Seshat: uma arquitetura de monitoração escalável para ambientes em nuvem

Vinicius S. Conceição[1], Nestor D. O. Volpini[1,2], Dorgival Guedes[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG

{vconceicao,nestor,dorgival}@dcc.ufmg.br

[2]Centro Federal de Educação Tecnológica
de Minas Gerais (CEFET-MG) – Divinópolis, MG

nestor@div.cefetmg.br

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. Especialmente para nuvens, a monitoração deve tratar diferentes tipos de recursos virtualizados de comportamento dinâmico, como métricas de desempenho da infraestrutura, logs de aplicações e dados sobre execução em ambientes distribuídos. Este artigo apresenta uma arquitetura de monitoração em nuvens concebida em camadas que é escalável, elástica, responsiva e extensível. A arquitetura foi validada através de uma implementação baseada em ferramentas open-source para atuar sobre um ambiente de processamento de dados massivos, onde foi possível comprovar sua escalabilidade.*

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. This paper presents a layered cloud monitoring architecture which is scalable, elastic, responsive and extensible. Such architecture has been tested through an implementation based on open-source tools to monitor a massive data processing environment in which its scalability has been confirmed.*

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. A flexibilidade no provisionamento de recursos favorece o modelo de negócios *pay-per-use*, no qual consumidores são tarifados por demanda. Qualquer que seja sua finalidade e o nível do serviço ofertado, nuvens se beneficiam do uso de máquinas virtuais (VMs) interconectadas por canais de comunicação de alta velocidade. Seu uso favorece tanto provedores como seus clientes. 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 as aplicações que executam.

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

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 *datacenters* [Kutare et al. 2010]. Com o intuito de monitorar informações, existem diversas abordagens no mercado [Ryder 2016, Massie et al. 2004, VMWare 2018, Prometheus 2018]. 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.

A contribuição deste trabalho é apresentar *Seshat*, uma arquitetura de monitoração elástica, escalável, resiliente e extensível para sistemas em nuvem[1] que oferece flexibilidade na montagem de soluções. Ao identificar 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[2]. Outra contribuição do trabalho foi implementar um sistema de monitoração composto por ferramentas *open source* 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 nosso conhecimento, essa abordagem ainda não foi considerada.

2. 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.

2.1. Demandas da arquitetura

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

[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`)

[2] (*Openstack*, `https://www.openstack.org/`)

de sistemas de monitoração e a forma como são interligados. 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**, **responsividade** e **extensibilidade** [Aceto et al. 2013]. No que se refere a uma **visão micro**, é necessário conhecer a infraestrutura e os sistemas 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. 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 coletados.

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 4.2.

Um sistema de monitoração também deve permitir avaliar periodicamente o estado do sistema computacional e facilitar o rastreio 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].

2.2. Métricas e *logs*

Em sistemas de monitoração para 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 em 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 suas naturezas diversas, todo sistema de monitoração em nuvem deve reconhecer as diferenças entre essas duas categorias e estar preparado para tratá-las de forma

diferenciada. Métricas, pela 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 podem exigir mecanismos de detecção de padrões complexos para extrair as diversas informações que podem estar contidas em uma única mensagem.

3. A arquitetura de monitoração *Seshat*

Para simplificar o processo de identificação das diversas atividades envolvidas no processo de monitoração em um ambiente em nuvem, *Seshat* foi organizada com base em quatro camadas principais: (i) Coleta, (ii) Transporte, (iii) Armazenamento e (iv) Visualização de dados. Além dessas, duas camadas opcionais oferecem funcionalidades extras: (v) Alertas e (vi) Correlação de Eventos. Essa organização é ilustrada na Figura. 1.

