

***SESHAT*: UMA ARQUITETURA DE
MONITORAÇÃO ESCALÁVEL PARA
AMBIENTES EM NUVEM**

VINICIUS SILVA CONCEIÇÃO

***SESHAT*: UMA ARQUITETURA DE
MONITORAÇÃO ESCALÁVEL PARA
AMBIENTES EM NUVEM**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: DORGIVAL OLAVO GUEDES NETO

Belo Horizonte

Julho de 2018

© 2018, Vinicius Silva Conceição
Todos os direitos reservados

**Ficha catalográfica elaborada pela Biblioteca do ICEx -
UFMG**

Conceição, Vinicius Silva.

C744s Seshat: uma arquitetura de monitoração escalável
para ambientes em nuvem / Vinicius Silva Conceição
— Belo Horizonte, 2016.
xxiii, 79 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal
de Minas Gerais – Departamento de Ciência da
Computação.

Orientador: Dorgival Olavo Guedes Neto.

1. Computação – Teses. 2. Sistemas distribuídos
3. Big Data. 4. Computação em nuvem I. Orientador.
II. Título.

CDU 519.6*73(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

Seshat: uma arquitetura de monitoração escalável para ambientes em nuvem

VINICIUS SILVA CONCEIÇÃO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. DORGIVAL OLAVO GUEDES NETO - Orientador
Departamento de Ciência da Computação - UFMG


PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG


PROF. HUMBERTO TORRES MARQUES NETO
Departamento de Ciência da Computação - PUC/MG

Belo Horizonte, 13 de julho de 2018.

Dedico este trabalho à minha família, aos meus amigos e a todos os fóruns de suporte na internet.

Agradecimentos

Gostaria de dedicar esse pequeno espaço para agradecer a todos que de alguma forma fizeram parte da minha caminhada. Embora não seja a pessoa mais brilhante do mundo quando se trata de colocar emoções em palavras, farei o melhor para não me esquecer de ninguém. Sem qualquer sombra de dúvida, agradeço primeiramente aos meus pais pelo apoio incondicional em todas as etapas da minha vida, pela educação que me proporcionaram, ensinamentos, pela formação do meu caráter e os incansáveis esforços em se manter presentes mesmo quando resolvo ir para os lugares mais longínquos.

Além disso, gostaria de agradecer a minha esposa Rachel por todo o carinho e apoio durante os momentos difíceis, pelas diversas conversas dos mais variados assuntos (nos últimos anos, não consigo me lembrar de um único dia que não tenha conversado com ela), pelas experiências que estamos vivendo e pelo crescimento como ser humano que me proporciona diariamente. Não posso também me esquecer de um pequeno ser de quatro patas chamado Atum, embora ele não possa me compreender e por mais estranho que pareça agradecer um gato, este animalzinho sozinho foi responsável por aliviar(e criar também) diversos momentos de stress durante todo o mestrado. Seu pelo quentinho e sua companhia certamente auxiliaram minha saúde física e mental durante o inverno de Illinois.

Em Belo Horizonte, tive a felicidade de conhecer e trabalhar com colegas muito talentosos. Gostaria de agradecer todo pessoal do laboratório do e-Speed. Agradeço ao Waltinho, a pessoa mais cheia de ideias que já encontrei, um dia conseguiremos encontrar uma ideia que sirva para criar um aplicativo milionário. Agradeço ao Derick, um matemático/programador brilhante responsável por scripts obscuros e resolução de questões de maneira inusitada e a criação de piadas internas(*memes*) como "Kappa Guava" e "Vou Vuando Wolverine". Agradeço ao Gui Maluf, um cara detentor de um conhecimento sobre infraestrutura que muito provavelmente beira o infinito, psicopata do software livre e desenvolvimento estruturado, posso dizer que sem o Gui salvando a minha pele na manutenção e utilização de uma enorme quantidade de ferramentas este trabalho certamente não teria sido concluído, espero não ter dado tanto trabalho

quanto imagino :p . Agradeço ao Nestor, o engenheiro eletricitista mais programador que conheço, obrigado pela ajuda em todas as etapas de desenvolvimento deste trabalho, por corrigir e oferecer sugestões em todo o texto, me acompanhar nas publicações que conseguimos, e por todas as reflexões e conversas que tivemos. Espero que os frutos deste trabalho lhe auxiliem na conclusão de seu doutorado.

Por último, agradeço a todos os professores do DCC e de outras instituições que estudei e que contribuíram para o conhecimento que possuo hoje. Particularmente, gostaria de agradecer ao meu orientador Dorgival Guedes, pela sua sabedoria e experiência ao me colocar na direção correta durante todo o trabalho e também à sua flexibilidade em me orientar mesmo que distante. Tive o privilégio de aprender aspectos técnicos com um excelente profissional dos quais jamais me esquecerei. Mesmo com toda a pressão durante meus estudos, não houve uma única vez em que não tenha me sentido mais aliviado após conversar com ele.

“It’s not how much time you have, it’s how you use it.”
(Ekko - The boy who shattered time)

Resumo

A monitoração de sistemas computacionais cumpre papel essencial para gerir, viabilizar a manutenção e oferecer *feedback* através da coleta de dados de seus usuários. Para o caso específico de nuvens computacionais, a monitoração deve tratar diferentes tipos de recursos virtualizados de comportamento dinâmico, métricas de desempenho da infraestrutura, *logs* de aplicações e dados sobre a execução em ambientes distribuídos. Atender a todos esses requisitos e lidar com a complexidade envolvida em sistemas em nuvem é certamente um desafio. Neste trabalho investigamos as necessidades de um sistema de monitoração em nuvem e após identificarmos algumas funções essenciais, propusemos uma arquitetura de uso geral para supri-las. A arquitetura de monitoração em nuvem concebida em camadas é elástica, escalável, resiliente e extensível.

Aplicamos os conhecimentos obtidos no projeto e implementação de um sistema de monitoração composto de ferramentas de código aberto que instancia a arquitetura proposta. Como objeto de monitoração, selecionamos um ambiente virtualizado de processamento de dados massivos que é comumente utilizado em nuvem. Nossos experimentos comprovaram a escalabilidade do sistema conseguindo reduzir até pela metade o atraso na recepção de mensagens. Adicionalmente, incluímos uma série de casos de uso relacionados as nossas experiências com a ferramenta desenvolvida.

Palavras-chave: *Big Data*, Computação em Nuvem, Monitoração, Sistemas Distribuídos.

Abstract

System monitoring has an essential role in management, maintenance and to provide feedback through data collection. Particularly, cloud monitoring must address several virtualized resources of dynamic behavior, such as infrastructure performance metrics, application logs, and data produced in distributed environments. To meet all of these requirements and deal with complexity involved in cloud systems is certainly a challenge. This work has investigated the needs of a cloud monitoring system and after identifying some essential roles, we propose a general use architecture to supply them. The layered cloud monitoring architecture is elastic, scalable, resilient and extensible.

Lastly, we apply knowledge obtained designing a monitoring system based on open source tools that deploys the proposed architecture. As a monitoring subject, we have selected a virtualized big data environment that is usually deployed in clouds. Our experiments proved the system's scalability, achieving up to half the delay time in message reception. Additionally, we have included a number of use cases related to our experiences with the system.

Keywords: Big Data, Cloud-Computing, Network Monitoring, Distributed Systems.

Lista de Figuras

4.1	Ilustração das camadas básicas de um sistema de monitoração de nuvem. .	26
4.2	Estrutura interna da camada de Transporte	29
4.3	Arquitetura de um sistema de monitoração	35
5.1	Camada de Infraestrutura	38
5.2	Camadas de Infraestrutura e Sistemas Distribuídos	38
5.3	Agentes instalados em um host enviando dados ao servidor remoto.	41
5.4	Arquitetura Senu.	42
5.5	Camadas de Infraestrutura, Sistemas Distribuídos e de Coleta.	43
5.6	Exchanges em RabbitMQ. Fonte adaptada de [Pivotal, 2007]	44
5.7	Exemplo de mensagem não tratada.	45
5.8	Exemplo de mensagem tratada.	46
5.9	Camadas de Infraestrutura, Sistemas Distribuídos, Coleta e Transporte. . .	46
5.10	Camadas de Infraestrutura, Sistemas Distribuídos, Coleta, Transporte e Armazenamento.	48
5.11	Exemplo de <i>dashboard Kibana</i> . No canto esquerdo existem informações relacionadas aos campos indexáveis dos <i>logs</i> coletados. Em rosa estão apresentadas as porcentagens de mensagens para três níveis de verbosidade diferentes (TRACE, DEBUG, INFO). Ao centro, um histograma apresenta o volume de mensagens em função do tempo e, logo abaixo, um pequeno conjunto de mensagens de log mais recentes de acordo com seu <i>timestamp</i> .	49
5.12	Exemplo de <i>dashboard Grafana</i> . As cores em cada uma das barras representam a quantidade de dados recebidos ou transmitidos por uma interface de rede.	49

5.13	Exemplo de <i>dashboard Uchiwa</i> . Podemos observar uma lista contendo os nomes e os endereços IP de todos os coletores registrados no sistema. Além disso, ainda são apresentadas informações como a ocorrência de um evento, a versão do coletor e quanto tempo se passou desde a última comunicação com o sistema monitor. Os agentes marcados com tarja vermelha apresentam problemas críticos. Nesse caso, foram desconectados ou estão inaptos a coletar dados.	50
5.14	Camadas de Infraestrutura, Sistemas Distribuídos, Coleta, Transporte, Armazenamento e Visualização.	51
5.15	Exemplo de criação de alarmes no <i>Grafana</i> . Um alarme é configurado para criar notificações via e-mail caso a utilização de CPU atinja 50%. Além disso, a condição do alarme é avaliada a cada intervalo de 60 segundos. Esse tipo de alarme pode ser definido rapidamente por qualquer usuário. .	52
5.16	Modelo de execução <i>Docker</i> . Fonte adaptada de [Combe et al., 2016] . . .	53
5.17	Serviço <i>HTTP Listener</i> com três réplicas. [Docker, 2016]	55
5.18	Serviço global e replicado em <i>swarm</i> . [Docker, 2016]	55
6.1	<i>Dashboard</i> de monitoração da fila de mensagens durante o experimento sobre a escalabilidade no sistema de processamento de <i>logs</i>	62
6.2	Dashboard de monitoração acompanhando um grupo de hosts, onde pode-se acompanhar a utilização de CPU, consumo de memória, escritas em disco e tráfego de rede enquanto uma aplicação Spark é executada em um <i>cluster</i> virtualizado.	64
6.3	Dashboard de monitoração acompanhando a utilização de memória heap e de eden-space durante a execução de uma aplicação Spark no momento em que o problema foi detectado.	65
6.4	Dashboard de monitoração acompanhando a utilização de memória heap e de eden-space durante a execução de uma aplicação Spark no momento após a resolução do problema.	66

Lista de Tabelas

6.1	Desempenho e escalabilidade do sistema de monitoração sob volume elevado de dados de métricas	61
6.2	Desempenho e escalabilidade do sistema de monitoração sob volume elevado de mensagens de <i>logs</i>	63

Sumário

Agradecimentos	ix
Resumo	xiii
Abstract	xv
Lista de Figuras	xvii
Lista de Tabelas	xix
1 Introdução	1
1.1 Objetivos	3
1.2 Desafios	3
1.3 Contribuições do trabalho	4
1.4 Organização do texto	4
2 Conceitos básicos e trabalhos relacionados	7
2.1 Conceitos de sistema de monitoração	7
2.2 Ambiente de computação em nuvem	9
2.2.1 Níveis de abstração	10
2.2.2 Métricas e <i>logs</i>	10
2.3 Ambiente de processamento de dados massivos	11
2.4 Soluções de código aberto	12
2.5 Soluções comerciais	14
2.6 Soluções alternativas	15
3 Monitoração em Ambientes de Nuvem	17
3.1 A necessidade de monitoração em nuvem	17
3.1.1 Planejamento de capacidade	17
3.1.2 Gerenciamento de recursos	18

3.1.3	Acordo de nível de serviço	18
3.1.4	Cobrança	18
3.1.5	Solução de problemas	19
3.1.6	Gerenciamento de desempenho	19
3.2	Definição do sistema	19
3.3	Requisitos para um ambiente de monitoração em nuvem	21
3.3.1	Demandas da arquitetura	21
4	A arquitetura de monitoração <i>Seshat</i>	25
4.1	Camada de coleta	26
4.1.1	Coleta de métricas	27
4.1.2	Coleta de logs	27
4.1.3	Modelos <i>pull</i> e <i>push</i>	27
4.2	Camada de transporte	27
4.3	Camada de armazenamento	30
4.4	Camada de visualização	32
4.5	Camada de alertas	32
4.6	Camada de correlação de eventos	33
4.7	Colocando os elementos em operação	34
5	Ferramenta de Monitoração	37
5.1	Ambiente e métricas	37
5.2	Critérios de seleção	40
5.3	Camada de coleta	41
5.4	Camada de transporte	43
5.5	Camada de armazenamento	47
5.6	Camada de visualização	48
5.7	Camada de alertas	50
5.8	Camada de correlação de eventos	50
5.9	Uso de <i>containers</i>	52
5.9.1	<i>Docker</i>	53
5.9.2	<i>Containers</i> distribuídos: <i>Docker swarm</i>	54
5.10	Escalabilidade e tolerância a falhas	56
5.11	Modularização do sistema	57
6	Resultados	59
6.1	Carga de trabalho	60
6.2	Desempenho e escalabilidade	60

6.2.1	Experimento de <i>stress</i> por métrica	60
6.2.2	Experimentos de <i>stress</i> por <i>log</i>	62
6.3	Casos de uso	63
7	Conclusão	67
	Referências Bibliográficas	69
	Glossário	79

Capítulo 1

Introdução

A facilidade e o baixo custo para se obter recursos computacionais em ambientes de nuvem faz com que soluções desse tipo se tornem cada vez mais populares, oferecendo disponibilidade de processamento, comunicação e armazenamento virtualmente infinitos. Essa flexibilidade no provisionamento de recursos favorece o modelo de negócios *pay-per-use*, no qual consumidores são tarifados por demanda. Esse modelo é atrativo tanto para provedores como seus clientes. Qualquer que seja sua finalidade e o serviço ofertado, nuvens se beneficiam do uso de máquinas virtuais (VMs) interconectadas por canais de comunicação de alta velocidade. Dessa forma, nuvens são complexas se considerarmos sua infraestrutura de hardware (constituída por servidores, estrutura de armazenamento, ativos de redes, refrigeração, etc.), as plataformas que suportam (muitas vezes constituídas por múltiplos sistemas operacionais) e suas aplicações.

Devido a isso, aplicações que nela executam também demonstram complexidade. Tomemos como exemplo aplicações de processamento de dados massivos [Zaharia et al., 2010], conhecidas pelo termo *big data*, que são executados em ambientes especiais de forma distribuída e manipulam grandes volumes de dados. Esse tipo de aplicação produz uma variedade de informações de grande importância como: registros de execução(*logs*), medidas de utilização de recursos (CPU, memória, disco, etc) e dados de tráfego de rede em cada máquina (largura de banda, pacotes, etc). É importante lembrar que, por se tratar de aplicações que usam ambientes em nuvem, que são virtualizados, ainda há que se considerar os *logs* do ambiente de virtualização e os sensores das máquinas físicas.

Dada essa complexidade, administrar os usuários, processos e consumo de recursos em qualquer nível, seja do ponto de vista do provedor da nuvem ou de um cliente que tenha muitas VMs se torna impraticável sem um sistema adequado de **monitoração** que possa gerir essa estrutura de maneira eficiente e escalável [Jain et al., 2007]. Além

disso, a criação de recursos virtuais acrescenta mais um nível de complexidade, já que permite a criação e remoção VMs a qualquer momento. Sistemas de monitoração para nuvem, de maneira diferente de sistemas de monitoração usuais, precisam ser capazes de lidar com essa volatilidade no estado do sistema [Park et al., 2011].

As formas de utilização de um sistema de monitoração em nuvem são variadas. Primariamente as informações provenientes de *logs* e métricas de toda a infraestrutura podem ser utilizadas tanto para detecção como resolução de problemas e acompanhamento do desempenho de aplicações [Romano et al., 2011]. Além disso, as mesmas informações também podem auxiliar no gerenciamento dos recursos e na avaliação da capacidade da infraestrutura [Wang et al., 2011]. Adicionalmente, provedores de nuvem passam por auditorias e, nesse contexto, a monitoração é obrigatória para verificar sua conformidade com acordos de nível de serviço [Massonet et al., 2011]. Por último, as métricas provenientes da monitoração podem auxiliar na tarifação dos serviços de nuvem. Por exemplo, para serviços do tipo SaaS (*Software as a Service*) o critério de cobrança pode ser a quantidade de usuários ativos, para serviços do tipo PaaS (*Platform as a Service*) a utilização de CPU ou tempo de duração de uma tarefa, e para serviços do tipo IaaS (*Infrastructure as a Service*) a quantidade de VMs utilizadas [Li et al., 2010]. Assim, a monitoração viabiliza o modelo de negócios adotado pela nuvem.

Estima-se que ao menos 25% dos dados corporativos seja proveniente de sistemas de monitoração, com previsão de crescimento seguindo o aumento de tamanho dos *data centers* [Kutare et al., 2010]. A Netflix, por exemplo, coleta milhares de métricas em seus micro-serviços a intervalos de 5 segundos [Coburn Watson, 2015]. Com os conjuntos de dados gerenciais crescendo em uma taxa de 10 bilhões de registros por dia [Anwar et al., 2015], técnicas tradicionais de monitoração e ferramentas de análise são incapazes de lidar com tal volume de dados. Com o intuito de monitorar esse volume de informações, existem diversas abordagens no mercado [CloudWatch, 2009; Hasselmeyer & d’Heureuse, 2010; Azure, 2010; CloudMonix, 2015; Monitis, 2006; Nagios, 1999]. Na maioria das vezes as soluções existentes surgem na forma de ferramentas que desempenham diversas funções e acompanham uma variedade de *plugins*, mas que apresentam limitações quando aplicadas ao ambiente de nuvem. Além disso, por se tratar de uma área multidisciplinar, um grande desafio na monitoração reside na forma de integração da variedade de ferramentas e informações a serem coletadas.

Tendo esses argumentos em mente, a contribuição deste trabalho é apresentar *Seshat*, uma arquitetura de monitoração elástica, escalável, resiliente e extensível para sistemas em nuvem¹ que oferece flexibilidade na montagem de soluções. Ao identificar

¹*Seshat* é o nome de uma deusa egípcia considerada “guardadora de registros, responsável por registrar a passagem do tempo” (Wikipedia, <https://en.wikipedia.org/wiki/Seshat>)

os diversos elementos de um sistema de monitoração de forma abstrata e explicitar os seus requisitos em termos da monitoração pretendida, *Seshat* permite reunir diferentes ferramentas e componentes com funções específicas de forma que seja possível avaliar o estado de um *cluster* como um todo ou de uma aplicação implementada em nuvem e fornecer *insights* relevantes sobre seu desempenho. A arquitetura proposta foi validada em um cenário real, para a monitoração de aplicações desenvolvidas para mineração de dados, as quais são executadas em ambiente Spark sobre HDFS, em um conjunto de VMs gerenciadas pelo controlador de nuvens Openstack. Outra contribuição do trabalho foi implementar um sistema de monitoração composto por ferramentas de código aberto que foram configuradas e integradas, inclusive contornando algumas de suas limitações, estruturada via rede distribuída de contêineres, atendendo aos requisitos da arquitetura. De acordo com nossos estudos essa abordagem nos pareceu promissora e original em sua concepção.

1.1 Objetivos

Considerando que a monitoração de sistemas em nuvem é complexa e requer a utilização de componentes distintos, um modelo indicando as funções específicas necessárias para a criação de um sistema desse tipo é essencial para a evolução desses ambientes. Assim, os objetivos deste trabalho são: (1) propor uma arquitetura generalizada de monitoração de sistemas em nuvem subdividida em componentes; (2) projetar e implementar uma ferramenta nos moldes indicados pela arquitetura e, sempre que possível aproveitando sistemas já existentes; e (3) demonstrar a viabilidade da ferramenta através de testes e casos de uso reais. Para isso, consideramos que o ambiente a ser monitorado é composto por um *cluster* de máquinas que podem ser virtuais ou físicas, onde aplicações de processamento distribuído em grandes volumes de dados, tipicamente, Apache Spark e Apache Hadoop são executadas.

1.2 Desafios

As infraestruturas de sistemas de computação em nuvem são complexas, o que requer maiores esforços para seu gerenciamento e monitoração. Em especial, podemos organizar os desafios relacionados à sua criação sob duas perspectivas: uma *visão macro* sobre as propriedades gerais que precisa apresentar, tais como: escalabilidade, elasticidade, extensibilidade, e uma *visão micro* sobre a complexidade imposta pelos diversos componentes necessários à sua construção e a forma como são interligados. Podemos

citar o caráter multidisciplinar que envolve desde um simples agente de coleta de dados até um banco de dados distribuído. Diferente de outros sistemas, onde componentes são simples programas ou scripts, em um sistema de monitoração componentes são sistemas inteiros por si só. Devido aos requisitos que devem ser atendidos é inviável a elaboração de um sistema de monitoração partindo do zero. Logo, surge um novo desafio, o de realizar a interconexão entre estes sistemas distintos (conceito conhecido na área de computação em nuvem como *plumbing*). Além disso, é necessário considerar os esforços para instalar e configurar corretamente cada um desses sistemas, que muitas vezes são desenvolvidos em linguagens diferentes. Adicionalmente, como discutido por Aceto et al. [2013], ainda é muito difícil encontrar na literatura padrões sobre procedimentos, formatos e métricas para avaliar sistemas de monitoração em nuvem.

Assim, o desafio no desenvolvimento de um sistema de monitoração está relacionado tanto a características gerais que o sistema precisa apresentar, quanto à complexidade das partes que o constituem. Em particular, sua multidisciplinaridade e a diversidade nas ferramentas envolvidas demonstram porque existem tantas abordagens empregadas pela comunidade.

1.3 Contribuições do trabalho

Tendo esses objetivos em mente, consideramos que este trabalho tem duas contribuições principais:

1. Uma **arquitetura de monitoração em nuvem**, organizada em componentes, que permite obter elasticidade, escalabilidade, resiliência e, principalmente, extensibilidade.
2. Uma **ferramenta de monitoração em nuvem** desenvolvida com base no conhecimento obtido durante a elaboração da arquitetura de monitoração. A ferramenta é capaz de armazenar tanto logs de sistema como métricas coletadas pelos agentes para fornecer informações a respeito da infraestrutura monitorada.

1.4 Organização do texto

Com base nessa argumentação, o restante deste trabalho está organizado da seguinte forma: o capítulo 2 discute conceitos e trabalhos relacionados na monitoração de sistemas, e o capítulo 3 discute o que se espera de uma solução de monitoração para sistemas em nuvem. O capítulo 4 descreve os detalhes da solução proposta, enquanto

o capítulo 5 descreve um sistema de monitoração que instancia a arquitetura. Finalmente, o capítulo 6 apresenta resultados de sua utilização, incluindo casos de uso reais e testes de desempenho, o capítulo 7 apresenta algumas conclusões e discute trabalhos futuros.

