  - Projetos colaborativos estão sendo desenvolvidos;
  - Um aplicativo SaaS na nuvem tem uma estratégia de segurança bem implementada;
  - Uma incrementação do aplicativo é necessária por conta do horário (a capacidade de adicionar mais recursos à nuvem, em horários de pico, para não ocorrer problemas de indisponibilidade do serviço por conta do grande número de acessos).

- Nuvens Privadas: são aquelas projetadas exclusivamente para um único usuário ou organização. Nesse caso, a infraestrutura utilizada pertence ao usuário, assim ele possui total controle sobre os aplicativos que são implementados. Geralmente, elas são construídas sobre *data centers* privados, o que faz com que o usuário tenha que comprar e manter todo o software e infraestrutura da nuvem. Caso o usuário queira aumentar os recursos da nuvem, ele deverá adquirir novos equipamentos, já que a sua nuvem é limitada a sua capacidade física, ao contrário da nuvem pública, onde seus recursos são facilmente escaláveis. É uma boa escolha quando o negócio em questão são os dados e as aplicações da empresa, onde controle e segurança são fundamentais.

- Nuvens Híbridas: são combinações de nuvens públicas com privadas. Uma vantagem é que uma nuvem privada pode ter os seus recursos ampliados a partir de recursos de uma nuvem pública. Apesar dessa vantagem, a complexidade para se determinar

a maneira como as aplicações serão distribuídas não é irrelevante. Uma grande quantidade de dados para serem processados em uma nuvem pública pode não ser favorável, pois pode ser muito custoso passar essa quantidade de uma nuvem privada para uma nuvem pública.

A maioria das empresas servidoras de nuvem possui as características apresentadas nesta Seção 2.1, apesar disso, existem algumas divergências, que serão demonstradas na Subseção 2.1.4.

2.1.4 Comparação entre os Principais Provedores de Nuvem

Na Tabela 2.1 tem-se algumas importantes características de grandes provedores de computação em nuvens, as quais são a Amazon [35], a Microsoft [27] e a Google [16].

Tabela 2.1: Comparação entre provedores de computação em nuvens (adaptado de G. Magalhães [26]).

Serviço	**AWS**	**Windows Azure**	**App Engine**
Provedor	Amazon	Microsof	Google
Lançamento	2002	2010	2008
Categoria	IaaS	PaaS	PaaS
Interface	API e linha de comando	API	API
Licença Comercial	Proprietário	Proprietário	Proprietário
Sistemas Operacionais Compatíveis	Linux e Windows Server 2003 e 2008	Windows Server 2003 e 2008	Linux e Windows Server 2008
Linguagens de programação suportadas	Java, PHP, Python e Ruby	Java, PHP, Python e .NET	Java, Python e Go
Tempo garantido de disponibilidade	99,95%	99,90%	99,90%

Em relação à disponibilidade, que é uma das características essenciais em uma plataforma de nuvem, foi observado que a Amazon oferece o maior tempo com uma pequena porcentagem maior que as outras duas plataformas. Apesar disso, todas as três plataformas tem disponibilidade suficiente para uso em sistemas domésticos ou de empresas, que não dependem totalmente do serviço de computação em nuvem.

Analisando as linguagens de programação suportadas pelas plataformas, percebe-se que o Java está em todas as plataformas. Provavelmente, isto ocorre pelo fato do Java possuir uma portabilidade própria, que faz com que um código fonte escrito uma única vez possa ser executado em várias plataformas diferentes, sem a necessidade de uma adaptação do programa. O PHP foi adotado pela Microsoft e pela Amazon, e isso possibilita um uso na interação dos seus bancos de dados em servidores hospedados na nuvem. A Google App Engine adotou a linguagem Go e Python, que também foram adotadas pela Amazon, ambas linguagens possibilitam a programação paralela [26].

Além dessas diferenças, existem algumas restrições. Na Azure o protocolo ICMP [27] é bloqueado, não permitindo assim, por exemplo, um teste de *ping*, teste o qual pode ser resolvido usando o utilitário para descoberta e auditória segura de redes, o Nmap [28], que utiliza quatro técnicas para escanear o *ping*. A primeira técnica é mandar uma requisição *ICMP*, se não houver resposta da requisição o Nmap tenta um *TCP PING* para determinar se o *host* está *online* ou somente com o protocolo *ICMP* bloqueado mandando uma mensagem do tipo *SYN* ou *ACK* para qualquer porta, por padrão escolhe-se a 80, se o *RST* ou *SYN/ACK* retornarem, então o *host* está *online*.

Um outro aspecto é em relação ao DNS, pois enquanto as máquinas virtuais da Amazon, ao serem desligadas perdem seu IP público e o DNS; na Azure, o DNS é mantido, designando assim um DNS único para as máquinas virtuais, facilitando o acesso via *ssh*, onde o endereço não muda se as máquinas forem desligadas.

Mesmo com todo o potencial ofertado, as nuvens não estão conseguindo suprir o aumento exponencial da quantidade de dados produzidos por pessoas, pois os seus recursos são limitados e os dados continuam crescendo. A partir deste problema surgiu a ideia das federações de nuvens, que buscam agregar recursos de outras nuvens, formando uma arquitetura onde os recursos dão a visão de que são ilimitados. Na Seção 2.2 serão apresentados os conceitos básicos de uma federação de nuvens.

2.2 Federação de Nuvens

Com o objetivo de suportar um grande número de consumidores de serviços de todo o mundo, os provedores de infraestrutura estabeleceram centros de dados em múltiplas localizações geográficas para fornecerem redundância e assegurar confiabilidade em casos de falhas. A Amazon, por exemplo, possui centros de dados nos EUA (um na Costa Leste e outro na Costa Oeste) e na Europa. No entanto, atualmente, eles esperam que seus clientes na nuvem (os provedores de SaaS) expressem uma preferência sobre o local que desejam hospedar os seus dados. Eles não fornecem mecanismos automáticos para escalar os seus serviços hospedados em vários centros de dados distribuídos geograficamente. Este tipo de abordagem apresenta algumas deficiências:

- É difícil para os clientes da nuvem determinarem com antecedência a melhor localização para hospedar os seus serviços;
- Os provedores de SaaS podem não ser capazes de atender as expectativas de qualidade de serviço (*QoS* - *Quality of Service*) dos consumidores que provenham de múltiplas localidades geográficas.

Segundo Buyya [3], isso exige a construção de mecanismos para uma federação de provedores de nuvens, com o objetivo de cumprir metas de qualidade de serviços (*QoS*). Além disso, nenhum fornecedor de infraestrutura na nuvem será capaz de estabelecer o seu *data center* em todos os locais do mundo. Como os fornecedores de aplicativos nas nuvens terão dificuldades de cumprir as expectativas de qualidade de serviço (*QoS*) para todos os consumidores, ocorre a tentativa de realizar o melhor uso dos serviços de múltiplos provedores de infraestrutura na nuvem, para poderem oferecer um suporte melhor para os usuários de acordo com a sua necessidade.

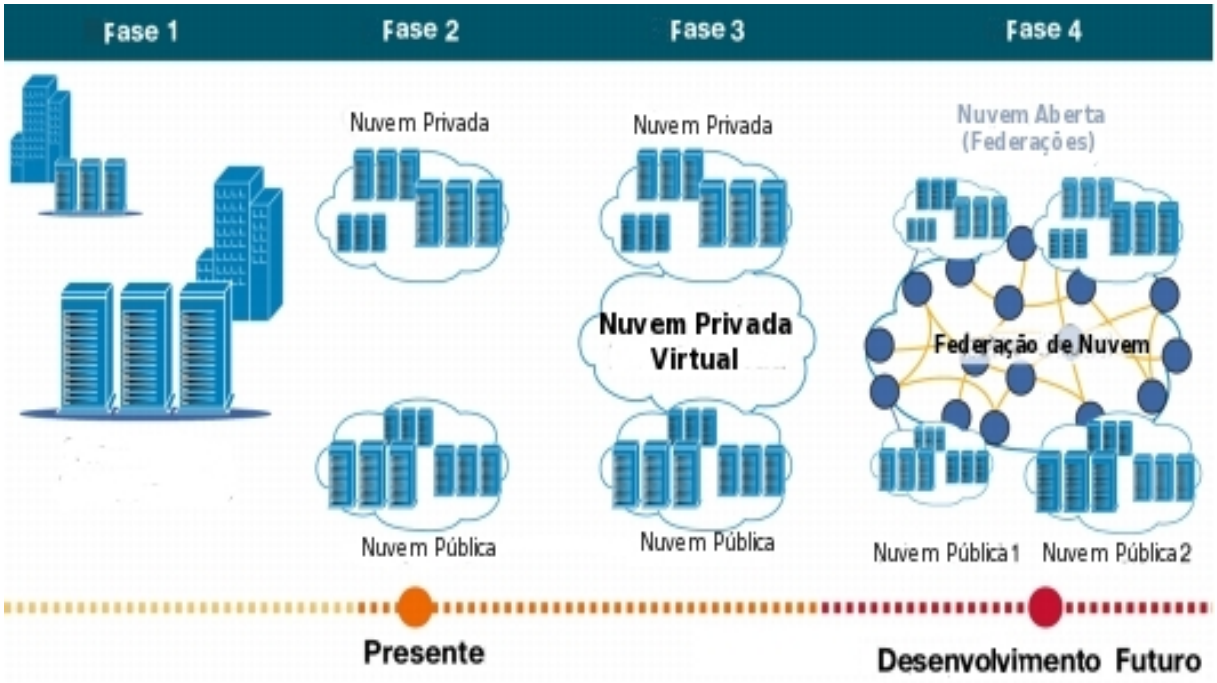

Figura 2.4: As Fases da Computação em Nuvem (segundo White E. et al. [41]).

Nesse cenário, uma federação de nuvens é uma forma de interligar ambientes de computação em nuvem de dois ou mais provedores, com a finalidade de controlar e balancear a demanda computacional. Para isso, é necessário um provedor para prover recursos de uma nuvem para outra, esses podem ser temporários ou permanentes, dependendo do acordo realizado entre as nuvens federadas. Uma vantagem da federação é a utilização destes recursos que estariam ociosos de uma nuvem para a outra, pois não ocorreria nenhum desperdício [31].

A federação de nuvens, em todo o seu conceito, passou por duas fases, sendo que existem mais duas que estão sendo trabalhadas para serem melhoradas e padronizadas. Na Figura 2.4 tem-se as quatro fases. Na primeira nota-se os *data centers* sem a utilização da tecnologia de nuvem, ou seja, eles estão espalhados pelo mundo e realizam todo o trabalho sozinhos, contando apenas com a infraestrutura que dispõem localmente. Não ocorre cooperação com nenhum outro *data center*.

Atualmente, está ocorrendo a segunda fase, na qual encontram-se as nuvens fornecendo os seus serviços. Na fase três, as nuvens já conseguem trocar recursos entre si por meio de um provedor virtual que realiza todo o balanceamento das nuvens, de forma que nenhuma fique ociosa enquanto há outras que se encontram com a utilização máxima de seus recursos. Nesse caso uma solicitação seria enviada para a nuvem com recursos ociosos, e esses seriam utilizados pela nuvem sobrecarregada.

Na última fase, encontram-se as várias federações espalhadas pelo mundo, e essas federações trabalhariam interoperando entre si, ou seja, dividindo os seus recursos da mesma forma que uma nuvem dentro de uma federação faz. Esse conceito da fase final formaria um cenário onde os recursos ociosos e desperdiçados na nuvem seriam pequenos. Uma outra vantagem também seria o possível balanceamento na carga entre as nuvens e federações, fazendo com que os usuários tenham uma ótima experiência de uso das aplicações hospedadas nas nuvens.

Com esse conceito de federação de nuvens apresentado nesta Seção 2.2, foi proposta uma arquitetura de federação em nuvens implementada por Hugo Saldanha, o BioNimbus [34], o qual será detalhado no Capítulo 3.

Em uma arquitetura de federação de nuvens, um dos principais serviços ofertados, é o armazenamento em nuvem, serviço no qual é apresentado alguns conceitos básicos, seguidos de um modelo que o exemplifica na Seção 2.3.

2.3 Armazenamento em Nuvem

Um dos principais usos da computação em nuvem é o armazenamento de dados. Com o armazenamento em nuvem, os dados são gravados em vários servidores, podendo assim optar-se por armazenar em servidores terceiros, em vez de servidores de armazenamento tradicionais dedicados. O local de armazenagem pode variar dia a dia ou até mesmo minuto a minuto, simplesmente pelo fato da nuvem gerenciar dinamicamente o espaço de armazenamento disponível. Apesar da localização dos dados ser virtual, o usuário vê uma localização estática, e pode, realmente, controlar o seu espaço de armazenamento como se fosse o seu próprio computador.

O armazenamento em nuvem possui algumas vantagens financeiras e de segurança contra desastres naturais. Em relação a segurança, os dados armazenados na nuvem são seguros de falhas de hardware ou apagamentos acidentais, porque eles são replicados em várias máquinas físicas. Como múltiplas cópias dos dados são mantidas continuamente, a nuvem continua a funcionar, normalmente, mesmo quando uma ou mais máquinas venham a ficar *off-line*. Esse recurso é chamado de *failover*, ou seja, se uma máquina falhar, os dados replicados em outras máquinas na nuvem assumem essa atividade, sem prejudicar o processo da atividade, tornando assim um serviço confiável e seguros.

Financeiramente, os recursos virtuais na nuvem são, normalmente, mais baratos do que os recursos físicos de um computador pessoal ou de rede [42]. O custo para se manter um centro de armazenamento de dados está aumentando de uma forma drástica, fazendo com que empresas procurem outros meios de armazenar seus dados. Dessa forma, uma opção mais econômica é utilizar o armazenamento da computação em nuvem para esta tarefa, ao invés de gerenciar todos os dados por uma estrutura proprietária e cara, que gasta muito mais em manutenção, tanto do hardware como do software. Assim, é melhor gastar menos com infraestrutura e serviço, comprando esses serviços sob demanda, ou seja, armazenamento em nuvem é simplesmente uma entrega de armazenamento virtualizado sob demanda, o termo formal utilizado para isso é *Armazenamento de Dados como Serviço* (*DaaS*) [36].

Os serviços de armazenamento apresentam diferentes opções. As escolhas variam desde arquivos tradicionais como o *(NFS)*, bancos de dados SQL em *clusters* locais até uma variedade de serviços na nuvem (por exemplo, a Amazon possui o S3, EBS, SimpleDB, RDS e ElastiCache) [32].

Com o passar do tempo a forma de armazenar dados em nuvem veio sofrendo uma constante evolução, como pode ser demonstrado na Figura 2.5, gerando assim uma grande variedade de formas de como se armazenar dados em nuvem.

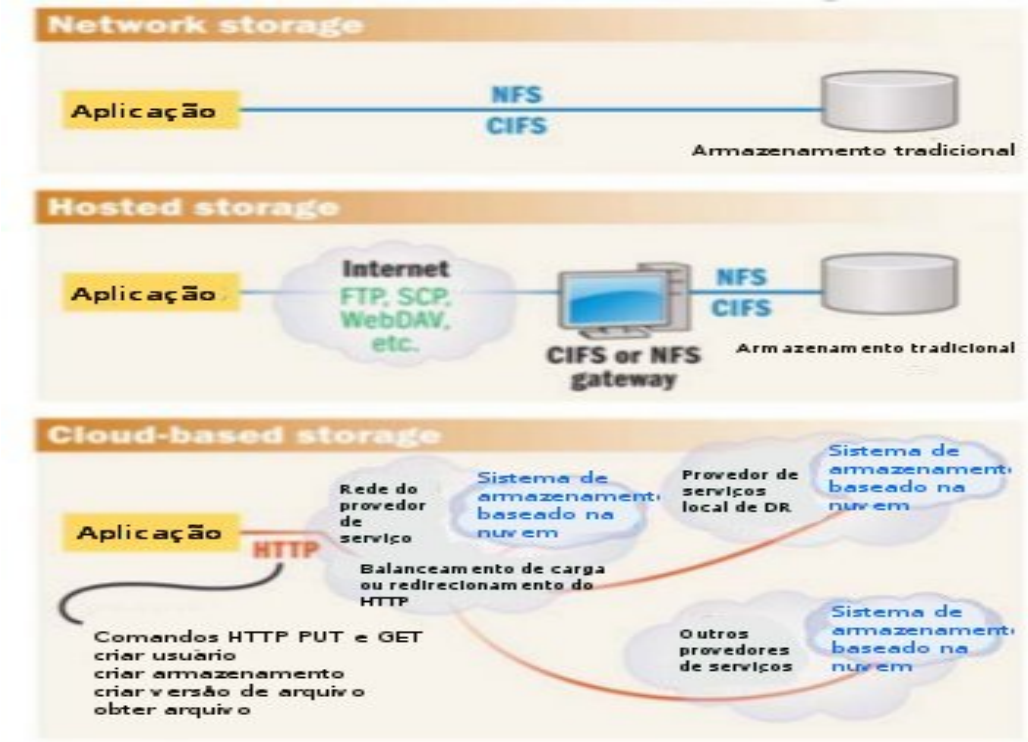

Figura 2.5: Evolução do armazenamento de dados na nuvens (adaptado de J. Wu [42]).