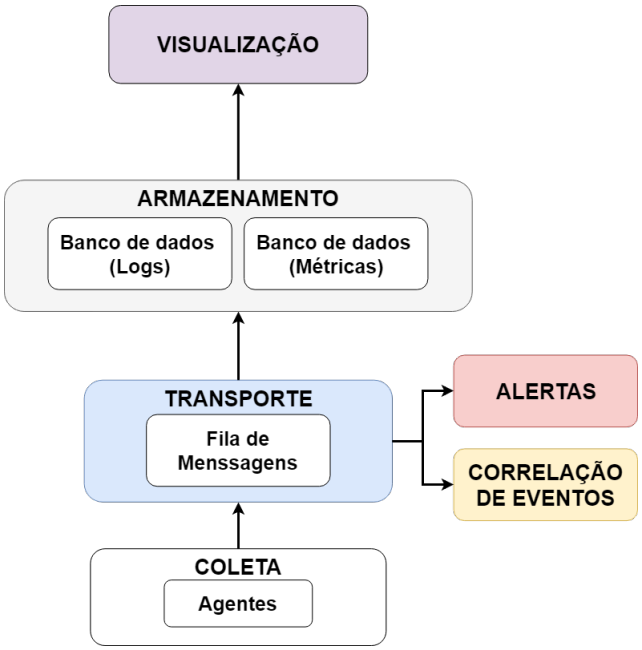

Figura 1. Elementos da arquitetura de monitoração de nuvem *Seshat*.

O fluxo de dados se inicia na camada de **Coleta**, que faz a aquisição dos dados a serem monitorados, estabelecendo o primeiro contato entre a monitoração e o serviço ou *host* alvo. Nessa camada, agentes coletores de dados são inseridos em *hosts* e aplicações para que suas informações sejam enviadas para a camada de **Transporte**. Nesta, são reunidas as informações de todos os agentes coletores através de mensagens. Em seguida, essas mensagens são disponibilizadas para serem processadas. Mensagens brutas adquiridas dessa forma passam por transformações como filtragens e agregações para se adequarem ao formato desejado. Logo depois todas as mensagens são entregues para a camada de **Armazenamento**, onde são inseridas em um bancos de dados adequado ao tipo de informação (métrica ou *log*). 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 certas combinações de valores sejam encontradas nos dados monitorados. Paralelamente, a camada de **Correlação de Eventos** manipula todas as informações

recebidas em busca de padrões mais complexos, a fim de produzir conhecimento mais elaborado sobre a condição do sistema. As próximas seções detalham o funcionamento de cada camada.

3.1. Coleta

A camada de Coleta faz a interface entre o sistema e as entidades monitoradas. Assim, para monitorar um determinado recurso é necessário instalar um agente adequado. A aquisição de dados nesse nível pode se dar de duas formas básicas: *push* (o coletor obtém e envia os dados monitorados de forma autônoma ou pró-ativa) ou *pull* (a infraestrutura de coleta aciona o coletor para cada dado que deseja obter). *Seshat* adota o modelo *push*, pois ele distribui a tarefa de coleta, reduzindo a ocorrência de gargalos. Cada agente coleta os dados sob sua responsabilidade e os envia para o restante do sistema sem a necessidade de uma entidade central realizar consultas sobre agentes. Assim, a coleta de dados apresenta escalabilidade horizontal, diferindo da escalabilidade vertical de sistemas de monitoração de propósito geral. Os agentes precisam ser 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.

Quando um agente coleta métricas, ele executa verificações (*checks*) continuamente que por sua vez produzem eventos (ou simplesmente mensagens). Um evento é um bloco de dados que possui *timestamp*, informações a respeito do host de origem, versão do agente e o resultado de cada *check* executado. Um *check* pode verificar variadas métricas como: uso de CPU, ocupação do disco rígido ou uso da interface de rede. O resultado de um *check* pode ser em formato binário ou, na maioria das vezes, um valor numérico.

Quando um agente coleta *logs*, ele é configurado para observar mudanças em arquivos de *logs* de aplicações ou do 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.

3.2. Transporte

Devido à natureza imprevisível do fluxo de dados de eventos e *logs* de aplicações, as rajadas de dados (*bursts*) são comuns em sistemas de grande porte e podem causar perdas de dados se tratadas incorretamente [Armbrust et al. 2010]. Assim, para evitar gargalos de recepção na camada de Armazenamento, filas de mensagens (também chamadas de *brokers*) são introduzidas para o armazenamento temporário a fim de garantir o transporte de informações cruciais aos banco de dados. No entanto, *brokers* normalmente não possuem mecanismos configurados para o envio de informações aos demais elementos da arquitetura, exigindo a inclusão de dois novos componentes: o *Shipper* e o *Indexer*.