Capítulo 2

Conceitos básicos e trabalhos relacionados

Neste capítulo são introduzidos os conceitos iniciais que servem de alicerce para a compreensão deste trabalho. A primeira seção apresenta alguns conceitos a respeito da monitoração de sistemas e como ela pode ser utilizada. Essa discussão fundamenta qualquer proposta de sistema de monitoração. Na segunda seção, são apresentadas características principais do ambiente em nuvem que pretendemos monitorar. Por fim, introduzimos o ambiente de processamento de dados massivos e como ele servirá de objeto de estudo deste trabalho. Na segunda metade deste capítulo, apresentamos o panorama da monitoração sob o ponto de vista tecnológico. Os trabalhos relacionados foram organizados em três seções. Na primeira discutimos com maior detalhe ferramentas de código aberto que possuem objetivos similares ao trabalho proposto. Na segunda discutimos ferramentas de monitoração comerciais existentes no mercado atualmente. Por fim, na última seção são discutimos estilos alternativos de monitoração.

2.1 Conceitos de sistema de monitoração

A monitoração de sistemas é uma área em Tecnologia da Informação quase tão antiga quanto a própria criação de sistemas de computação. Monitoração é o processo pelo qual se utiliza ferramentas e medições para o gerenciamento de sistemas de TI [Turnbull, 2014]. Através dela, se proporciona a tradução de métricas produzidas por sistemas e aplicações em valor de negócio. O sistema de monitoração traduz métricas em uma forma mensurável da qualidade do serviço, que por sua vez, fornece feedback ao usuário para garantir que o serviço atende à demanda esperada. Para compreender melhor a forma como sistemas de monitoração vem sendo utilizados, podemos classifica-los em

três estágios de maturidade:

- **Manual ou não monitorado:** monitoração realizada de forma manual, iniciada pelo usuário ou não monitorado. Neste nível de maturidade, a monitoração é usualmente feita através de *checklists*, scripts simples e outras formas não automatizadas. Na maioria das vezes são monitorados somente componentes que já tenham apresentado falhas anteriormente, falhas essas que são resolvidas pela ação manual do administrador. O objetivo da monitoração neste nível é uma tentativa de reduzir *downtime*, fornecendo pouco ou nenhum valor que indique a qualidade do serviço.
- **Reativo:** monitoração realizada de forma automática, com apenas alguns componentes monitorados de forma manual ou não monitorados. Neste nível, ferramentas de variada sofisticação são utilizadas para realizar a monitoração, e usualmente, coletam apenas métricas simples como CPU, memória e em alguns casos dados sobre desempenho. A finalidade da monitoração é verificar a disponibilidade dos recursos, sendo apenas alguns dados utilizados para medir a qualidade do serviço. A maioria dos dados precisa ser manipulado ou transformado antes de ser utilizado, e existe pouco ou nenhum esforço na criação de *dashboards* para sua visualização.
- **Proativo:** monitoração é considerada essencial para o gerenciamento da infraestrutura e ao modelo de negócios. A monitoração é completamente automatizada e gerada por gerentes de configuração. Além disso, são utilizadas múltiplas ferramentas para a monitoração que realizam funções específicas e são conectadas para permitir o fluxo de informações. As verificações e coleta de dados são orientadas a medir a qualidade do serviço ou o desempenho de aplicações. Os dados de desempenho são frequentemente utilizados em análises e resolução de falhas. O objetivo é medir a qualidade do serviço e auxiliar o setor de TI. A maioria dos dados fornecidos são aplicados diretamente aos negócios, times de desenvolvimento de aplicações através de *dashboards* e relatórios.

Neste trabalho, apresentaremos uma arquitetura de monitoração no capítulo 4 que visa orientar desenvolvedores a estabelecer uma monitoração proativa, tendo sua funcionalidade validada por implementação e avaliação de alguns casos de uso, apresentados no capítulo 6.

2.2 Ambiente de computação em nuvem

As características gerais dos três modelos de serviço da computação em nuvem (SaaS, PaaS e IaaS) incluem auto-serviço sob demanda, acesso à banda larga, agrupamento de recursos, rápido provisionamento de recursos e monitoração de recursos até um certo nível [Spring, 2011a,b]. De acordo com a Cloud Security Alliance [Brunette et al., 2009], uma infraestrutura de nuvem pode ser modelada com sete camadas: infraestrutura, rede, *hardware*, sistema operacional, *middleware*, aplicação, e usuário. Cada uma delas é detalhada a seguir incluindo quais informações podemos extrair sob a perspectiva da monitoração.

- **Estrutura física:** esta camada considera as instalações físicas abrangendo *datacenters* que realizam a computação, incluindo energia e refrigeração.
- **Rede:** esta camada considera as ligações entre hosts dentro da nuvem e entre a nuvem e o próprio usuário. A monitoração desta camada permite uma visão de como se comporta o tráfego na rede, os padrões de comunicação entre as máquinas e possíveis pontos de gargalo de rede.
- **Hardware físico e virtual:** esta camada considera os componentes físicos como CPU's, memória, tomadas de energia, além de componentes virtuais como VCPU's, discos rígidos virtuais, memória virtual, etc. Essa é uma das camadas de maior importância para a monitoração, uma vez que são os componentes desta camada que fornecem a maioria das métricas relacionadas ao uso de recursos e desempenho. A partir dela podemos obter uma visão de como a utilização de recursos virtuais influencia os recursos físicos e verificação de falhas de componentes.
- **Sistema operacional:** nesta camada são considerados os componentes que formam o sistema operacional de um host, podendo ser o sistema operacional executando em uma máquina física ou virtual. A monitoração nessa camada fornece uma visão a respeito dos processos sendo executados, o consumo de tempo de cada um deles, trocas de contexto do sistema e paginação de memória.
- **Middleware:** esta camada considera o software executando entre o sistema operacional e a aplicação de usuário. Embora este seja um tópico abrangente, podemos citar como exemplo hipervisores como os que coordenam a virtualização dos recursos físicos e bibliotecas especializadas. A monitoração nessa a camada permite obter informações que correlacionam máquinas físicas e virtuais, instanciação de novas máquinas virtuais e redes virtuais.

- **Aplicação:** esta camada considera as aplicações executadas por usuários da nuvem. Em especial, podemos citar o processamento de dados massivos que são o objeto de estudo deste trabalho. Nela são executadas as ferramentas de processamento de dados distribuídos, como por exemplo, Spark e Hadoop.
- **Usuário:** esta camada considera o usuário final de sistemas em nuvem e as aplicações que executam fora da nuvem. Por exemplo, um *browser* que executa no computador do usuário.

No contexto de monitoração em nuvem, essas camadas servem de orientação sobre onde devem ser colocados agentes. Na verdade, a camada na qual agentes são inseridos influencia diretamente o tipo de informações coletadas, assim como o que pode ser observado. É importante lembrar que o acesso e a monitoração de cada uma dessas camadas está intimamente relacionada com o serviço de nuvem utilizado, variando de acordo com as limitações impostas pelo provedor da nuvem.

2.2.1 Níveis de abstração

Em nuvem podemos classificar a monitoração entre alto nível e baixo nível, sendo ambas importantes [Caron et al., 2012]. A monitoração de alto nível está relacionada à informações sobre o estado da plataforma virtual e da aplicação. Isto é, informações coletadas nas camadas de *middleware*, aplicação e usuário. No contexto de SaaS, são de maior interesse para o usuário, por estar relacionadas à qualidade do serviço. Por outro lado, a monitoração de baixo nível se refere a informações coletadas a respeito do estado físico da nuvem (métricas como CPU, memória, uso de disco, temperatura, tensão elétrica, etc) e são geralmente de interesse do provedor. Vale observar que métricas deste ultimo tipo são, usualmente, coletadas pelos provedores e não apresentadas ao consumidor. No contexto de serviços do tipo IaaS, a monitoração de alto e de baixo nível são de interesse para ambos, provedores e consumidores.

2.2.2 Métricas e logs

De maneira geral, para a monitoração de ambientes em nuvem, os dados de interesse podem ser classificados segundo essas duas categorias: *métricas* são dados simples, normalmente de tipos numéricos (inteiros ou de ponto flutuante), obtidos periodicamente. Nessa categoria se enquadram dados como a porcentagem de utilização de uma CPU, o volume de dados recebidos ou enviados por unidade de tempo, etc. Normalmente métricas são disponibilizadas através de pontos de medição que podem ser acessados para se obter o valor corrente de uma certa grandeza. O produto resultante

da monitoração periódica é uma série temporal. Cada dado isoladamente é bastante simples e a evolução de seus valores ao longo do tempo tende a ser mais relevante que valores individuais. Por outro lado, *logs* (ou registros de execução) são normalmente mensagens textuais de formato mais complexo, sem um padrão global pré-definido, que podem trazer em uma única linha uma variedade de informações, como detalhes de uma requisição de um cliente recebida pelo sistema, ou as condições de um erro detectado durante a operação, por exemplo. Normalmente são produzidos por comandos inseridos no código de uma aplicação ou no sistema operacional pelo programador para registrar a ocorrência de determinados eventos e são armazenados inicialmente em arquivos de *logs* configurados pelo administrador do sistema.

Por sua natureza diversa, todo sistema de dados de monitoração em nuvem deve reconhecer as diferenças entre essas duas categorias e estar preparado para tratá-las de forma diferenciada. Métricas, por sua simplicidade, ocupam isoladamente pouco espaço, mas devem ser facilmente recuperadas em longas sequências, sempre com a informação de tempo associada. *Logs* precisam ser tratados como registros complexos que ocupam individualmente mais espaço, com padrões que podem variar entre uma e outra mensagem. Nesse caso, o acesso a eles pode exigir mecanismos de detecção de padrões complexos para extrair as diversas informações que podem estar contidas em uma única mensagem.

2.3 Ambiente de processamento de dados massivos

Atualmente, é importante observar um considerável crescimento no processamento de um tipo de carga que tem se tornado cada vez mais comum, o processamento de dados massivos [Mashayekhy et al., 2014; Gu et al., 2011]. Com um grande volume de informações provenientes de seus clientes das mais diversas formas, cada vez mais companhias estão interessadas obter informações que forneçam melhores decisões de negócios, operações mais eficientes, melhoria nos lucros e clientes mais felizes [Ferguson, 2016]. Desta forma, dado o elevado e crescente volume de dados que são produzidos todos os dias, é necessário lançar mão do uso de ambientes de processamento de dados massivos para extrair informações que favoreçam às tomadas de decisão dentro do negócio em questão.

Em uma infraestrutura típica para processamento de dados massivos, computadores são ocupados de forma colaborativa (*cluster*) trabalhando em paralelo, sendo que cada um assume uma parte da carga de processamento. Ao final as tarefas distribuídas entre eles são agregadas a fim de se gerar um resultado. Por se tratar de uma

grande quantidade de servidores e recursos e a serem gerenciados, o processamento de dados massivos é realizado na nuvem, onde existe grande flexibilidade e escalabilidade a partir da utilização de recursos virtuais. Isso é particularmente interessante tanto para provedores que desejam aproveitar o máximo de sua infraestrutura, quanto para usuários que procuram processar seus dados sem se preocupar com os custos envolvidos na manutenção de infraestrutura. Como sistemas de processamento de dados massivos podemos citar ferramentas tais como: Hadoop [Shvachko et al., 2010] que realiza o processamento de informações em *batch*, Spark [Zaharia et al., 2012] vem ganhando espaço por realizar o processamento em memória, Giraph [Ching, 2013] que realiza o processamento em grafos de redes sociais com trilhões de conexões e Hive [Barbierato et al., 2013] que atua como um armazém de dados capaz de realizar consultas similares às feitas em SQL sobre *big data*. Para nossos testes consideramos mais interessante trabalhar sobre o ambiente Spark, o que será mais detalhado na nossa arquitetura apresentada na seção 3.2.

Todo o ambiente de processamento de dados massivos descrito até este momento é solo fértil para monitoração. A maioria desses serviços não é ciente dos detalhes de implementação de outros serviços, e todos eles se comunicam por interfaces bem estabelecidas e estáveis. Nesse tipo de infraestrutura cada serviço mantém seu próprio *log* independentemente. Dentre eles estão: registros de requisição de usuários, resultados de execução de tarefas, quanto tempo durou uma tarefa, se houve erro, etc. Dados de *log* são independentes por padrão, e como consequência, são espalhados e armazenados em lugares isolados que precisam ser agregados para construir uma visão completa do que realmente acontece quando um usuário executa uma aplicação em nuvem. Nesse caso, um sistema de monitoração em nuvem que consiga centralizar essas informações pode beneficiar tanto a provedores de nuvem realizarem cobrança sobre os recursos utilizados [Anwar et al., 2015] como aos usuários buscando um detalhamento de suas aplicações [Lin & Ryaboy, 2013].

2.4 Soluções de código aberto

Sob a óptica tecnológica, a monitoração de informações em sistemas distribuídos vem sendo tratada de diferentes formas e ferramentas. Nagios [1999] é um exemplo já consolidado no mercado que oferece monitoração em larga escala. No entanto, sua arquitetura centralizada e baseada em *polling* não se adéqua ao dinamismo na monitoração de recursos virtuais, que são frequentemente utilizados no ambiente de nuvem. O mesmo tipo de limitação é apresentada por sistemas de monitoração em grid como os

descritos por Andreozzi et al. [2005], Newman et al. [2003], Buyya [2000] e Cooke et al. [2003]. Outro problema apresentado em Nagios são as limitações de escalabilidade. Projetado para funcionar em um único servidor para aumentar sua capacidade de monitoração é necessário criar novas instâncias autônomas que não compartilham dados, dificultando o gerenciamento. Por outro lado, Ganglia Massie et al. [2004] apresenta uma arquitetura distribuída que permite escalabilidade. No entanto, foi projetado para a monitoração de clusters bem definidos e com poucas mudanças na infraestrutura, dificultando também o trato de recursos virtuais dinâmicos.

Dentre os sistemas de monitoração de código aberto, Kutare et al. [2010] e Wang et al. [2011] apresentaram esforços relacionados à alta disponibilidade através do sistema Monalytics. Pelo uso da estrutura de DCG (*Distributed Computation Graph*) o sistema é capaz de prover escalabilidade e efetividade em cenários dinâmicos de Nuvem. Os autores também discutem a aplicação de algoritmos de análise de dados dentro dos próprios agentes de coleta de dados. Argumentam que máquinas apresentam limitação de desempenho na maioria das vezes pelo uso de memória do que pelo uso de CPU. Tal argumento é inválido atualmente, já que existem variadas aplicações em nuvem que apresentam uso intensivo de CPU Peng et al. [2015]. Além disso, o sistema não apresenta uma forma de armazenamento dos dados de longo prazo para análises posteriores e não há qualquer indício de que seja possível coletar *logs*.

Hasselmeyer & d’Heureuse [2010] apresentaram esforços na direção de um sistema de monitoração escalável através do uso de fila de mensagem para o transporte das informações oriundas dos agentes de coleta de dados. No entanto, como observado pelos próprios autores, com o aumento do número de agentes enviando informações podem ocorrer gargalos devido ao caráter centralizado apresentado pela fila de mensagens. Além disso, para o armazenamento dos dados no longo prazo foi utilizado um banco de dados comum, que é inadequado ao dinamismo apresentado em sistema de nuvem, como será discutido na seção 4.2.

Por fim, Monasca [2015] é um projeto desenvolvido pela comunidade OpenStack. A ferramenta apresenta estrutura de um sistema de monitoração completo com: *multi-tenancy*, escalabilidade e tolerância a falhas. Adicionalmente, possui capacidade de armazenamento tanto de logs do sistema como também métricas. Segundo o website da ferramenta, existem módulos responsáveis pela criação de alarmes e notificações através de limiares nas métricas monitoradas. Monasca é uma solução baseada na integração de ferramentas já adotadas pela comunidade, como: Kafka, Influxdb, Elasticsearch, MySQL e Apache Storm. No entanto, ao experimentar a ferramenta foi possível perceber uma elevada complexidade em sua implantação, consumindo semanas somente na instalação de seus componentes. Como descrito em Hasselmeyer & d’Heureuse [2010]

um sistema de monitoração precisa ser simples em dois aspectos: primeiramente, as interfaces do sistema devem ser de fácil entendimento, uso e implementação. Além disso, o sistema deve ser de fácil instalação e manutenção para operadores e clientes. Embora Monasca demonstre grande potencial para a monitoração de nuvens, ainda precisa evoluir no quesito simplicidade de implantação.

2.5 Soluções comerciais

Algumas ferramentas comerciais também vêm sendo utilizadas em ambientes de larga escala. Dentre elas podemos destacar o CloudWatch [2009], desenvolvido pela Amazon com o objetivo de monitorar serviços como o EC2, que de maneira geral são relacionados a plataformas virtuais. CloudWatch coleta diversos tipos de informação e os armazena por no máximo duas semanas. Períodos maiores de armazenamento são possíveis, porém a resolução (intervalo entre medições) é agregada em intervalos maiores. Por exemplo, medições realizadas a cada 60 segundos são disponíveis por um máximo de 15 dias; após esse período os dados apenas estão disponíveis em uma janela de medição de 5 minutos. Tal característica representa uma perda de informação em alguns casos.

O CloudMonix [2015], sucessor do AzureWatch, uma ferramenta desenvolvida para monitorar métricas chave de recursos providos pelo Azure [2010] como: instâncias de banco de dados, armazenamento e aplicações web. De acordo com informações disponíveis no site da empresa, a nova ferramenta inclui a capacidade de recuperação de serviços interrompidos, maior detalhamento na captura e apresentação dos dados e interface com painéis de exibição.

Monitis [2006] adota um esquema de instalação de agentes para monitorar recursos e informar os usuários sobre o desempenho dos serviços através de um servidor web que também é capaz de produzir alertas caso algum recurso se torne escasso. Além disso, os agentes executam periodicamente verificações pré-configuradas e as transmite ao servidor central utilizando o protocolo HTTPS. O que torna Monitis um sistema peculiar é o fato de que seus usuários não são responsáveis por manter o servidor: as informações coletadas pelos agentes são enviadas aos servidores da empresa. A partir daí, os usuários podem acessá-las através de um servidor web, limitando a escalabilidade pela banda de rede entre o sistema monitorado e os servidor Monitis. Embora a ferramenta ofereça uma API de desenvolvimento de plugins, assim como todas as outras soluções comerciais de monitoramento em nuvem, também possui bloqueio do fornecedor que limita o acesso ao código-fonte e possíveis ajustes na ferramenta.

2.6 Soluções alternativas

Observando cenários alternativos, Park et al. [2011] propõem um serviço de monitoramento baseado em cadeias de Markov para nuvens compostas por aparelhos móveis. O modelo desenvolvido analisa e prevê o estado dos recursos, tornando o sistema mais resistente a falhas ocasionadas pela volatilidade e o dinamismo de aparelhos móveis. Demers et al. [2007] introduz um sistema de monitoração escalável para processamento de eventos. Os autores desenvolveram uma linguagem própria para a expressão de eventos complexos que permite correlacionar sequências de eventos. Viratanapanu et al. [2010], com o objetivo de entender melhor os efeitos de desempenho que máquinas virtuais possuem em sistemas de nuvem, desenvolveram o sistema Pantau. Os autores argumentam que a monitoração de recursos de maneira independente é insuficiente para compreender como máquinas virtuais alocadas em um mesmo servidor afetam umas às outras. Dessa forma, através de uma arquitetura projetada para permitir a interação com hipervisores, a ferramenta é capaz de prover informações de grão fino a respeito dos recursos utilizados por máquinas virtuais.

Capítulo 3

Monitoração em Ambientes de Nuvem

Neste capítulo são apresentados alguns aspectos relacionados à monitoração em Nuvem. Inicialmente a seção 3.1 descreve atividades comuns ao ambiente de nuvem e como a monitoração é importante em cada uma delas. A seção 2.2 apresenta conceitos básicos de monitoração para contextualizar o leitor, enquanto a seção 3.3 analisa uma série de requisitos necessários à monitoração em nuvem. Por último, a seção 2.3 detalha o ambiente de processamento de dados massivos que será utilizado como referência de monitoração.

3.1 A necessidade de monitoração em nuvem

Monitoração em nuvem é uma tarefa importante tanto para provedores quanto consumidores. Por um lado, é uma forma de controlar e gerenciar toda a infraestrutura e servidores; por outro lado, registra as informações sobre sistemas e aplicações. Monitoração certamente contribui para as atividades cobertas por uma infraestrutura em nuvem. Nas próximas seções apresentamos algumas dessas atividades descrevendo o papel da monitoração em cada um delas.

3.1.1 Planejamento de capacidade

Para garantir que o desempenho de aplicações e serviços, desenvolvedores precisam quantificar a capacidade e a quantidade de recursos que devem ser providos a fim de atender tal demanda. Isso envolve uma série de fatores, como a maneira como aplicações foram projetadas, a carga sobre a qual devem executar e os níveis de qualidade que devem ser mantidos [Hasselmeyer & d’Heureuse, 2010]. Dessa forma, a monitoração se faz essencial para que provedores de nuvem consigam prever e manter

um histórico a respeito da evolução desses parâmetros [Katsaros et al., 2011] e assim planejar sua infraestrutura e recursos para se adequar a tais garantias.

3.1.2 Gerenciamento de recursos

Inicialmente para gerenciar uma infraestrutura complexa como a de uma nuvem é necessário um sistema de monitoração capaz de avaliar seu estado como um todo [Viratanapanu et al., 2010]. Virtualização vem ganhando importância ao longo dos anos por abstrair a heterogeneidade dos servidores em uma infraestrutura. Por sua vez, a tecnologia introduz um novo nível de complexidade no gerenciamento, uma vez que provedores precisam lidar com recursos virtualizados e físicos ao mesmo tempo [Katsaros et al., 2011]. Em geral, quando uma infraestrutura em nuvem é utilizada, espera-se que ela esteja disponível a todo momento. Assim, um sistema de monitoração confiável para todo o ambiente de nuvem é necessário para garantir essa disponibilidade esperada [Padhy et al., 2011].

3.1.3 Acordo de nível de serviço

Um acordo de nível de serviço (ANS) (ou ainda, **SLA**, do inglês *Service Level Agreement*) é uma acordo de comprometimento oficial entre um provedor de serviço e um cliente. Aspectos particulares do serviço como qualidade e disponibilidade são acordadas entre o provedor e o usuário do serviço [Wieder et al., 2011]. Nesse contexto, monitoração é obrigatória e contribui para certificar conformidade com o SLA durante atividades de auditoria pelo órgão regulador [Massonet et al., 2011]. Além disso, a monitoração também permite aos provedores a criação de modelos SLA mais realísticos e melhores preços através da exploração dos dados de seus usuários.

3.1.4 Cobrança

Uma das características nativas da nuvem é permitir que o consumidor pague proporcionalmente pelo uso do serviço sob diferentes métricas e granularidades, de acordo com o tipo de serviço e o modelo de preços adotado [Iyer et al., 2009]. Por exemplo, para serviços do tipo SaaS (*Software as a Service*) o critério de cobrança pode se basear na quantidade de usuários ativos, já para serviços do tipo PaaS (*Platform as a Service*) a utilização de CPU ou tempo de duração de uma tarefa, e para serviços do tipo IaaS (*Infrastructure as a Service*) a quantidade de VMs utilizadas [Li et al., 2010]. Para cada um dos critérios mencionados anteriormente, monitoração é necessária tanto

para que provedores realizem a cobrança, quanto para que os clientes acompanhem sua utilização e comparem os custos de diferentes provedores.