Um modelo de serviço capaz de explicar e definir de forma geral o armazenamento de dados em nuvem, é o *Armazenamento de dados como Serviço*(*Data-storage-as-a-Service - DaaS*), termo usado para explica os serviços ofertados por nuvens para armazenamento, e que engloba os serviços de armazenamento.

2.3.1 *Data-storage-as-a-Service (DaaS)*

Ao abstrair o armazenamento de dados por meio de um conjunto de interfaces de serviço e entregá-los sob demanda, um grande conjunto de ofertas e implementações são possíveis para aplicações de armazenamento em nuvem. O único tipo que é excluído dessa definição é aquele que, em vez de ser entregue sob demanda, é entregue com capacidade fixa de recursos, aplicações dedicadas.

A diferença entre comprar uma aplicação dedicada e uma de armazenamento em nuvem não é simplesmente por ser uma interface funcional, mas também pelo motivo em que o armazenamento é entregue sob demanda. Os clientes pagam pelo o que eles estão utilizando no momento, ou o que eles alocaram para usar. No caso do armazenamento em bloco, a granularidade da alocação é o número de unidade lógica (*Logic Unit Number - LUN*, uma unidade endereçável ou um volume lógico exclusivos e discretos que possam residir dentro de um ou mais dispositivos de armazenamento simples ou de *array* [38]). Para protocolos de arquivos, um sistema de arquivos é a unidade de granularidade. Nos dois casos, o armazenamento pode ser provisionado e cobrado sob demanda pela atual utilização [36].

A gestão destes dados, armazenados nestes serviços de dados, muitas vezes são tipicamentes feitos por um tipo de gerenciamento remoto, permitindo aos administradores de

TI se conectarem ao controlador de gerenciamento de um computador, quando o mesmo estiver nos modos de inatividade, ou hibernação ou se não tiver respondendo por meio do sistema operacional (*out-of-band management*) [38]. Por essas interfaces de padrão de armazenamento de dados, seja por uma *API* ou mais comumente, por uma interface administrativa baseada em navegador pode-se invocar outros serviços de dados como *snapshot* e cópia do sistema (*cloning*).

Do grande conjunto de ofertas e implementações possíveis em nuvem, ao se abstrair o armazenamento de dados, por meio de um conjunto de interfaces e entregando-as sob demanda, deriva-se uma forma de oferta que vem se destacando no mercado por ser uma extensão da forma de como se armazena os dados das aplicações em servidores locais e de possível acesso por meio do SQL, denominada de *Database-as-a-Service*(*DbaaS*), que se diferencia do *DaaS*, justamente por fornecer um meio de acesso aos dados pela linguagem SQL, ofertando assim um banco de dados em nuvem.

2.4 Considerações Finais

Neste capítulo foram descritos os conceitos de computação em nuvens, a sua arquitetura e os seus principais modelos de serviços. Além disso, foram apresentados os problemas encontrados na plataforma da nuvem por conta dos seus recursos limitados. Nesse cenário, discutiu-se também a plataforma de federação de nuvens, que busca implantar a ideia de que os recurso são infinitos.

Dentre os desafios e problemas encontrados em uma federação de nuvens, o armazenamento de dados é um problema em aberto, o qual foi abordado na Seção 2.3, apresentado esses conceitos sobre a computação em nuvens e seus problemas surgiu a ideia de uma plataforma que pudesse tentar resolver esses desafios citados, por este motivo, ela será abordada no Capítulo 3 deste trabalho.

Capítulo 3

Do BioNimbus para o ZooNimbus

Neste capítulo é descrito como a proposta da arquitetura de federação em nuvens desenvolvida por Saldanha [34], o BioNimbus, apresentado na Seção 3.1, foi integrado com o programa Apache ZooKeeper, apresentado na Seção 3.2 e com o Apache Avro, descrito na Seção 3.3, essa integração é explicada explicada na Seção 3.4 e 3.5, o que possibilitou transformar o BioNimbus em ZooNimbus, versão descrita na Seção 3.6.

3.1 BioNimbus

O BioNimbus[33], uma arquitetura para federação de nuvens computacionais híbridas [34], permite a integração e o controle de diferentes provedores de infraestrutura com suas ferramentas de bioinformática oferecidas como serviço, de maneira transparente, flexível e tolerante a falhas, o que significa que provedores independentes, heterogêneos, privados ou públicos de nuvens computacionais podem oferecer seus serviços conjuntamente, mantendo suas características e políticas internas. Neste contexto, o BioNimbus procura oferecer a ilusão de que os recursos computacionais disponíveis são ilimitados e que as demandas dos usuários sempre serão atendidas.

Atualmente, federações de nuvens são implementadas por meio de um *middleware* que permite a interação entre diversas nuvens. A proposta do BioNimbus é justamente ser esse *middleware*, possibilitando a interação entre um ou mais serviços, hospedados em um ou mais provedores de nuvem.

A Figura 3.1 ilustra a arquitetura que permite que um novo provedor seja incluído por meio de um *plug-in* de integração que é responsável por coletar informações sobre os recursos computacionais disponíveis, além de permitir a comunicação entre o provedor e os serviços controladores da federação. Esses serviços são implementados no núcleo do BioNimbus, e oferecem informações sobre os recursos computacionais, tais como, armazenamento de dados, capacidade de processamento e latência na rede.

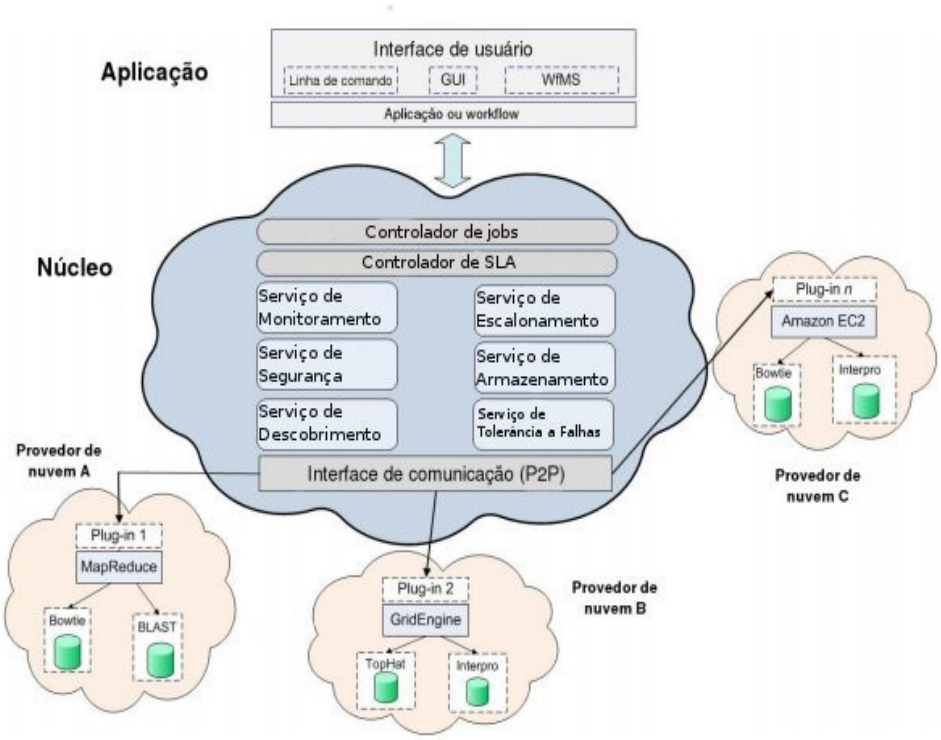

Figura 3.1: BioNimbus uma Arquitetura de Federação de Nuvens Computacionais para Aplicações de Bioinformática (adaptado de [34]).

Para isso, a arquitetura foi projetada de forma que todos os componentes e respectivas funcionalidades sejam bem definidos e divididos, permitindo simplicidade e eficiência na inclusão de novos provedores de nuvens.

3.1.1 Arquitetura do BioNimbus

Como pode ser visto na Figura 3.1, a arquitetura é composta pelos serviços de interação com o usuário (camada de aplicação), por um núcleo (camada de núcleo) e os *plug-ins* utilizados para a comunicação entre o núcleo e a infraestrutura (camada dos provedores de nuvem). Esses serviços são responsáveis pela interação do usuário com a federação. Dessa forma, o usuário interage com esses serviços, que ficam responsáveis pela comunicação com o núcleo e os *plug-ins*. A seguir são detalhados todos esses serviços.

Camada de Aplicação

A interação com o usuário pode ser feita de várias maneiras: páginas web, linhas de comando, interface gráficas, sistemas gerenciados de *workflows*, etc. Essas maneiras são implementadas por meio de serviços de interação que são responsáveis por coletar as ações as quais os usuários desejam fazer, e repassá-las para a federação de nuvens. Além disso, é função desta camada mostrar informações com os *feedback* da execução para os usuários.

Para isso os serviços precisam conectar-se a rede P2P e comunicarem com os serviços do núcleo da arquitetura do BioNimbus. Essa comunicação pode realizar tarefas tais como listar os arquivos armazenados na federação, fazer *upload* de um arquivo ou enviar

um *workflow* para ser executado. Dessa forma, nota-se que a interface com o usuário é responsável por interagir com os usuários, e redija a comunicação com o núcleo.

Camada de Núcleo

O chamado núcleo da arquitetura BioNimbus é responsável pela gerência da federação. O núcleo do BioNimbus possui seis serviços principais: serviço de descobrimento, serviço de monitoramento, serviço de tolerância a falhas, serviço de escalonamento, serviço de segurança e serviço de armazenamento. Além destes, possui mais dois controladores: controlador de *jobs* e controlador de SLA, para exercerem suas funcionalidades, os serviços e os controladores também interagem com os demais componentes da arquitetura por meio de mensagens enviadas pela rede P2P. Esses serviços são [34]:

- **Serviço de descobrimento:** identifica os provedores de serviços e consolida as informações sobre a capacidade de armazenamento, processamento, latência de rede e recursos disponíveis. O serviço de descobrimento é responsável por identificar os provedores de serviço que fazem parte da federação, e consolidar as informações sobre a capacidade de armazenamento e processamento, as ferramentas oferecidas, os detalhes sobre os parâmetros de execução e os arquivos de entrada e saída.

  Para realizar sua função, o serviço de descobrimento envia uma mensagem para todos os provedores da federação, os quais respondem com as informações referentes à infraestrutura e às ferramentas disponíveis. Para consolidar esses dados o serviço de descobrimento mantém uma estrutura de dados que é atualizada para cada nova resposta dos provedores. Além disso, o serviço remove dessa estrutura os provedores que não respondem as requisições.

  Finalmente, tendo realizado a consolidação das informações sobre o estado atual da federação de nuvens, o serviço de descobrimento é capaz de atender às requisições enviadas pelos demais componentes da federação;

- **Serviço de monitoramento:** verifica se um serviço requisitado está disponível no provedor de nuvem, procura por um novo provedor na federação caso não esteja disponível, e garante que todas as tarefas de um processo sejam realmente executadas. Além disso, informa ao serviço de escalonamento quando uma tarefa finaliza sua execução;

- **Serviço de armazenamento:** coordena a estratégia de armazenamento dos arquivos consumidos e produzidos pelas tarefas executadas no BioNimbus, decidindo sobre a distribuição, replicação e controle de acesso dos arquivos entre os diferentes serviços.

  O serviço de armazenamento é responsável por coordenar a estratégia de armazenamento dos arquivos a serem consumidos ou produzidos pelas tarefas executadas na federação. Para isso, o serviço deve decidir sobre a distribuição e a replicação dos arquivos entres os diferentes provedores.

  Para realizar esta função o serviço de armazenamento pode comunicar-se com o serviço de descobrimento para obter acesso as informações sobre a federação. Com isso, o serviço saberá as condições atuais de armazenamento de cada um dos provedores que faz parte da federação.

Quando o serviço de armazenamento recebe um pedido para gravar um arquivo, ele deve decidir em qual federação o arquivo será, efetivamente, armazenado. Para isso uma estratégia de armazenamento é definida, de forma que essa escolha receba as informações sobre o arquivo (contidas na classe *FileInfo*) e retorne o provedor (contido na classe *PluginInfo*) no qual o arquivo será armazenado.

Além disso, esse serviço mantém uma tabela com os arquivos armazenados na federação. Essa tabela é responsável por mapear o identificador de um arquivo ao local, e a federação no qual o arquivo se encontra, de fato, armazenado.

- **Serviço de escalonamento:** distribui dinamicamente as tarefas entre os provedores de nuvem. Para isso, mantém um registro de todas as tarefas alocadas. Além disso, controla a carga de cada provedor da nuvem e redistribui as tarefas quando os recursos estão sobrecarregados.

  O serviço de escalonamento é responsável por receber os pedidos de execução de tarefas submetidas pelos usuários da federação, mantendo um registro com o estado de execução de cada um deles.

  Antes da tarefa ser enviada para um provedor ela passa pelo serviço de escalonamento, que utiliza uma ou mais política de escalonamento para decidir em qual provedor a tarefa deve ser executada. Cada política recebe uma lista de tarefas a serem escalonadas, e retorna um mapeamento das tarefas e dos respectivos provedores que ficarão responsáveis por executá-la. Para isso, a política pode comunicar-se com o serviço de descobrimento para obter acesso às informações da federação.

  Por tratar-se de uma arquitetura de federação de nuvens, a política de escalonamento deve considerar vários aspectos, como por exemplo a latência, o tempo na fila de espera de uma tarefa, a capacidade de processamento, os serviços disponíveis, etc.

- **Serviço de segurança:** garante a segurança dos usuários do sistema, implementando um sistema de *login* e não permitindo que um usuário acesse pastas e tarefas de outro usuário.

- **Controlador de jobs:** é responsável por fazer a ligação entre o núcleo da arquitetura e a camada de aplicação. Uma de suas atribuições consiste em realizar o controle de acesso dos usuários que tentam acessar a federação para realizar pedidos de execução. O controlador acessa o serviço de segurança para fazer uma verificação nas credenciais do usuário. Além disso, o controlador é responsável por gerenciar os pedidos dos vários usuários, de forma a fazer o controle por usuário, e manter os resultados para posterior consulta.

- **Controlador de SLA:** segundo Buyya [43], um *Service-level Agreement* (SLA) é um contrato formal entre provedores de serviço e consumidores para garantir que as expectativas de qualidade de serviço do usuário sejam atingidas. O controlador de SLA no BioNimbus é responsável por implementar o ciclo de vida de um SLA, o qual possui seis passos: descoberta de provedores de serviço, definição de SLA, estabelecimento do acordo, monitoramento de violação do acordo, término de acordo e aplicação de penalidades por violação. Para identificar os parâmetros do acordo, são usados *templates*. Um *template* de SLA representa, entre outras coisas, os parâmetros de QoS que um usuário negociou com a arquitetura. Ele preenche esse

template por meio da camada de interação com o usuário com os valores necessários, os quais podem descrever requisitos funcionais - tais como número de núcleos de CPU, frequência de CPU, tamanho de memória, versão mínima de aplicações ou tamanho de armazenamento; e não funcionais - como tempo de resposta, custo a pagar, taxa de transferência de arquivos, disponibilidade ou tempo máximo de execução.

O controlador de SLA tem a responsabilidade de investigar se os requisitos especificados no *template* preenchido pelo usuário podem ser suportados pela federação de nuvens naquele dado momento. Para tomar essa decisão, o controlador utiliza os dados de SLA informados pelo *plug-in* de integração de cada provedor. Para cada pedido de execução feito pelo usuário para a arquitetura BioNimbus, a negociação SLA procede da seguinte forma:

- O usuário faz seu pedido com um *template* preenchido com os parâmetros de SLA desejados à arquitetura;
- O controlador de SLA cria uma sessão de negociação, se ela ainda não existe, e repassa o pedido de execução para o serviço de monitoramento com os requisitos mínimos extraídos do *template*;
- O serviço de monitoramento requisita, então, um escalonamento para o escalonador de tarefas repassando aqueles parâmetros, e aguarda sucesso ou falha;
- Em caso de falha, o processo é reiniciado com um novo *template* SLA até que se chegue a um acordo.

Após a negociação de um acordo ter obtido sucesso, o controlador de SLA mantém a sessão do acordo por meio de um identificador único, retornando-o para o controlador de jobs, que inclui essa informação na sessão do usuário. Com o pedido do usuário em execução, é preciso acompanhá-lo de forma que o acordo não seja violado. Essa é uma das responsabilidades do serviço de monitoramento. Uma interface realiza a integração destes serviços na arquitetura, e pode ser utilizada para adicionar novas e não previstas funcionalidades à arquitetura, sem impactar no seu funcionamento, e aumentando sua flexibilidade. Esta interface exige que algumas funções operacionais sejam implantadas, como inicialização, atualização de estado e finalização do serviço, bem como oferece a sinalização de eventos na rede P2P.

Camada de Infraestrutura

Os *plug-ins* tem como objetivo serem a interface de comunicação entre o seu respectivo provedor de serviço e os demais componentes da arquitetura BioNimbus. Para que a comunicação seja feita com sucesso, o *plug-in* precisa mapear as requisições vindas dos componentes da arquitetura para ações correspondentes a serem realizadas na infraestrutura do provedor de serviço. Por isso, para cada tipo de infraestrutura é preciso haver uma implementação diferente do *plug-in*.