3.2.1. Fila de mensagens

Devido à sua alta taxa de produção, é inviável que os dados sejam enviados diretamente para a camada de armazenamento. Em geral, a taxa de ingestão de bancos de dados tende a ser menor que a taxa de produção de dados pelos agentes em momentos de pico. Assim, diversas mensagens seriam perdidas no processo de envio, e ocasionalmente, ocorreria

degradação de desempenho. Dessa forma, filas de mensagens atuam como um intermediário (*buffer*) para garantir que os dados coletados cheguem eventualmente à camada de armazenamento.

Filas de mensagem funcionam segundo o paradigma *Publish/Subscribe*. Produtores publicam suas informações em filas e consumidores se inscrevem nessas filas para receberem informações [Eugster et al. 2003]. Em geral, as filas são divididas em tópicos de acordo com o assunto das mensagens. A separação entre *logs* de sistema e métricas é um exemplo de divisão de tópicos para o caso da monitoração. Vale citar que as interações entre produtores e consumidores através da fila de mensagens evita que seja preciso bloquear um produtor ou consumidor. Desta forma, a velocidade com que cada produtor ou consumidor gera ou consome as mensagens não tem relaçao direta, resolvendo o problema da disparidade 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 de *stress* onde ocorre o uso critico de memória, a fila é capaz de armazenar parte dos dados no disco rígido, oferecendo ainda mais disponibilidade na recepção de mensagens.

3.2.2. Shipper e Indexer

Uma vez adicionada ao sistema, a fila de mensagens é incapaz de buscar informações dos componentes da camada de coleta e também de enviá-las diretamente à camada de Armazenamento. Cada um desses sistemas pode ter suas peculiaridades, que não necessariamente se adaptam ao modelo da fila. Dessa forma, as informações devem passar por dois componentes que fazem a adaptação entre esses sistemas, como ilustrado na Figura. 2.

O primeiro componente é o *shipper*, responsável pela recepção de dados obtidos por agentes em variados 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 ponte na comunicação entre os agentes e a fila. É importante observar que existem casos onde os agentes coletores já possuem *plugins* de comunicação com a fila, tornando o uso do *shipper* opcional.

O segundo componente, o *indexer*, é mais complexo e é responsável por consumir as mensagens da fila e enviá-las à camada de Armazenamento. Além disso, realiza operações mais custosas como filtragem e *parsing* de mensagens. De maneira análoga ao *shipper*, componentes da camada de Armazenamento também podem ter *plugins* para acessar diretamente a fila. No entanto, para esses casos, o *indexer* ainda é necessário para tratar dados brutos.

3.3. Armazenamento

Estabelecidos a coleta e o transporte, é necessário registrar/armazenar os dados. Em vista disso, a camada de Armazenamento guarda de forma estruturada as informações recebidas das camada inferiores e permite a criação de um histórico para análises mais detalhadas. Para se adequar ao volume de informações e as necessidades de escalabilidade, *Seshat* prevê o emprego de bancos de dados especializado para séries temporais.