3.1.5 Solução de problemas

A complexa infraestrutura da Nuvem apresenta um grande desafio para a resolução de problemas, por exemplo identificação da origem de um erro, uma vez que a causa do problema precisa ser investigada em diversos componentes como o host de origem, a rede ou o serviço em execução. Além disso, cada um deles pode ser composto por várias camadas como recursos físicos ou virtuais e até mesmo envolver diferentes sistemas operacionais. Logo, um sistema de monitoração é necessário para que provedores identifiquem a causa de problemas em sua infraestrutura e consumidores esclareçam problemas de desempenho ou falhas em suas aplicações [Romano et al., 2011].

3.1.6 Gerenciamento de desempenho

Um dos motivos do modelo de nuvem ser atrativo aos consumidores é porque a manutenção dos servidores é delegada ao provedor. Assim, apesar de provedores estarem sempre preocupados com a qualidade de seus serviços, alguns servidores podem apresentar desempenho ordens de magnitude pior do que outros [Armbrust et al., 2009]. Nesse contexto, o gerenciamento de desempenho fornece ao consumidor a habilidade de identificar o quão bem suas aplicações estão executando através da monitoração do desempenho percebido e, caso haja degradação de desempenho aplicar ações corretivas.

3.2 Definição do sistema

Seguindo as diretrizes conceituais definidas no capítulo anterior, o ambiente que pretendemos monitorar começa a tomar forma. No contexto do ambiente de nuvem, selecionamos o provisionamento de nuvem realizado pelo *OpenStack* [Pepple, 2011], que é um sistema provedor de nuvem gratuito e de código aberto capaz de fornecer serviços no formato *infrastructure as a service (IaaS)*. Com *OpenStack* é possível criar máquinas virtuais com diversos tipos de configuração para a instanciación de um *cluster*, além de facilitar o escalonamento horizontal da infraestrutura.

Para compor a camada de aplicação e, definir o ambiente de processamento de dados massivos para o *cluster* em nuvem provisionado foram selecionadas duas ferramentas: Hadoop e Spark. Hadoop [Shvachko et al., 2010] pode ser visto como a implementação de software livre do paradigma de programação MapReduce [Dean &

Ghemawat, 2008], que se caracteriza por ser um modelo de processamento distribuído de grandes volumes de dados, de elevada robustez e escalabilidade. Outro aspecto a ser considerado é que o sistema de arquivos do Hadoop, o HDFS [Shvachko et al., 2010], distribui dados entre os discos rígidos das máquinas no *cluster* para aumentar desempenho de processamento tirando proveito da computação de dados locais. Já para contornar falhas, o HDFS estabelece uma política de replicação de blocos de dados distribuídos de forma a favorecer a conclusão do processamento, mesmo que algum nó falhe. HDFS utiliza o modelo de execução mestre-escravo. Uma entidade mestre única chamada *NameNode* é responsável por coordenar o acesso aos arquivos e distribuir tarefas às outras máquinas (escravos). Os escravos por sua vez são chamados *DataNodes*, geralmente um em cada nó do *cluster*, e gerenciam o armazenamento dos respectivos nós em que executam.

Por sua vez, Spark [Zaharia et al., 2012] se propõe a ser uma plataforma de processamento de dados massivos que adota um modelo de programação mais flexível. Spark se apresenta como um *framework* veloz e de uso geral para processamento de dados massivos. Tem chamado a atenção por ser possível trabalhar associado ou disjuncto do próprio Hadoop, mas tem sido alvo de consideração por oferecer nativamente armazenamento temporário persistente distribuído em memória [Zaharia et al., 2010]. Isso permite o reaproveitamento dos dados processados de forma acumulativa, o que tem elevado consideravelmente sua utilização. Assim como Hadoop, também utiliza o modelo de execução mestre-escravo. Aplicações Spark executam como um conjunto de tarefas independentes, coordenadas por um programa *Driver*. Mais especificamente, para executar Spark em um *cluster*, o *Driver* se conecta à um gerenciador de *cluster*. Uma vez conectado, Spark adquire executores a partir dos nós do *cluster*, que são processos responsáveis pela execução de tarefas. Frequentemente um *cluster* Spark é executado sobre o gerenciador de *cluster* chamado YARN [Vavilapalli et al., 2013], que atua como escalonador dos recursos que são oferecidos aos executores. Adicionalmente, também oferece uma interface gráfica chamada *History Server* que apresenta o histórico da execução das aplicações.

Finalmente, para fornecer ao leitor um panorama do sistema temos a ferramenta *OpenStack* que gerencia os recursos físicos disponíveis e de onde serão criadas máquinas virtuais para a composição de um *cluster*. Com o *cluster* definido, utilizaremos o sistema de arquivos HDFS para a distribuição e acesso de dados que serão processados através de uma aplicação executada na plataforma Spark. Esse conjunto define o objeto que pretendemos monitorar e de onde serão extraídas métricas e *logs* os quais envolvem as camadas de rede, hardware físico e virtual, sistema operacional e aplicação.

3.3 Requisitos para um ambiente de monitoração em nuvem

Ao considerarmos uma solução para monitoração de sistemas em nuvem, devemos abordar tanto as características esperadas do sistema como um todo, quanto as demandas específicas geradas pelos dados que devem ser tratados.

3.3.1 Demandas da arquitetura

A monitoração de ambientes em nuvem pode ser abordada sob duas perspectivas: uma *visão micro* que apresenta a complexidade imposta pelos componentes necessários à construção de sistemas de monitoração e a forma como são interligados, e uma *visão macro* sobre as propriedades gerais que um sistema precisa apresentar.

No que se refere a uma **visão micro**, é necessário conhecer a infraestrutura e os sistemas que nela operam para a aplicação correta de agentes coletores de dados. Além disso, é preciso conhecer os protocolos usados na comunicação entre coletores e as outras partes do sistema de monitoração, como por exemplo, o de banco de dados. O armazenamento das informações também possui papel relevante, o que requer a aplicação de um banco de dados que seja capaz de lidar com o volume de informações coletadas. Outro desafio que surge a partir do armazenamento, é que dados coletados e armazenados não fornecem conhecimento por si só. Dessa forma, técnicas de mineração e visualização de dados permitem extrair mais informação a partir dos dados. Como será detalhado no capítulo 4, é necessário coordenar e estruturar a forma como todos esses sistemas interagem e definir funções chave para que um sistema de monitoração em nuvem seja eficiente.

Na **visão macro**, sistemas em nuvem, se comparados a sistemas de hospedagem tradicionais, exigem monitoração mais complexa, escalável e robusta. Portanto, esses sistemas precisam ser capazes de atender a uma série de atributos, tais como **escalabilidade**, **elasticidade**, **extensibilidade**, **pontualidade**, **adaptabilidade** e **resiliência** [Aceto et al., 2013].

- **Escalabilidade:** Um sistema de monitoração é escalável se ele consegue lidar com o crescimento do número de agentes [Clayman et al., 2010]. Tal propriedade é essencial devido ao grande número de parâmetros monitorados em uma quantidade enorme de recursos na nuvem. A importância da escalabilidade aumenta quando se considera a virtualização, uma vez que, dentro de um único servidor físico, podem ser alocados muitos recursos virtuais. Devido a isso, para obter o

estado da nuvem é gerado um grande volume de dados provenientes de múltiplas fontes. Logo, um sistema de monitoração escalável precisa ser capaz de coletar, transportar e analisar esse volume de dados sem prejudicar o funcionamento normal da nuvem.

- **Elasticidade:** Um sistema de monitoração é elástico se ele consegue lidar com as mudanças nas entidades monitoradas. Isto é, precisa garantir que recursos virtuais criados e destruídos por expansão ou contração da nuvem sejam monitorados adequadamente [Clayman et al., 2010]. Tal propriedade implica na existência de escalabilidade e também adiciona o requisito de se oferecer suporte em tempo de execução de aumento e diminuição do conjunto de recursos monitorados.
- **Extensibilidade:** Um sistema de monitoração é extensível se pode receber novas funcionalidades, seja através da criação de *plug-ins*, filtros ou pela adição de novos componentes sem grandes modificações no sistema. A importância dessa propriedade consiste na possibilidade de se instalar e manter uma única infraestrutura de monitoração, realizando alterações à medida em que se tornem necessárias.
- **Pontualidade:** Um sistema de monitoração é pontual se os eventos detectados estão disponíveis em tempo para o uso pretendido [Wang et al., 2011]. Monitoração é essencial para as atividades de provedores e consumidores, logo falhas em conseguir informações necessárias a tempo para uma resposta apropriada tornam a monitoração ineficaz. Por exemplo, quando um recurso crítico monitorado apresenta problemas e o alarme responsável por indicar irregularidades não é acionado imediatamente devido a atrasos na recepção de métricas. De maneira mais específica, o tempo entre a ocorrência de um evento e sua notificação no sistema é composta por: coleta, análise e o atraso de comunicação. Quanto menor o tempo para realizar a coleta, menor o atraso entre o momento em que o evento ocorre e o momento em que é capturado pelo sistema. É importante observar que menores intervalos de medição favorecem a pontualidade, no entanto, possuem maior impacto sobre o uso de recursos do host (como CPU). O atraso gerado pela análise se deve principalmente a situações em que é necessário reunir muitas informações e processá-las para depois criar um evento. Por último, o atraso devido a comunicação esta relacionada à quantidade de links no qual dados precisam passar para encontrar os nós de monitoração.
- **Adaptabilidade:** Um sistema de monitoração é adaptável se consegue se adaptar a cargas variáveis de computação e rede para não ser invasivo, isto é, impedir

outras atividades [Clayman et al., 2010]. Devido à complexidade da nuvem, adaptabilidade é importante para evitar ao máximo o impacto negativo das tarefas de monitoração sobre o funcionamento das atividades da nuvem. A carga gerada pela medição contínua de métricas aliada ao custo de coleta e transmissão exige, da infraestrutura de nuvem, recursos computacionais e de comunicação. Assim, a habilidade de ajustar as atividades de monitoração para que sejam o menos invasivas possível é de grande importância no gerenciamento de nuvens. Fornecer adaptabilidade não é trivial [Park et al., 2011; Katsaros et al., 2011; Kutare et al., 2010], uma vez que é necessário balancear entre o intervalo de coleta e a propriedade de pontualidade.

- **Resiliência:** Um sistema de monitoração é resiliente quando a garantia de entrega do serviço é confiável. Isto é, o sistema de monitoração é capaz de resistir a falhas de componentes e continuar a operar normalmente, isto é, continuar a coletar, analisar e apresentar os dados monitorados. A continuidade de seu funcionamento geralmente é mantida através de mecanismos de tolerância a falhas e recuperação em caso de desastres.

Por se tratar de um sistema direcionado à monitoração de parâmetros de computação em Nuvem, escalabilidade passa a ser não só uma característica desejável, mas essencial quando se considera cargas de elevados volumes e variabilidades. A escalabilidade pode ser obtida de diversas formas através da arquitetura proposta, como será ilustrado na seção 6.2. Um sistema de monitoração também deve permitir avaliar periodicamente o estado do sistema computacional [Viratanapanu et al., 2010] e facilitar o rastreamento das causas de falhas. Esse desafio requer que uma série de componentes sejam analisados, englobando desde servidores físicos, redes e máquinas virtuais até logs de aplicações. Fica evidente, portanto, a necessidade de uma plataforma abrangente e confiável de monitoração para que provedores consigam localizar problemas dentro de sua infraestrutura e para que consumidores compreendam se as causas de problemas relacionados a desempenho são provenientes da infraestrutura ou da própria aplicação [Romano et al., 2011].

Capítulo 4

A arquitetura de monitoração Seshat

Como visto no capítulo anterior, ambientes de nuvem demandam atenção especial a uma série de propriedades previamente desconsideradas por sistemas de monitoração de propósito geral. Com isso em mente, as próximas seções serão dedicadas a apresentação de uma arquitetura subdividida em componentes que facilite o processo de monitoração e seja escalável. A arquitetura de monitoração pode ser definida através de 4 camadas básicas em uma orientação *bottom-up*: (1) Coleta, (2) Transporte, (3) Armazenamento e (4) Visualização. E além dessas, 2 camadas adicionais: (5) Alertas e (6) Correlação de Eventos, como pode ser observado na Figura 4.1.

O fluxo se inicia na camada mais inferior chamada de **Coleta**, que é responsável pela aquisição dos dados. Ela estabelece o primeiro contato entre a monitoração e o host. Nesta camada, agentes coletores de dados são inseridos em hosts e aplicações para que suas informações sejam enviadas até a camada de transporte. Na camada de **Transporte** são reunidas as informações de todos os agentes coletores através de mensagens. Mensagens brutas adquiridas desta forma passam por transformações como por exemplo filtragens e agregações para adequá-las ao formato desejado. Em seguida, as mensagens são disponibilizadas para o consumo em uma fila de mensagens. Logo depois, consumidores transmitem todas as mensagens atreladas à um *timestamp* para a camada de **Armazenamento**, onde são armazenadas em bancos de dados. Finalmente, na camada de **Visualização** informações gerais e específicas a respeito de toda a infraestrutura são apresentadas de forma sistematizada e de fácil compreensão. Adicionalmente, a camada de **Alertas** é responsável por avisar o administrador caso algo errado aconteça nos hosts monitorados. Por fim, a camada de **Correlação de Eventos** realiza a manipulação de todas as informações recebidas, a fim de produzir conhecimento mais elaborado a respeito da situação do sistema. As seções seguintes detalham o funcionamento de cada camada.

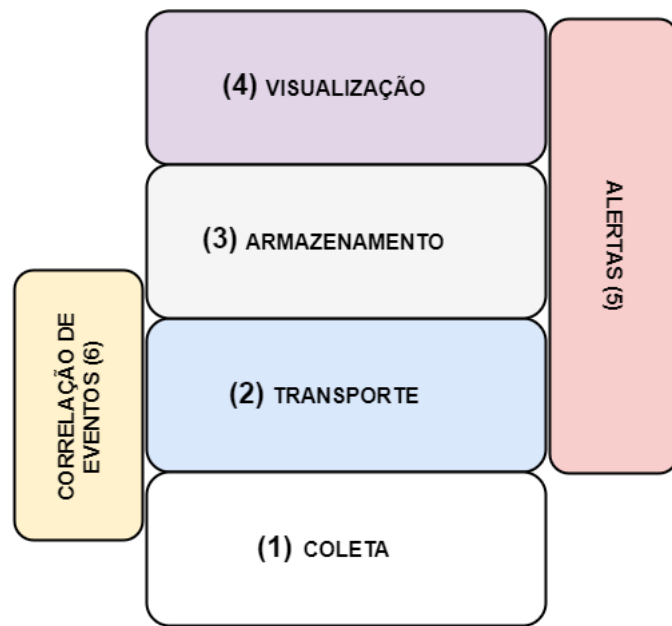


Figura 4.1. Ilustração das camadas básicas de um sistema de monitoração de nuvem.

4.1 Camada de coleta

O objetivo da camada de Coleta é ser o primeiro contato entre o sistema e a entidade monitorada. Assim, para que seja possível monitorar recursos é necessária a instalação de um agente (também conhecido como *probe*) para cada novo host na infraestrutura. Cada agente é responsável por ativamente coletar dados e envia-los ao sistema, sem a necessidade de uma entidade central realizar consultas sobre agentes. Dessa forma, a coleta de dados possui a característica de escalabilidade horizontal em contrapartida a escalabilidade vertical apresentada em sistemas de monitoração de propósito geral. Adicionalmente, é importante ressaltar que agentes precisam ser componentes leves e eficientes para se adequar à propriedade de Adaptabilidade. Quanto mais simples as instruções executadas, menos invasivo é o agente e consequentemente menor o custo computacional de instrumentação. Dessa forma, agentes podem desempenhar duas funções diferentes dependendo do tipo de informação que coletam: coleta de métricas (seção 4.1.1) ou coleta logs (seção 4.1.2).

4.1.1 Coleta de métricas

Quando um agente coleta métricas, ele é responsável por executar continuamente verificações (*checks*) que produzem eventos (ou simplesmente mensagens). Um evento é um bloco de dados que possui um *timestamp*, informações a respeito do host de origem, sobre a versão do agente e o resultado de cada verificação executado. Um *check* pode verificar diferentes tipos de métricas como: utilização de CPU, espaço vazio do disco rígido ou a utilização da interface de rede. O resultado de um *check* pode ser representado por formato categórico ou, na maioria das vezes, um valor numérico.

4.1.2 Coleta de logs

Quando um agente coleta logs, ele é configurado para observar mudanças em arquivos de log de aplicações ou do próprio sistema operacional. Para cada nova linha adicionada ao arquivo, o agente cria um novo evento com o conteúdo da linha (sequência de caracteres), *timestamp* e outras informações relacionadas ao host de origem.

4.1.3 Modelos pull e push

Existem dois modelos conceituais para a coleta de dados: *pull* e *push*. A maioria dos sistemas de monitoração são do tipo *pull* (também conhecido como *polling-based*). Esse tipo de sistema realiza o trabalho de consultar os hosts sendo monitorados, isso significa que, quanto maior a quantidade de hosts e serviços sendo gerenciados maior a quantidade de consultas executadas e processadas pelo servidor. Dessa forma, para garantir escalabilidade a monitoração precisa escalar verticalmente, e além disso, também é necessário que a monitoração seja centralizada. Por outro lado, no modelo *push*, hosts, serviços e aplicações são os emissores de seus próprios dados. Esse tipo de coleta é completamente distribuída, uma vez que, os próprios objetos monitorados realizam o trabalho de coleta e envio de suas informações. Como consequência, a monitoração passa a apresentar escalabilidade horizontal em relação aos objetos monitorados. Adicionalmente, o modelo *push* é inerentemente mais seguro, uma vez que os emissores não precisam ficar continuamente aguardando conexão de rede. Isso reduz a possibilidade de ataque aos hosts, serviços e aplicações.

4.2 Camada de transporte

Devido à natureza imprevisível do fluxo de dados derivados de eventos e logs de aplicações, rajadas de dados são comuns em sistemas de grande porte [Armbrust

et al., 2010] e podem comprometer a coleta causando perdas de dados. A camada de transporte lida com as rajadas e as disparidades nas taxas produção e recepção de mensagens. Isso pode ser alcançado através da inclusão filas de mensagens (também chamadas *brokers*), um componente de armazenamento temporário robusto para garantir o transporte de informações cruciais ao banco de dados. No entanto, filas de mensagem usualmente não possuem mecanismos de envio de informações para outros componentes, isso exige a inclusão de dois componentes: o *Shipper* e o *Indexer*. A seguir são abordados os pormenores da camada de Transporte.

Devido à alta taxa de produção de dados oriundos dos agentes na camada de coleta se torna inviável que essas informações sejam enviadas diretamente a camada de armazenamento. Considerando que a taxa de ingestão dos bancos de dados pode ser muito menor do que a taxa máxima de produção dos agentes, diversas mensagens poderiam ser perdidas no processo de envio e, ocorreria degradação no desempenho. Dessa forma, filas de mensagem são introduzidas como um componente intermediário de *buffer* para garantir que os dados coletados realmente cheguem à camada de armazenamento.

Filas de mensagem funcionam sob o paradigma de *Publish/subscribe*, no qual produtores publicam suas informações em filas e consumidores se inscrevem nessas mesmas filas para receber informações [Eugster et al., 2003]. Em geral filas são divididas em tópicos de acordo com o assunto das mensagens nelas contidas. Adicionalmente, consumidores somente recebem informações a respeito das filas em que se inscrevem, não possuindo qualquer informação a respeito das mensagens de outras filas. A separação entre logs de sistema e métricas é um exemplo de divisão de tópicos para o caso da monitoração. Pode-se também organizar a fila por fonte de dados ou tipo de informação.

Outro ponto importante é que as interações entre produtores, consumidores e a fila de mensagens não precisa bloquear sincronamente a execução de um produtor/consumidor. Em termos gerais, a velocidade com que cada produtor/consumidor é capaz de produzir/consumir mensagens não afeta a outra parte, resolvendo o problema da disparidade momentânea, entre as taxas de produção da camada de coleta e inserção da camada armazenamento.

Por fim, vale mencionar o atributo de persistência. Em geral, filas de mensagem mantêm seus dados armazenados em memória. No entanto, para situações extremas onde ocorre o uso crítico de memória, a fila é capaz de armazenar parte dos dados em disco, oferecendo ainda mais disponibilidade na recepção de mensagens.

Uma vez adicionada ao sistema a fila de mensagem por si só é incapaz tanto de buscar informações dos componentes da camada de coleta, como também enviar informações diretamente aos componentes da camada de Armazenamento; Funciona

essencialmente como uma camada de *buffer*. Dessa forma, o processamento das informações é dividido através de dois novos componentes como ilustrado na figura 4.2.

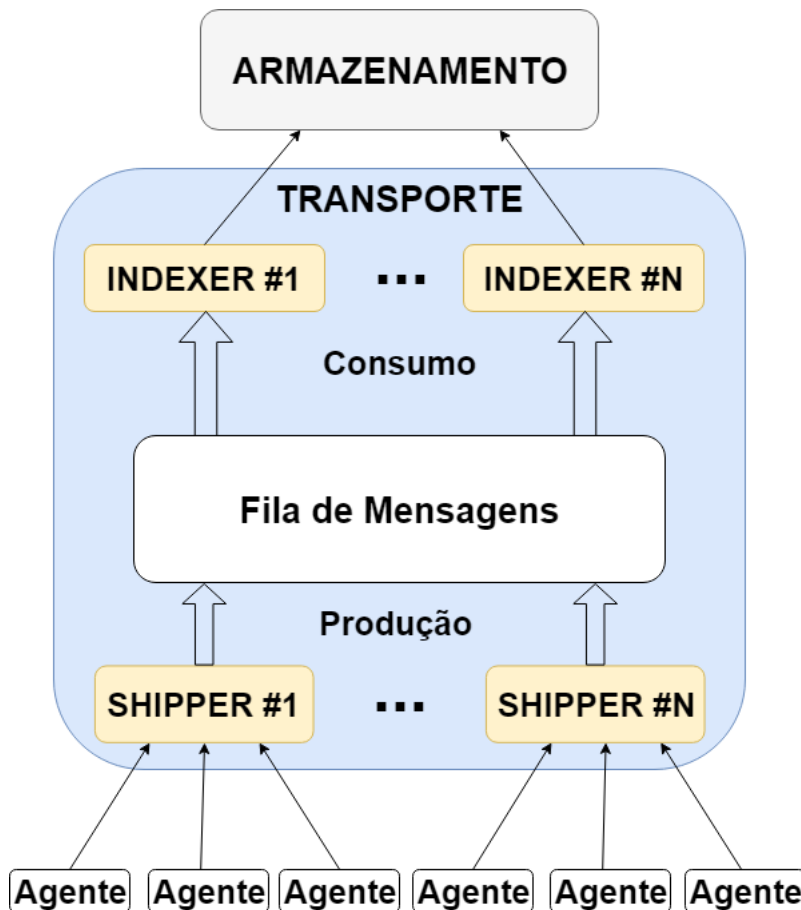


Figura 4.2. Estrutura interna da camada de Transporte

O primeiro componente chamado **Shipper** é responsável pela recepção de dados obtidos por diversos agentes em diferentes tipos de protocolos e imediatamente enviá-los à fila, desempenhando assim o papel de produtor. Sua função é servir como um receptor universal e atuar como uma ponte na comunicação entre os agentes e a fila. É importante observar que existem casos onde os próprios agentes coletores de dados já possuem *plugins* específicos para comunicação direta com a fila, fazendo com que o uso do **Shipper** seja opcional.