O *plug-in* necessita tratar três tipos de requisição, as quais são informações sobre a infraestrutura do provedor de serviço, gerenciamento de tarefas e transferência de arquivos, o último será detalhado na Subseção 3.1.3.

3.1.2 Classes do BioNimbus

A seção anterior mostrou uma visão macro do BioNimbus, porém para compreender a arquitetura é necessário aprofundar alguns detalhes de implementação, para isso algumas classes de informações são detalhadas a seguir:

- PeerInfo: informações referentes a um servidor, que é um membro da rede P2P. Nessa classe são armazenadas informações, tais como o endereço de rede, a latência de rede e o tempo que está *on-line* na rede;
- PluginInfo: informações referentes a um *plug-in* e o provedor referenciado por ele. Nessa classe são armazenadas informações como o *PeerInfo* do provedor, a quantidade total de CPUs, a quantidade de CPUs livres, a capacidade de armazenamento e o tamanho de armazenamento livre;
- ServiceInfo: informações referentes a um serviço oferecido por um provedor que faz parte da federação. Essa classe armazena informações como o nome da ferramenta, os parâmetros de configuração, os arquivos de entrada e os arquivos de saída;
- JobInfo: informações referentes a um trabalho, ou seja, uma requisição de execução de um serviço na federação. Nessa classe são armazenadas informações como o identificador do serviço solicitado e os parâmetros para execução do serviço;
- TaskInfo: informações referentes a uma tarefa, ou seja, uma instância de um trabalho. Essa classe representa uma execução de um trabalho em um determinado *plug-in*. Nessa classe são armazenadas informações, como o *JobInfo*, que possui dados sobre o trabalho a ser executado, e o *PluginInfo* com informações do local em que será executado. Além disso, essa classe armazena o estado de execução da tarefa;
- FileInfo: informações referentes a um arquivo armazenado na federação. Nessa classe são armazenadas informações como, por exemplo, o nome e o tamanho do arquivo;
- PluginFileInfo: informações referentes a uma instância de um arquivo que pode estar armazenado em vários provedores da federação. Essa classe armazena informações como o *FileInfo* do arquivo, e o *PluginInfo* do local onde o arquivo está armazenado.

Essas classes armazenam informações essenciais para a execução do BioNimbus, e são utilizadas na comunicação entre os componentes da arquitetura.

3.1.3 Serviço de Armazenamento no BioNimbus

O serviço de armazenamento é responsável por coordenar a estratégia de armazenamento dos arquivos a serem consumidos ou produzidos pelas tarefas executadas na arquitetura BioNimbus, decidindo sobre a distribuição e a replicação dos arquivos entre os diferentes provedores.

Para realizar essa função, primeiramente, o serviço de armazenamento tem acesso às informações sobre a federação, enviando ao serviço de descobrimento mensagens do tipo

CloudReq. Com isso, o serviço saberá as condições atuais de armazenamento de cada um dos provedores que faz parte da federação.

Quando um pedido de armazenamento de arquivo é feito, a decisão do local para onde efetivamente será feito o *upload* do arquivo é realizada pela estratégia configurada por meio da interface *PluginInfo chooseStorage(FileInfo file)*.

Uma estratégia de armazenamento deve implementar esta interface de forma que o serviço de armazenamento repasse as informações sobre o arquivo (dados em *FileInfo*) e ela retorne o provedor de infraestrutura (dados em *PlugInfo*), para onde será feito o *upload* do arquivo baseado em algoritmo próprio e os dados obtidos do serviço de descobrimento. Assim, a arquitetura BioNimbus possibilita a utilização de diferentes estratégias de armazenamento.

Além disso, é esse serviço que mantém uma tabela com os arquivos armazenados na federação, com seus respectivos identificadores e localização. Essa tabela, chamada *File-Tables* pode ser persistida de uma ou mais maneiras. A cada armazenamento confirmado por um *plug-in* de integração, após um *upload* de sucesso, o serviço de armazenamento percorre sua coleção de tabelas, acionando em cada uma delas a interface, *void storeFile(ID id, FileInfo file, List<PluginFileInfo> pluginFiles)*. Assim, cada tabela é responsável por mapear o identificador de um arquivo às suas informações originais e aos locais onde ele está armazenado. Para realizar essas suas funções, o serviço de armazenamento receberá três grupos de mensagens, as quais deverá atender. O primeiro grupo trata do processo de decisão do local de armazenamento de arquivo:

- StoreReq: requisição de escolha do local de armazenamento de um arquivo. Deve conter dados da classe *FileInfo* para que sejam usados pela estratégia de armazenamento;
- StoreReply: resposta do serviço de armazenamento indicando o local de armazenamento para o *peer* que solicitou o armazenamento. Deve conter, além dos dados da requisição, dados da classe *PluginInfo*;
- StoreAck: mensagem enviada por um *plug-in* de integração após a finalização do *upload* de um arquivo à sua infraestrutura. Ela contém dados da classe *PluginFile-Info* que serão armazenados nas tabelas de arquivo do serviço de armazenamento.

O segundo grupo de mensagens refere-se às consultas sobre os arquivos armazenados na federação de nuvens:

- ListReq: requisição de listagem de arquivos armazenados na federação de nuvens. Ela não contém nenhum conteúdo adicional;
- ListReply: resposta do serviço de armazenamento com uma coleção de dados da classe *File-Info*, que representa a consolidação de todos os arquivos mantidos na federação naquele momento.

O último grupo de mensagens enviadas ao serviço de armazenamento refere-se à indicação do local de armazenamento de um arquivo, a partir do qual poderia ser feito um *download*:

- GetReq: requisição sobre o local de onde pode ser feito o *download* de um dado arquivo. Deve conter, pelo menos, o identificador único do arquivo;

- GetReply: resposta do serviço de armazenamento com dados da classe *PluginInfo* sobre o provedor, para onde deve ser enviada uma mensagem do tipo *FilePrepReq* para iniciar o processo de *download*.

Para definir estratégias de armazenamento eficientes para uma federação de nuvens, é necessário, primeiramente, conhecer as características dos dados. Neste trabalho foram considerados as características dos dados biológicos, entre elas:

- Normalmente, ocupam enormes volumes de armazenamento;
- Durante as execuções de ferramentas de bioinformática não há atualização simultânea de dados por mais de um usuário;
- Fragmentação e replicação podem ser feitos de diferentes modos, dependendo da aplicação;

Assim, considerando essas características e o ambiente de federação de nuvens, uma alternativa é a utilização de bancos de dados NoSQL (*Not only SQL*), tais como o HBase [15], o Cassandra [10] ou o MongoDB [5], pois além de suas características específicas para ambiente em nuvem e de larga escala, possuem características que permitem a conjugação de dados (arquivos biológicos) com conjuntos de informações (proveniência de dados).

Para as funcionalidades de replicação e de fragmentação, podem ser projetados mecanismos que estejam em uma camada acima da infraestrutura de armazenamento. Tais mecanismos devem levar em conta parâmetros como formato dos dados, tamanho de armazenamento disponível, taxa de transferência da rede, localização geográfica, entre outros. Com o intuito de diminuir a quantidade de dados transferidos entre os membros da federação, e aumentar o paralelismo nas execuções das aplicações.

Essa arquitetura integrada com os programas Apache Zookeeper [14], como coordenador de tarefas, e Apache Avro [11], como responsável pela comunicação entre os computadores envolvidos na arquitetura, fizeram com que uma reestruturação no BioNimbus fosse feita para suprir algumas necessidades no sistema. O Apache ZooKeeper e o Apache Avro são apresentados na Seção 3.2 e na Seção 3.3.

3.2 Apache ZooKeeper

ZooKeeper [14] é um serviço de coordenação distribuída, de código-aberto para aplicações distribuídas. ZooKeeper é simples e permite que os processos distribuídos coordenem-se por meio de um espaço de nomes hierárquico, compartilhado, que é organizado de forma semelhante a um sistema de arquivos padrão.

Conforme mostra a Figura 3.2, o ZooKeeper é composto por vários servidores, sendo que um deles é o líder e os outros são chamados de seguidores. Caso o servidor líder desconecte-se do ZooKeeper, um algoritmo de eleição é executado novamente para que dentre os servidores restantes, um deles torne-se o novo líder.

Os dados do ZooKeeper são chamados *znodes*, e estes são semelhantes aos arquivos e diretórios. Ao contrário de um sistema de arquivo comum, que é projetado para o armazenamento, os dados do ZooKeeper são mantidos na memória, o que faz com que ele consiga números baixos de latência e uma alta taxa de transferência. ZooKeeper é

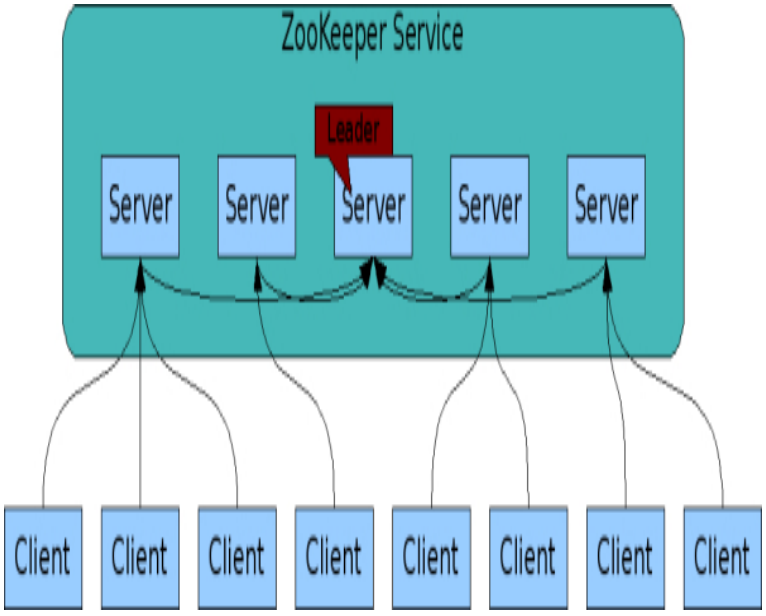

Figura 3.2: Modelo de Serviço do Apache Zookeeper[14].

replicado, como os processos distribuídos que coordena, o próprio ZooKeeper se destina a ser replicado ao longo de um conjunto de máquinas [14].

O ZooKeeper mantêm uma imagem na memória, juntamente com os *logs* de transação e *snapshots*, em um armazenamento persistente. Os *snapshots* são imagens que o ZooKeeper faz de todos os *znodes* presentes em sua estrutura. O ZooKeeper escreve os *snapshots* no sistema de arquivos, ele não realiza a exclusão dos *snapshots*, isto é uma responsabilidade do operador do sistema.

Os clientes conectam-se a um único servidor ZooKeeper. O cliente mantém uma conexão TCP através da qual se envia pedidos, recebe respostas, recebe eventos de *watchers*, e envia *heartbeats*. Se a conexão TCP entre o cliente e o servidor cai, o cliente irá se conectar a um servidor diferente. *Heartbeats* são pacotes que o ZooKeeper envia para os servidores, esperando respostas de cada um. Ele faz isso para saber se todos os servidores estão *on-line* no sistema. O tempo de espera para uma resposta de um servidor é, por padrão, definido em 2s.

As operações no ZooKeeper são ordenadas, são utilizados selos para cada atualização com um número que reflete a ordem de todas as transações ZooKeeper. Operações subsequentes podem ser usadas a fim de implementar abstrações de nível superior, como primitivas de sincronização. O ZooKeeper é especialmente rápido em cargas de trabalho com ambientes onde operações de leitura são dominantes. Aplicações ZooKeeper são executadas em milhares de máquinas, e ele executa melhor, onde as leituras são mais comuns do que as escritas, em proporções de cerca de 10:1.

O espaço de nomes fornecido pelo ZooKeeper é muito parecido com o de um sistema de arquivos padrão. Um nome é uma sequência de elementos de caminho separados por uma barra (/). Cada nó em nome do espaço ZooKeeper é identificado por um caminho. Na Figura 3.3 tem-se um exemplo de como funciona a hierarquia de *Znodes* do ZooKeeper.

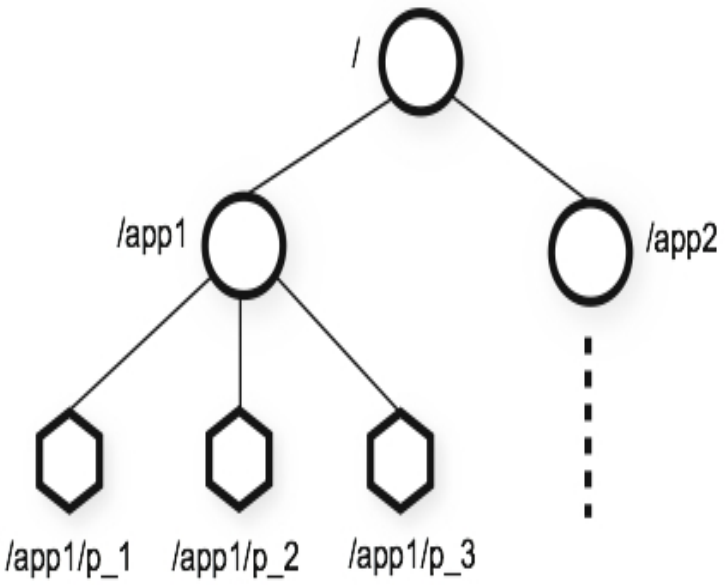

Figura 3.3: Hierarquia de ZNodes no ZooKeeper.

3.2.1 *Znodes* Persistentes e *Znodes* Efêmeros

Cada *znode* em um *namespace* no ZooKeeper pode ter dados associados a ele, bem como os seus filhos. É como ter um sistema de arquivos que permite que um arquivo seja também um diretório. ZooKeeper foi projetado para armazenar dados de coordenação: informações de *status*, configurações, informações de localização, etc. Os dados armazenados em cada *znode* são, geralmente, pequenos, na faixa de *byte* para *kilobyte*. Cada *znode* armazena no máximo 1Mb de dados.

Znodes mantêm uma estrutura estatística que inclui números de versão para alterações de dados, mudanças de lista de controle de acesso (*Access Control List* - ACL) e carimbos de tempo, para permitir validações de cache e atualizações coordenadas. Cada vez que os dados de um *znode* são alterados, o número da versão aumenta. Sempre que um cliente recupera dados recebe também a versão destes dados.

Os dados armazenados em cada *znode* em um *namespace* é lido e escrito atomicamente. Operações de leitura obtêm todos os *bytes* de dados associados com um *znode*, e uma operação de gravação substitui todos os dados. Cada nó tem uma ACL, que restringe quem pode fazer o que com cada *znode*. O ZooKeeper tem também a noção de *znodes* efêmeros. Estes *znodes* existem enquanto a sessão que criou o *znode* está ativa. Quando a sessão termina o *znode* é excluído.

3.2.2 *Watchers*

ZooKeeper [14] suporta o conceito de *watchers*. Os clientes podem definir um *watcher* em um *znode*. Um *watcher* é uma espécie de aviso e serve para dizer que um *znode* foi alterado ou deletado no ZooKeeper. Quem insere o *watcher* em um determinado *znode* são clientes que desejam saber da situação deste *znode*, sendo assim, quando esse *znode* for deletado ou tiver seus dados alterados, somente quem solicitou receber um *watcher* deste *znode* que irá receber o aviso.

Watchers são inseridos apenas uma vez, ou seja, depois que ele foi disparado para os clientes que solicitaram receber aviso de um *znode*, ele deixa de existir, sendo então necessário que outro *watcher* seja inserido novamente no mesmo *znode*, caso se queira saber das operações futuras que possam acontecer com ele.

3.2.3 Garantias

ZooKeeper é muito rápido e simples. Uma vez que o seu objetivo é ser uma base para a construção de serviços mais complexos, como a sincronização. ZooKeeper fornece um conjunto de garantias, que são as seguintes [14]:

- Consistência sequencial: atualizações de um cliente serão aplicadas na ordem em que foram enviadas;
- Atomicidade: atualizações retornam sucesso ou fracasso, ou seja, não há resultados parciais;
- Sistema único de imagem: todos os clientes vão ter o mesmo ponto de vista do serviço, pois quando qualquer alteração for feita no ZooKeeper, todos que estão conectados a ele irão obter uma imagem atualizada dos dados do ZooKeeper com as alterações feitas, independentemente do servidor em que ele estiver conectado;
- Confiabilidade: uma vez que uma atualização foi aplicada, ele irá persistir a partir daquele momento, até que um cliente substitua a atualização.

3.2.4 API Simples

Um dos objetivos do ZooKeeper é fornecer uma interface de programação muito simples. Como resultado, ele suporta apenas as seguintes operações [14]:

- *create*: cria um *znode* numa localização na árvore do ZooKeeper;
- *delete*: realiza a exclusão de um *znode* no ZooKeeper;
- *exist*: testa se existe um *znode* em um determinado local;
- *getdata*: faz a leitura dos dados a partir de um *znode*;
- *setdata*: grava dados em um *znode*;
- *getchildren*: recupera uma lista de filhos de um determinado *znode*;
- *sync*: sincroniza os dados no ZooKeeper a partir do ponto de vista de um cliente.