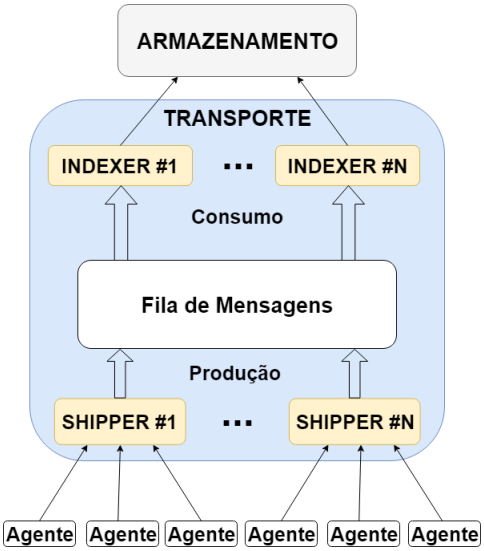

Figura 2. Estrutura interna da camada de Transporte

Independente do tipo de dado considerado, um sistema de monitoração deve ser capaz de armazenar todos os dados recebidos, atender a consultas eventuais sobre os valores armazenados e apenas raramente executar operações de exclusão de dados. Por sua natureza, dados resultantes de monitoração não devem ser alterados. Por isso, há elevada demanda por escrita, eventual leitura e muito raramente exclusões. Esse comportamento define a necessidade de uso de bancos de dados específico para séries temporais. Adicionalmente, sistemas de monitoração de nuvem também possuem grande demanda por armazenamento escalável e robusto [Goldschmidt et al. 2014]. Bancos de dados relacionais padrão são melhor executados em um único servidor a fim de manter a integridade dos mapeamentos de tabela e evitar a complexidade envolvida no computação distribuída, apresentando baixa elasticidade. Portanto, um banco de dados do tipo *NoSQL* é mais adequado por sua facilidade em se adaptar a variações da carga de trabalho através de escalabilidade horizontal.

Apesar de tanto métricas quanto *logs* poderem ser considerados séries temporais, existem peculiaridades no tratamento de cada um. Por exemplo: (i) após o processo de *parsing* de *logs* podem possuir vários campos indexáveis, enquanto métricas possuem apenas um ou dois campos indexáveis além do *timestamp*; (ii) em geral, *logs* possuem diversos caracteres em cada campo, enquanto métricas possuem apenas um único valor numérico; (iii) consultas relacionadas a cadeias de caracteres são bem distintas de consultas relacionadas a valores numéricos e, potencialmente, mais complexas. Considerando todas essas particularidades, não existe necessariamente um componente de armazenamento capaz de endereçar de maneira eficiente e simultânea todas essas questões. Por isso, podem ser necessários bancos de dados distintos para *logs* e métricas.

3.4. Visualização

A camada de visualização permite que os dados armazenados sejam apresentados de forma gráfica. Esse é o ponto da arquitetura onde *dashboards* interativos e técnicas estatísticas podem ser usados para avaliação das informações e acompanhamento contínuo do estado sistema. Dessa maneira, o administrador/usuário é capaz de receber notificações e tomar decisões. Dada a 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. Como será explicado na Seção 4, neste trabalho foram utilizadas ferramentas distintas para apresentação de métricas e *logs* em função de suas diferenças.

3.5. Alertas

Alertas têm a função de chamar a atenção do administrador a respeito de algum evento inesperado. Por exemplo, se a utilização de CPU de um *host* monitorado alcançar 99% quando na verdade não deveria passar dos 80%, o sistema deve reportar a ocorrência incomum ao administrador. Dessa forma, esta camada realiza suas funções através da configuração de *alarmes* e envio de *notificações*. Alarmes são checagens pré-estabelecidas pelo administrador, por exemplo, para métricas que ultrapassem um dado limiar, ou algum recurso que falhe. A partir desses eventos, notificações são disparadas sob forma de e-mails, mensagens SMS ou qualquer tipo de comunicação que informe o administrador sobre o contexto do sistema.

3.6. Correlação de eventos

Ainda derivando da camada de transporte, muita "inteligência" pode ser obtida a partir do uso de uma camada de correlação de eventos. A correlação de eventos (*Event Correlation*) serve como um componente de manipulação de *streams* de eventos, capaz de agregar informações, criar filtros e produzir conhecimento mais elaborado sobre a situação do sistema, permitindo um melhor tratamento da informação durante sua coleta.

4. Aplicação da arquitetura a um caso real

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 na Seção 2 e os detalhes da arquitetura descritos na Seção 3. Para a camada de **coleta**, consideramos ferramentas de coleta que operam no modelo *push*, por serem mais adequadas para garantir escalabilidade. Usamos agentes *Sensu*[3] como a ferramenta básica para extração de métricas, servidor *Sensu* como um ambiente integrado mais flexível para configuração de agentes de coleta, e quando possível, *Beaver*[4] como extrator de *logs*, responsável por ler os arquivos de *logs* gerados pelos diversos sistemas e encaminhar as novas mensagens encontradas para a camada de transporte. Para a camada de **Transporte** utilizamos *RabbitMQ*[5] como a fila de mensagens e *Logstash*[6] para implementar os *shippers* e *indexers* para o processamento de *logs*. Na camada de **armazenamento**, utilizamos *Influxdb*[7] para o armazenamento de métricas e *ElasticSearch*[8] para o armazenamento de *logs* e também métricas. As ferramentas de **visualização** utilizadas foram *Kibana*[9] (*logs*), *Grafana*[10] (métricas em geral) e *Uchiwa*[11] (acompanhamento dos agentes *Sensu*). *Riemann*[12] foi