Em contrapartida, o segundo componente chamado **Indexer** é mais robusto e responsável por consumir as mensagens da fila e enviá-las para a camada de Armazenamento. Além disso, realiza operações mais custosas, como filtragem e *parsing* de mensagens. É importante observar que componentes da camada de Armazenamento também podem possuir *plugins* de acesso direto à fila. No entanto, para esses casos, o **Indexer** ainda é necessário para realizar tratamento sobre os dados brutos.

A necessidade de uso de uma fila se deve principalmente a dois tipos de cenários: O primeiro cenário se baseia na natureza imprevisível do volume de informações no sistema. Quando há muitos eventos sendo gerados, ocorre igualmente a criação de um grande volume de mensagens. Se o sistema não conseguir processá-las isso se torna um gargalo, impossibilitando o armazenamento na camada superior. Esse tipo de *burst* é bem comum em ambientes multi-usuário como o comércio eletrônico [Kusters et al., 2017].

O segundo cenário considera situações em que um dos componentes da camada de Armazenamento se torne temporariamente inacessível. Se, por exemplo, um banco de dados precisar de manutenção ou atualização de software, isso pode significar uma interrupção no serviço onde informações são coletadas, mas não são registradas. Nesse tipo de situação, uma fila de mensagem ou *broker* é útil para armazenar momentaneamente as informações oriundas da camada de coleta até que a situação seja normalizada.

É importante observar que a adição de um *broker* em um sistema de monitoração insere mais uma camada de software que requer atenção e eventualmente manutenção. Dessa forma, é necessário avaliar se logs e métricas precisam estar disponíveis assim que produzidos. Para sistemas onde uma maior latência na pesquisa de informações é permitida, o uso desse tipo de componente é desnecessário. Ferramentas como Log4j [Gülçü, 2003] e FileBeat [Elastic, 2016b] aceitam o arquivamento de logs obsoletos (também conhecido como *log rotation*), para casos em que aplicações emitam mais informações do que a camada de armazenamento seja capaz de absorver: essas informações são armazenadas em arquivos temporários no próprio host. Deve ser observado que neste último caso, o sistema de arquivos do próprio host é utilizado como *buffer*.

4.3 Camada de armazenamento

Com os alicerces de coleta e transporte definidos é necessária uma forma de manter registrados os dados adquiridos. Em vista disso, a camada de Armazenamento é responsável por registrar de forma estruturada as informações recebidas das camadas inferiores e posteriormente permitir a criação de histórico para análises mais detalhadas. Levando em conta as características apresentadas por sistemas em nuvem descritas do capítulo 3, para se adequar ao volume de informações e às necessidades de escalabilidade, são empregados bancos de dados especializados em séries temporais. Mais especificamente, são utilizados dois componentes de bancos de dados, um para o armazenamento de métricas e outro para o armazenamento de logs, como pode ser visto na

figura 4.3 e em maiores detalhes ao longo desta seção.

Ao se estudar o fluxo de dados em um sistema de monitoração é possível observar um certo padrão, uma série de valores numéricos indexados por um timestamp, que se encaixa exatamente na definição de uma série temporal (o mesmo cenário é válido para logs se considerados como blocos de informação). Outra característica muito importante nesse tipo de sistema é a grande demanda por escrita, eventual leitura e muito raramente deleções. Os dois argumentos citados anteriormente são mais do que suficientes para justificar o uso deste tipo de banco de dados. Adicionalmente, como estudado por [Goldschmidt et al., 2014], sistemas de monitoração de nuvem também possuem grande demanda por armazenamento escalável e robusto. De maneira geral, bancos de dados relacionais padrão são projetados para executar em um único servidor a fim de manter a integridade dos mapeamentos de tabela e evitar a complexidade envolvida na computação distribuída. Embora seja possível alcançar a distribuição em bancos de dados relacionais, tal tarefa ainda é complicada. Além disso, uma vez que espaço adicional é alocado se torna ainda mais difícil encolher o tamanho do *cluster*, indicando baixa elasticidade. Portanto, um banco de dados do tipo *NoSQL* é mais adequado por ser capaz de contornar as dificuldades mencionadas anteriormente e pela sua capacidade de se adaptar a variações da carga de trabalho através de sua escalabilidade horizontal. Logo, pode-se dizer que bancos de dados ideais para esta camada precisam ser otimizados para séries temporais e do tipo *NoSQL*.

Tanto métricas como logs de sistema podem ser considerados séries temporais. No entanto, ainda existem certas peculiaridades no tratamento de cada um deles, como por exemplo: (1) após o processo de *parsing*, logs podem possuir diversos campos indexáveis, enquanto métricas possuem apenas um ou dois campos indexáveis além do timestamp; (2) em geral, logs de sistema possuem diversos caracteres em cada campo, enquanto métricas possuem apenas um único valor numérico; (3) consultas relacionadas a cadeias de caracteres são bem distintas de consultas relacionadas a valores numéricos e potencialmente mais complexas. Considerando as particularidades acima e de acordo com nosso conhecimento, não existe um componente de armazenamento capaz de endereçar de maneira eficiente a todas essas questões. Por isso, é aconselhável o uso de bancos de dados distintos para o armazenamento de logs e métricas.

4.4 Camada de visualização

A camada de visualização é responsável por permitir que os dados que foram coletados e armazenados sejam visualizados de forma gráfica. Esse é o ponto da arquitetura onde telas configuráveis (também chamadas de dashboards) com as quais métricas e logs podem ser exibidos sob a forma pretendida pelo usuário. Além disso, podem ser aplicadas técnicas estatísticas para avaliação das informações e acompanhamento contínuo do estado sistema.

Dashboards são o primeiro contato entre o usuário e o sistema de monitoração. De maneira geral, podemos dizer que um dashboard deve apresentar todas as informações importantes em uma única tela e mostrar indicadores chave. Tomando como exemplo a monitoração de desempenho, é importante que um dashboard apresente métricas como CPU, memória RAM, uso de disco e tráfego de rede a respeito de cada uma das máquinas que estão sendo monitoradas e de forma que o usuário consiga identificar cada uma delas na visualização escolhida. Outro ponto importante é a interatividade, ações como a filtragem de informações e o detalhamento são especialmente importantes para a monitoração. Usualmente um sistema de monitoração nunca para de funcionar, e portanto, nunca para de receber uma grande quantidade informações que precisam ser avaliadas a todo tempo. Logo, escolher quais dessas informações devem ser observadas e a janela de tempo é certamente uma das funções mais benéficas a um sistema de monitoração.

Através dessa camada o administrador do sistema é capaz de tomar decisões e receber notificações caso aconteçam falhas. Devido à variedade de opções para ferramentas gráficas, não são citados componentes fixos ou limitações na quantidade de componentes utilizados. Como será explicado no capítulo 5, foram utilizadas ferramentas distintas para apresentação de métricas e logs considerando as diferenças inerentes entre eles.

4.5 Camada de alertas

A camada de Alertas, como o próprio nome sugere, tem a função de chamar a atenção do administrador a respeito de algum evento fora do normal que ocorra no sistema. Por exemplo, se a CPU de um dos hosts monitorados atinge utilização de 95% quando na verdade não deveria passar dos 40%, o sistema deve reportar a atividade incomum ao administrador. Dessa forma, essa camada realiza suas funções através da configuração de **alarmes** e do envio de **notificações**.

Notificações são uma das formas primárias de saída em sistemas de monitoração. Podem ser criadas de diversas formas, como e-mails, mensagens SMS ou qualquer tipo de interface que forneça informações ao administrador. Por outro lado, alarmes são checagens permanentes pré-estabelecidas pelo administrador a respeito dos valores observados nos dados coletados. Caso uma determinada métrica ultrapasse o limiar determinado em um alarme, uma notificação do contexto do sistema é enviada ao administrador para que uma ação seja tomada.

Atualmente ferramentas de visualização já possuem funcionalidades simples de alarmes e notificações pré-instaladas. Apesar de alarmes serem conceitualmente simples, existe grande complexidade envolvida na sua execução, como por exemplo: qual tipo informações fornecer em uma notificação para que seja possível resolver um problema em potencial? Caso um alarme seja ativado, com que frequência enviar notificações ao administrador? Se um alarme for mal configurado e gerar muitas notificações, administradores serão incapazes de resolvê-los e eles serão possivelmente negligenciados. No escopo deste trabalho, a função da camada de Alertas pode ser alcançada através de alarmes e notificações simples presentes em ferramentas de visualização, ou conectando um módulo com esse propósito específico à camada de transporte.

4.6 Camada de correlação de eventos

Ao avaliar o sistema de monitoração até este ponto é possível coletar, armazenar, visualizar e gerar notificações a respeito de eventos de interesse. Todas as informações obtidas até esse ponto são importantes, mas possuem pouco tratamento analítico. Através da recepção de eventos (métricas e logs) inseridos na camada de transporte, a camada de Correlação de Eventos (ou simplesmente *Event Correlation Engine*) serve como um poderoso componente de manipulação de eventos em formato de stream, capaz de agregar informações, criar filtros e produzir conhecimento mais elaborado a respeito da situação do sistema. De forma similar à camada de Alertas, também é conectada à camada de Transporte para a recepção de informações.

Retomando o assunto de alarmes, como exemplo, em geral os valores utilizados como limiares em alarmes são fixos e definidos arbitrariamente por administradores de sistemas. Observe que além desses valores poderem não ser considerados alarmantes em todas as máquinas da infra-estrutura, eles também podem mudar dependendo do stress sobre o qual o *cluster* está sendo exposto ao longo do tempo. Ao invés de utilizar valores fixos, através do *Event Correlation Engine* é possível realizar o cálculo de percentis sobre métricas em uma janela de tempo e definir limiares dinâmicos e

customizados na criação de alarmes. Dessa forma, *Event Correlation Engine* pode ser visto como uma ferramenta para a incorporação de inteligência sobre os dados e permitir a programação orientada a eventos.

4.7 Colocando os elementos em operação

Para finalizar este capítulo, esta última seção apresenta uma visão geral de como o fluxo de informações ocorre na arquitetura e que maneira estão estabelecidas as premissas que garantem escalabilidade. Analisando a figura 4.3, a base da monitoração se encontra na camada de Coleta, que é feita por agentes instalados diretamente nas entidades a serem monitoradas. Como visto anteriormente nas seções 4.1.1 e 4.1.2, os agentes são capazes de coletar métricas e logs e transmiti-los na forma de eventos. Os eventos são enviados para a camada de Transporte de duas formas distintas: a primeira forma acontece quando os próprios agentes possuem capacidade de comunicação direta com a fila de mensagem (por meio de um protocolo de comunicação); a segunda forma acontece através da recepção dos dados pelo *shipper*(4.2) e redirecionamento para a fila de mensagens. Dentro do *broker* os eventos são separados por tópicos (métricas e logs) e distribuídos para potenciais consumidores. O tipo primário de consumidor são os componentes *indexer*, onde eventos passam por um processo de adequação de seu formato e, em seguida, são armazenados em bancos de dados específicos para cada tipo de dado (métrica ou log). O tipo secundário de consumidor são as camadas de Alertas e Correlação de eventos. É possível observar que os mesmos eventos podem ser enviados para esses dois tipos de consumidores sem que haja duplicação de dados, graças à propriedade de roteamento da fila de mensagens. Em seguida, na camada de Armazenamento, as informações são distribuídas e replicadas pelos bancos de dados com o intuito de se obter tolerância a falhas e escalabilidade. Finalmente, na camada de Visualização o resultado de todo o processo toma forma. São apresentadas as informações armazenadas pelos bancos de dados na forma de *dashboards* interativos, notificações oriundas da camada de Alertas e manipulações realizadas na camada Correlação de eventos.

Por se tratar de um sistema direcionado à monitoração de parâmetros de computação em Nuvem, escalabilidade passa a ser não só uma característica desejável, mas essencial quando se considera cargas com elevados volumes e variabilidades. A escalabilidade pode ser obtida de diversas formas através da arquitetura apresentada neste capítulo.

Como visto anteriormente, na camada de Coleta agentes são inseridos em cada um

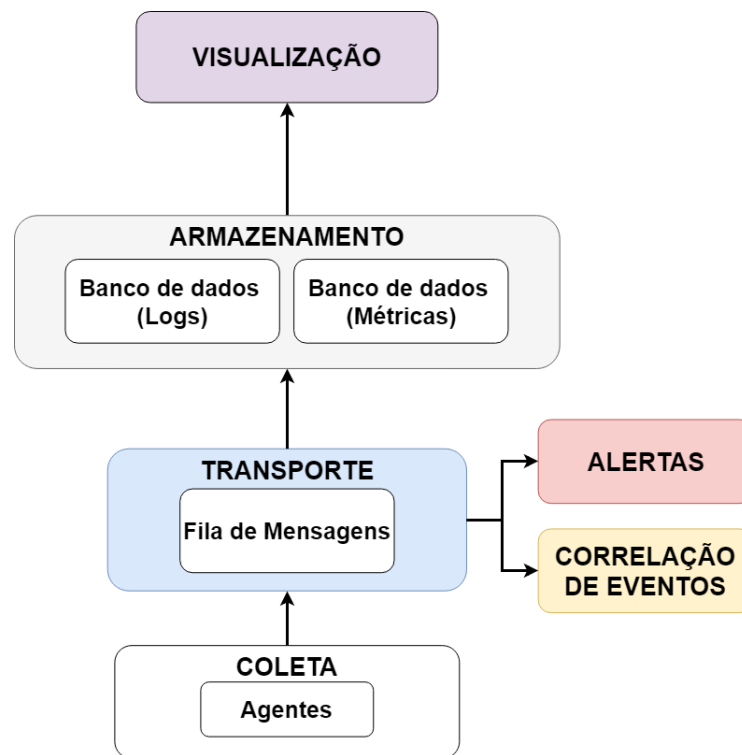


Figura 4.3. Arquitetura de um sistema de monitoração

dos hosts a serem monitorados e são executados os *checks* para obtenção de métricas, eventos, etc. Portanto, os próprios hosts executam as instruções e enviam seus dados ao sistema de monitoração, apresentando escalabilidade horizontal na coleta das informações. Além disso, uma grande quantidade de hosts não impacta o desempenho da camada de Coleta, uma vez que pouco recurso é consumido em cada host para o funcionamento do agente. Logo em seguida, na camada de Transporte existem três pontos onde a escalabilidade pode ser aplicada: *shipper*, fila de mensagens e *indexer*. Caso a taxa de recepção de dados esteja baixa, novos componentes *shipper* podem ser instanciados juntamente com própria fila de mensagens, atuando de forma distribuída a fim de aumentar a inserção de dados. Para o caso em que a taxa de recepção de dados seja muito maior do que a de envio à camada de armazenamento, novas instâncias do componente *indexer* podem ser disparados para agilizar o processo de consumo de mensagens. Dessa forma, tanto o ingresso das informações na camada de Transporte quanto seu roteamento se tornam escaláveis. Por último, para casos onde a restrição ocorra na inserção de mensagens nos bancos de dados, é altamente desejável que eles também possuam a capacidade de distribuição, para dessa forma também escalarem.

Para avaliar a arquitetura *Seshtat*, ela foi instanciada em uma estrutura completa para monitoração de um ambiente em nuvem real: um cluster de processamento de

dados massivos. Os detalhes serão discutidos nos capítulos a seguir.

Capítulo 5

Ferramenta de Monitoração

Existem várias ferramentas disponíveis no mercado que realizam parte do que se espera de um sistema de monitoração. Devido à essa grande quantidade, se torna uma tarefa custosa selecionar os componentes adequados. Em vista disso, a arquitetura *Seshat* é formada por vários componentes com diferentes papéis que identificam as funcionalidades mais importantes. Com o planejamento de cada camada realizado no capítulo 4, podemos nos guiar pelas funcionalidades necessárias (como por exemplo, escalabilidade) e iniciar nosso segundo objetivo, a implementação do sistema de monitoração. Primeiramente, a seção 5.1 define o ambiente a ser monitorado e as métricas a serem coletadas. Na seção 5.2 são apresentados alguns critérios para a seleção das ferramentas do sistema. Em seguida, as seções 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 mapeiam as camadas definidas no capítulo anterior em ferramentas de código aberto, instanciadas de forma distribuída através de *containers* (seção 5.9). Por último, as seções 5.10 e 5.11 discutem os aspectos de escalabilidade, tolerância a falhas e modularização.

5.1 Ambiente e métricas

Como previsto na seção 3.2, neste ambiente, todo o provisionamento de recursos (incluindo a instanciação de VMs e gerenciamento dessa infraestrutura) é feito pelo OpenStack. Além disso, uma unidade de distribuição e monitoramento de energia conhecida como PDU (Power Distribution Unit) [Lincoln & Slack, 1991] é inserida para controlar a energia. A partir de suas tomadas, o PDU distribui e monitora energia elétrica para as máquinas físicas que constituem o servidor da nuvem. A figura 5.1 retrata a infraestrutura descrita. Sobre a camada de infraestrutura são executados os sistemas de processamento distribuído como Spark e Hadoop (figura 5.2).

Definido o ambiente a ser monitorado, inicialmente são coletadas informações

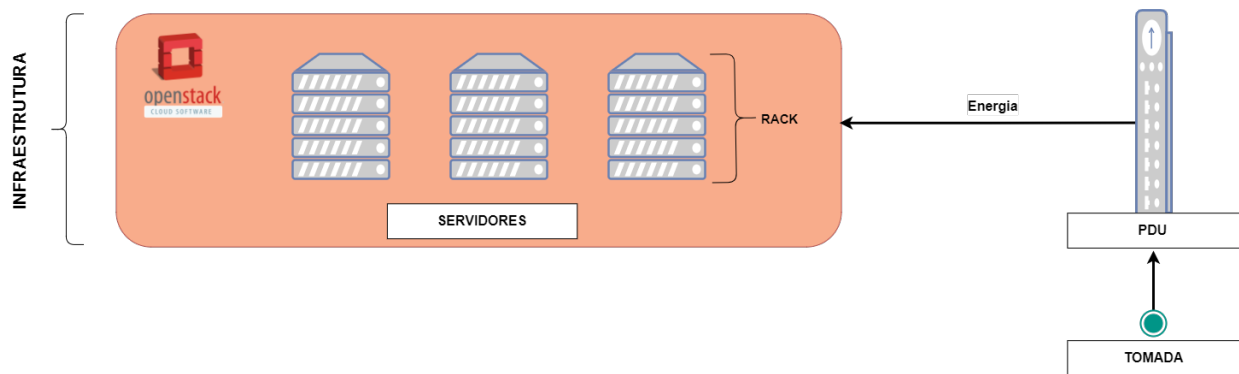


Figura 5.1. Camada de Infraestrutura

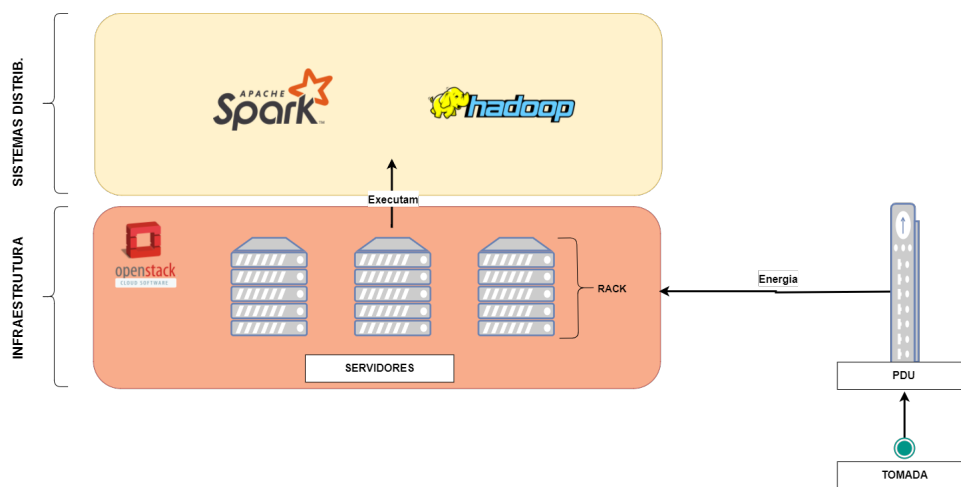


Figura 5.2. Camadas de Infraestrutura e Sistemas Distribuídos

básicas a respeito do uso de recursos das máquinas que compõem o *cluster*. É estabelecida uma coleta de dados genérica, isto é, em todos os hosts por padrão são coletadas as mesmas métricas de sistema. O conjunto básico de métricas monitoradas inclui:

- **CPU** - Mede o uso de CPU de acordo com a carga imposta. A métrica de CPU é dividida pelos tipos de processos (user, guest, system, etc).
- **Memory** - Mede a porcentagem de memória utilizada, alocada para cache, buffers etc.
- **Load** - Apresenta o *system load*, uma estimativa da utilização do host, definida como o número de tarefas executáveis na fila de execução, dividido entre médias de 1, 5 e 15 minutos.
- **Swap** - Mede quanto de SWAP está sendo utilizado.

- **Disk** - Mede o quanto de espaço do disco está sendo utilizado e quanto está disponível no sistema de arquivos.
- **Disk IO** - Mede as taxas de leitura e escrita realizadas nos discos.
- **Network** - Mede o tráfego de dados nas interfaces de redes.

As métricas mencionadas até o momento fornecem informações somente a nível do sistema operacional e não das aplicações sendo executadas. Dessa forma, além das métricas relacionadas ao sistema, também são monitoradas métricas relacionadas às aplicações. Dentre elas estão:

Métricas do Spark - Métricas relacionadas à execução de uma aplicação Spark, por exemplo:

- Leitura e Escrita realizadas no HDFS
- Uso de memória Heap
- Tarefas ativas e completas

Métricas do Hadoop - Métricas relacionadas ao sistema de arquivos HDFS, por exemplo:

- Capacidade e uso do HDFS
- Gerenciamento de blocos do HDFS
- Métricas de utilização fornecidas pelos componentes do sistema

Adicionalmente, também são coletadas métricas referentes à utilização energética. Estas são coletadas a partir da unidade de distribuição de energia (PDU). Entre as métricas estão:

- Potência ativa
- Tensão
- Corrente
- Fator de potência

Essas métricas são fornecidas pelo PDU tanto para o valor agregado de entrada, quanto indiretamente para cada tomada do mesmo.