3.3 Apache Avro

Apache Avro [11] é um sistema de serialização de dados e de chamadas remotas de procedimentos (RPC - *Remote Procedure Call* [23]) desenvolvido dentro do projeto Hadoop[12] da Apache. Ele é um sistema que oferece:

- Estrutura de dados rica;

- Formato compacto e rápido de dados binários;
- Um *container* de arquivos para armazenar dados persistentes;
- Integração simples com linguagens dinâmicas. Não é necessária a geração de código para ler e escrever arquivos de dados, nem para usar ou implementar protocolos RPC. Geração de código funciona como uma opção de otimização, só vale a pena implementar para linguagens de tipagem estática.

3.3.1 Esboços

Quando os dados do Avro são lidos, o esboço utilizado na escrita deles estão sempre presentes com os dados. Isso permite que cada dado escrito não tenha despesas de valor, tornando a serialização de dados rápida e pequena, pois os dados, juntos com os seus esboços, são totalmente auto-explicativos, isto facilita o uso de linguagens de *script* dinâmicas. Se o programa de leitura dos dados espera um esboço diferente, isso pode ser facilmente resolvido, uma vez que ambos os esboços estão presentes.

Os esboços Avro são escritos em um arquivo *.avdl* que contêm as funções utilizadas que são chamadas remotamente. Na Figura 3.4 encontra-se um exemplo de um arquivo *.avdl*.

```
276
277     string cancelJob(string jobID);
278
279       record NodeInfo {
280       string peerId;
281       string address;
282       float freesize;
283       double latency;
284     }
285
286     record FileInfo{
287       string fileId;
288       long size;
289       string name;
290     }
291
292      //Set information file that will be uploaded
293      void setFileInfo(FileInfo file, string kindService) oneway;
294
295     //Get peers from server
296     array<NodeInfo> getPeersNode();
297
298     //Enviar plugins
299     // void sendPlugins(array<NodeInfo> plugins) oneway;
```

Figura 3.4: Exemplo de Arquivo *.avdl* Utilizado pelo Avro no ZooNimbus.

O arquivo contêm os metadados dos dados que são chamados remotamente. Quando os dados Avro são armazenados em um arquivo, seu esboço é armazenado junto com ele, de forma que os arquivos possam ser processados por qualquer outro programa futuramente.

Quando o Avro é usado em RPC, o cliente e o servidor usam trocas de esboços de dados quando uma conexão é estabelecida. Uma vez que o cliente e o servidor têm esboços do outro, a correspondência entre os mesmos campos nomeados, campos em falta, campos extras, etc, podem ser facilmente resolvidas. Logo, esboços Avro são definidos com JSON [7]. Isso facilita a implementação em linguagens que já possuem bibliotecas JSON.

3.3.2 Comparando o Avro com outros Sistemas

O Avro oferece serialização de dados por meio de chamadas RPC, semelhante a outros sistemas, como Thrift [13], Protocol Buffers [20], etc. Contudo, o Avro difere destes sistemas nos seguintes aspectos fundamentais:

- Tipagem dinâmica: Avro não requer que o código seja gerado. Os dados são sempre acompanhados por um esboço, que permite o processamento completo desses dados, sem a geração de código, tipos de dados estáticos, etc. Isso facilita a construção de sistemas de processamento de dados e linguagens de genéricos.
- Dados sem etiqueta: desde que o esboço esteja presente quando os dados são lidos, informações consideravelmente de menor tipo precisam ser codificadas com os dados. O Avro fornece um esquema com dados binários que permite que cada dado seja escrito sem sobrecarga, resultando um uma codificação de dados mais compacta, e um processamento dos dados mais rápido.
- Atribuição automática de ID's de campos: quando um esboço muda, o antigo e o novo estão sempre presentes durante o processamento dos dados, de modo que as diferenças podem ser resolvidas simbolicamente, usando nomes de campo.

3.4 Integração com o Apache ZooKeeper

O Apache ZooKeeper [14] foi integrado ao BioNimbus para fornecer uma nova forma de troca de informações entre os servidores e clientes. As funções P2P presentes no BioNimbus foram substituídas pela centralização de dados oferecida pelo ZooKeeper. Com o uso do sistema P2P, os servidores realizavam troca de mensagens entre si por meio da rede P2P, além disso, tinham a necessidade de um servidor ficar enviando mensagens em *broadcast* para conhecer todos os outros servidores da federação. Com a integração do ZooKeeper, todos os servidores realizam consultas de dados dos outros servidores dentro ZooKeeper.

Toda vez que um servidor é iniciado dentro da federação, é criado um *znode* dentro do ZooKeeper com os seus dados. Isto fornece uma alta centralização no controle dos dados, visto que o ZooKeeper é quem irá armazenar os dados dos servidores. Assim sendo, quem for consultar ou atualizar estes dados, fará estas operações dentro do próprio ZooKeeper.

Um servidor ZooKeeper é iniciado em um servidor e todos os outros comunicam-se com o ZooKeeper por meio da porta 2181, conforme mostra a Figura 3.5. Caso o servidor com o serviço ZooKeeper caia, um algoritmo de eleição é executado pelo ZooKeeper para escolher qual o servidor ZooNimbus que será o novo líder. A partir dai, qualquer mudança de dado ou leitura é feita neste servidor. Para consultar os dados que estão inseridos é necessário iniciar o cliente do ZooKeeper e se conectar ao servidor ZooKeeper.

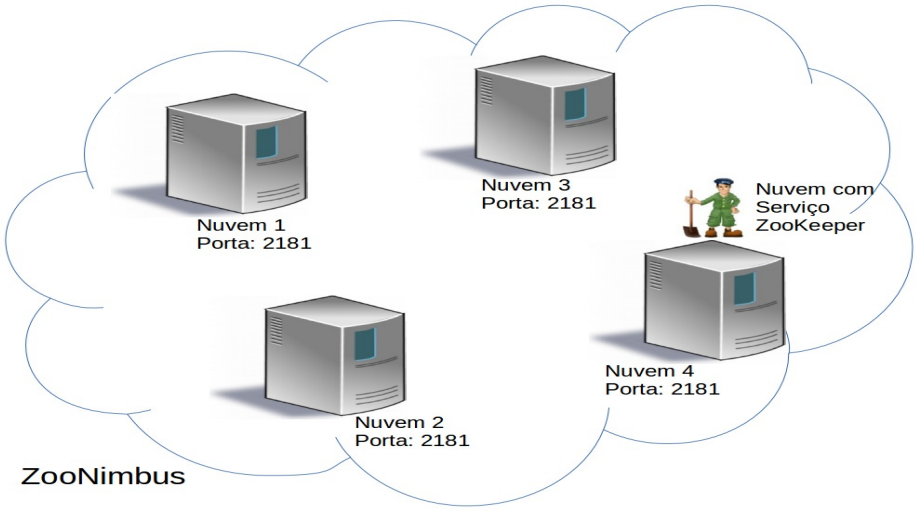

Figura 3.5: Exemplo de Conexão entre o ZooNimbus e o ZooKeeper.

O ZooNimbus utiliza, com frequência, os *watchers* do ZooKeeper, para que todos os outros servidores fiquem sabendo quais dados de um *znode* foram alterados ou deletados. Para se ter um controle dos arquivos que são inseridos na federação, o ZooNimbus cria um *znode*, chamado *pending saves*. Neste *znode* são armazenados dados de todos os arquivos que possuem uma requisição de *upload* e que irão ser inseridos no ZooNimbus.

As requisições de *upload* podem ocorrer por um cliente ou por um servidor. No caso de uma requisição por parte do cliente, o processo começa com a requisição de um *upload*, logo depois, a política de armazenamento decide o melhor servidor para receber o arquivo. Neste momento é adicionada na *pending saves*, os dados do arquivo que será enviado ao ZooNimbus, esses dados só são apagados depois que for verificado que o arquivo foi enviado com sucesso.

No caso da replicação de dados, é adicionado na *pending saves*, um *znode* com os dados do arquivo que será replicado. Assim, o servidor que ficar responsável pela duplicação dos dados irá replicar o arquivo na quantidade de cópias definidas, após isso ele irá deletar o *znode* do arquivo replicado. A quantidade de cópias é definida no *Storage Service*, por meio da variável **replicationfactor**. Inicialmente, definiu-se duas cópias para cada arquivo, porém esse valor pode ser facilmente alterado. O servidor responsável conversa com todos os outros servidores por meio de RPC, utilizando para isso o Apache Avro.

3.5 Integração com o Apache Avro

A comunicação entre os servidores e os clientes no BioNimbus era realizada por meio de mensagens enviadas pela arquitetura de rede P2P. Esta forma de comunicação foi substituída pelas chamadas RPC [23]. O Apache Avro [11] trabalha com RPCs fornecendo uma interface de comunicação mais simples e robusta ao ZooNimbus. Como o Avro é um programa que trabalha com recursos de rede para gerar e mandar as RPCs, ele não consegue garantir que uma chamada seja executada corretamente, caso ocorra alguma problema na rede no momento em que uma chamada é realizada. O Avro gerou um arquivo

.avdl dentro do ZooNimbus, onde estão implementados os métodos que são chamados remotamente, tanto por clientes quanto por servidores.

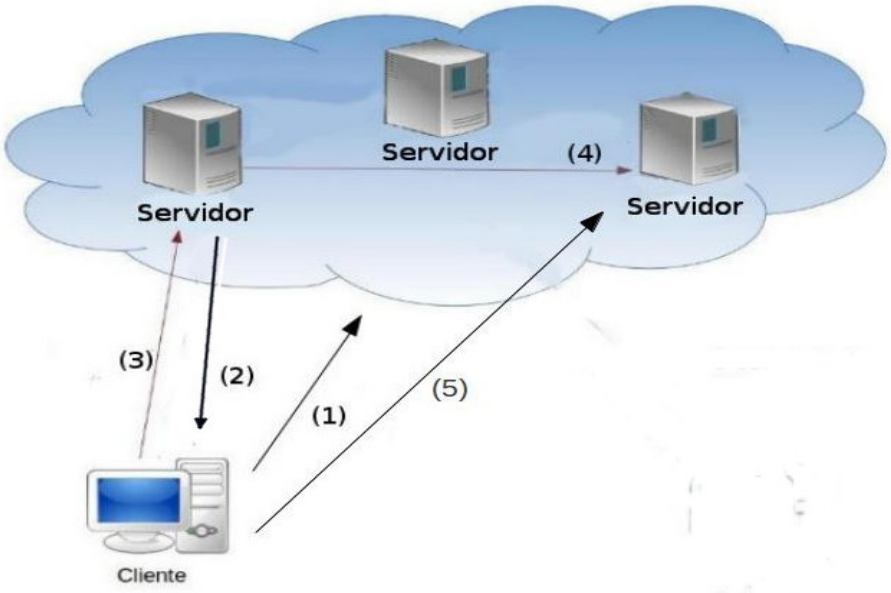

Figura 3.6: Chamadas RPC Passo-a-Passo no ZooNimbus.

A Figura 3.6 mostra, de forma simples, como ocorre a utilização do Avro no ZooNimbus no caso de uma operação de *upload*. Como mostrado na Figura 3.6 (1) o cliente faz uma conexão com um servidor na federação e (2) recebe uma lista com os servidores disponíveis para receber o arquivo, (3) depois o cliente calcula a latência entre ele e os servidores e envia uma chamada RPC para o servidor em que ele está conectado com os dados que ele gerou e que serão utilizados no cálculo da política de armazenamento. Após esse servidor receber os dados, decide qual o melhor servidor para colocar o arquivo com base nos cálculos da ZooClouS a ser descrito no próximo capítulo, (4) e faz uma chamada RPC para este servidor com os dados do arquivo que o cliente pretende enviar para o ZooNimbus. Após isso, (5) o cliente abre uma conexão com o servidor de destino e envia o arquivo. Todo o processo de cálculo da política e do destino do arquivo é detalhado na Seção 4.2.

3.6 A Nova Versão do BioNimbus - ZooNimbus

O BioNimbus teve várias modificações na sua estrutura original, que foi proposta por Hugo Saldanha [34]. Porém essas alterações não mudaram os objetivos e as propostas iniciais da arquitetura de federação em nuvens. Assim, o principal objetivo do ZooNimbus é a oferta de ferramentas como um serviço em uma plataforma de federação em nuvens, de maneira transparente, flexível e tolerante a falhas com grande capacidade de processamento e armazenamento. Assim, a nova versão, ZooNimbus, continua provendo uma plataforma que constrói uma federação em nuvem que executa serviços de bioinformática com recursos heterogêneos, públicos ou privados.

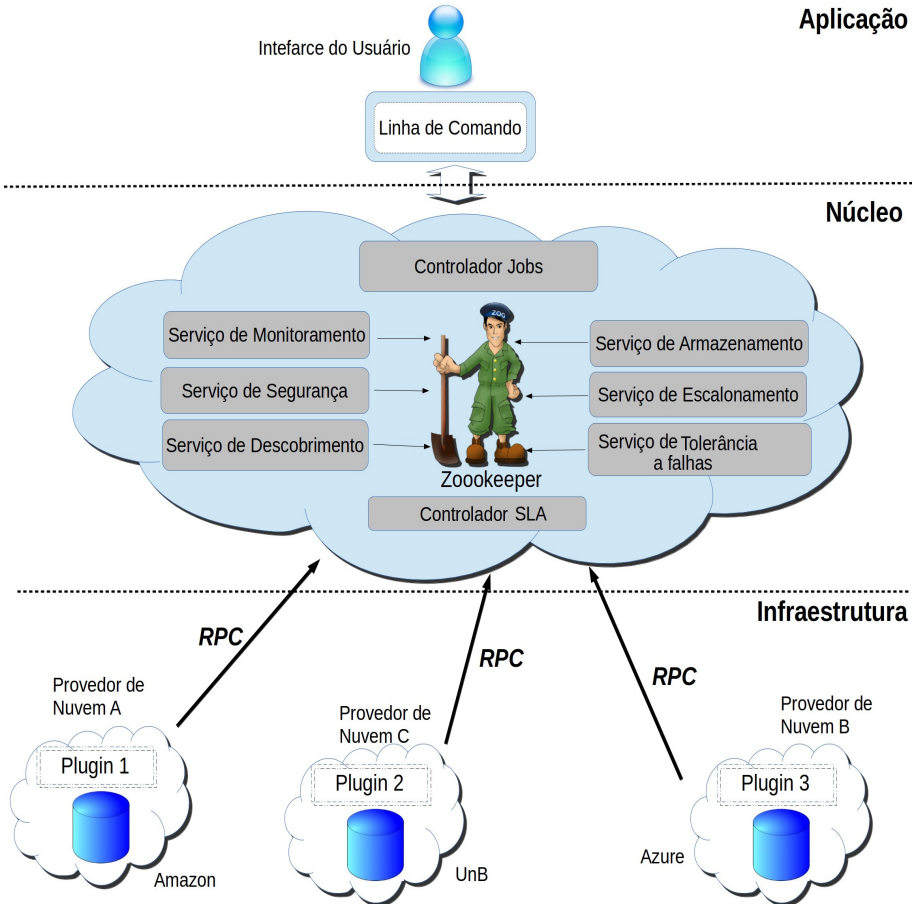

Figura 3.7: Arquitetura de Federação em Nuvens ZooNimbus.

Conforme visto na Figura 3.7, os meios de comunicação entre as nuvens no BioNimbus também foram substituídas. Assim, as chamadas, que antes eram feitas por meio da arquitetura de rede P2P, foram substituídas por Chamadas de Procedimento Remotas - RPC [23]. Essa mudança foi feita através do Avro Apache, que fornece uma forma simples e robusta de comunicação a arquitetura do ZooNimbus.

A arquitetura do ZooNimbus possui três camadas, na camada de aplicação encontra-se o usuário, atualmente toda a interação é feita por meio de linha de comando. No núcleo encontra-se os serviços do ZooNimbus, da mesma forma que a arquitetura do BioNimbus [34], todos os serviços foram mantidos. Além disso, os serviços de armazenamento e de escalonamento comunicam-se com o ZooKeeper, pois nele estão os dados utilizados por estes serviços para que possam realizar as tarefas e rotinas implementadas. Na camada inferior, a de infraestrutura, encontra-se as nuvens integradas ao ZooNimbus. Tanto nuvens públicas ou privadas podem ser integradas a federação, mostrando que o ZooNimbus é uma arquitetura bastante heterogênea.

A integração do Apache ZooKeeper foi bastante significativa, pois proporcionou um serviço de centralização de dados, o qual ofereceu uma interface fácil e simples para organizar o acesso aos dados do sistema.