[3](*Sensu*, `https://sensu.io/`)
[4](*Beaver*, `https://python-beaver.readthedocs.io/en/latest/`)
[5](*RabbitMQ*, `https://www.rabbitmq.com/`)
[6](*Logstash*, `https://www.elastic.co/products/logstash`)
[7](*Influxdb*, `https://www.influxdata.com/time-series-platform/influxdb/`)
[8](*ElasticSearch*, `https://www.elastic.co/products/elasticsearch`)
[9](*Kibana*, `https://www.elastic.co/products/kibana`)
[10](*Grafana*, `https://grafana.com/`)
[11](*Uchiwa*, `https://uchiwa.io/#/`)
[12](*Riemann*, `http://riemann.io/`)

utilizada como a ferramenta de correlação de eventos. Para nossos testes, não foi necessário utilizar uma ferramenta para geração de alarmes, apenas os recursos de geração de alarmes das ferramentas de visualização utilizadas já foram suficientes.

Para a implantação (*deployment*) do sistema, os serviços foram disparados em contêineres *Docker*[13] e lançados como serviços em um *cluster Docker Swarm*[14]. Para alguns serviços foram utilizadas imagens *Docker* disponibilizadas pelas próprias empresas desenvolvedoras, enquanto outras precisaram ser "transformadas"em contêineres no formato *Docker*. É importante ressaltar que, mesmo quando disponíveis em imagens pré-construídas, existe um considerável esforço para configurar e compatibilizar formatos de entrada e saída de dados esperados por cada serviço. Além disso, uma série de redes inter-contêineres foram configuradas para orquestrar a comunicação entre os serviços. Adicionalmente, foram incluídas configurações para permitir que os contêineres que fornecem esses serviços sejam disparados e configurados automaticamente entre os servidores dedicados à monitoração através da plataforma *Swarm*. Dessa forma, o sistema se ajusta a um aumento de carga. Como produto final, foi desenvolvida uma receita em *Docker Swarm* onde todas as configurações e alocações de serviços são instanciadas de forma automática e distribuída. Podemos analogamente afirmar que, assim como um sistema é composto por vários programas, *Seshat* é composto por diversas ferramentas e a integração entre elas é determinada pela arquitetura proposta.

4.1. 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 cada máquina virtual do *cluster*, bem como as máquinas físicas que as hospedam e os dados de consumo de energia de cada *host*, obtidos de um PDU (*Power Distribution Unit*) inteligente. A Figura 3 apresenta um dos *dashboards*[15] 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.

Ao longo da nossa experiência com o sistema, diversos foram os casos onde a adoção da arquitetura simplificaram 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 para acompanhar a carga das máquinas virtuais. Verificamos então que naquelas execuções, a aplicação era suspensa temporariamente. Buscando nos *logs* armazenados os eventos que ocorreram no momento da suspensão, foi possível observar que o problema estava associado ao processo de coleta

[13](*Docker*, `https://www.docker.com/what-docker`)

[14](*Docker Swarm*, `https://docs.docker.com/engine/swarm/key-concepts/`)

[15]Os *dashboards* foram gerados diretamente pela ferramenta e seguem sua identidade visual