Métricas são geralmente úteis para oferecer informações a respeito do desempenho e manter o histórico de estado do sistema. No entanto, *logs* têm potencial de expressividade muito maior. *Logs* oferecem contexto sobre determinada situação e ressaltam

algo que tenha ocorrido. Para propósito de diagnóstico, registros de logs são bastante utilizados. Dessa forma, além da coleta de métricas, também são coletados *logs* das aplicações. Exemplos de tipo de *logs* coletados no contexto do sistema considerado aqui são:

- *Logs* do HDFS
- *Logs* do Hadoop
- *Logs* do Yarn, o escalonador de tarefas usado
- *Logs* dos Executores Spark que mantêm detalhes sobre cada execução
- *Logs* do History Server (Spark) que registram o histórico de tarefas executadas

5.2 Critérios de seleção

Como descrito no capítulo anterior, a arquitetura *Seshat* apresenta uma série de funções gerais que, quando agregadas, permitem o funcionamento de uma sistema de monitoração em nuvem. Considerando as funções isoladamente, podemos encontrar no mercado uma variedade de ferramentas para desempenhar cada uma delas. Entretanto, por se tratar de uma arquitetura, *Seshat* é completamente independente das ferramentas utilizadas, desde que se respeite as características descritas. Para facilitar a escolha de qual ferramenta usar para prover uma determinada função, utilizamos os seguintes critérios:

- Adequação à função descrita: o quão bem a ferramenta desempenha o papel para o qual foi escolhida? É seu objetivo principal desempenhar aquela função? Ela se enquadra nos requisitos necessários?
- Qualidade da documentação disponível: os desenvolvedores fornecem uma documentação completa sobre a utilização da ferramenta?
- Adoção da ferramenta pela comunidade: é uma ferramenta com uma quantidade considerável de usuários? O *feedback* a respeito daquela ferramenta é bom?
- Sinergia com outras ferramentas escolhidas: a ferramenta escolhida possui formas de se comunicar com as outras já presentes no sistema ou seria necessário alterações?
- Suporte e atualização: a ferramenta escolhida vem sendo atualizada constantemente? Os *bugs* encontrados estão sendo corrigidos pelos desenvolvedores? Existe algum tipo de fórum de ajuda da ferramenta?

Através destes critérios, selecionamos as ferramentas descritas nas próximas seções, além de apresentarmos alternativas para cada uma delas.

5.3 Camada de coleta

Assim como projetado na seção 4.1, os agentes de coleta são responsáveis por buscar informações das entidades a serem monitoradas. Observando a figura 5.3, inicialmente a plataforma *Docker* é instalada no host e dois agentes são instanciados: Sensu e Beaver. Ambos submetem as informações coletadas a um servidor remoto responsável pelo tratamento das informações. Assim, o procedimento de coleta é dividido em 2 fluxos: coleta de *logs* e coleta de métricas.

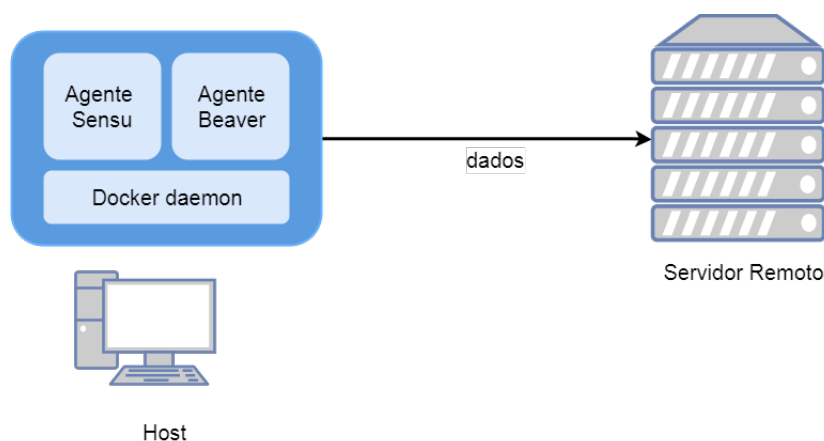


Figura 5.3. Agentes instalados em um host enviando dados ao servidor remoto.

O fluxo da coleta de *logs* pode ser realizada de duas formas, a primeira delas é através do envio de *logs* pela ferramenta Log4j [Gülcü, 2003] presente em aplicativos escritos na linguagem Java (como Spark e Hadoop) e a outra forma é através do agente Beaver [Python-Beaver, 2012]. Desenvolvido na linguagem Python, apresenta uma método leve para o envio de *logs* locais para um servidor remoto. Mais especificamente, durante a configuração um conjunto de arquivos de *logs* são selecionados para serem observados. A cada nova linha de log adicionada em um arquivo, o agente cria uma mensagem com seu conteúdo e uma série de tags de identificação. A mensagem é então enviada para a fila de mensagens RabbitMQ (detalhada na próxima seção), localizada no servidor remoto, através do protocolo AMQP [Vinoski, 2006].

Por outro lado, o fluxo da coleta de métricas começa com o agente Sensu [Sensu, 2013]. Como apresentado no esquema da figura 5.4, inicialmente o servidor Servidor Sensu publica na fila de mensagens um conjunto de *checks* que devem ser executados

pelos agentes (1). Os agentes acessam a fila e executam localmente os *checks* necessários (2). Finalmente, representado pelas setas pontilhadas, os agentes publicam os resultados obtidos na fila, de onde são levados ao Servidor Senu e posteriormente enviados para armazenamento (3).

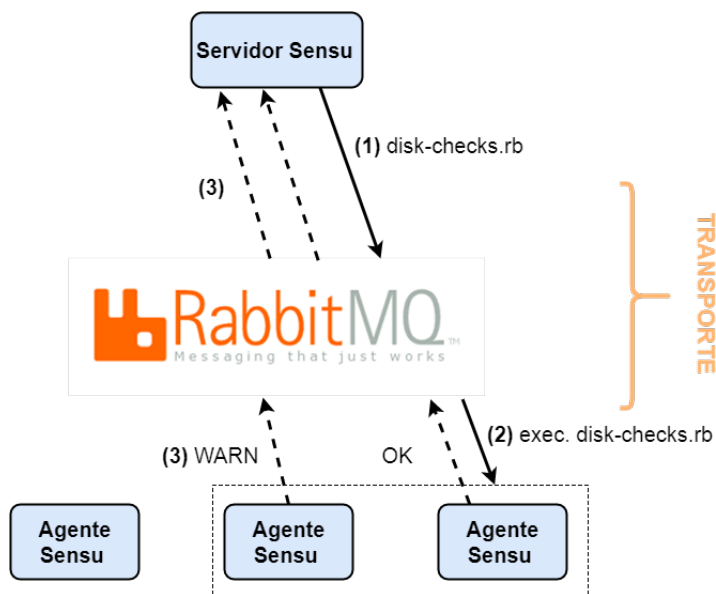


Figura 5.4. Arquitetura Senu.

As características de funcionamento do Senu são muito interessantes para a monitoração de ambientes em nuvem. Primeiramente, com uma arquitetura *push-based*, hosts, serviços e aplicações se tornam emissores, enviando dados à um coletor central. Dessa forma, a tarefa de coleta é completamente distribuída horizontalmente e delegada aos emissores, resultando em escalabilidade linear [Turnbull, 2014]. Além disso, o modelo *Publish/Subscribe* utilizando a fila de mensagens para a publicação dos *checks* elimina o esforço de configurar cada agente localmente, sendo necessário somente que o agente possua o endereço da fila. Por último, uma vez que os resultados são recebidos pelo servidor, o mesmo registra informações a respeito do agente de origem. Assim, agentes são registrados ou removidos dinamicamente no sistema atendendo ao requisito de elasticidade.

Adicionalmente, as informações relevantes ao consumo de energia presentes no PDU são coletadas através de um agente Collectd [Collectd, 2017]. Por se tratar de um dispositivo simples com capacidade computacional limitada, o PDU possui poucas formas de compartilhamento de seus dados. Dentre as formas disponíveis de comunicação, está o protocolo SNMP [Stallings, 1998], muito comum em dispositivos como modems, roteadores e switches. Assim, o agente Collectd é instalado no próprio servi-

dor de monitoração remoto e periodicamente faz consultas ao PDU através do protocolo SNMP. Apesar do agente Sensu realizar uma função similar ao agente Collectd, existem pontos que dificultam seu uso no PDU. O agente Sensu padrão precisa ser instalado em um sistema operacional para executar suas funções e, como mencionado anteriormente, o PDU é composto por uma controladora simples inviabilizando a instalação de qualquer tipo de software. Alternativamente, o agente Sensu oferece uma funcionalidade chamada de agente proxy [Sensu, 2018], que não necessita de instalação, mas que ainda não é completamente funcional [Sensu, 2016]. Por fim, adicionamos a primeira camada relacionada ao sistema de monitoração na figura 5.5.

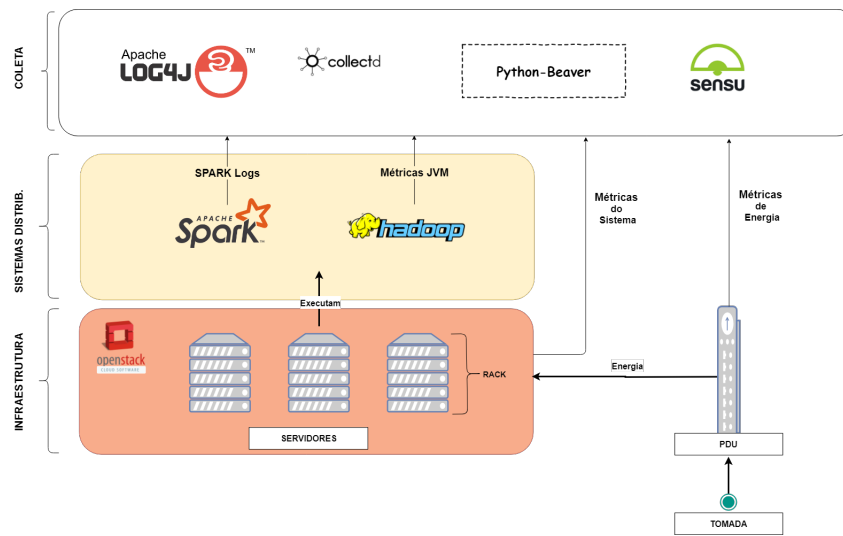


Figura 5.5. Camadas de Infraestrutura, Sistemas Distribuídos e de Coleta.

Como opções para substituir os coletores apresentados, as seguintes ferramentas podem ser utilizadas: *FileBeat* [Elastic, 2016b], *StatsD* [Etsy, 2018] e *MetricBeat* [Elastic, 2016c].

5.4 Camada de transporte

Como mencionado na seção 4.2, a natureza imprevisível do fluxo de dados oriundos da camada de coleta pode causar perdas de dados se tratada incorretamente. Dessa forma, duas ferramentas foram selecionadas para compor a camada de transporte: RabbitMQ [Videla & Williams, 2012] que realiza a função de fila de mensagens e Logstash [Turnbull, 2013] para o tratamento dos dados.

Inicialmente existem 2 caminhos possíveis de entrada de informações nessa camada. O primeiro deles é através de *Shippers*, que são componentes do Logstash. Os

componentes são configurados para aceitar informações de uma variedade de entradas diferentes (HTTP, TCP/UDP, arquivos, Syslog, Log4j, etc) e transmiti-los através do protocolo AMQP à fila de mensagens, sem alterações nos dados brutos. No segundo caminho, as informações são enviadas diretamente à fila de mensagens RabbitMQ, que é responsável por organizar os dados em filas de acordo com o seu tipo (métrica ou log) e armazená-los até que os consumidores (*Indexers*) os reivindiquem. A razão para que existam esses dois caminhos reside no fato de que alguns agentes possuem plugins para envio de dados diretamente para a fila de mensagens, como mencionado na seção 4.2.

Antes de introduzir os consumidores, serão discutidas algumas características importantes a respeito do RabbitMQ. A ideia principal de seu modelo de mensagens é que um produtor nunca envia suas mensagens diretamente a uma fila específica. Ao invés disso, um produtor, no caso um agente de coleta ou um *shipper*, somente envia mensagens para uma estrutura chamada *exchange*. O papel do *exchange* é: receber mensagens dos produtores e inseri-las em filas segundo regras configuráveis. No exemplo da figura 5.6, um produtor P envia mensagens ao *exchange* X, que por sua vez distribui a mensagem para duas filas. Através desse mecanismo, o *exchange* é capaz de inserir logicamente uma mesma mensagem em filas diferentes sem que seja necessário duplicá-la fisicamente. Assim, a requisição de execução de *checks* realizado pelo servidor sensu pode ser feito de maneira eficiente, publicando mensagens de requisição em um único *exchange* que distribui para as filas de todos os agentes. Além disso, o modelo também favorece a alimentação de mensagens para qualquer outro componente interligado à fila, como os elementos das camadas de correlação de eventos e alertas.

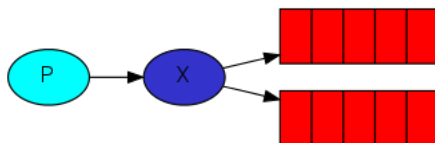


Figura 5.6. Exchanges em RabbitMQ. Fonte adaptada de [Pivotal, 2007]

Outro ponto importante sobre o RabbitMQ é sua escalabilidade. A ferramenta permite a criação de um *cluster* de processos que implementam o controle das filas. Tanto a inserção quanto a remoção de nós é facilmente realizada, apresentando a propriedade de elasticidade que é de grande importância em sistemas de nuvem. Além disso, a adição de novos nós em um *cluster* é completamente transparente para consumidores. O consumo de mensagens de uma fila pode ser realizado a partir de qualquer nó do cluster.

Por último, o *Indexer* compõe a parte final do fluxo de informações na Camada de Transporte. Além de possuir a função de consumir as mensagens presentes em cada fila, o *Indexer* analisa e transforma cada mensagem recebida.

```

1      {
2          "client": {
3              "name": "host01",
4              "address": "192.168.0.1",
5              "version": "0.23.2",
6          },
7          "check": {
8              "type": "metric",
9              "command": "/opt/sensu/embedded/bin/metrics-load.rb -p",
10             "interval": 1,
11             "name": "load-metrics",
12             "duration": 0.812,
13             "output": "load_avg.one 4.34 load_avg.five 3.56 load_avg.fifteen 3.44",
14             "status": 0,
15         },
16         "timestamp": 1510250912
17     }

```

Figura 5.7. Exemplo de mensagem não tratada.

A figura 5.7 ilustra uma mensagem não tratada no formato JSON que representa a medição da carga na máquina *host01*. A maioria dos campos é facilmente traduzida no esquema chave valor necessária para inserção adequada no banco de dados. No entanto, o campo *output* da linha 13 possui os valores das três métricas (*avg one*, *avg five* e *avg fifteen*) agregados em uma única linha. Nesse caso, se a mensagem for enviada ao banco de dados o valor do campo *output* seria considerado uma string e os valores das métricas não seriam acessíveis para computação ou apresentação gráfica. Assim, na figura 5.8, após o *parsing* realizado pelo *indexer*, é possível observar que o campo *output* foi removido e seu valor re-organizado em 3 novos campos nas linhas 13, 14 e 15. Além de possuírem valores acessíveis dentro do banco de dados, esses novos campos agora também são indexáveis, facilitando buscas sobre eles.

Finalmente, tanto para produtores (*shippers*) quanto para consumidores (*indexers*) escalabilidade por ser obtida através da criação de novas instâncias do mesmo componente, onde cada uma é responsável por processar um conjunto de mensagens, sejam elas para produção ou consumo. Por fim, a figura 5.9 ilustra adição de mais uma camada ao sistema de monitoração.

Alternativamente, como substitutos das ferramentas utilizadas temos: *Kafka* (fila de mensagens) [Kreps et al., 2011], *Syslog-ng* (*shipper* e *indexer*) [LLC, 2018].

```

1      {
2          "client": {
3              "name": "host01",
4              "address": "192.168.0.1",
5              "version": "0.23.2",
6          },
7          "check": {
8              "type": "metric",
9              "command": "/opt/sensu/embedded/bin/metrics-load.rb -p",
10             "interval": 1,
11             "name": "load-metrics",
12             "duration": 0.812,
13             "avg.one": 4.34,
14             "avg.five": 3.5,
15             "avg.fifteen": 3.44,
16             "status": 0,
17         },
18         "timestamp": 1510250912
19     }

```

Figura 5.8. Exemplo de mensagem tratada.

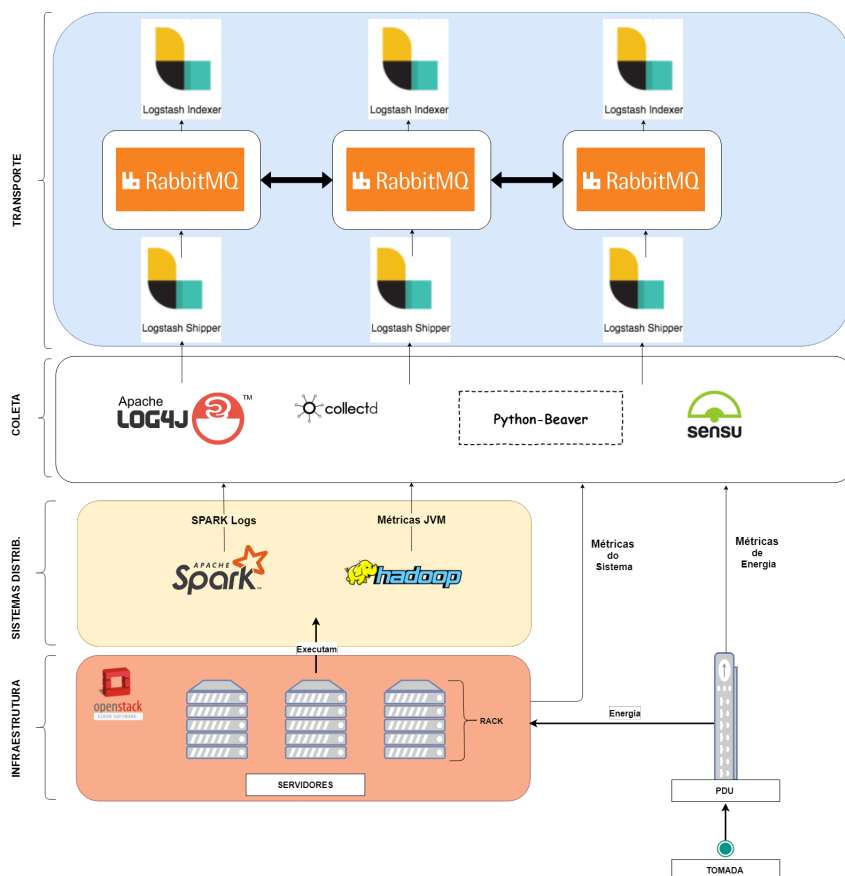


Figura 5.9. Camadas de Infraestrutura, Sistemas Distribuídos, Coleta e Transporte.

5.5 Camada de armazenamento

Monitorar sistemas de computação em nuvem requer uma especial atenção voltada para o armazenamento das informações, visando garantir confiabilidade ao registrá-las, facilidade na forma de recuperá-las e compatibilidade com a camada de visualização. Não menos importante, justamente por se tratar de nuvens computacionais, deve-se observar com bastante atenção a escalabilidade dos bancos de dados escolhidos, além do espaço que será ocupado para o armazenamento.

Neste trabalho foram selecionados dois bancos de dados específicos para o tratamento de séries temporais. O primeiro deles é o Elasticsearch [Gormley & Tong, 2015] criado pela empresa *Elastic*. Desenvolvido utilizando como base o projeto Apache Lucene [McCandless et al., 2010], Elasticsearch é um mecanismo de busca capaz de realizar buscas completas por texto através de uma interface web e o uso de documentos JSON. Mais precisamente, neste trabalho sua função é servir como um banco de dados para as informações relacionadas a *logs* no sistema. Algumas de suas características favorecem seu uso no domínio de *logs*. A primeira delas é o fato de ser uma plataforma *near real time*, que significa pequeno atraso (normalmente 1 segundo) entre o tempo em que um documento é indexado até o momento em que se torna disponível, atendendo à propriedade de Pontualidade. Outro ponto importante se refere à sua escalabilidade, permitindo a criação de *clusters* sob o modelo mestre-escravo com capacidades de pesquisa e indexação em todos os nós que o compõem.

Por outro lado, o banco de dados Influxdb foi selecionado para o armazenamento de métricas. Assim como o sistema anterior, também possui características como ser *NoSQL*, armazenar informações em pequenos blocos e ser especializado em séries temporais. Embora Influxdb e Elasticsearch tratem do mesmo problema, séries temporais, atributos de desempenho tornam o uso do Influxdb mais adequado para o armazenamento de métricas. Segundo o relatório de *benchmark* [Persen, 2016] que compara o uso de Influxdb e Elasticsearch para o armazenamento de métricas, o primeiro possui de 4 a 16 vezes melhor compressão de dados, ingestão de dados aproximadamente 8 vezes mais veloz e resposta em consultas 4 a 10 vezes mais rápidas. O atributo de compressão de dados é especialmente importante ao considerar que sistemas de monitoração atuam continuamente sobre um grande volume de informações, e dessa forma, reduzir o custo de espaço é essencial. Além disso, a ingestão de dados também é importante na escolha de banco de dados, uma vez que precisa ser capaz de absorver todos os dados produzidos. Por fim, ambos os sistemas citados no início deste parágrafo foram utilizados durante o desenvolvimento e assim adicionamos a camada de armazenamento na figura 5.10.

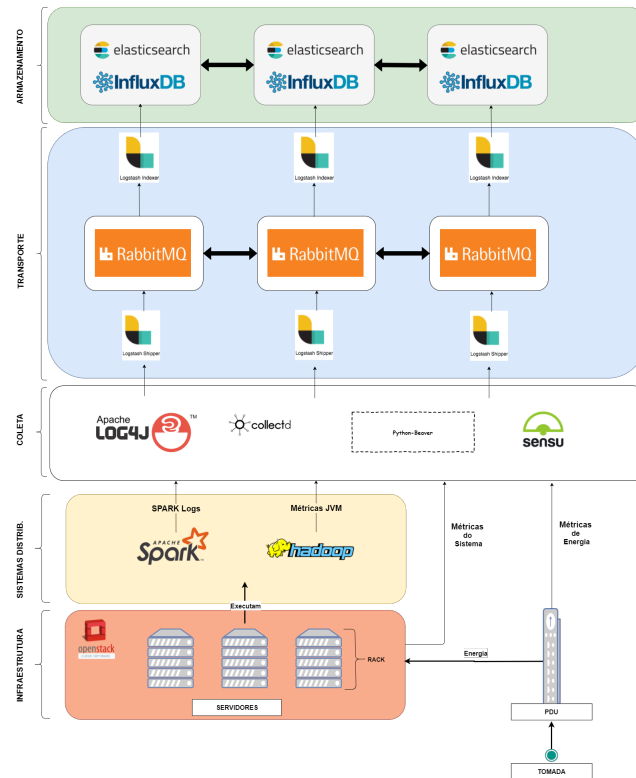


Figura 5.10. Camadas de Infraestrutura, Sistemas Distribuídos, Coleta, Transporte e Armazenamento.

Alternativamente, como substitutos desses bancos de dados podemos citar: *Prometheus* [Prometheus, 2018], *DalmatinerDB* [FiFo, 2014] e *OpenTSDB* [OpenTSDB, 2017].