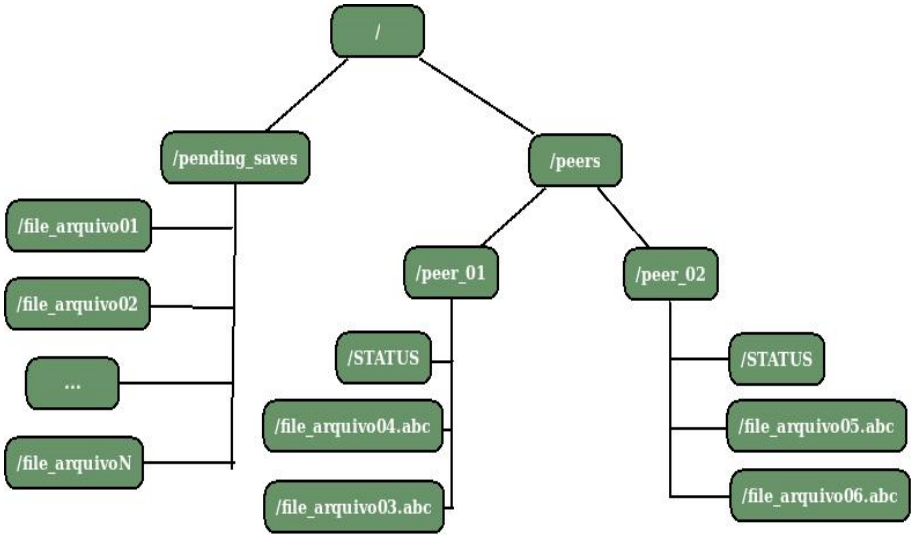

Figura 3.8: Estrutura do ZooKeeper com Relação ao *Storage Service*.

A Figura 3.8 mostra como está organizado a estrutura interna do ZooKeeper com relação ao *Storage Service*. Cada servidor tem uma quantidade de arquivos x e, para cada arquivo, é criado um *znode* ZooKeeper com as suas informações. As informações salvas nos *znodes* dos arquivos são o seu nome, seu ID, o seu tamanho e uma lista, chamada *pluginList*. Nesta *pluginList* são guardados os ID's dos servidores que possuem aquele arquivo salvo em disco. Quando um servidor se desconecta da federação, todos recebem um *watcher* avisando que este servidor se desconectou, para que assim possa iniciar a rotina de tratamento de replicação de arquivos por meio da rotina de tolerância a falhas da ZooClouS, descrita no próximo capítulo. Cada *znode* que possui informações sobre o arquivo que está inacessível é acessado e o ID do servidor que se desconectou da federação é retirado da *pluginList*. Em seguida, os arquivos são replicados, e então o ID do novo servidor que recebeu o arquivo é adicionado na *pluginList* deste arquivo. Além disso, também é adicionado um *znode* com as informações ao arquivo no servidor que o recebeu.

3.7 Considerações Finais

Neste capítulo foi descrita a arquitetura de federação em nuvens, o BioNimbus, que sofreu integração de dois programas, essenciais para coordenação de tarefas e comunicação entre os computadores envolvidos na arquitetura, transformando-se assim numa nova versão, denominada de Zoonimbus.

Capítulo 4

A Política de Armazenamento ZooClouS

O objetivo deste capítulo é apresentar o ZooClouS, a política de armazenamento de arquivos que foi implementada no ZooNimbus. Na Seção 4.1 são explicadas todas as mudanças realizadas em outros módulos de serviço do BioNimbus. Na Seção 4.2 estão todas as mudanças feitas no *Storage Service*, suas novas funcionalidades e como foi implementada o seu relacionamento com o ZooKeeper e com o Avro.

4.1 Mudanças nos outros Serviços do BioNimbus

Com as integrações dos *znodes* e *watchers* presentes no Apache ZooKeeper, e das chamadas RPC do Apache Avro, os módulos de serviços *Monitoring Service*, *Discovery Service*, *Scheduling Service* e *Fault Tolerance Service* sofreram alterações na forma em que implementavam as suas tarefas. Para a utilização do ZooKeeper na coordenação e 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 serviço ZooKeeper são corretamente identificadas e utilizadas de forma personalizada em cada módulo, modificando então a interface de implementação dos serviços, essa modificação ocorreu baseada na interface exemplificada na Tabela 4.1, como a interface funcionava anteriormente e na Tabela 4.2, pode ser visualizado as modificações implementadas na interface, para poder atender a questão da comunicação entre os módulos sobre alterações e criações de atividades do ZooNimbus.

Na Tabela 4.1 mostra a interface de comunicação dos serviços, na versão do BioNimbus, onde os módulos de serviços realizavam as seguintes funções, iniciar o serviço por meio da função *start(P2PService p2p)*, desligar o serviço pela função *shutdown()* e obter o estado do módulo do serviço por via do método *getStatus()*.

Tabela 4.1: Interface Implementada pelos Serviços no BioNimbus.

Service
+start(P2PService p2p);
+shutdown();
+getStatus();

Para atender as novas funcionalidades implementadas na versão ZooNimbus, a interface dos módulos de serviços recebeu novos métodos, o *verifyPlugins()* e o *event(WatchedEvent eventType)*, métodos implementados pelos módulos de serviços para verificar o *znode STATUS* de cada servidor e os *watchers* disparados pelo Zookeeper, respectivamente. Como a comunicação P2P foi retirada na versão ZooNimbus, o método *start()* seu parâmetro que era do tipo P2P foi retirado. Alterações descritas na Tabela 4.2.

Tabela 4.2: Nova Interface Implementada pelos Serviços no ZooNimbus.

Service
+start();
+shutdown();
+getStatus();
+verifyPlugins();
+event(WatchedEvent eventType);

Dessa 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. Esta classe implementada não realiza o tratamento da notificação, ela envia essa notificação para o serviço que fez o pedido de criação do *watcher*. Ele então recebe um alerta no novo método *event()* da interface *Service*, que pode ser visualizado na Tabela 4.2. Assim, 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*, *znode* que armazena as informações dos servidores, 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 ZooKeeper referente a este servidor. As mudanças nos serviços relativos à sua forma de funcionamento e integração ao ZooKeeper, estão descritas de forma mais detalhada nas próximas seções.

4.1.1 *Discovery Service*

Neste módulo 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 dos servidores, pois esses dados são lançados no servidor ZooKeeper assim que o novo servidor inicia sua

execução no ZooNimbus. O novo *znode* criado para o novo servidor e os seus *znodes* filhos contém as informações que são utilizadas pelo ZooNimbus para gerenciar a federação e para que os módulos funcionem de forma correta. Dessa forma, após o lançamento dos dados, o serviço de descoberta executa uma rotina de reconhecimento dos mesmos dados lançados por outros servidores que já iniciaram a execução do ZooNimbus, conhecendo as informações dos demais servidores. Assim, sempre que for necessário verificar quais servidores estão disponíveis podem ser realizadas leituras de seus dados no ZooKeeper, além da utilização de *watchers* para notificar as mudanças nos servidores.

Com a utilização desses *watchers* não há a necessidade do serviço de descoberta realizar constantes verificações de indisponibilidade dos servidores, provendo assim uma maior consistência, já que os servidores contidos no ZooNimbus não irão ficar indisponíveis sem que os demais servidores sejam avisados. A utilização da classe *PluginInfo* se manteve, classe na qual é abstraido os dados de cada servidor, dados que são armazenados no *znode peer_ IdPlugin* do ZooKeeper, podendo ser facilmente acessados por qualquer outro servidor.

O *znode STATUS*, mostrado na Figura 3.8, representa a disponibilidade do servidor. Ele é verificado sempre que necessário, e existirá até que o servidor fique indisponível. Ao ficar indisponível, o *znode STATUS* é automaticamente apagado pelo próprio ZooKeeper. Assim, é enviado um alerta para todos os servidores, e um novo *znode*, chamado *STATUSWAITING* é criado, permitindo aos demais serviços iniciem a recuperação das informações necessárias contidas nos *znodes* daquele servidor antes que sua estrutura seja apagada pelo módulo de monitoramento, o *Monitoring Service*. Desta forma, mantém-se um controle consistente da tolerância a falha, pois 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 servidor. Todos os módulos realizam e solicitam notificações de indisponibilidade de todos os servidores da federação. Para isso, um *watcher* é criado no *znode* de cada servidor do ZooNimbus.

Esta nova implementação de descoberta da rede que forma a 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.

4.1.2 *Monitoring Service*

O atual módulo de monitoramento modifica em partes sua proposta inicial de pedido de execução e acompanhamento do estado das tarefas que foram enviadas para execução. Na nova implementação, o seu principal acompanhamento é referente aos servidores que formam a federação em nuvem. Esse acompanhamento é feito de forma integrada ao ZooKeeper, com a utilização de *watchers* e tratamento de alertas dos mesmos, realizando assim um trabalho de monitoramento livre de envio e solicitação de mensagem via rede P2P. Isso evita consumo de banda e possíveis erros com o não recebimento de dados. Agora, as solicitações e o envio de dados são todos direcionados ao servidor ZooKeeper.

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

Dessa forma, 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. Como não é possível garantir a ordem cronológica do envio dos avisos realizados pelo ZooKeeper, todos os demais módulos que solicitaram o aviso de indisponibilidade dos servidores, podem realizar a criação do *znode STATUSWAITING* ao receberem a notificação de indisponibilidade para iniciarem suas rotinas de recuperação dos dados. 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 contidas 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 de monitoramento irá aguardar até que seja feita a recuperação, para então excluir o *znode*.

4.1.3 *Scheduling Service*

A forma de realizar o serviço de escalonamento foi completamente refeita, pois foi baseada nas novas tecnologias utilizadas, ZooKeeper Apache [14] e Avro Apache [11]. A comunicação entre servidores para verificar o estado das tarefas em execução ou para escalonamento é agora feita com a utilização do ZooKeeper, deixando de ser direcionada servidor a servidor. Os procedimentos são realizados quando é necessário informar o estado, a atualização e a inclusão das tarefas. Esses dados são enviados e recuperados diretamente do servidor ZooKeeper.

Quando um cliente está conectado a um servidor ZooNimbus, ou seja, conectado a qualquer servidor que participe da federação em nuvem, ele 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*, informações que são necessárias para realizar a execução do *job*. 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 4.1 que retorna um mapa das tarefas, representado por *Map<JobInfo,PluginInfo>* na assinatura da Figura 4.1, enviadas para o escalonamento, com o servidor eleito, representado pela variavél *PluginInfo* e que está contida no retorno do mapa, pelo escalonador para executar a tarefa. Para a chamada desse método deve ser informado as tarefas à serem escalonadas, tarefas que estão contidas na coleção *Collection<JobInfo> jobinfos*, e também a conexão utilizada com o Zookeeper,a qual a variavél *zk* está representando.

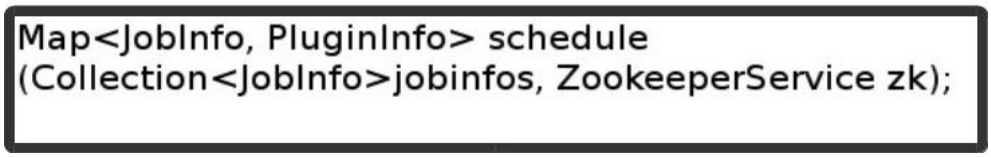

Figura 4.1: Assinatura que Deve ser Implementada ao Utilizar a Interface *SchedPolicy*.

Após a realização da rotina e da política de escalonamento, um servidor é eleito para executar o *job*, e essa informação é disponibilizada por meio do ZooKeeper. Um novo *znode task_IdJob*, contendo as informações da tarefa, é criado dentro do *znode task* do servidor eleito para o escalonamento e o *znode job_IdJob*, que pertencia ao *znode jobs*,*znode* que armazena as requisições de execução de tarefas, 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. Assim, quando a execução é finalizada, o método que realizou sua execução irá modificar o estado da tarefa no ZooKeeper, de tarefa "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 contêm os dados que serão processados pelos serviços de bioinformática, e geram um ou mais arquivos de saída. Esses arquivos devem existir em algum recurso da federação antes de ser solicitada a execução do serviço. Caso esse arquivo não exista em nenhum servidor, o ZooNimbus informa ao cliente que solicitou a execução, e que os arquivos de entrada devem ser disponibilizados na plataforma. Assim, 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 elege para qual servidor o arquivo de entrada será enviado.

Considerando que o módulo de armazenamento poderá enviar o arquivo de entrada para um servidor diferente daquele que o módulo de escalonamento solicitar a execução do serviço, ambos os módulos devem realizar algum procedimento que minimize esse conflito que pode aumentar o tempo de execução de uma tarefa. O procedimento realizado para esse cenário consiste na utilização de um parâmetro comum, que é utilizado por ambos os módulos para eleger um servidor, a latência, que é calculada entre o 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, podendo alterar o valor da probabilidade de sua escolha do recurso.

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

4.1.4 *Fault Tolerance Service*

Na nova implementação, este serviço está presente em todos os demais serviços, o que favorece o seu desempenho e garante um maior sucesso na instalação da tolerância a falhas.

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

No *Scheduling Service*, o *Fault Tolerance* está presente quando ele também recebe o alerta informando sobre a indisponibilidade do servidor e inicia a recuperação de todas as tarefas que foram escalonadas para lá, e ainda não foram executadas ou estavam em execução, de acordo com os meta-dados armazenados 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 servidor, 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, assim como pela verificação da inclusão de algum arquivo nos recursos que o módulo de armazenamento não tenha reconhecido.

Toda essa recuperação é possível porque os recursos da federação se conhecem e mantêm um *watcher* nos seus respectivos *znodes STATUS*. As alterações citadas nas seções anteriores mostraram as mudanças que ocorreram no ZooNimbus. Assim, na próxima seção será explicado detalhadamente todo o processo que o *Storage Service* passou e quais as mudanças que ocorreram com relação ao tratamento de arquivos na federação em nuvens.

4.2 *Storage Service*

O ZooNimbus tem sido usado para executar, não de forma exclusiva, aplicações de bioinformática que necessitam de grande poder de processamento e armazenamento. Essas aplicações de um modo geral possuem características semelhantes, as principais são:

- Um conjunto de arquivos de entrada que serão analisados pela ferramenta, que podem ser originados a partir da saída da máquina de sequenciamento ou de outra ferramenta de bioinformática;
- Um conjunto de arquivos de saída contendo o resultado da análise realizada pela ferramenta;
- Um conjunto de parâmetros de execução que podem influenciar, por exemplo, na utilização dos recursos de hardware ou no formato dos arquivos de entrada e de saída.

Assim, a arquitetura prevê uma série de estudo de caso que tentam se adequar às características listadas. O objetivo é prover um conjunto completo de ações que possam ser realizadas sobre a arquitetura ZooNimbus, de forma que os serviços de interação com o usuário sejam capazes de atender às necessidades dos pesquisadores [34].

O *Storage Service* passou por várias modificações, anteriormente, quando um *upload* era feito na federação, o servidor no qual o cliente estava conectado recebia uma solicitação

de *upload* e mandava o arquivo para o primeiro servidor que estivesse disponível, sem realizar nenhuma análise que pudesse escolher o servidor mais adequado para armazenar o arquivo solicitado.

Assim, serão descritas, nas próximas seções, as formas de execução dos principais serviços da política de armazenamento que foram implementadas no *Storage Service*, as quais são: *upload*, replicação, *download* e tolerância a falhas. Também será descrito a forma que a ZooClouS coleta os dados dos servidores para o cálculo do custo de armazenamento, e a maneira de como é feito este cálculo.

4.2.1 *Upload* de Arquivos

Para executar ferramentas de bioinformática no ZooNimbus é necessário que os arquivos de entrada utilizados pelas ferramentas estejam dentro da federação. Assim, um *upload* é necessário também, caso o arquivo de saída gerado seja armazenado em uma infraestrutura diferente da que realizou o processamento. Logo, conforme mostra a Figura 4.2, o *upload* de arquivos é realizado por meio dos seguintes passos:

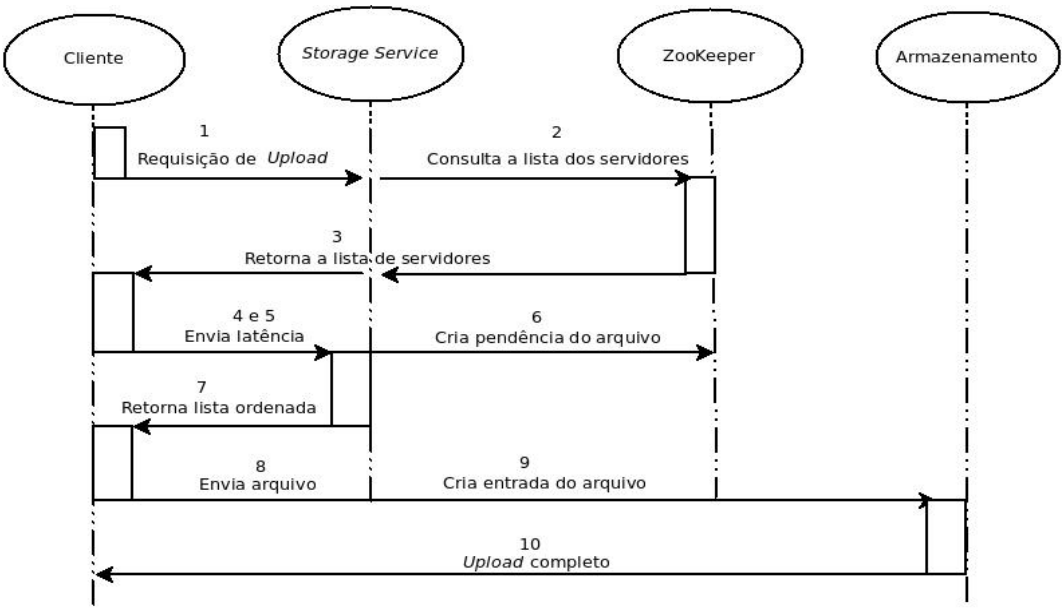

Figura 4.2: Sequência de Ações que Ocorrem Durante o Processo de *Upload*

1. O cliente realiza uma requisição de *upload* por meio do console, passando o caminho do arquivo de entrada na máquina onde ele está se conectando ao ZooNimbus.