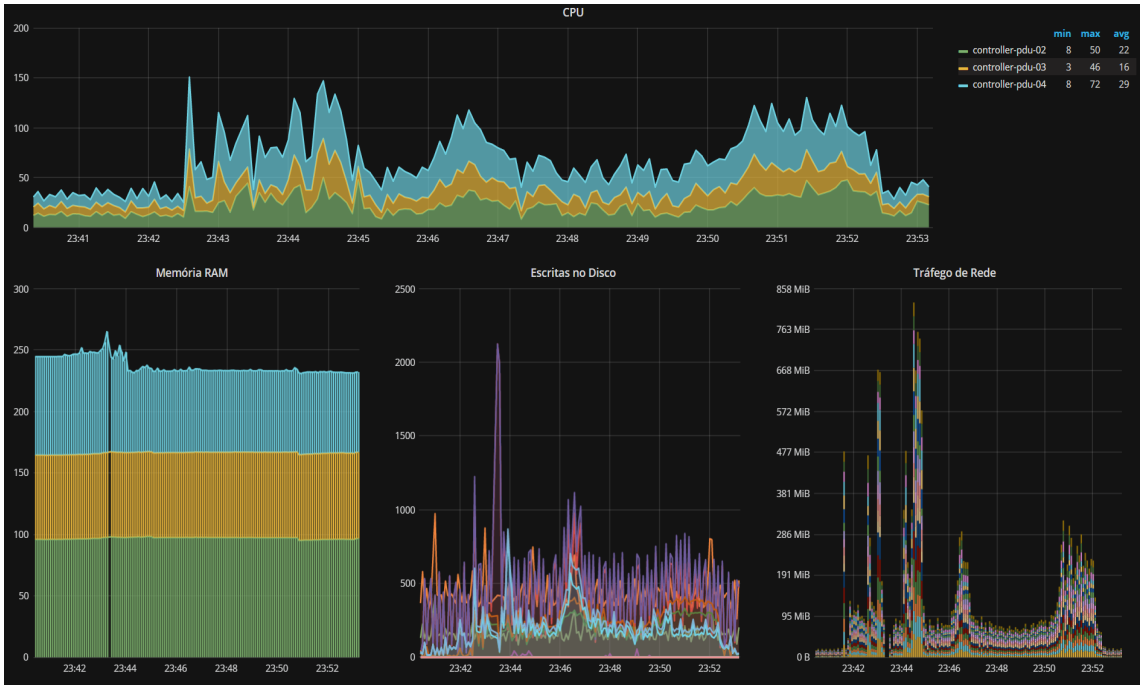

Figura 3. *Dashboard* de monitoração acompanhando um grupo de hosts, onde pode-se acompanhar a utilização de CPU, consumo de memória, escritas em disco e tráfego de rede enquanto uma aplicação Spark é executada em um *cluster* virtualizado.

de lixo das JVMs envolvidas. Uma reconfiguração da memória e outros elementos das JVMs resolveu o problema.

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, verificarmos que a versão de código aberto do *Influxdb* não inclui os recursos para operação distribuída. Depois de alguma análise, decidimos adotar um outro *Elastic Search* com pequenas otimizações 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.

4.2. Desempenho e escalabilidade

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. Foram testadas 2 configurações: (i) todo o sistema executando em um servidor único com processador core-i7 2600 3,4GHz com 16 GB de memória RAM e (ii) executando de forma distribuída em 3 servidores idênticos ao anterior dedicados exclusivamente à monitoração. Para um servidor único, havia apenas uma instância de cada componente do sistema(*shipper*, *indexer*, fila de mensagem, etc), enquanto o distribuído contou com instâncias da fila de mensagens e de bancos de dados em cada uma de 3 máquinas disponíveis. Além disso, foram utilizadas 9 instâncias de *indexers*, também distribuídos entre as 3 máquinas, para consumir da fila. Para os testes foram coletadas 7 tipos de métricas diferentes de cada uma das 24 máquinas virtualizadas na nuvem cliente, composta por 6 máquinas servidoras Intel Xeon E5-2620v4 2.1GHz com 32 GB de RAM. A orquestração da nuvem cliente foi feita pelo Openstack v2.3.1 em outro servidor, também com processador core-i7 2600 3,4GHz com 16 GB de memória RAM. Toda essa estrutura se encontra interligada por um *switch gigabit* 3Com Baseline 2928-SFP Plus *ethernet*. 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. A Tabela 1 apresenta os resultados obtidos nesse teste com a média das 5 execuções.

Tabela 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 1 apresenta a comparação entre os intervalos de medição em cada configuração de servidor, a quantidade média de mensagens perdidas $\bar{x}$ junto ao comprimento de seu intervalo 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 menagens perdidas.

Podemos observar que até nas configurações com maior volume de mensagens por unidade de tempo, as perdas foram baixas, mesmo no ambiente com um único 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.