5.6 Camada de visualização

Como mencionado na seção 4.4, devido à variedade de opções para ferramentas gráficas, não há uma escolha fixa ou limitação de componentes a serem utilizados para apresentar os dados que estão armazenados. As ferramentas de **visualização** neste trabalho foram selecionadas para suprir necessidades específicas. Por exemplo, a ferramenta *Kibana* permite visualizar *logs* e fazer consultas. A figura 5.11 apresenta um exemplo de *dashboard* com *logs* de uma aplicação Spark. Por outro lado, *Grafana* é mais adequado à visualização de métricas. Pela figura 5.12 é possível observar o tráfego de rede acumulado dos hosts ao longo da execução de uma aplicação. Por fim, a ferramenta *Uchiwa* foi incluída para acompanhar os agentes *Sensu*. A figura 5.13 apresenta um exemplo de acompanhamento de agentes. Finalmente, adicionamos a camada de visualização na figura 5.14.

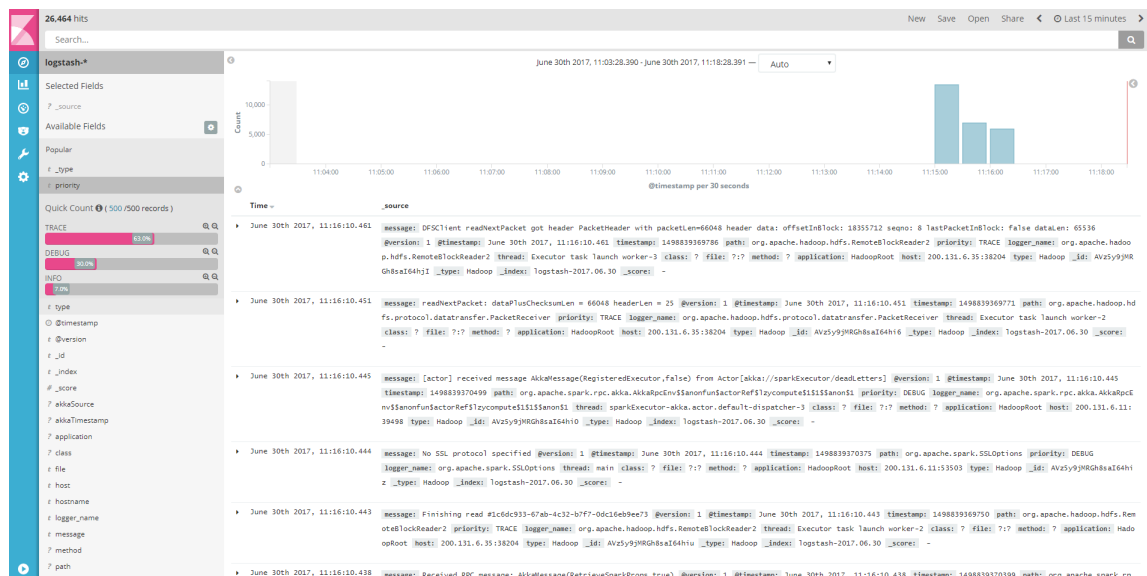


Figura 5.11. Exemplo de *dashboard Kibana*. No canto esquerdo existem informações relacionadas aos campos indexáveis dos logs coletados. Em rosa estão apresentadas as porcentagens de mensagens para três níveis de verbosidade diferentes (TRACE, DEBUG, INFO). Ao centro, um histograma apresenta o volume de mensagens em função do tempo e, logo abaixo, um pequeno conjunto de mensagens de log mais recentes de acordo com seu *timestamp*.

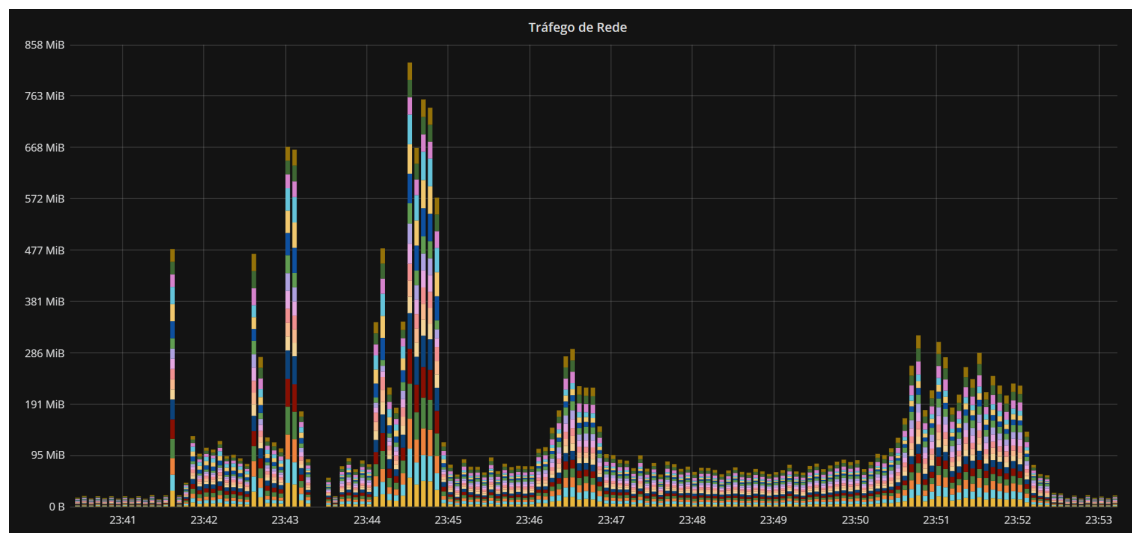
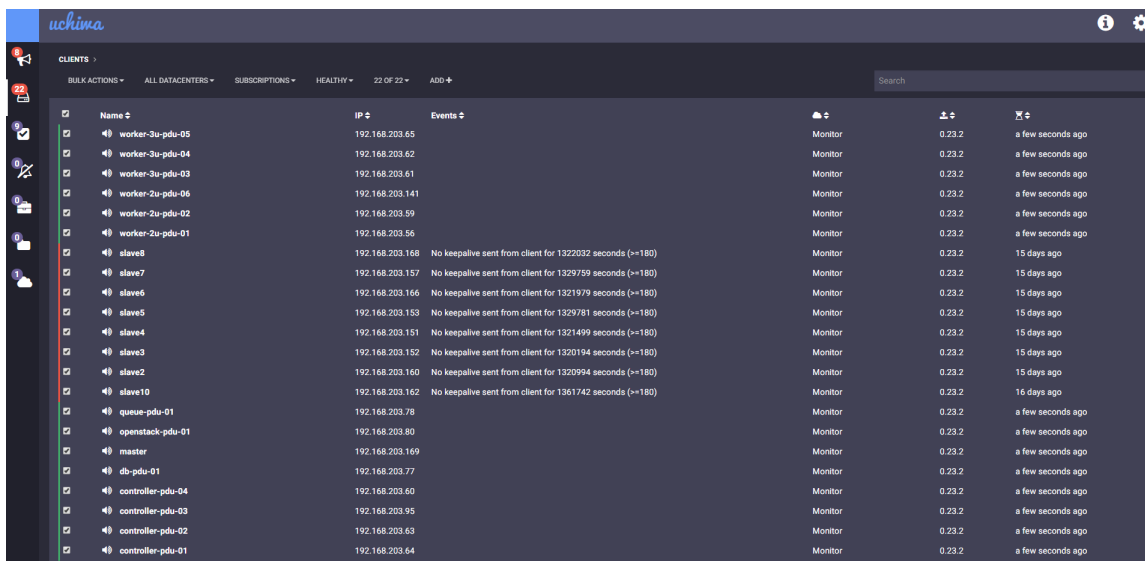


Figura 5.12. Exemplo de *dashboard Grafana*. As cores em cada uma das barras representam a quantidade de dados recebidos ou transmitidos por uma interface de rede.

Alternativamente, como substitutos dessas ferramentas podemos citar: *Graphite* [Dixon, 2017] e *PagerDuty* [PagerDuty, 2018].



Name	IP	Events	Status	Version	Last Event
worker-3u-pdu-05	192.168.203.65		Monitor	0.23.2	a few seconds ago
worker-3u-pdu-04	192.168.203.62		Monitor	0.23.2	a few seconds ago
worker-3u-pdu-03	192.168.203.61		Monitor	0.23.2	a few seconds ago
worker-2u-pdu-06	192.168.203.141		Monitor	0.23.2	a few seconds ago
worker-2u-pdu-02	192.168.203.59		Monitor	0.23.2	a few seconds ago
worker-2u-pdu-01	192.168.203.56		Monitor	0.23.2	a few seconds ago
slave8	192.168.203.168	No keepalive sent from client for 1322032 seconds (~180)	Monitor	0.23.2	15 days ago
slave7	192.168.203.157	No keepalive sent from client for 1329799 seconds (~180)	Monitor	0.23.2	15 days ago
slave6	192.168.203.166	No keepalive sent from client for 1321979 seconds (~180)	Monitor	0.23.2	15 days ago
slave5	192.168.203.153	No keepalive sent from client for 1329781 seconds (~180)	Monitor	0.23.2	15 days ago
slave4	192.168.203.151	No keepalive sent from client for 1321499 seconds (~180)	Monitor	0.23.2	15 days ago
slave3	192.168.203.152	No keepalive sent from client for 1320194 seconds (~180)	Monitor	0.23.2	15 days ago
slave2	192.168.203.160	No keepalive sent from client for 1320994 seconds (~180)	Monitor	0.23.2	15 days ago
slave10	192.168.203.162	No keepalive sent from client for 1361742 seconds (~180)	Monitor	0.23.2	16 days ago
queue-pdu-01	192.168.203.78		Monitor	0.23.2	a few seconds ago
openstack-pdu-01	192.168.203.80		Monitor	0.23.2	a few seconds ago
master	192.168.203.169		Monitor	0.23.2	a few seconds ago
db-pdu-01	192.168.203.77		Monitor	0.23.2	a few seconds ago
controller-pdu-04	192.168.203.60		Monitor	0.23.2	a few seconds ago
controller-pdu-03	192.168.203.95		Monitor	0.23.2	a few seconds ago
controller-pdu-02	192.168.203.63		Monitor	0.23.2	a few seconds ago
controller-pdu-01	192.168.203.64		Monitor	0.23.2	a few seconds ago

Figura 5.13. Exemplo de *dashboard Uchiwa*. Podemos observar uma lista contendo os nomes e os endereços IP de todos os coletores registrados no sistema. Além disso, ainda são apresentadas informações como a ocorrência de um evento, a versão do coletor e quanto tempo se passou desde a última comunicação com o sistema monitor. Os agentes marcados com tarja vermelha apresentam problemas críticos. Nesse caso, foram desconectados ou estão inaptos a coletar dados.

5.7 Camada de alertas

A camada de alertas foi implementada de forma simplificada no sistema, seguindo as ideias expressas na seção 4.5. Foram incluídas duas formas de criação de alarmes. A primeira aproveita uma característica da ferramenta *Grafana*; como pode ser visto na figura 5.15. A segunda forma de criação de alarmes faz uso de uma característica da ferramenta *Sensu*; nela o administrador configura um arquivo JSON contendo o endereço de um servidor SMTP e as respectivas credenciais para que *Sensu* direcione suas notificações ao servidor, de onde eventualmente serão transmitidas via e-mail. Esse método requer um servidor SMTP previamente configurado e maior conhecimento do usuário para sua configuração. É importante ressaltar que componentes mais complexos de alertas podem ser facilmente acoplados ao sistema, seja através da camada de Transporte ou como uma forma de saída do *Sensu*.

5.8 Camada de correlação de eventos

Como visto anteriormente, métricas e *logs* coletados são direcionados para a fila *RabbitMQ*. Sendo assim, para correlacioná-los e analisá-los, é preciso obtê-los da fila. Esse fluxo de informações coletado da fila possui o formato de *stream* [Babcock et al., 2002].

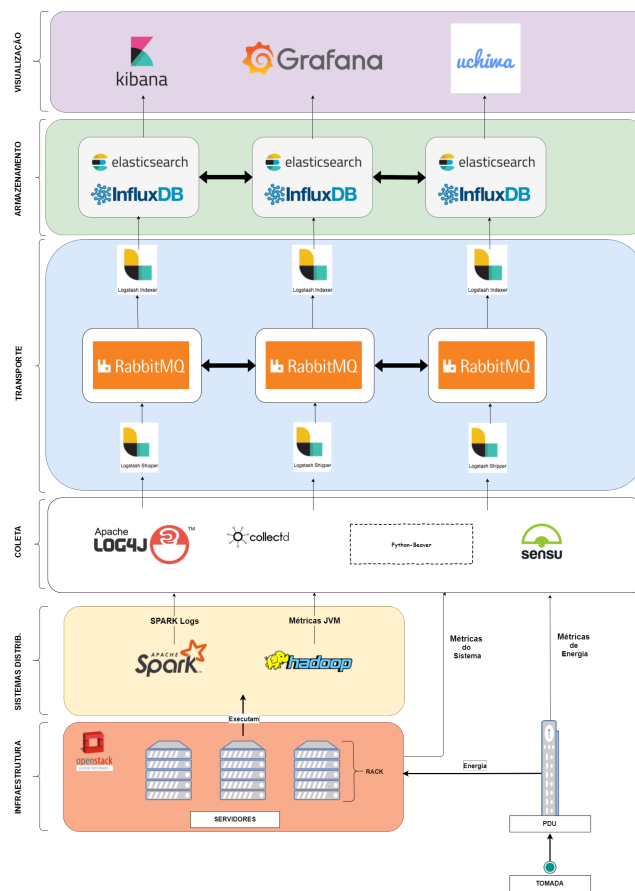


Figura 5.14. Camadas de Infraestrutura, Sistemas Distribuídos, Coleta, Transporte, Armazenamento e Visualização.

Nesse caso, a ferramenta *Riemann* foi escolhida por ser um poderoso processador de *streams*, construído especialmente para processá-las a partir de informações de monitoração de ambientes distribuídos. *Riemann* é uma ferramenta de código aberto escrita em Clojure que é executada sobre Máquina Virtual Java (JVM). Foi construída para explorar o máximo de paralelismo possível em Clojure e na JVM, pode tratar de milhões de eventos por segundo e recebe os eventos com latências de milissegundos. Usando Clojure, linguagem de programação com paradigma funcional dialeto de Lisp, é possível criar várias transações através de fluxos para manipular *streams*. Exemplos de ações possíveis são checar o estado de um serviço em um host, emitir avisos ao detectar uma sequência específica de eventos e enviar correlações analisadas para as ferramentas de visualização. Dentro do sistema de monitoração, *Riemann* é ligado aos *exchanges* de logs e métricas, permitindo que os dados sejam enviados à ferramenta e aos bancos de dados de forma eficiente. Alternativamente, como um potencial substituto dessa ferramenta podemos citar o *Apache Storm* [van der Veen et al., 2015].

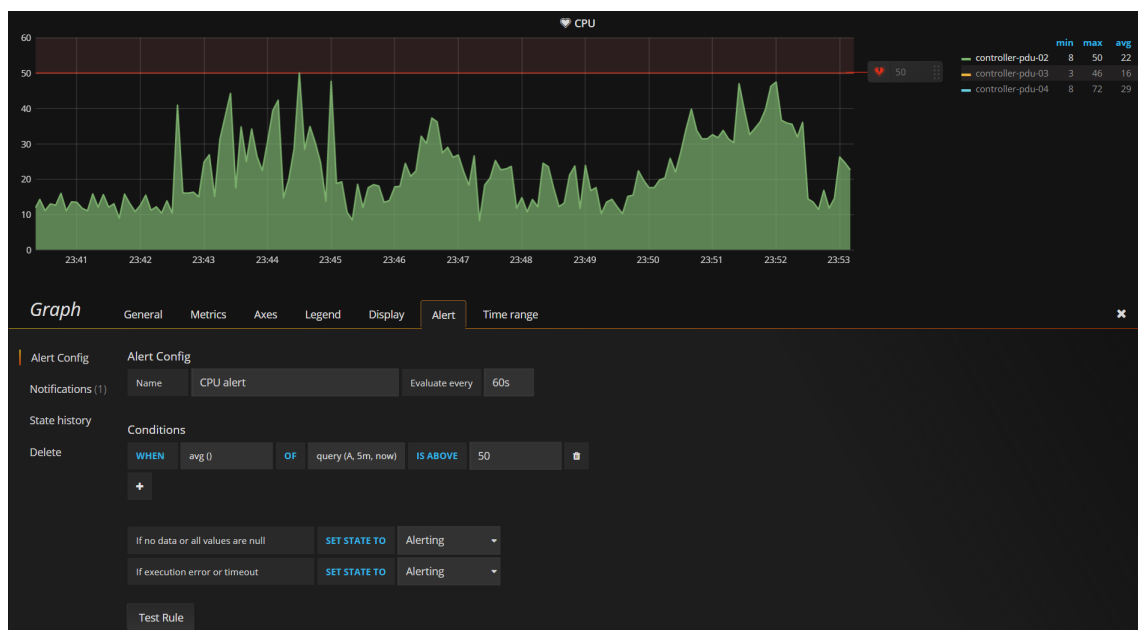


Figura 5.15. Exemplo de criação de alarmes no *Grafana*. Um alarme é configurado para criar notificações via e-mail caso a utilização de CPU atinja 50%. Além disso, a condição do alarme é avaliada a cada intervalo de 60 segundos. Esse tipo de alarme pode ser definido rapidamente por qualquer usuário.

5.9 Uso de containers

Por se tratar de um sistema que deve ser escalável, é inviável que todas as ferramentas descritas anteriormente sejam instanciadas em único servidor. Além disso, como o objetivo é distribuir apenas componentes desse sistema entre os servidores dedicados à monitoração, a criação de máquinas virtuais para tal fim resultaria em *overhead* desnecessário. Assim, decidimos optar pelo uso de *containers* para cada um dos componentes do sistema de monitoração.

Em ambientes virtualizados comuns, máquinas virtuais (VMs) executam sobre o hardware de máquinas físicas usando um hipervisor. *Containers* por sua vez compartilham o kernel do sistema operacional do host em que executam, o que pode ser chamado de virtualização a nível de sistema operacional. O isolamento significa que caso haja problema em um determinado *container*, como por exemplo consumo excessivo de recursos por um processo, isto afetará apenas os recursos do mesmo, e não todos os recursos do host em que executa. Duas características influenciaram a decisão de usar *containers* ao invés de máquinas virtuais. A primeira, como já mencionado, por executarem apenas aplicações *containers* são bem mais leves, logo um servidor comporta mais *containers* que máquinas virtuais. A segunda, *containers* são altamente flexíveis, podendo ser facilmente destruídos e criados em máquinas diferentes

sem grande *overhead*.

5.9.1 Docker

Docker é uma plataforma de código aberto para desenvolvimento, envio e implantação de diversas aplicações de maneira rápida. Além do conceito de *containers*, *Docker* auxilia no rápido envio, teste e implantação de código diminuindo o ciclo entre o momento em que o código é escrito e sua execução. *Docker* faz isso oferecendo nos *containers* uma estrutura que gerencia e implanta aplicações [Docker, 2013]. *Docker* utiliza características de isolamento de recursos (CPU, memória, rede, etc) do kernel Linux como *cgroups* e *namespaces* para permitir que *containers* independentes executem em uma mesma instância Linux sem que seja necessário o uso de diferentes máquinas virtuais.

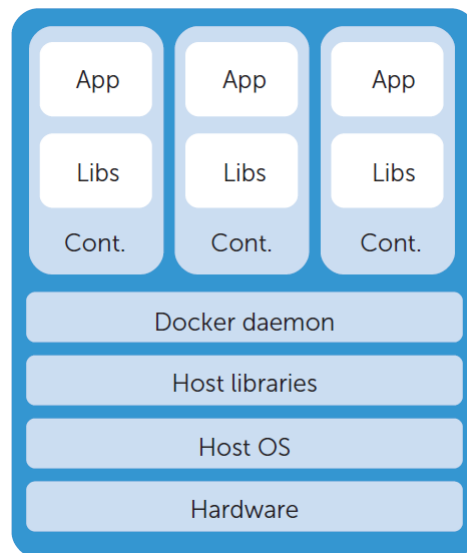


Figura 5.16. Modelo de execução *Docker*. Fonte adaptada de [Combe et al., 2016]

A tecnologia *Docker* é de vital importância para este trabalho, por uma série de razões. A primeira delas reside na facilidade de implantação de aplicações, uma vez que são necessários diversos componentes para a criação de um sistema de monitoração completo. Dessa forma, o uso de *containers* praticamente extingue os esforços de instalação desses componentes. Além disso, a característica de isolamento evita possíveis (muitas vezes constantes) conflitos de dependências. Outra razão é a manutenibilidade, considerando a velocidade com que novos softwares e suas versões são liberadas no mercado: caso uma aplicação precise ser atualizada ou mesmo substituída (com pequenas alterações no código) é possível trocar um *container* inteiro em poucos minutos. Por

último, como mencionado anteriormente, apesar de *containers* compartilham recursos do host, eles são isolados entre si. Assim, para que o sistema funcione de maneira adequada, é necessário permitir a interconexão entre esses componentes. Para isso, *Docker* oferece suporte à criação de múltiplas redes virtuais entre *containers*. Para cada *container*, *Docker* cria uma pilha de rede independente e pode ligá-las a switches virtuais, permitindo que aplicações executando em diferentes *containers* possam interagir de maneira transparente e segura [Bui, 2015].

5.9.2 Containers distribuídos: Docker swarm

A instanciação de *containers* comumente realizada pelo *Docker* é possível somente em um único servidor. Para aproveitar todo o potencial de escalabilidade do sistema, precisamos que *containers* estejam distribuídos entre os servidores de monitoração e sejam capazes de se comunicar estando em hosts distintos. Dessa forma, para escalar os *containers Docker* foi utilizado o *Docker swarm*. Executar *Docker* em modo *swarm*¹ permite a criação de um cluster de *Docker Engines* (os Daemons/Servidores *Docker*). Um *swarm* é composto de um ou mais nós, sejam eles máquinas físicas ou virtuais, executando *Docker Engine* em modo *swarm*.

Cada nó *swarm* pode atuar como **administrador** ou **trabalhador**, ou desempenhar os dois papéis ao mesmo tempo. Nós administradores são responsáveis por manter informações a respeito do cluster e pela distribuição de unidades de trabalho para os nós trabalhadores chamadas **tarefas**. Uma tarefa é a unidade atômica de escalonamento de um cluster *swarm*. Cada tarefa instancia um *container* com todos os comandos que nele devem ser executados. Uma vez atribuída a um nó trabalhador, uma tarefa não pode ser movida para outro, ou seja, executa no nó em questão ou retorna sem sucesso. Nós trabalhadores notificam aos administradores sobre o status da tarefa: caso o *container* falhe, a tarefa é retornada sem sucesso e o nó administrador cria uma nova tarefa réplica.