2. O servidor ZooNimbus em que ele está conectado consulta as informações do arquivo e verifica quais servidores dentro da federação possuem espaço livre suficiente para receber o arquivo, sendo que cada servidor utiliza no máximo 90% de sua carga de armazenamento. Este valor foi definido pela necessidade de espaço em disco no servidor, para que este possa iniciar o sistema operacional e também para que pastas que geram arquivos temporários tenham espaço em disco para a criação destes arquivos. Esse valor poder ser facilmente alterado, bastando apenas modificar a variável *MAXCAPACITY* presente no *Storage Service*.

3. O servidor retorna para o cliente uma lista com todos os servidores disponíveis para o armazenamento.
4. No cliente é realizado o cálculo da latência entre ele e cada servidor disponível na federação, logo após ele devolve a lista com a latência de cada servidor em relação ao cliente para o servidor.
5. O servidor recebe a lista e envia para o *Storage Service*, onde será feito o cálculo da política de armazenamento de cada servidor, este cálculo será detalhado na Seção 4.2.7.
6. O servidor irá criar uma entrada no *znode* chamado *pending saves*, com os dados do arquivo que será colocado na federação. Esta entrada informa a todos os servidores que um arquivo será enviado para a federação e que este envio ainda não foi finalizado, ou seja, está pendente a sua conclusão.
7. Com o cálculo feito, o servidor retorna para o cliente uma lista ordenada com todos os servidores disponíveis de acordo com o custo de armazenamento de cada um (o primeiro servidor possui o menor custo, o segundo servidor, o segundo menor custo, e assim por diante).
8. Após receber a lista, o cliente então realiza a transferência do arquivo via *SSH File Transfer Protocol - SFTP* para o melhor servidor. Em caso de erro durante o envio do arquivo, falha na rede ou qualquer outro problema, o envio do arquivo será automaticamente direcionado para o próximo servidor da lista.
9. O servidor irá criar uma entrada no *znode* do servidor onde o arquivo foi armazenado com os seus dados (nome, ID, tamanho, etc).
10. Em seguida, o cliente recebe uma mensagem confirmando que seu arquivo foi enviado com sucesso.

Quando for finalizado o *upload*, será possível ver se o arquivo está no ZooNimbus por meio do comando *files*, o qual lista todos os arquivos contidos em todos os servidores da federação. Depois que o cliente recebe uma confirmação de que o arquivo foi colocado na federação com sucesso, o servidor que recebeu o arquivo irá iniciar a rotina de replicação, descrita na Subseção 4.2.2.

4.2.2 Replicação de Arquivos

Toda vez que um arquivo é colocado dentro do ZooNimbus, o servidor que recebeu o arquivo irá iniciar o processo de replicação deste arquivo em outros servidores na federação de nuvens de acordo com o fator de replicação definido, por padrão, este valor foi definido em 2. Logo, isto faz com que no mínimo uma cópia do arquivo fique disponível em outro servidor. Conforme a Figura 4.3, o processo de replicação segue os passos abaixo:

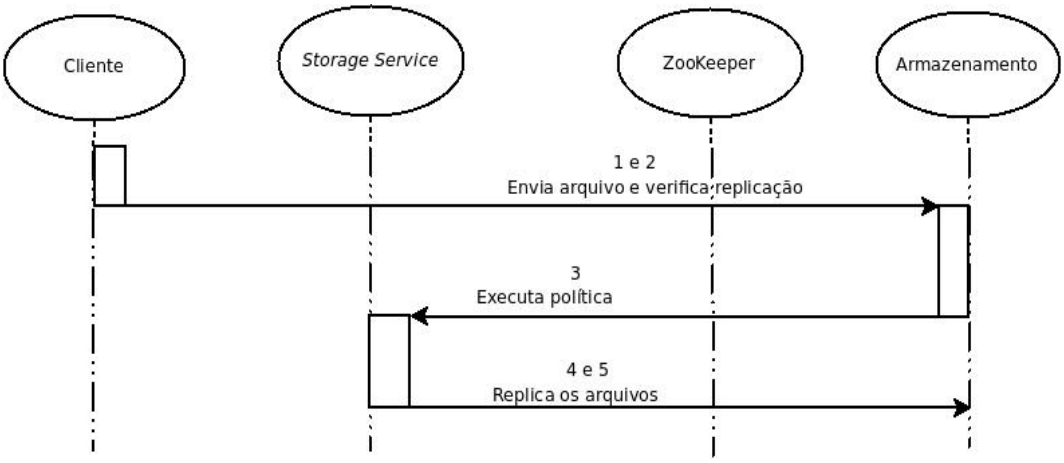

Figura 4.3: Sequência de Ações que Ocorrem Durante o Processo de Replicação dos Arquivos.

1. O *upload* é concluído no melhor servidor de acordo com o cálculo do custo de armazenamento feito pela política de armazenamento.
2. O servidor que recebeu o arquivo consulta uma lista com todos os servidores que possuem espaço disponível para armazenar o arquivo.
3. De acordo com o número de cópias definidas, no caso em estudo é dois, o ZooNimbus consulta os servidores disponíveis e copia o arquivo para o melhor servidor em relação ao mesmo que possue o arquivo, através do protocolo SFTP [46] até que a operação seja concluída com sucesso.
4. Para cada servidor que recebe a cópia do arquivo, é criada uma entrada em seu respectivo *znode* com os dados do arquivo que foi copiado.
5. Com o número de cópias alcançado, o servidor interrompe a rotina e termina a replicação.

Esta sequência de ações ocorre após um envio de um arquivo na federação. Existe outra situação que também utiliza o processo de replicação, que é a tolerância a falhas, que será descrita na Subseção 4.2.3.

4.2.3 Tolerância a Falhas

Para garantir que um arquivo possua uma quantidade mínima de cópias dentro da federação, foi criada uma rotina de tolerância a falhas. A Figura 4.4 mostra a sequência de ações durante o processo de tolerância a falhas.

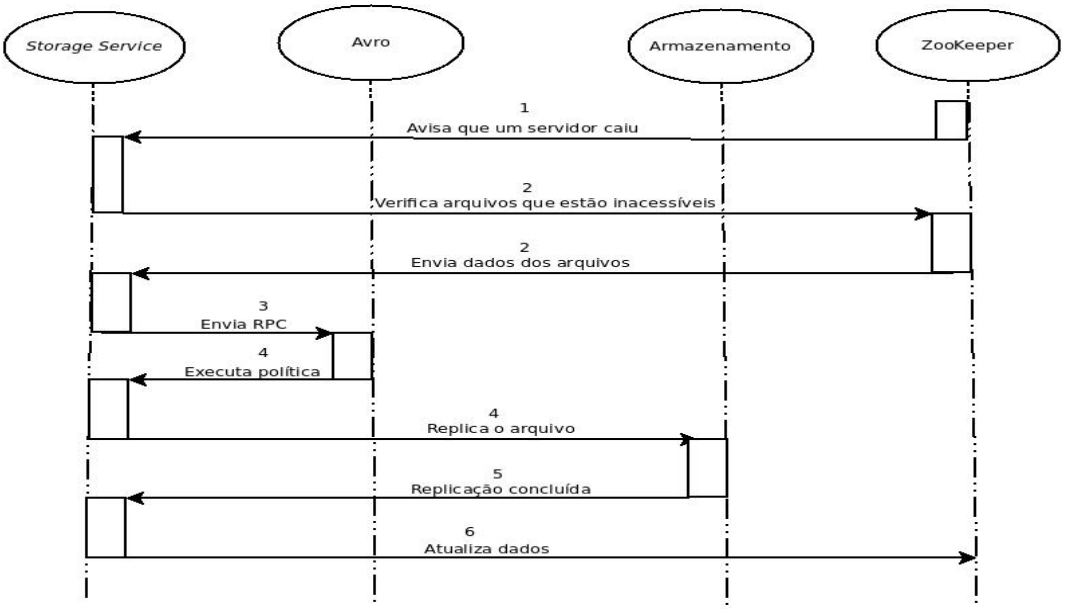

Figura 4.4: Sequência de Ações que Ocorrem Durante o Processo de Tolerância a Falhas.

1. Quando um servidor cair ou se desconectar da federação. Como cada servidor possui uma quantidade x de arquivos, esses arquivos são replicados para outros servidores garantindo o número mínimo de cópias definido, por meio do Zookeeper ocorre o alerta sobre a situação do servidor.
2. Os recursos, assim como arquivos do servidor tornam-se inacessíveis para os outros servidores. Assim sendo, para garantir que todos os arquivos que ficaram inacessíveis tenham a quantidade mínima de cópias, a tolerância a falhas irá verificar quais arquivos ficaram inacessíveis.
3. Após isso, o ZooNimbus irá contactar via RPC o servidor que possuir uma cópia do arquivo que está inacessível avisando que ele deve replicar aquele arquivo para algum servidor.
4. O servidor que tiver o arquivo irá rodar o cálculo do custo de armazenamento entre ele e os servidores da federação, e irá replicar para o que tiver o menor custo de armazenamento para ele.
5. A replicação é concluída.
6. Por último, os dados do arquivo são atualizados no ZooKeeper de cada servidor que tiver o arquivo e o *znode* do servidor que se desconectou é deletado.

A tolerância a falhas garante que nenhum arquivo dentro da federação tenha um número de cópias menor do que *REPLICATIONFACTOR*. Com isto, a disponibilidade de um arquivo aumenta. Assim, se um servidor se desconectar e ao mesmo tempo uma requisição para um *download* dos arquivos que se tornaram inacessíveis seja feita, o ZooNimbus irá buscar a cópia deste arquivo em outro servidor que tenha o arquivo, enquanto isso outro servidor iniciará o tratamento dos arquivos que não estão mais disponíveis.

4.2.4 *Download* de arquivos

No ZooNimbus existem duas situações diferentes de requisições de *download*. Na primeira, o cliente/usuário deseja baixar um arquivo que é o resultado de algum processamento ou que apenas está armazenado na federação. O segundo caso é quando um cliente/servidor precisa de determinado arquivo dentro da federação para que ele seja utilizado como entrada para a execução de outro processamento. O *download* de arquivos pelo cliente pode ser descrito nos seguintes passos, demonstrados na Figura 4.5:

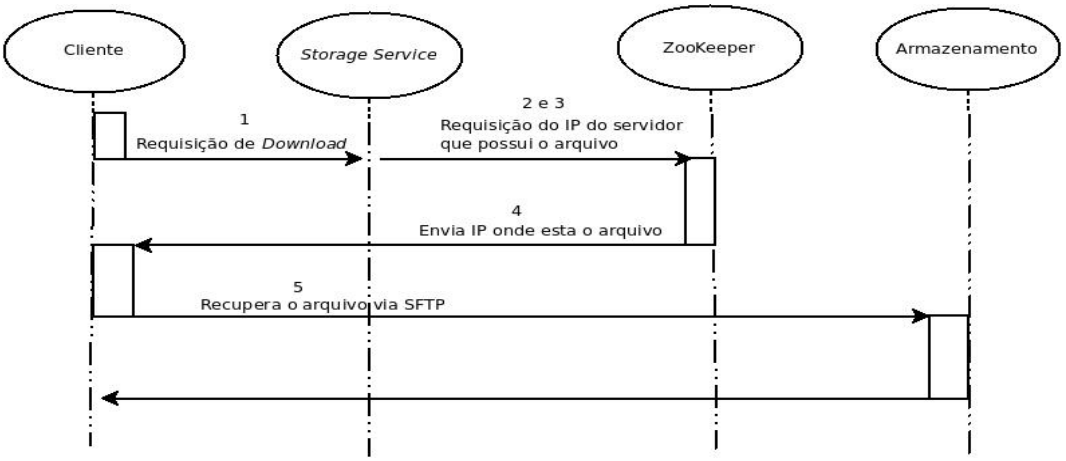

Figura 4.5: Sequência de Ações que Ocorrem Durante o Processo de *Download*.

1. O cliente solicita um arquivo para a federação enviando o seu nome para o servidor ZooNimbus em que está conectado.
2. Este servidor, via Avro [11] faz uma chamada RPC para recuperar uma lista com todos os servidores da federação, e seus respectivos arquivos que estão armazenados nele.
3. A lista com todos os servidores e arquivos é recebida pelo servidor. O *Storage Service* realiza uma rotina para percorrer todos os arquivos e ir comparando com o nome do arquivo solicitado.
4. Se o arquivo é encontrado, o servidor ZooNimbus retorna o endereço IP do servidor que possui o arquivo solicitado.
5. Após receber o IP, o cliente abre uma conexão SFTP com o servidor passando o arquivo que deseja como parâmetro. A conexão é estabelecida e o o *download* do arquivo é realizado.

A sequência de ações mostrada na Figura 4.5, descreve situação em que o cliente/usuário solicita o arquivo. No caso de um cliente/servidor precisar de um arquivo, as ações são as seguintes:

1. Um serviço de um cliente/servidor precisa de um arquivo para realizar um processamento, esse serviço consulta no ZooKeeper onde tal arquivo está localizado, retornando o *ip* do servidor onde está situado o arquivo.

2. Com o *ip* do servidor que possui o arquivo, uma conexão é feita entre o cliente/servidor e o servidor que possui o arquivo, e o arquivo é transferido.

3. Após o arquivo ser transferido para o cliente/servidor, para o mesmo é criado um *znode* no respectivo cliente/servidor.

A diferença entre as duas chamadas está no fato do cliente/servidor não precisar fazer requisição de um arquivo para o *Storage Service*, ele realiza a operação apenas descobrindo o *ip* do servidor que possui o arquivo.

4.2.5 Listagem de Arquivos

A listagem de arquivos é necessária para se obter todos os arquivos contidos na federação, não importando em qual servidor o usuário está conectado. A sequência de passos é mostrada na Figura 4.6.

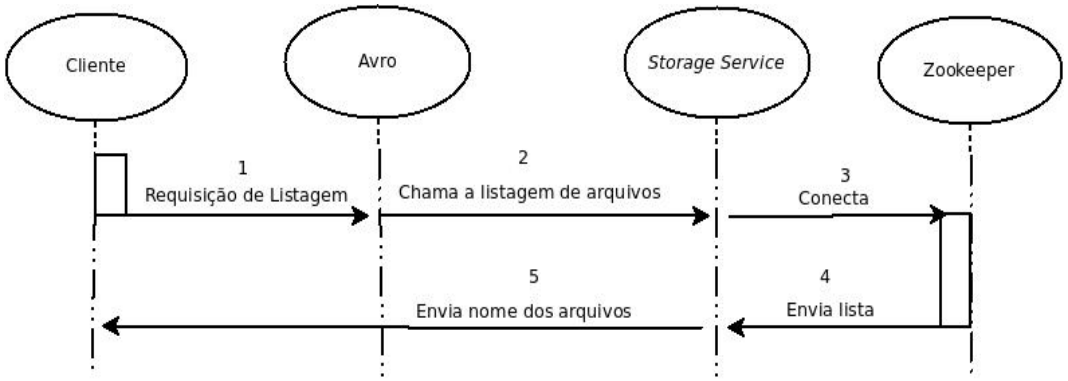

Figura 4.6: Sequência de Ações que Ocorrem Durante o Processo de Listagem dos Arquivos.

1. O cliente realiza uma requisição de listagem de arquivos.

2. O servidor o qual o cliente está conectado passa o comando para o *Storage Service*.

3. O *Storage Service* irá realizar uma consulta no ZooKeeper para obter os arquivos de cada servidor da federação.

4. O ZoooKeeper retorna a lista de todas as entradas de arquivos presenetes no mesmo.

5. O *Storage Service*, realiza um tratamento na lista retornada, enviando uma lista contendo nomes únicos dos arquivos presentes na federação.

Os nomes dos arquivos são mostrados em forma de uma única lista, dando a ilusão de que o usuário está conectado em apenas um computador, e que este único computador possui todos os arquivos mostrados. Todos os arquivos contidos na federação possuem ao menos uma cópia, porém quando o comando de listagem é executado, os arquivos são mostrados apenas uma vez, escondendo do usuário a quantidade de cópias existentes dos arquivos encontrados. Caso o usuário queira baixar um dos arquivos na lista, basta apenas digitar o comando *download* seguido do nome do arquivo que deseja transferir para sua máquina. O armazenamento de arquivos em um ambiente de federação de nuvens recebe

várias demandas de várias fontes diferentes. Neste cenário, é necessário um algoritmo para realizar a escolha do melhor servidor para se armazenar o arquivo enviado para o ZooNimbus.

Na Subseção 4.2.6 será explicado o Algoritmo A*, no qual foi utilizado a sua base para a implementação do algoritmo proposto na Subseção 4.2.7, o ZooClouS.