O segundo experimento avaliou o comportamento do sistema sob stress quando submetido a um elevado volume de mensagens de *log*. De maneira similar ao primeiro foram utilizadas as mesmas configurações de servidor único e distribuído, com exceção dos *indexers*, que foram aumentados para 21 instâncias no caso distribuído. Como carga, usamos uma aplicação Spark sobre HDFS executando em 10 máquinas virtuais da nuvem cliente. Também variamos a verbosidade do sistema de *logs* do Spark entre INFO e

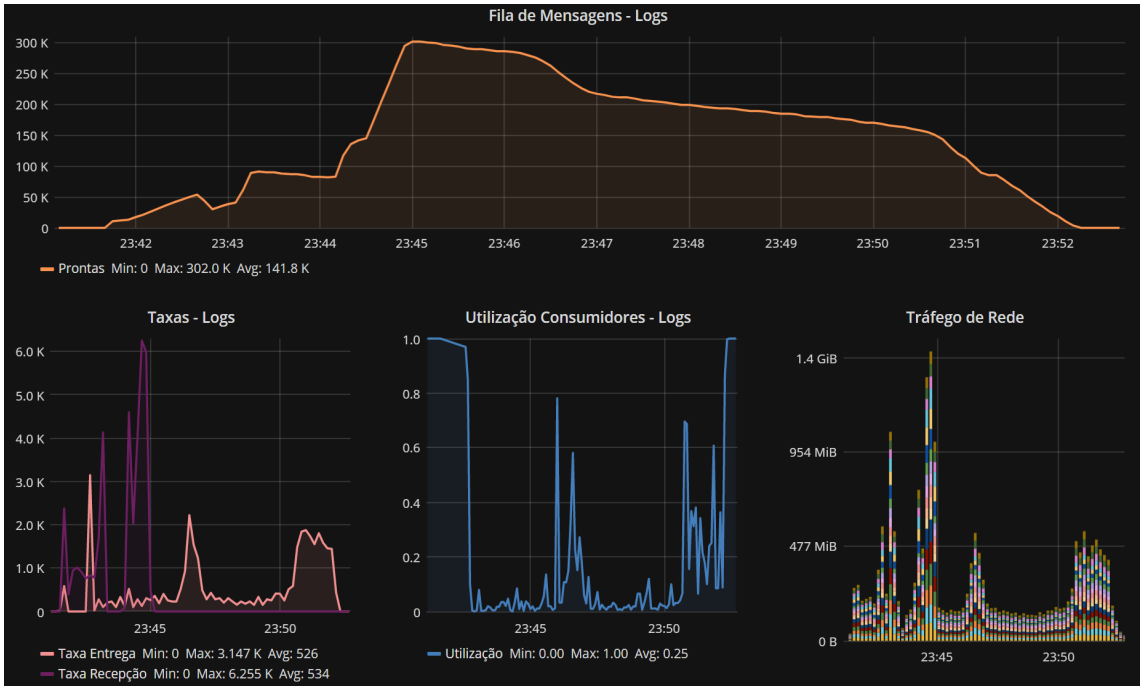

Figura 4. *Dashboard* de monitoração da fila de mensagens durante o experimento sobre a escalabilidade no sistema de processamento de *logs*.

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. O *dasboard* da Figura 4 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 em que as mensagens estão sendo consumidas (gráfico de linha azul, no centro) e o trafego de rede agregado dos 3 servidores da monitoração (gráfico de barras coloridas à direita).

A Tabela 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 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 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 a

taxa de recepção não há diferença significativa entre as configurações de servidor único e distribuído. No entanto, quando comparados entre a verbosidade a diferença chega a aproximadamente o triplo. Isso se deve principalmente ao volume de mensagens enviadas por cada agente de coleta.

5. Trabalhos Relacionados

Monitoração é o processo pelo qual se utiliza ferramentas e medições para o gerenciamento de sistemas de TI [Turnbull 2014], sendo uma área em Tecnologia da Informação quase tão antiga quanto a própria criação de sistemas de computação. Uma contextualização da necessidade de se monitorar sistemas deste tipo foi descrita por [Aceto et al. 2013], onde também apresenta uma série de propriedades chave desejáveis em qualquer sistema de monitoração. A monitoração em nuvem definida por [Spring 2011] apresenta desejavelmente sete camadas. No entanto, a descrição de cada uma delas é feita de maneira abstrata, por vezes considerando somente o ponto de vista dos provedores. Adicionalmente, não existe qualquer menção a respeito de uma camada responsável pela visualização das informações.