De maneira geral, quando um usuário deseja criar uma aplicação no cluster *swarm* ele o faz através de um serviço. Ao criar um **serviço**, ele especifica qual imagem (sequência de comandos) o *container* deve usar, portas que tornarão o serviço disponível fora do cluster *swarm*, a rede virtual na qual os *containers* devem ser interconectados, limitações de CPU e memória e, por fim, a quantidade de réplicas do mesmo *container* que devem ser executadas. Quando um serviço é definido, um nó administrador se responsabiliza pelo serviço e agenda tarefas a serem executadas pelos nós trabalhadores para que o serviço seja disponibilizado pelo cluster *swarm*. Por exemplo,

¹*Swarm* em inglês significa enxame

o diagrama da figura 5.17 apresenta um cenário onde se deseja balancear 3 instâncias de um *HTTP Listener*. Um serviço *swarm* é especificado com 3 réplicas e utiliza a imagem do listener *nginx*. Logo depois, o administrador lança uma tarefa em cada nó trabalhador disponível.

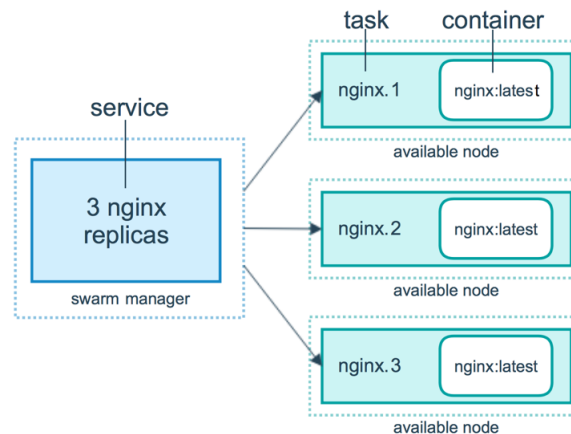


Figura 5.17. Serviço *HTTP Listener* com três réplicas. [Docker, 2016]

Existem dois tipos de serviço, o replicado e o global. No modelo de serviço replicado, um nó administrador distribui entre os nós disponíveis um número específico de tarefas replicadas na quantidade escolhida. O modelo global, uma tarefa (instância) do serviço é executada em cada nó disponível do cluster *swarm*. Além disso, toda vez que um novo nó é adicionado ao cluster uma tarefa do serviço global é atribuída a ele. A figura 5.18 apresenta um serviço com replicação 3 e um serviço global.

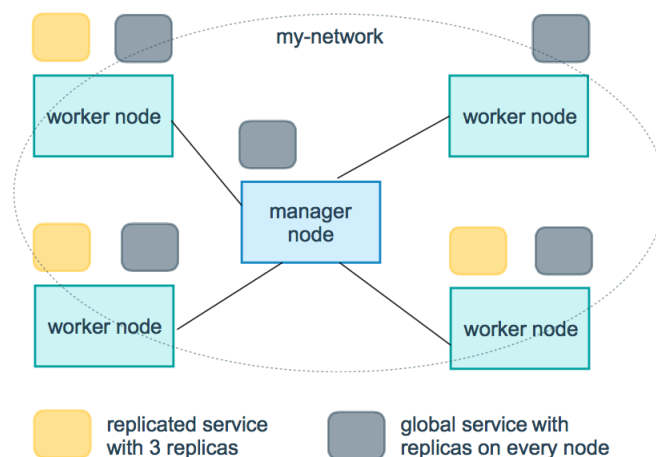


Figura 5.18. Serviço global e replicado em *swarm*. [Docker, 2016]

É importante ressaltar que uma *swarm* possui uma camada de DNS interna que automaticamente realiza o balanceamento de carga entre os *containers* de cada serviço. Além disso, o acesso a qualquer serviço disponibilizado pelo cluster *swarm* pode ser feito através de qualquer um dos nós. Caso um nó receba uma requisição por um serviço para o qual não possua uma instância, a requisição é direcionada a outro nó do cluster que possua tal instância.

Por essas características, a estrutura *swarm* realiza um papel importante na escalabilidade do sistema. Ambos os bancos de dados usado são beneficiados pela distribuição entre diferentes hosts; dessa forma, foram empregados como serviços globais a fim de distribuir os dados pela maior quantidade possível de nós. Além disso, a fila de mensagens e os componentes de *shipper* e *indexer* podem ser escalados e reduzidos sob demanda através de serviços replicados. Agentes coletores podem enviar seus dados a qualquer um dos nós do cluster *swarm*, escalando a capacidade de recepção de dados horizontalmente. Adicionalmente, todos os componentes do sistema foram empregados como serviço, logo, caso uma tarefa que instancie um componente falhe, *swarm* automaticamente cria uma nova instância replicada que desempenha o mesmo trabalho da anterior. Por consequência, caso um nó inteiro da infraestrutura falhe, todos os serviços são replicados entre os outros nós *swarm* disponíveis.

5.10 Escalabilidade e tolerância a falhas

Além dos pontos de escalabilidade fornecidos pela arquitetura citados na seção 4.7, podemos observar mais algumas características interessantes nas ferramentas escolhidas. A integração entre as ferramentas *Sensu* e *RabbitMQ* é escalável tanto durante a coleta dos dados como também na sua recepção durante a execução em modelo *push* idealizado pela arquitetura *Seshtat*. Além disso, por estarem instanciados como serviços dentro de um cluster *swarm*, tanto *Logstash* (*indexers* e *shippers*) quanto *RabbitMQ* podem ser replicados dinamicamente a fim de aumentar a robustez do sistema. Como mencionado, ambos os bancos de dados trabalham de forma distribuída, logo, foram instanciados como serviços globais para que estivessem presentes em todo o cluster de monitoração.

A tolerância a falhas está presente principalmente nas camadas de **Transporte** e **Armazenamento**. *RabbitMQ* oferece diferentes opções de persistência em disco para as filas de mensagem [Pivotal, 2016a]. Além disso, também possui mecanismos para re-envio de mensagens [Pivotal, 2016b], com restrições mais relaxadas, uma vez que a perda ou duplicação de poucas mensagens é aceitável. Adicionalmente, ambos

os bancos de dados possuem mecanismo de tolerância a falhas através da replicação de dados em blocos [Elastic, 2016a; Betts, 2017] (também conhecidos como *shards* ou *chunks*), tornando possível restaurar dados caso algo seja corrompido ou um nó falhe. Falhas em agentes de coleta podem acarretar perda temporária de informações sobre um host específico. Entretanto, esse tipo de situação é em geral resolvida através da geração de alertas e notificações de sistema para que administradores ou ferramentas automatizadas tomem decisões cabíveis em relação à falha, como por exemplo, reiniciar o agente.

5.11 Modularização do sistema

Como discutido na introdução deste capítulo, um dos grandes desafios na construção de um sistema de monitoração reside na seleção das ferramentas que o compõem e sua interação. É inviável construir do zero todas as ferramentas necessárias para um sistema de tal escala, principalmente devido a complexidade dos componentes envolvidos. Por exemplo, um sistema completo de banco de dados otimizado para trabalhar com séries temporais requer uma equipe inteira durante vários meses para ser desenvolvido. Dessa forma, é preciso decidir quais ferramentas se encaixam melhor no sistema de forma modular, ou seja, adicionando componentes altamente especializados para que realizem determinada função.

Muitas vezes para um mesmo componente existe mais de uma opção que apresenta características desejáveis. Um exemplo disso são as filas de mensagem. Tanto *RabbitMQ* quanto *Kafka* [Kreps et al., 2011] possuem a capacidade de servir como um intermediário robusto para recepção de dados. Neste trabalho, *RabbitMQ* foi escolhido por possuir sinergia com a ferramenta de coleta *Sensu*. No entanto, caso novos requisitos fossem adicionados, a inclusão de *Kafka* poderia ser feita sem um grande impacto ao sistema graças a organização modular da arquitetura *Seshat*. Para sistemas planejados de forma ‘engessada’, ou seja, sem modularização, talvez essa mudança nem mesmo fosse possível. Nesse contexto, a arquitetura de monitoração com as definições de funcionalidades de cada componente, aliada à utilização dos *containers*, faz com que exista flexibilidade de alteração nos componentes, desde que sejam respeitadas as interfaces de comunicação oferecidas. Através do sistema de criação de imagens oferecido pelo *Docker* e a facilidade de criação de *containers*, a troca ou adição de um determinado componente pode ser realizado com poucos comandos, tornando o sistema extensível.

Adicionalmente, mesmo depois que as ferramentas são selecionadas e devidamente instaladas, eventualmente novas versões contendo mais funcionalidades são liberadas a

todo tempo. Nesse contexto, o isolamento fornecido pelos *containers* e a modularização praticamente eliminam a quebra de dependências, facilitando a manutenção do sistema e reduzindo seu *downtime*.

No capítulo a seguir, apresentamos testes de escalabilidade sobre a ferramenta em um ambiente em nuvem real e discutimos os resultados obtidos.

Capítulo 6

Resultados

Para avaliar a arquitetura *Seshat*, ela foi instanciada em uma estrutura completa para monitoração de um ambiente em nuvem real: um *cluster* de processamento de dados massivos em ambiente Spark sobre HDFS em máquinas virtuais orquestradas pela ferramenta Openstack. Nesse processo, consideramos os requisitos discutidos no Capítulo 3 e os detalhes da arquitetura descritos no Capítulo 4. Ao longo do capítulo serão apresentados experimentos de desempenho realizados, assim como uma série de casos de uso relacionados as nossas experiências com a ferramenta desenvolvida.

Os experimentos foram direcionados para avaliar a escalabilidade do sistema através do *stress* na inserção de um grande volume de métricas e *logs*. Para isso, foram utilizados 2 conjuntos de servidores. O primeiro, responsável pela computação e que será objeto da monitoração, foi instalado o ambiente Apache Spark versão 1.5.2 e o sistema de arquivos Hadoop/HDFS v2.6.0. Com a finalidade de realizar as medições de energia, foi instalado e configurado um PDU (*Power Distribution Unit*) Raritan PX-2 da série 5000, que é uma unidade de distribuição de energia capaz de monitorar variados parâmetros elétricos. A orquestração dos servidores virtualizados foi feita pelo OpenStack v2.3.1 com sua estrutura montada em uma máquina servidora core-i7 2600 3,4GHz com 16 GB de memória RAM. Para os nós de computação foram usadas 6 máquinas servidoras Intel Xeon E5-2620v4 2.1GHz com 32 GB de RAM, duas interfaces de rede Gigabit Ethernet e disco SATA de 2 TB, sobre os quais foi possível criar 24 máquinas virtuais com 8 GB de RAM e discos rígidos de 80GB. Todos os dados produzidos sobre consumo de energia, uso de CPU, disco, ocupação de memória, carga da máquina e tráfego de rede, além dos *logs* do processamento do Spark e da utilização do HDFS, foram monitorados e registrados em um segundo *cluster* de monitoração montado seguindo a arquitetura *Seshat*, que contou com três servidores core-i7 2600 3,4GHz com 16 GB de memória RAM. Em cada um dos três servidores foi instalado

Docker versão 17.05.0-ce e estabelecido um *cluster* de containers na plataforma *swarm*.

6.1 Carga de trabalho

Como carga de trabalho a ser monitorada, foi usada uma aplicação Spark que processava 8 GB de dados sobre um conjunto de *posts* do Twitter, contendo *tweets* coletados usando a API da plataforma, em busca de padrões frequentes usando o algoritmo *Twidd* [Dias et al., 2016]. A aplicação *Twidd* é uma implementação do algoritmo FPGrowth sobre a abstração de RDD's (Resilient Distributed Datasets) e resolve o problema de mineração de padrões frequentes, no qual dada uma série de transações, cada uma composta por um conjunto de itens, e um suporte, ele encontra todos os subconjuntos de itens que ocorrem no mínimo a quantidade de vezes especificada por esse suporte. A motivação para o uso dessa aplicação no trabalho reside no grande custo computacional inerente ao processo de geração dos subconjuntos de itens para cada transação, o que consequentemente produz diversas mensagens de *log* e valores variados de métricas. Para o algoritmo em função da base de dados adotada, foram definidas 128 partições, com suporte mínimo de 0,1.

6.2 Desempenho e escalabilidade

Com o objetivo de avaliar a escalabilidade da solução, dois experimentos foram realizados sobre a ferramenta de monitoração. O primeiro avaliou o comportamento do sistema sob *stress* devido à inserção de um elevado volume de métricas e o segundo o comportamento do sistema sob *stress* devido à inserção de um elevado volume de *logs*. Foram testadas 2 configurações para o servidor de monitoração: (i) todo o sistema executando em um servidor único e (ii) executando de forma distribuída, nas 3 máquinas dedicadas à monitoração. Para um servidor único, havia apenas uma instância de cada componente do sistema, enquanto o distribuído contou com instâncias da fila de mensagens e de bancos de dados em cada uma de 3 máquinas disponíveis.

6.2.1 Experimento de stress por métrica

Para o teste de *stress* por métricas, foram coletadas 7 tipos de métricas diferentes de cada um de 24 *hosts* do laboratório que se encontravam em estado *idle*. O intervalo das medições de métrica foi variado entre 1, 2 e 5 segundos. Foram realizadas medições em 5 intervalos de 30 minutos espaçados por uma janela de descanso de 5 minutos. Além disso, para o caso de servidor distribuído foram utilizadas 9 instâncias de *indexers* para

consumir da fila. A tabela 6.1 apresenta os resultados obtidos nesse teste com a média das 5 execuções.

Tabela 6.1. Desempenho e escalabilidade do sistema de monitoração sob volume elevado de dados de métricas

Servidor	Interv.	Total msgs.	Msgs. perdidas	% perdas
Único	1 s	280800	5795.8 ± 250.8	2.06%
	2 s	151200	111.4 ± 37.0	0.07%
	5 s	60480	22.2 ± 12.1	0.04%
Distribuído	1 s	280800	4535.2 ± 79.3	1.62%
	2 s	151200	90.8 ± 29.6	0.06%
	5 s	60480	6.6 ± 10.9	0.01%

A tabela 6.1 apresenta a comparação entre os intervalos de medição em cada configuração de servidor, a quantidade média de mensagens perdidas \bar{x} e sua variação em um nível de confiança de 90% e a porcentagem das mensagens perdidas em função do total de mensagens esperadas. Nesse experimento, sabemos o total de mensagens que deveriam ser armazenadas, já que o experimento é determinístico nesse sentido. Ao consultar o sistema após o experimento para verificar quantas mensagens foram realmente armazenadas, podemos determinar o número de mensagens perdidas.

O que chama de imediato a atenção na tabela 6.1 são as mensagens perdidas para os intervalos de 1 segundo, chegando próximo de 6 mil. No entanto, essa perda equivale a no máximo 2% do total esperado, mostrando que o sistema, mesmo executando em um servidor único, é capaz de receber a grande maioria das mensagens. Aprofundando a análise, para intervalos de 1s podemos constatar que além de uma média maior a variabilidade é muito superior em servidor único quando comparado ao distribuído.

Também podemos observar que o aumento no intervalo de medições para 5s teve muito impacto na quantidade de mensagens perdidas em ambas as configurações (único e distribuído), chegando a ser três ordens de grandeza menor. Isso acontece devido ao menor número de mensagens e menor stress.

Por fim, mesmo para as configurações com maior volume de mensagens por unidade de tempo, as perdas totais foram baixas para ambas as configurações de servidor. Com os recursos disponíveis no nosso laboratório, não fomos capazes de gerar uma configuração que levasse o serviço a um colapso. Apesar disso, podemos ver que a versão distribuída do sistema se mostrou capaz de lidar melhor com a carga, tendo perdas menores e exibindo uma variabilidade menor no seu comportamento.

6.2.2 Experimentos de stress por log

Para o teste de *stress* por *logs*, como carga executamos a aplicação *Twidd* na plataforma Spark em 10 máquinas virtuais da nuvem cliente. Também variamos a verbosidade do sistema de *logs* do Spark entre INFO e DEBUG. Cada teste foi executado 10 vezes em cada um dos níveis de verbosidade e intervalados de 15 minutos para permitir que todas as mensagens presentes na fila fossem consumidas. Além disso, os *indexers* foram aumentados para 21 instâncias no caso de servidor distribuído e mantida a instância única para o caso de servidor único.

O *dashboard* da figura 6.1 apresenta um exemplo das informações disponíveis a respeito da fila de mensagens na versão distribuída durante uma execução da aplicação Spark na verbosidade DEBUG. Nela é possível observar ao longo do tempo o acúmulo de mensagens na fila (gráfico de linha laranja no topo), como variam as taxas de entrega e recepção de mensagens (gráfico de linhas rosa, à esquerda), a eficiência com que as mensagens estão sendo consumidas (gráfico de linha azul, no centro) e o tráfego de rede agregado dos 3 servidores da monitoração (gráfico de barras coloridas à direita).

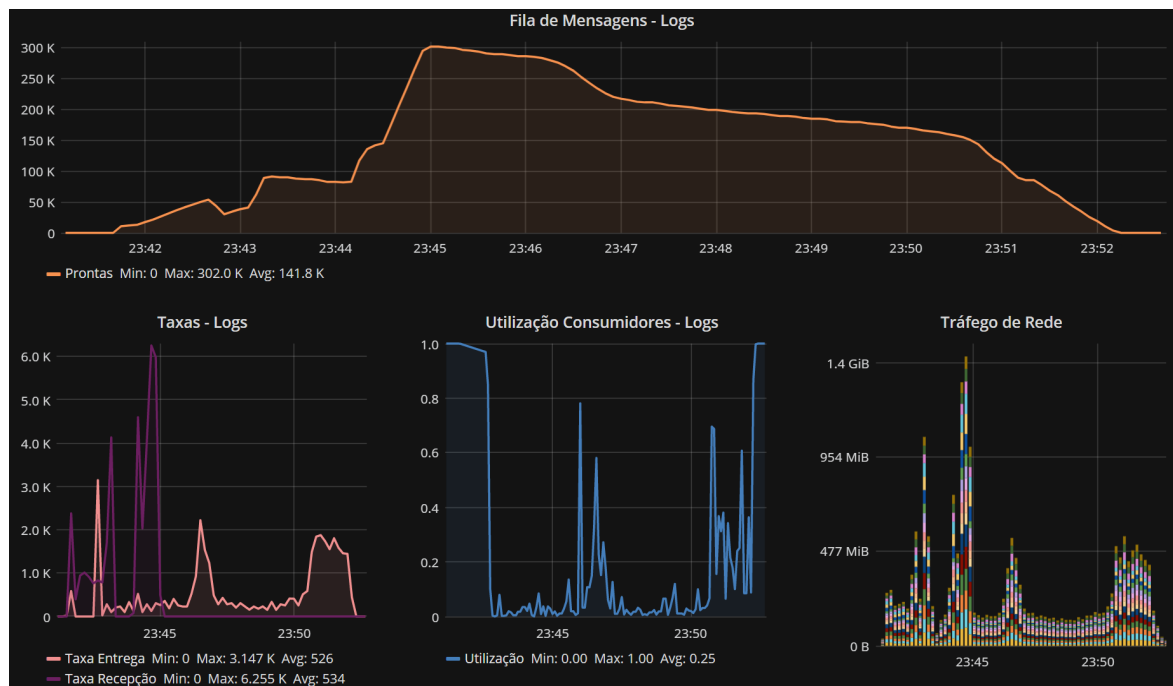


Figura 6.1. *Dashboard* de monitoração da fila de mensagens durante o experimento sobre a escalabilidade no sistema de processamento de *logs*.

A tabela 6.2 apresenta uma comparação entre as configurações de servidor para dois níveis de verbosidade diferentes em uma aplicação Spark. São eles: (i) INFO, que é o volume comum de mensagens de *log* de uma aplicação Spark e (ii) DEBUG

com um elevado volume de mensagens de *log*. Para o experimento foi avaliado o atraso médio das mensagens dado em segundos, a taxa de entrega das mensagens por segundo aos consumidores e a taxa de recepção das mensagens por segundo na fila. Esse atraso é caracterizado pela diferença de tempo entre o momento que a mensagem é produzida pelo agente e o momento que é inserida no banco de dados. Todos os valores são acompanhados de seus respectivos comprimentos de intervalo em um nível de confiança de 90%.

Tabela 6.2. Desempenho e escalabilidade do sistema de monitoração sob volume elevado de mensagens de *logs*.

Servidor	Verbosidade	Núm. msg	Atraso	Taxa de Entrega	Taxa de Recepção
Único	INFO	128769	$163.9s \pm 0.8$	188.3 ± 4.3	510.5 ± 26.8
	DEBUG	2726140	$449.5s \pm 0.5$	356.9 ± 16.6	1804.4 ± 146.3
Distribuído	INFO	128209	$107.0s \pm 0.5$	250.4 ± 12.1	493.6 ± 25.2
	DEBUG	2699265	$192.7s \pm 0.2$	778.9 ± 52.6	2044.7 ± 182.4

Ao analisar a tabela 6.2 podemos constatar que o atraso médio chega a ser o dobro para o servidor único quando comparado ao servidor distribuído. Além disso, a taxa de entrega chega a ser o dobro dada a maior quantidade de consumidores na fila. Quanto à taxa de recepção não há diferença significativa entre as configurações de servidor único e distribuído. No entanto, quando comparadas as taxas de recepção entre a verbosidade a diferença chega a aproximadamente o triplo. Isso se deve principalmente ao volume de mensagens enviadas por cada agente de coleta. Assim, podemos observar um impacto significativo no atraso das mensagens quando comparamos as configurações de servidor único e distribuído, esse argumento é reforçado ao analisarmos as diferenças nas taxas de entrega entre eles.

6.3 Casos de uso

Com a monitoração em operação, todos os nossos experimentos na área de processamento de dados massivos com Spark passaram a ser monitorados: uma vez configurado e disparado um *cluster* virtualizado para execução de uma aplicação, os *logs* das máquinas virtuais Java, do ambiente Spark e do HDFS são coletados, assim como diversas métricas de interesse de mencionadas na seção 5.1 sobre cada máquina virtual do *cluster*, bem como as máquinas físicas que as hospedam e os dados de consumo de energia de cada *host*, obtidos do PDU.

A figura 6.2 apresenta um dos *dashboards*¹ desenvolvidos no ambiente para a exploração visual dos dados. Nela podemos observar como se comportam ao longo do tempo as métricas de uso de CPU, memória RAM, escritas em disco e tráfego de rede nos servidores de monitoração durante a execução de uma aplicação Spark.