4.2.6 O Algoritmo A*

O Algoritmo A* [42] é um algoritmo de busca de melhor caminho, muito utilizado em jogos, cuja função é buscar o melhor caminho entre um ponto inicial e um ponto final. Para um ambiente em nuvens foi implementado por Ren Xun-Yi [45], um algoritmo baseado no A*, com o intuito de otimizar o custo de armazenamento. Segundo Ren Xun-Yi [45], dado um ambiente de armazenamento em nuvens, N usuários necessitam armazenar D = (D1, D2, ..., Dn) dados no ambiente, que possui M nós CM = (C1, C2, ..., CM). Para um usuário, o custo de armazenamento de um dado D_i em um nó C_j inclui duas partes: é o custo do tempo necessário para transferir o dado D para o nó C_j, e a outra parte é o preço de armazenamento de C_j. A fórmula desta situação é mostrada na Equação 4.1, onde *trans* é a função que calcula o custo do tempo de transferência do arquivo, passando como parâmetro os dados do arquivo, representado pela variável D, os dados do cliente representado pela variável *user* e as informações do nó destino representado por C_j e s é a função que calcula o preço do armazenamento do arquivo D no nó C_j:

$$StorageCosts = trans(D, user, C_j) + s(D, C_j) \tag{4.1}$$

O custo total de acessos de N usuários é a transmissão de dados dos nós da nuvem para todos os usuário, conforme mostrada na Equação 4.2.

$$AccesCosts = \sum_{i=1}^{N} trans(D, C_j, user_i) \tag{4.2}$$

O objetivo da otimização é que o custo total de armazenamento e de acesso sejam mínimos, conforme é mostrado na Equação 4.3:

$$TotalC = Min(\sum_{k=1}^{M} \alpha * AccessCost_k + \beta * Storagecosts_k) \tag{4.3}$$

Onde α e β são fatores de peso, e $\alpha + \beta = 1$. A restrição do problema de otimização de armazenamento é indicada na Equação 4.4

$$AvailStorage \geq SizeofReqData \tag{4.4}$$

Um fator simples que deve ser considerado é que nuvens respondam de forma eficiente o acesso do usuário. A otimização do A* utiliza como possível solução do custo de armazenamento a Equação 4.5:

$$EV(s) = D(s) + V(s) \tag{4.5}$$

D(s) é o custo pago, e V(s) é o custo potencial. Assumindo que a solução ótima é V(s)*, então o atual custo é mostrado na Equação 4.6:

$$EV(s) = D(s) + V(s)* \tag{4.6}$$

Uma vez que V(s)* é desconhecido, usa-se o menor *Limite superior* de V(s): Ls(V(s)) em vez de V(s)*, dado pela Equação 4.7:

$$Ls(V(s)) \leq V(s)* \tag{4.7}$$

Em seguida, $D(s) + Ls(V(s)) < D(s) + V(s)*$. Usando $D(s) + V(s)*$ para calcular a atual solução proposta, independentemente de D(s) ser bom ou ruim, uma solução não é perdida com isto. Para o ambiente de nuvens, o A* é combinado com o algoritmo Min-Min [44] para procurar a melhor solução. A descrição do algoritmo A* é mostrada na Figura 4.7.

A* based Storage Optimization Algorithm

```
1 :  BestCost←Get_Minmin()
2 : h←Get_height(T) //get hight of tree
3 : n←0
4 : while (T<Set(Time))and(ε>Set(ε))
5:      if n==h then
6 :         UPDATE-Bestcost(Bestcost,Currentcost);
7:      else
8:           {DS←Get_Cost1;
9 :          VS←Get_Cost2;}
10:     If (Ds+Vs< bestcost)
11:          Select_next_site;
13:     If current level is travel finished
15:          n←n+1;
16:     end while
```

Figura 4.7: Algoritmo A* Otimizado para Armazenamento em Nuvens

O algoritmo A* tem um efeito de otimização baixo se um algoritmo de escolha ao acaso for utilizado para uma solução inicial. Por conta disto, na linha 1 do algoritmo é utilizado o algoritmo Min-Min [44] para gerar uma melhor solução aproximada. O algoritmo Min-Min distribui dados para um nó na nuvem com um custo de armazanamento mínimo, um de cada vez, em seguida, remove o nó ocupado da lista. Ocorre uma consideração apenas de distribuições ideais no algoritmo Min-Min, pois uma remoção cega de um nó de armazenamento pode apagar uma solução ideal e, portanto, não alcançar o melhor custo global entre as nuvens.

Para evitar esse problema, a altura da árvore é obtida na segunda linha e, em seguida, é gerada uma árvore inicial, onde qualquer dado N pode ser armazenado em um nó M na nuvem, gerando a alocação $N * M$, como resultado, a árvore inicial tem N camadas

e cada camada tem M ramos. Com isso, uma remoção cega é evitada. No *loop while* (linhas 4 a 14 da Figura 4.7), uma viagem na árvore é realizada. Em primeiro lugar, é verificado se o custo total de armazenamento de um nó atende a Equação 4.3, depois utiliza-se a Equação 4.4 para calcular o custo total. Se o atual custo calculado for de um nó folha, *Bestcost* é atualizado, se $CurrentCost < BestCost$, *BestCost* é substituido por *CurrentCost*, através deste, o algoritmo mantém a solução ideal nas linhas 5 e 6 da Figura 4.7. Custo V(S) é calculado se o nó atual está na camada do meio, este cálculo é feito a partir do nó raiz para o nó atual e, em seguida, comparar o provável custo pago com *BestCost* para determinar se deve ou não podar a árvore. Portanto, o algoritmo deve determinar a escolha da solução calculada somente com o custo minimo do nível seguinte, ele não faz a partir do nó raiz para os nós folha, com isso, consegue-se o efeito da poda rápida da árvore.

Neste cenário, um algoritmo de armazenamento foi proposto neste trabalho para o ZooNimbus, chamado ZooClouS, o qual será apresentado na Seção 4.2.7.

4.2.7 ZooNimbus Cloud Storage - ZooClouS

Quando um arquivo é enviado para a federação de nuvens, um custo de armazenamento é calculado para aquele arquivo. Este custo é feito transparentemente entre o cliente que requisitou o *upload* e cada servidor que possui espaço em disco suficiente para receber o arquivo. Para o ZooNimbus, foram utilizadas três variáveis para o cálculo do custo de armazenamento:

- *Free-Size*: espaço livre em disco do servidor;
- *Uptime*: tempo em que o servidor está *on-line* no sistema ZooNimbus. É o segundo peso mais importante no cálculo do custo de armazenamento, visto que um servidor que está há mais tempo no sistema, gera uma maior confiabilidade com relação ao fato de que um servidor pode se desconectar da rede e consequentemente da federação. Toda vez que um servidor é criado, é adicionado um valor em milissegundos no seu *znode*, a partir da consulta deste valor, torna-se conhecido o tempo em que este servidor está *on-line* no sistema.
- *Latency*: latência entre o cliente e o servidor de destino, é o maior peso no cálculo do custo.

Baseado no algoritmo A* apresentado na Seção 4.2.6, proposto por Xun-Yi [45], o cálculo do custo de armazenamento pega dados dos servidores e define em cada um a variável *storagecost*, que será o custo de armazenamento do dado, de acordo com o cliente que realizou a requisição de envio. Para o próximo cliente, o *storagecost* é sobrescrito com um novo valor, pois o valor antigo foi configurado de acordo com o cliente que enviou o último arquivo.

Cada cliente pega instância dos servidores para calcular e setar o custo, isto evita que um custo seja sobrescrito por outro cliente antes de ser utilizado pelo cliente anterior. O custo de armazenamento utilizado na política é uma forma aproximada de definir o tempo de transferência entre origem e destino. Quanto menor o valor do custo, menor

será o custo gasto na transferência do arquivo. As três variáveis utilizadas para o cálculo do custo de armazenamento formam a seguinte equação:

$$S = ((U + F) * L) + Costs \tag{4.8}$$

Onde S é o custo de armazenamento, U é o tempo que o servidor está *on-line* (*uptime*), F é o espaço livre (*free size*) em disco, e L é a latência calculada entre origem e destino. Além disso, as variáveis utilizadas possuem pesos com o objetivo de priorizar os campos mais importantes no cálculo, assim tem-se a Equação 4.9:

$$\alpha + \beta + \gamma = 1.0 \tag{4.9}$$

Com base em vários testes feitos, os valores dos pesos que definiram um melhor custo de armazenamento foram α com um peso de 0.5, associado à latência, β com um peso de 0.2, associado ao *freesize*, e o γ atribuído ao *uptime* e possui um peso de 0.3, Conforme mostrado na Equação 4.10:

$$S = ((U * \gamma + F * \beta) * (L * \alpha)) \tag{4.10}$$

Esse cálculo forma a primeira parte do custo de armazenamento. Todavia, como o ZooNimbus trabalha com nuvens públicas ou privadas, pode acontecer de uma nuvem ser integrada à federação e ela cobrar um preço pela utilização de seu recurso de armazenamento. Por conta disso, é necessário adicionar ao custo de armazenamento o preço de armazenar um dado em uma nuvem que faça cobrança. No caso de nuvens que, geralmente, pertencem a uma organização privada ou instituição pública e que não cobrem valores para a utilização de seu disco, o preço do seu armazenamento é 0. Para outros casos, como por exemplo, a Amazon [25] ou Azure [6], um valor é cobrado por *gigabyte* de dado armazenado. Este valor é definido no arquivo de configuração da nuvem.

Com o valor do preço por *gigabyte* definido, este custo é adicionado à variável *costs*, que somado ao custo de armazenamento calculado daquela nuvem, da origem a Equação 4.11:

$$S = ((U * \gamma + F * \beta) * (L * \alpha)) + Costs \tag{4.11}$$

Depois de calculado o custo final de cada servidor, uma lista é recebida com todos os servidores e ordenada de acordo com os seus custos de armazenamento. A ordenação é feita de forma crescente, ou seja, do servidor que tem o menor custo até o maior, sendo que quanto menor o custo, melhor será a transferência de um arquivo para este servidor. Após devolver esta lista para quem está requisitando um envio de *upload*, o cliente irá tentar enviar o arquivo ao primeiro da lista, caso não consiga, continuará tentando o envio no próximo servidor até que o arquivo seja enviado.

4.3 Considerações Finais

Neste capítulo foi apresentado o ZooClouS, a política de armazenamento implementada no ZooNimbus, que buscou melhorar o seu desempenho fornecendo serviços que fazem com que a federação em nuvens consiga atender as requisições feitas por vários usuários ao sistema. O ZooClouS mudou a forma de tratamento de arquivos no ZooNimbus, pois

com as suas funções que foram implementadas, a busca de arquivos deixou de ser linear, tornando-se mais flexível e rápida.

No capítulo 5 estão os estudos de caso feitos com o ZooClouS no ZooNimbus, e as conclusões tiradas a partir da coleta de resultados destes testes.

Capítulo 5

Estudo de Caso

Neste capítulo é descrito o estudo de caso realizado com o ZooClouS no ZooNimbus. Para isto, foram federadas nuvens privadas e públicas para a criação de um ambiente de nuvens voltado para a execução de aplicações de bioinformática. Foram realizados diversos estudos de caso envolvendo o ZooClouS. Na Seção 5.1 estão os testes realizados com relação ao tempo de decisão que o ZooClouS atinge para escolher o melhor destino para um arquivo. Na Seção 5.2 teste realizado para comprovar a melhoria no tempo de execução de tarefas utilizando o Bowtie [24], na arquitetura ZooNimbus, provando que quanto mais recursos, mais rápido é a execução das tarefas.

5.1 Tempo de Decisão e Transferência

Em um ambiente de federação de nuvens, são encontrados vários servidores oferecendo espaço em disco para armazenar arquivos, e vários usuários enviando e solicitando arquivos contidos na federação. Com base neste cenário, foram realizados testes para identificar e medir o tempo de resposta que o ZooClouS emite para as requisições que são feitas ao ZooNimbus. Para isto foi criado um ambiente de execução detalhado na Seção 5.1.1.

5.1.1 Ambiente de Execução

Para o estudo de caso do tempo de decisão do ZooClouS, foi criada uma federação com três nuvens, as quais são:

- Uma nuvem na Universidade de Brasília, com três computadures e com as seguintes configurações, processador Core i7-3770 de 3,4 GHz, memória RAM de 8Gb, 2TB de disco rígido, com o sistema operacional Linux, distribuição Ubuntu 13.04. Essa nuvem está fisicamente localizada no Brasil;
- Uma nuvem na Microsoft Azure [6], com três máquinas com 8 núcleos, 14Gb de RAM e disco rígido de 30Gb, com sistema operacional Linux, distribuição Ubuntu 12.04 LTS, cada. A nuvem está localizada na Ásia Oriental;
- Uma nuvem na Amazon [25], com três máquinas com processador Intel Xeon 2.4 Ghz, 613MB de memória RAM, disco rígido de 80GB e sistema operacional Linux, distribuição Ubuntu 12.04.2. A nuvem está localizada nos EUA.

5.1.2 Escolha do Melhor Servidor para os Arquivos

Para o estudo da escolha do melhor servidor para o arquivo, foram feitas requisições de *upload* para se conhecer em qual local o ZooClouS enviará os arquivos devido a localidade de quem requisitou o envio e dos servidores disponíveis.

Foram realizados nove requisições de *upload* vindas de um cliente com um computador localizado no Brasil, sendo que foram feitos nove envios para a versão anterior do ZooNimbus (BioNimbus), onde o tratamento e a escolha do melhor servidor para receber os arquivos eram inexistentes, os arquivos eram inseridos na federação de forma aleatória, onde a escolha era feita de acordo com o endereço que o cliente conhecesse.

Logo após, foram feitos mais nove envios com a política de armazenamento ZooClouS, com um arquivo de 70 megabytes de tamanho, do tipo binário. Os resultados deste teste são mostrados na Figura 5.1.

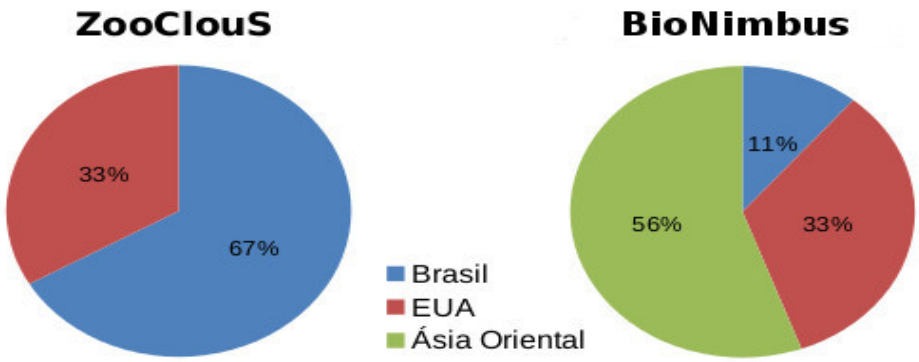

Figura 5.1: Destino de Arquivo Enviado ao ZooNimbus com a Implementação da ZooClouS, e sem a Implementação da ZooClouS.

Conforme mostrado na Figura 5.1, com a ZooClouS, observou-se que em 67% dos envios, o arquivo foi enviado para a nuvem localizada no Brasil e 33% para a nuvem nos EUA. Com a política de armazenamento aleatória, 56% dos envios resultaram na nuvem localizada na Ásia, que encontra-se mais distante da origem do arquivo, e apenas 11% resultaram no Brasil, local que está mais perto do local onde o arquivo foi enviado. Com um servidor mais distante escolhido de quem envia o arquivo, uma latência e um tempo de transferência maiores são gastos no *upload*.

Esses resultados demonstram que o ZooClouS, com base no cálculo de custo feito, envia o arquivo para os servidores que estiverem mais próximos de quem solicita o envio, pois a latência é menor quando a origem e o destino estão mais próximos geograficamente. Como a política considera a latência como fator mais importante, o resultado do tratamento é o envio do arquivo ao servidor mais próximo possível do cliente. Foram observados durante os testes, que a latência entre o solicitante no Brasil e os servidores ZooNimbus localizados nos EUA diferenciavam em poucos milissegundos, fazendo com que o espaço livre em disco e o tempo de disponibilidade tornassem os fatores decisivos na escolha do melhor servidor.

A partir dos resultados da escolha de envio do arquivo, foi medido o tempo de transferência de um arquivo que foi enviado ao ZooNimbus e tratado com a ZooClouS. O tempo encontrado foi comparado com o tempo de transferência encontrado nos envios de arquivos feito ao ZooNimbus utilizando a forma aleatória presente na sua antiga versão

(BioNimbus). A Figura 5.2 demonstra o tempo médio obtido na transferência dos *uploads* realizados.