Do ponto de vista tecnológico, a monitoração de informações em sistemas distribuídos vem sendo tratada de diferentes formas e ferramentas. Nagios [Ryder 2016] é um exemplo já consolidado no mercado que oferece monitoração em larga escala. No entanto, sua arquitetura centralizada e baseada em *polling* não se adequa ao dinamismo na monitoração de recursos virtuais, que são frequentemente utilizados no ambiente de nuvem. Outro problema apresentado em Nagios, que também é recorrente no Hyperic [VMWare 2018] e Prometheus [Prometheus 2018], são as limitações de escalabilidade. Todos os três sistemas foram projetados para funcionar em um único servidor, e 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. Finalmente, Ganglia e Prometheus não realizam a coleta de *logs*, assim como outras ferramentas no mercado que por padrão coletam somente métricas ou somente *logs*.

6. Conclusão e Trabalhos Futuros

Monitoração de ambientes e aplicações é uma demanda crescente na comunidade de computação em nuvem. Neste trabalho apresentamos *Seshat*, uma arquitetura de monitoração concebida em camadas e organizada em componentes, com uma definição clara dos comportamentos e funcionalidades esperados em cada caso. Uma implementação da arquitetura foi realizada com base em ferramentas de código aberto e aplicada a uma nuvem computacional voltada para o processamento de dados massivos. A experiência com o sistema tem gerado diversos casos de uso bem sucedidos, como discutidos no texto.

Experimentos para avaliar a escalabilidade do sistema sob condições de carga elevada mostraram que o sistema se comporta bem, com perdas baixas. Testes com uma versão distribuída do sistema mostraram uma redução nas mensagens perdidas, redução de atrasos e redução da variabilidade. Esses resultados indicam que o sistema tem potencial para escalar horizontalmente com o aumento da carga.

Como trabalhos futuros, pretendemos realizar mais experimentos para variar as configurações do sistema distribuído para melhor compreender o impacto de elementos como a quantidade de *Shippers* e *Indexers*.

Agradecimentos

Este trabalho foi parcialmente financiado por Fapemig, CAPES, CNPq, e pelos projetos MCT/CNPq-InWeb (573871/2008-6), FAPEMIG-PRONEX-MASWeb (APQ-01400-14), H2020-EUBR-2015 EUBra-BIGSEA e H2020-EUBR-2017 Atmosphere.

Referências

Aceto, G., Botta, A., De Donato, W., and Pescapè, A. (2013). Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115.

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.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131.

Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., and Breivold, H. P. (2014). Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 602–609. IEEE.

Kutare, M., Eisenhauer, G., Wang, C., Schwan, K., Talwar, V., and Wolf, M. (2010). Monalytics: online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th international conference on Autonomic computing*, pages 141–150. ACM.

Massie, M. L., Chun, B. N., and Culler, D. E. (2004). The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840.

Park, J., Yu, H., Chung, K., and Lee, E. (2011). Markov chain based monitoring service for fault tolerance in mobile cloud computing. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 520–525. IEEE.

Prometheus (2018). Prometheus overview. https://prometheus.io/docs/introduction/overview/. Acessado em Abril de 2018.

Romano, L., De Mari, D., Jerzak, Z., and Fetzer, C. (2011). A novel approach to qos monitoring in the cloud. In *Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, pages 45–51. IEEE.

Ryder, T. (2016). *Nagios core administration cookbook*. Packt Publishing Ltd.

Spring, J. (2011). Monitoring cloud computing by layer, part 1. *IEEE Security & Privacy*, 9(2):66–68.

Turnbull, J. (2014). *The Art of Monitoring:*. James Turnbull.

VMWare (2018). Vmware vfabric hyperic, manage application performance across physical, virtual and cloud infrastructures. https://docs.vmware.com/en/vRealize-Hyperic/index.html. Acessado em Abril de 2018.