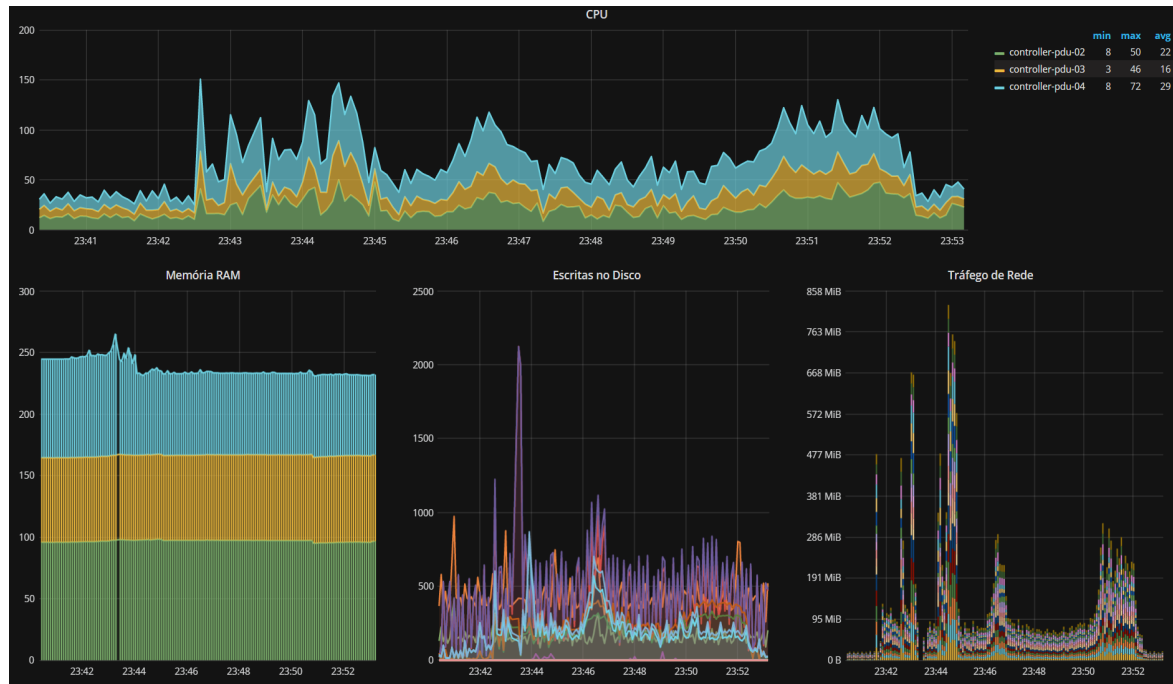


Figura 6.2. Dashboard de monitoração acompanhando um grupo de hosts, onde pode-se acompanhar a utilização de CPU, consumo de memória, escritas em disco e tráfego de rede enquanto uma aplicação Spark é executada em um *cluster* virtualizado.

Ao longo da nossa experiência com o sistema, diversos foram os casos onde a adoção da arquitetura simplificou tarefas de administração e gerência, e onde as informações coletadas foram úteis para na análise de comportamento de diferentes aplicações. Três desses casos são descritos a seguir.

Coletas de lixo excessivas por erros de configuração: ao investigar casos em que algumas execuções de uma aplicação Spark levavam muito mais tempo que outras para terminar, utilizamos um *dashboard* Grafana representado pela figura 6.3 para acompanhar a utilização de memória heap da aplicação Spark nas máquinas virtuais. Pela figura, podemos verificar que naquelas execuções mais longas, a aplicação era suspensa temporariamente(indicado pelas linhas estendidas). Buscando nos *logs* armazenados os eventos que ocorreram no momento da suspensão, foi possível observar que o problema estava associado ao processo de coleta de lixo das JVMs envolvidas. Uma

¹Os *dashboards* foram gerados diretamente pela ferramenta e seguem sua identidade visual

reconfiguração da memória e de outros elementos das JVMs resolveu o problema. Após uma nova execução da mesma aplicação, podemos observar na figura 6.4 em que não ocorre a coleta de lixo durante a execução e portanto não existem as linhas estendidas.



Figura 6.3. Dashboard de monitoração acompanhando a utilização de memória heap e de eden-space durante a execução de uma aplicação Spark no momento em que o problema foi detectado.

Perfis de consumo de energia imprevistos: em outra situação, observamos um consumo anormal de energia nas máquinas do *cluster*, com um comportamento periódico e de longa duração. Montamos então um *dashboard* com métricas de carga de CPU, tráfego de rede e consumo de energia das máquinas, onde pudemos verificar que o consumo de energia não tinha relação direta com as aplicações em execução. Observamos que o aumento de consumo tinha relação direta com mensagens enviadas pelo controlador OpenStack do *cluster* para o restante do sistema, o que permitiu identificar uma tarefa de atualização periódica que precisou ser reconfigurada.

A modularidade da arquitetura simplifica a mudança de escolhas: Outra situação em que a aplicação da arquitetura se mostrou extremamente útil foi durante o início dos testes de escalabilidade descritos a seguir. Inicialmente, havíamos escolhido *Influxdb* como nosso sistema de armazenamento de métricas. Quando iniciamos os testes de escalabilidade, que exigiam a instalação em um conjunto de máquinas para distribuir a tarefa de armazenamento, verificamos que a versão de código aberto do *Influxdb* não inclui os recursos para operação distribuída. Depois de alguma análise,



Figura 6.4. Dashboard de monitoração acompanhando a utilização de memória heap e de eden-space durante a execução de uma aplicação Spark no momento após a resolução do problema.

decidimos adotar um outro servidor *Elastic Search* com pequenas otimizações para atuar como base de dados para métricas, já que ele inclui recursos para operação distribuída. Graças à estrutura modular ditada pelo uso da arquitetura *Seshat*, a retirada de um sistema e a inclusão do outro foi feita de forma simples, exigindo poucas alterações no restante do sistema.

Capítulo 7

Conclusão

Monitoração de ambientes e aplicações é uma demanda crescente na comunidade de computação em nuvem, devido a suas variadas formas de uso e o valor que agregam. Observamos os diversos requisitos que diferenciam a monitoração em nuvem dos demais sistemas clássicos e a multidisciplinaridade envolvida no processo. A partir dessas observações, estruturamos *Seshat*, uma arquitetura de monitoração concebida em camadas e organizada em componentes, com uma definição clara dos comportamentos e funcionalidades esperados em cada caso. Como produto da arquitetura, desenvolvemos uma implementação da arquitetura que foi idealizada com base em ferramentas de código aberto e aplicamo-la a uma nuvem computacional voltada ao processamento de dados massivos.

Analisando o trabalho sob o ponto de vista tecnológico, adquirimos experiências enriquecedoras ao testarmos uma variedade de ferramentas para realizar a função de cada um dos componentes em cada camada. Adicionalmente, a utilização da virtualização por *containers* (*docker*) para os componentes do sistema acelerou o desenvolvimento de maneira surpreendente. Ao longo do projeto, a “clusterização” de *containers* (*docker swarm*) tomou uma posição de importância ainda maior na distribuição e roteamento dos serviços entre os servidores da monitoração permitindo, além de seu objetivo inicial de distribuição, realizar também o balanceamento de requisições ao sistema e fornecer mecanismos de tolerância a falhas. Essas últimas características foram além do esperado e certamente contribuíram para a melhoria na escalabilidade. Resaltamos também as dificuldades encontradas. Todas as ferramentas utilizadas foram desenvolvidas em linguagens de programação diferentes e possuem peculiaridades de configuração que exigiram entre alguns dias até semanas para alcançar seu funcionamento adequado. Mesmo operacionais, a integração entre as mesmas ainda consumiu um tempo elevado. Isso confirma o desafio sobre o caráter multidisciplinar discutido

na introdução desse trabalho.

Experimentos para avaliar a escalabilidade do sistema sob condições de carga elevada comprovaram que o sistema se comporta bem, com baixas perdas e sem comprometer sua funcionalidade. Testes com uma versão distribuída do sistema demonstraram sua eficiência ao diminuir significativamente a perda de mensagens e sua variabilidade, além da redução de atrasos. Nossos resultados indicam que o sistema tem potencial para escalar horizontalmente com o aumento da carga. Adicionalmente, a experiência com o sistema tem gerado diversos casos de uso bem sucedidos entre eles a publicação de dois artigos [Conceição et al., 2018; Volpini et al., 2018] e outros casos discutidos no texto. Dessa forma, comprovamos a aplicabilidade da arquitetura utilizando-a em um cenário real e que pode ser estendida à outros ambientes de processamento de dados massivos e modelos de nuvem.

Trabalhos futuros

Como trabalhos futuros, consideramos a oportunidade de avaliar mais experimentos, variando suas configurações com objetivo de determinar a melhor quantidade de *shippers* e o *indexers*, tendo como foco compreender melhor o comportamento das taxas de entrega e recepção.

Outra oportunidade a desenvolver de forma a complementar o sistema é estabelecer um mapeamento junto ao orquestrador de nuvem Openstack e junto aos hipervisores garantindo o rastreamento de ponta a ponta no processo, de modo que um evento que aconteça dentro de uma determinada aplicação que está sendo executado em uma determinada VM tenha sua identificação completa ao ser visualizada na monitoração, e.g. um executor do Spark que gere um evento de *log* pode ser visto em sua exata localização na VM em que se encontra.

Outro ponto interessante a ser explorado é a utilização do processador de *streams* sendo mais específico o Riemann. Por meio do Riemann será possível estabelecer correlações entre os dados coletados e os eventos ocorridos no ambiente. Consideramos também experimentar a monitoração em outros ambientes de processamento de dados massivos. Isso nos trará a oportunidade de amadurecer o sistema e torna-lo mais versátil e validar a generalidade da arquitetura.

Referências Bibliográficas

- Aceto, G.; Botta, A.; De Donato, W. & Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.
- Andreozzi, S.; De Bortoli, N.; Fantinel, S.; Ghiselli, A.; Rubini, G. L.; Tortone, G. & Vistoli, M. C. (2005). Gridice: a monitoring service for grid systems. *Future Generation Computer Systems*, 21(4):559–571.
- Anwar, A.; Sailer, A.; Kochut, A. & Butt, A. R. (2015). Anatomy of cloud monitoring and metering: A case study and open problems. Em *Proceedings of the 6th Asia-Pacific Workshop on Systems*, p. 6. ACM.
- Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I. et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.
- Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R. H.; Konwinski, A.; Lee, G.; Patterson, D. A.; Rabkin, A.; Stoica, I. et al. (2009). Above the clouds: A berkeley view of cloud computing. Relatório técnico, Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.
- Azure, M. (2010). Microsoft azure. <https://azure.microsoft.com/en-us/?v=17.14>. Acessado em agosto de 2017.
- Babcock, B.; Babu, S.; Datar, M.; Motwani, R. & Widom, J. (2002). Models and issues in data stream systems. Em *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1–16. ACM.
- Barbierato, E.; Gribaudo, M. & Iacono, M. (2013). Modeling apache hive based applications in big data architectures. Em *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '13, pp. 30–38, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

- Betts, R. (2017). Basic concepts. <https://www.influxdata.com/blog/influxdb-internals-101-part-one/>. Acessado em janeiro de 2018.
- Brunette, G.; Mogull, R. et al. (2009). Security guidance for critical areas of focus in cloud computing v2. 1. *Cloud Security Alliance*, pp. 1–76.
- Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv:1501.02967*.
- Buyya, R. (2000). Parmon: a portable and scalable monitoring system for clusters. *Software-Practice and Experience*, 30(7):723–740.
- Caron, E.; Rodero-Merino, L.; Desprez, F. & Muresan, A. (2012). *Auto-scaling, load balancing and monitoring in commercial and open-source clouds*. Tese de doutorado, INRIA.
- Ching, A. (2013). Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, p. 25.
- Clayman, S.; Galis, A. & Mamatas, L. (2010). Monitoring virtual networks with lattice. Em *Network operations and management symposium workshops (NOMS Wksp), 2010 IEEE/IFIP*, pp. 239–246. IEEE.
- CloudMonix (2015). Cloudmonix. <http://www.cloudmonix.com/aw/>. Acessado em agosto de 2017.
- CloudWatch (2009). Cloudwatch. <https://aws.amazon.com/pt/cloudwatch/>. Acessado em agosto de 2017.
- Coburn Watson, Scott Emmons, B. G. (2015). A microscope on microservices. Acessado em janeiro de 2018.
- Collectd (2017). Collectd. <https://collectd.org/>. Acessado em outubro de 2017.
- Combe, T.; Martin, A. & Di Pietro, R. (2016). To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62.
- Conceição, V. S.; Volpini, N. D. O. & Guedes, D. (2018). *Seshat: uma arquitetura de monitoração escalável para ambientes em nuvem*. Em *Anais do XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicação, Natal-RN. Sociedade Brasileira de Computação (SBC)*.

- Cooke, A.; Gray, A. J.; Ma, L.; Nutt, W.; Magowan, J.; Oevers, M.; Taylor, P.; Byrom, R.; Field, L.; Hicks, S. et al. (2003). R-gma: An information integration system for grid monitoring. Em *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, pp. 462–481. Springer.
- Dean, J. & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113. ISSN 0001-0782.
- Demers, A. J.; Gehrke, J.; Panda, B.; Riedewald, M.; Sharma, V.; White, W. M. et al. (2007). Cayuga: A general purpose event monitoring system. Em *CIDR*, volume 7, pp. 412–422.
- Dias, V.; Meira, W. & Guedes, D. (2016). Dynamic reconfiguration of data parallel programs. Em *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 190–197.
- Dixon, J. (2017). *Monitoring with Graphite: Tracking Dynamic Host and Application Metrics at Scale*. "O'Reilly Media, Inc."
- Docker (2013). Docker. <https://docs.docker.com/>. Acessado em outubro de 2017.
- Docker (2016). How services work. <https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/>. Acessado em 2017.
- Elastic (2016a). Basic concepts. https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html. Acessado em janeiro de 2018.
- Elastic (2016b). Filebeat. <https://www.elastic.co/products/beats/filebeat>. Acessado em setembro de 2017.
- Elastic (2016c). Metricbeat. <https://www.elastic.co/products/beats/metricbeat>. Acessado em agosto de 2018.
- Etsy (2018). Statsd. <https://github.com/etsy/statsd>. Acessado em agosto de 2018.
- Eugster, P. T.; Felber, P. A.; Guerraoui, R. & Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131.
- Ferguson, M. (2016). Why big data? <http://www.ibmbigdatahub.com/blog/why-big-data>. Accessed: 2018-06-06.
- FiFo, P. (2014). Dalmatinerdb. <https://dalmatiner.io/>. Acessado em agosto de 2018.

- Goldschmidt, T.; Jansen, A.; Koziol, H.; Doppelhammer, J. & Breivold, H. P. (2014). Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. Em *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pp. 602–609. IEEE.
- Gormley, C. & Tong, Z. (2015). *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc.
- Gu, X.; Hou, R.; Zhang, K.; Zhang, L. & Wang, W. (2011). Application-driven energy-efficient architecture explorations for big data. Em *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pp. 34–40. ACM.
- Gülcü, C. (2003). *The complete log4j manual*. QOS. ch.
- Hasselmeyer, P. & d'Heureuse, N. (2010). Towards holistic multi-tenant monitoring for virtual data centers. Em *Network Operations and Management Symposium Workshops (NOMS Wksp), 2010 IEEE/IFIP*, pp. 350–356. IEEE.
- Iyer, R.; Illikkal, R.; Zhao, L.; Newell, D. & Moses, J. (2009). Virtual platform architectures for resource metering in datacenters. *ACM SIGMETRICS Performance Evaluation Review*, 37(2):89–90.
- Jain, N.; Kit, D.; Mahajan, D.; Yalagandula, P.; Dahlin, M. & Zhang, Y. (2007). Known unknowns in large-scale system monitoring. *Review (October 2007)*.
- Katsaros, G.; Kübert, R. & Gallizo, G. (2011). Building a service-oriented monitoring framework with rest and nagios. Em *Services Computing (SCC), 2011 IEEE International Conference on*, pp. 426–431. IEEE.
- Kreps, J.; Narkhede, N.; Rao, J. et al. (2011). Kafka: A distributed messaging system for log processing. Em *Proceedings of the NetDB*, pp. 1–7.
- Kusters, N. P.; Smiley, J. R.; Brooker, M. J.; Guo, B.-J. & Levy, M. (2017). Dynamic burst throttling for multi-tenant storage. US Patent 9,639,397.
- Kutare, M.; Eisenhauer, G.; Wang, C.; Schwan, K.; Talwar, V. & Wolf, M. (2010). Monalytics: online monitoring and analytics for managing large scale data centers. Em *Proceedings of the 7th international conference on Autonomic computing*, pp. 141–150. ACM.
- Li, A.; Yang, X.; Kandula, S. & Zhang, M. (2010). Cloudcmp: comparing public cloud providers. Em *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 1–14. ACM.

- Lin, J. & Ryaboy, D. (2013). Scaling big data mining infrastructure: The twitter experience. *ACM SIGKDD Explorations Newsletter*, 14(2):6–19.
- Lincoln, F. & Slack, V. (1991). Power distribution unit. US Patent 5,073,120.
- LLC, O. I. (2018). Syslog-ng. <https://www.syslog-ng.com/products/open-source-log-management/>. Acessado em agosto de 2018.
- Mashayekhy, L.; Nejad, M. M.; Grosu, D.; Lu, D. & Shi, W. (2014). Energy-aware scheduling of mapreduce jobs. Em *Big Data (BigData Congress), 2014 IEEE International Congress on*, pp. 32–39. IEEE.
- Massie, M. L.; Chun, B. N. & Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840.
- Massonet, P.; Naqvi, S.; Ponsard, C.; Latanicki, J.; Rochwerger, B. & Villari, M. (2011). A monitoring and audit logging architecture for data location compliance in federated cloud infrastructures. Em *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1510–1517. IEEE.
- McCandless, M.; Hatcher, E. & Gospodnetic, O. (2010). *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co.
- Monasca (2015). Monasca. <https://wiki.openstack.org/wiki/Monasca>. Acessado em agosto de 2017.
- Monitis (2006). Monitis. <http://www.monitis.com/>. Acessado em agosto de 2017.
- Nagios (1999). Nagios. <https://www.nagios.org/>. Acessado em agosto de 2017.
- Newman, H. B.; Legrand, I. C.; Galvez, P.; Voicu, R. & Cirstoiu, C. (2003). Monalisa: A distributed monitoring service architecture. *arXiv preprint cs/0306096*.
- OpenTSDB (2017). Dalmatinerdb. <http://opentsdb.net/>. Acessado em agosto de 2018.
- Padhy, S.; Kreutz, D.; Casimiro, A. & Pasin, M. (2011). Trustworthy and resilient monitoring system for cloud infrastructures. Em *Proceedings of the Workshop on Posters and Demos Track*, p. 3. ACM.
- PagerDuty (2018). Pagerduty. <https://www.pagerduty.com/>. Acessado em agosto de 2018.

- Park, J.; Yu, H.; Chung, K. & Lee, E. (2011). Markov chain based monitoring service for fault tolerance in mobile cloud computing. Em *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pp. 520–525. IEEE.
- Peng, J.; Dai, Y.; Rao, Y.; Zhi, X. & Qiu, M. (2015). Modeling for cpu-intensive applications in cloud computing. Em *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICSS), 2015 IEEE 17th International Conference on*, pp. 20–25. IEEE.
- Pepple, K. (2011). *Deploying openstack*. "O'Reilly Media, Inc."
- Persen, T. (2016). Benchmarking influxdb vs. elasticsearch for time-series data, metrics and management. <https://www.influxdata.com/resources/benchmarking-influxdb-vs-elasticsearch-for-time-series/>. Acessado em janeiro de 2018.
- Pivotal (2007). Exchanges. <https://www.rabbitmq.com/tutorials/tutorial-three-python.html>. Acessado em novembro de 2017.
- Pivotal (2016a). Persistence configuration. <https://www.rabbitmq.com/persistence-conf.html>. Acessado em janeiro de 2018.
- Pivotal (2016b). Reliability guide. <https://www.rabbitmq.com/reliability.html>. Acessado em janeiro de 2018.
- Prometheus (2018). Prometheus. <https://prometheus.io/>. Acessado em agosto de 2018.
- Python-Beaver (2012). Python-beaver. <https://github.com/python-beaver/python-beaver>. Acessado em outubro de 2017.
- Romano, L.; De Mari, D.; Jerzak, Z. & Fetzner, C. (2011). A novel approach to qos monitoring in the cloud. Em *Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, pp. 45–51. IEEE.
- Sensu (2013). Sensu. <https://sensuapp.org/>. Acessado em outubro de 2017.
- Sensu, C. (2018). Client sensu. <https://sensuapp.org/docs/latest/reference/clients.html>. Acessado em outubro de 2017.

- Sensu, I. C. (2016). Issue client sensu. <https://github.com/sensu/sensu/issues/853>. Acessado em outubro de 2017.
- Shvachko, K.; Kuang, H.; Radia, S. & Chansler, R. (2010). The hadoop distributed file system. Em *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10. IEEE.
- Spring, J. (2011a). Monitoring cloud computing by layer, part 1. *IEEE Security & Privacy*, 9(2):66–68.
- Spring, J. (2011b). Monitoring cloud computing by layer, part 2. *IEEE Security & Privacy*, 9(3):52–55.
- Stallings, W. (1998). *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., Inc.
- Turnbull, J. (2013). *The Logstash Book*. James Turnbull.
- Turnbull, J. (2014). *The Art of Monitoring*:. James Turnbull. ISBN 9780988820241.
- van der Veen, J. S.; van der Waaij, B.; Lazovik, E.; Wijbrandi, W. & Meijer, R. J. (2015). Dynamically scaling apache storm for the analysis of streaming data. Em *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pp. 154–161. IEEE.
- Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; Saha, B.; Curino, C.; O'Malley, O.; Radia, S.; Reed, B. & Baldeschwieler, E. (2013). Apache hadoop yarn: Yet another resource negotiator. Em *Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13*, pp. 5:1–5:16, New York, NY, USA. ACM.
- Videla, A. & Williams, J. (2012). Rabbitmq in action. isbn: 9781935182979.
- Vinoski, S. (2006). Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89. ISSN 1089-7801.
- Viratanapanu, A.; Kamil, A.; Hamid, A.; Kawahara, Y. & Asami, T. (2010). On demand fine grain resource monitoring system for server consolidation. Em *Kaleidoscope: Beyond the Internet?-Innovations for Future Networks and Services, 2010 ITU-T*, pp. 1–8. IEEE.

- Volpini, N. D. O.; Conceição, V. S. & Guedes, D. (2018). Uma análise do consumo de energia de ambientes de processamento de dados massivos em nuvem. Em *Anais do XVII Workshop em Desempenho de Sistemas Computacionais e de Comunicação, Natal-RN. Sociedade Brasileira de Computação (SBC)*.
- Wang, C.; Schwan, K.; Talwar, V.; Eisenhauer, G.; Hu, L. & Wolf, M. (2011). A flexible architecture integrating monitoring and analytics for managing large-scale data centers. Em *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 141–150. ACM.
- Wieder, P.; Butler, J. M.; Theilmann, W. & Yahyapour, R. (2011). *Service level agreements for cloud computing*. Springer Science & Business Media.
- Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S. & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Em *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2. USENIX Association.
- Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S. & Stoica, I. (2010). Spark: Cluster computing with working sets. Em *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, pp. 10–10, Berkeley, CA, USA. USENIX Association.

Glossário

Broker Um message broker é um componentes intermediário que media a comunicação entre duas aplicações.

Buffer É uma região da memória utilizada para armazenar dados enquanto os mesmos estão sendo movidos de um lugar para outro.

Data burst É a transmissão de uma quantidade grande de dados em um curto período de tempo.

Log rotation É um processo automático utilizado na administração de sistemas no qual arquivos de logs obsoletos são arquivados.

Parsing É o processo de transformação de um conjunto de caracteres de entrada em um formato adequado ao da aplicação em que se deseja utiliza-los.

Plumbing É o termo utilizado para descrever a tecnologia e as conexões entre sistemas no modelo de computação em nuvem..

Timestamp uma cadeia de caracteres denotando a hora ou data que certo evento ocorreu.