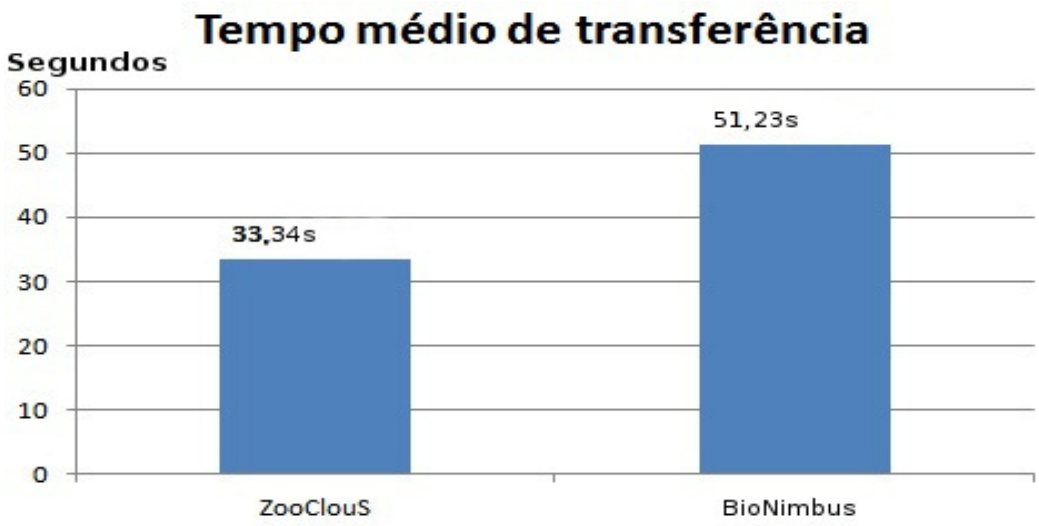

Figura 5.2: Tempo Médio de Transferência de Arquivo com o ZooClouS e de Forma Aleatória.

Como o ZooClouS escolhe os servidores de destino mais próximos de quem solicita o envio do arquivo, a transferência de dados torna-se mais rápida, fazendo com que a banda e os recursos utilizados na transferência de um arquivo sejam liberados com mais velocidade. A taxa de transferência média foi calculada com base no tempo médio de transferência do arquivo nas nove operações realizadas em cada algoritmo. Comparando a implementação do ZooClouS com a forma anterior de envio, o tempo de transferência foi melhorado em cerca de 65%. Na Seção 5.2 estão os testes feitos para a execução de um *pipeline* dentro do ZooClous.

5.2 Execução de Tarefas

Para os testes de execução de tarefas foram utilizadas duas federações, uma localizada na Universidade de Brasília (UnB) e outra na Amazon, ambas com três máquinas, estes testes foram realizados utilizando dados de bioinformática, dados descritos na Subseção 5.2.1.

5.2.1 Ambiente 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 [24], uma ferramenta de bioinformática, disponível como serviço nos servidores ZooNimbus.

Nesse cenário, foi criado um grupo de tarefas composto por cinco 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 5.1 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 segundos, 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 5.1. Para cada arquivo de entrada, havia um par de tarefas, totalizando cinco tarefas para o conjunto de tarefas.

Tabela 5.1: Nomes e Tamanhos 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

Para o estudo de caso do tempo de execução de uma tarefa utilizando dados de bioinformática, foi criado um ambiente com uma federação que possui duas nuvens, as quais são:

- Uma nuvem na Universidade de Brasília, localizada no Brasil, com três computadores com processadores Core i7-3770 de 3,4 GHz, memória RAM de 8Gb, 2TB de disco rígido, sistema operacional Linux, distribuição Ubuntu 13.04. Nuvem fisicamente localizada no Brasil;
- Uma nuvem na Amazon [25], com três máquinas virtuais que possuem as seguintes configurações: processador Intel Xeon 2.4 Ghz, 613MB de memória RAM, disco rígido de 80GB e sistema operacional Linux, distribuição Ubuntu 12.04.2. Nuvem localizada nos EUA.

5.2.2 Resultados Obtidos

Com o ambiente descrito na Subseção 5.2.1, foram definidos cinco tarefas com os arquivos de bioinformática citados anteriormente na Subseção 5.2.1. Foram realizadas duas execuções, uma com a nuvem localizada na Universidade de Brasília e a outra execução com a nuvem criada na Amazon integrada com a nuvem da UnB, ambas com o ZooClouS. Na primeira execução, foi utilizado apenas um servidor ZooKeeper para gerenciar os dados das nuvens. No segundo caso, observou-se durante os testes, que com mais servidores ZooKeeper o tempo de escalonamento e execução das tarefas tornou-se mais rápido, portanto foram levantados três servidores ZooKeeper, um na nuvem da UnB e os outros dois na nuvem da Amazon. A Figura 5.3 mostra os resultados obtidos nos testes realizados.

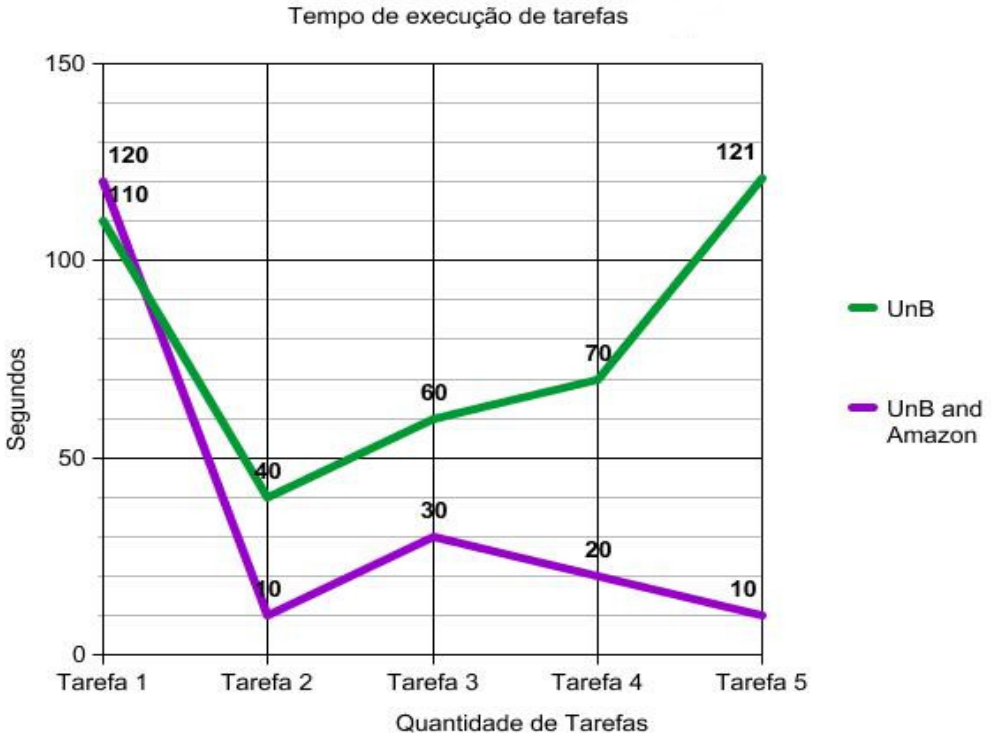

Figura 5.3: Tempo de Execução de Tarefas com o ZooNimbus.

No ambiente montado com as nuvens da UnB e Amazon, nota-se que a execução das tarefas tornou-se mais rápida, devido a quantidade maior de recursos disponiveis. Na nuvem criada somente na UnB, o tempo total de execução das cinco tarefas foi de 401 segundos. Na integração com a Amazon o tempo total foi de 190 segundos, o que resultou em uma execução 53% mais rápida.

5.3 Considerações Finais

De acordo com os testes analisados neste capítulo, chega-se a conclusão de que ao se utilizar a política de armazenamento para uma arquitetura de federação em nuvens, o ZooClouS. Tanto o tempo de transferência do arquivo, quanto a localidade onde os dados são armazenados, observou-se que tais dados são armazenados nas localidades mais próximas do cliente, no qual a latência é menor, concluindo assim, que o tempo de transferência utilizando o ZooClous reduz em relação ao tempo de transferência utilizado pelo ZooNimbus com uma política de armazenamento na qual existia anteriormente na versão BioNimbus, onde o cliente enviava o arquivo para aquele servidor o qual o mesmo conhecia o endereço, de forma aleatória, pois a conexão era feita de acordo com qualquer endereço que o cliente possua. Com as mudanças realizadas na arquitetura e para comprovar que quanto mais recursos a arquitetura possuir, mais rápido ocorre a execução de tarefas utilizando a aplicação Bowtie [24], como observado no gráfico da

Capítulo 6

Conclusão e Trabalhos Futuros

Neste trabalho foi proposto o ZooClouS, uma política de armazenamento de dados para a arquitetura de federação em nuvens ZooNimbus. Com base nos testes realizados, concluiu-se que o envio de arquivos na arquitetura de federação em nuvens ZooNimbus ocorre de forma menos custosa ao cliente e aos servidores, comparada à forma anterior do BioNimbus. Com as operações de replicação de dados, a disponibilidade dos dados aumentou e também a obtenção de arquivos pelo cliente e pelo servidor, com a operação de *download*. Com a implementação da tolerância a falhas ocorre a garantia de que todos os arquivos tenham ao menos uma cópia salva em outro servidor da federação, e caso um destes servidores se desconecte da federação, novamente a tolerância a falhas será acionada e replicará todos os arquivos indisponíveis daquele servidor.

Como trabalhos futuros propõe-se a implementação de uma lista de controle de acesso (ACL) a ser associada aos arquivos inseridos na federação, de modo que somente usuários autorizados possam utilizar e modificar tais arquivos. Além disso, pretende-se também implementar o Protocolo Paralelo de Transferência de Arquivos (P-FTP - *Parallelized File Transfer Protocol*) [37] para um ambiente de federação em nuvens, de forma a aumentar a velocidade na transferência de dados entre os clientes e os servidores. Como melhoria do cálculo da ZooClouS propõe-se a alteração do cálculo do *costs*, a última parte do custo de armazenamento. Atualmente esse custo está sendo computado de maneira fixa. O ideal seria que este custo seja calculado de acordo com o tamanho do arquivo, multiplicando o tamanho do arquivo pelo custo por *gigabyte* estabelecido pelo provedor de nuvem.

Para que a maioria dos protocolos de transferências de arquivos possam ser utilizados pelo ZooNimbus, propoêm-se a integração da *Interface de Gerenciamento de Dados em Nuvem* (CDMI - *Cloud Data Manage Interface*) [36], que trabalha com vários protocolos de transferência, gerando assim uma nova proposta, na qual se trate de uma política que resolva a escolha da melhor forma de se transferir o arquivo, por meio dos protocolos disponíveis na arquitetura de federação de nuvens.

Com a nova arquitetura do ZooNimbus, foram executadas tarefas por meio da ferramenta *Bowtie*, a execução das tarefas mostrou-se mais rápida quando a quantidade de recursos disponíveis na federação é maior, conforme os testes realizados neste trabalho. Porém as tarefas que foram executadas, foram escalonadas e seus arquivos de saída armazenados, sem ser utilizados posteriormente. Portanto, propõe-se também como trabalhos futuros, a execução de um *pipeline* dentro do ZooNimbus.

Referências

[1] A. P. F. Araújo. *Paralelização Autonômica de Metaheurísticas em Ambientes de Grid*. PhD thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, PUC-Rio, Brasil, 2008. 5, 6

[2] C. A. L. Borges. Escalonamento de tarefas em uma infraestrutura de computação em nuvem federada para aplicações em bioinformática, 2012. Monografia de graduação, Departamento de Ciência de Computação, Universidade de Brasília. 9

[3] R. Buyya, R. Ranjan, and R. N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Proceedings of the 10th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'10, pages 13–31, Berlin, Heidelberg, 2010. Springer-Verlag. 1, 12

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, jun. 2009. 7, 9

[5] Creative Commons. Mongo db. http://www.mongodb.org/, 2013. 24

[6] Microsoft Corporation. Windows azure. http://www.windowsazure.com/pt-br/pricing/free-trial/, 2012. 1, 7, 9, 49, 51

[7] Douglas Crockford. Json. http://json.org/, 2012. 28

[8] Olhar Digital. Qual o futuro da computação em nuvem ? http://olhardigital.uol.com.br/negocios/central_de_videos/qual-o-futuro-da-computacao-em-nuvem/14732/integra, 2012. 6

[9] Sales Force. Sales cloud. http://www.salesforce.com/sales-cloud/overview/, feb. 2013. 8

[10] Apache Software Foundation. Apache cassandra. http://cassandra.apache.org/, 2009. 24

[11] Apache Software Foundation. Apache avro). http://avro.apache.org/, 2012. 2, 24, 27, 30, 37, 44

[12] Apache Software Foundation. Apache hadoop). http://hadoop.apache.org/, 2012. 27

[13] Apache Software Foundation. Apache thrift. http://thrift.apache.org/, 2012. 29

[14] The Apache Software Foundation. Apache zookeeper. http://zookeeper.apache.org/, 2010. vii, 2, 24, 25, 26, 27, 29, 37

[15] The Apache Software Foundation. Apache hbase. http://hbase.apache.org/, 2013. 24

[16] Google. Google app engine. https://developers.google.com/appengine/docs/whatisgoogleappengine?hl=pt-br, 2013. 1, 7, 9, 11

[17] Google. Google drive. http://drive.google.com, feb. 2013. 8

[18] Google. Google file system. http://research.google.com/archive/gfs.html, feb. 2013. 9

[19] R. Hamann. http://www.tecmundo.com.br/infografico/9421-a-evolucao-dos-computadores.htm, 2011. 5

[20] Google Inc. Protocol buffers. https://developers.google.com/protocol-buffers/, 2012. 29

[21] Y. Jadeja and K. Modi. Cloud computing - concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference*, pages 877–880, mar. 2012. 6

[22] M. T. Jones. Cloud computing platforms and applications. http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux/l-cloud-computing/l-cloud-computing-pdf.pdf, 2008. 5

[23] L. Kwei-Jay and J.D. Gannon. Atomic remote procedure call. *Software Engineering, IEEE Transactions on*, SE-11(10):1126–1135, 1985. 27, 30, 32

[24] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-eficient alignment of short dna sequences to the human genome, 2009. 51, 53, 55

[25] Amazon Web Services LLC. Amazon elastic compute cloud (EC2). http://aws.amazon.com/pt/ec2/, 2012. 1, 7, 9, 49, 51, 54

[26] G. Magalhães, C. Schepke, and N. Maillard. Comparação entre plataformas de computação em nuvem. *ERAD 2012*, mar. 2012. viii, 11

[27] Microsoft. Windows azure. http://www.windowsazure.com/pt-br/, 2013. 11, 12

[28] Nmap. http://nmap.org/, 2013. 12

[29] U.S. National Library of Medicine. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/, 2013. 54

[30] S. Rajan and A. Jairath. Cloud computing: The fifth generation of computing. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 665 –667, jun. 2011. 9, 10

[31] M. Rouse. http://searchcloudprovider.techtarget.com/definition/What-is-cloud-federation, 2011. 13

[32] A. Ruiz-Alvarez and M. Humphrey. A model and decision procedure for data storage in cloud computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 572 –579, may 2012. 14

[33] H. Saldanha, E. Ribeiro, C. Borges, A. Araújo, R. Gallon, M. Holanda, R. Walter, M. E.and Togawa, and J. C. Setubal. *BioInformatics*. In Tech, 2012. 2, 17

[34] H. V. Saldanha. Bionimbus: uma arquitetura de dederação de nuvens computacionais híbrida para a execução de workflows de bioinformática. Master's thesis, Departamento de Ciência de Computação, Universidade de Brasília, 2012. vii, 3, 8, 9, 14, 17, 18, 19, 31, 32, 39

[35] Amazon Web Services. Amazon web services. http://aws.amazon.com/pt/, 2013. 11

[36] SNIA, http://snia.org/sites/default/files/CDMI%20v1.0.2.pdf. *Cloud Data Management Interface*, 1.0.2 edition, jun. 2012. 14, 15, 56

[37] S. Sohail, S. Jha, and H. ElGindy. Parallelized file transfer protocol (p-ftp). In *Local Computer Networks, 2003. LCN '03. Proceedings. 28th Annual IEEE International Conference on*, pages 624–631, 2003. 56

[38] Symantec. Glossário. http://www.symantec.com/pt/br/security_response/glossary, 2012. Security Response, Glossário. 15, 16

[39] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, dec. 2008. vii, 7, 8

[40] VMware. Diagram hybrid cloud. http://www.vmware.com/br/cloud-computing/public-cloud/index/, 2012. vii, 10

[41] E. White, L. McMillan, P. Romanski, M. O'Gara, and J. Bloomberg. Inter-cloud peering points. http://cloudcomputing.sys-con.com/node/1658700, 2010. vii, 13

[42] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu. Cloud storage as the infrastructure of cloud computing. In *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*, pages 380 –383, jun. 2010. vii, 14, 15

[43] L. Wu and R. Buyya. Service level agreement (sla) in utility computing systems. *CoRR*, abs/1010.2881, 2010. 20

[44] Y. Xiaogao and Y. Xiaopeng. A new grid computation-based min-min algorithm. In *Fuzzy Systems and Knowledge Discovery, 2009. FSKD 09. Sixth International Conference on*, volume 1, pages 43–45, 2009. 47

[45] Ren Xun-Yi and Ma Xiao-Dong. A* algorithm based optimization for cloud storage. *JDCTA*, 4(8):203–208, 2010. 3, 46, 48

[46] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. RFC 4253, Internet Engineering Task Force (IETF), jan 2006. 42