Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação
Mestrado e Doutorado em Ciência da Computação

MyDBaaS: Um Framework para o Monitoramento de Serviços de Banco de Dados em Nuvem

David Araújo Abreu

DISSERTAÇÃO DE MESTRADO

Fortaleza
Setembro - 2013

Universidade Federal do Ceará
Centro de Ciências
Departamento de Computação

David Araújo Abreu

MyDBaaS: Um Framework para o Monitoramento de Serviços de Banco de Dados em Nuvem

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. José Antônio F. de Macêdo
Co-orientador: Prof. Dr. Flávio R. C. Sousa

Fortaleza
Setembro - 2013

MyDBaaS: Um *Framework* para o Monitoramento de Serviços de Banco de Dados em Nuvem

David Araújo Abreu

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará como requisito para a obtenção do título de Mestre em Ciência da Computação.

Aprovada em: ______ . ______ . ________

____

Prof. Dr. José Antônio Fernandes de Macêdo
Universidade Federal do Ceará
Orientador

____

Prof. Dr. Flávio Rubens de Carvalho Sousa
Universidade Federal do Ceará
Co-Orientador

____

Prof. Dr. José Maria da Silva Monteiro Filho
Universidade Federal do Ceará

____

Prof. Dr. Javam de Castro Machado
Universidade Federal do Ceará

____

Prof. Dr. Ângelo Roncalli Alencar Brayner
Universidade de Fortaleza - UNIFOR

Dedico esta dissertação aos meus pais, namorada, irmã e família por sempre apoiarem meus sonhos e ambições, independente das dificuldades e das indisponibilidades momentâneas da vida, pelo amor, carinho, conforto, dedicação e compreensão ao longo de todos esses anos. Por serem meu porto seguro.

AGRADECIMENTOS

Agradeço primeiramente a Deus por nunca me deixar faltar nada durante esta caminhada. Aos meus pais que tanto amo, Lenilce e Giuseppi Roncalli, por proporcionarem a educação que me permitiu ser a pessoa que sou hoje, por estarem sempre dispostos a me ajudar a ultrapassar as barreiras que a vida impõe, me apoiando em todos os momentos com muito amor e dedicação. A minha irmãzinha Natália, tão importante e especial. A minha amada Camila, pela cumplicidade, amor e carinho que sempre dedicou a mim, estando ao meu lado em todos os momentos desta vitória. Aos meus queridos avós Severino, Maria Eugênia, Nair e José Floriano (em memória), por participarem incondicionalmente em minha vida, com conselhos, carinho, amor e exemplos de vida. A toda minha família e amigos, que independente da distância, dos encontros e desencontros da vida, das diferenças ou afinidades, sempre foram meu porto seguro em todos os momentos que necessitei.

Ao meu professor e orientador José Antônio, pela dedicação e permanente apoio. Aos seus ensinamentos e amizade, de fundamental contribuição no meu crescimento enquanto aluno, pesquisador e pessoa. Agradeço também ao meu co-orientador Flávio Sousa, pelo apoio, ensinamentos e amizade durante toda essa jornada. Agradeço também aos professores da banca, Jośe Maria, Javam Machado e Ângelo Brayner, pela participação e contribuições para o bom resultado deste trabalho. Por fim, aos grandes amigos que fiz e que partilharam desta mesma jornada e aos professores que contribuíram para a construção do meu conhecimento.

Meu muito obrigado!
David Araújo Abreu

Se você quer ser bem sucedido, precisa ter dedicação total, buscar seu último limite e dar o melhor de si.

—AYRTON SENNA

RESUMO

A adoção de serviços em nuvem está aumentando exponencialmente, e uma das razões é porque a sua arquitetura salienta os benefícios de serviços compartilhados e com pagamento baseado no uso. A computação em nuvem possui o foco de proporcionar uma economia em grande escala, possibilitando o acesso a diversos recursos computacionais em tempo real, como serviços de aplicações, infraestrutura e armazenamento, de modo que estes possam ser obtidos de modo dinâmico, elástico, escalável e rápido na medida em que forem consumidos, independente de quem os administra e onde estes recursos estejam alocados. Dentre esses serviços, o gerenciamento e armazenamento de dados são componentes críticos na pilha de software da nuvem, pois a maioria das aplicações são orientadas a dados. Esse serviço, conhecido por *Database as a Service* (DBaaS), nasce como um paradigma de gestão de dados, onde um provedor hospeda e gerencia todo ambiente necessário ao funcionamento dos sistemas de banco de dados e o terceiriza como um serviço para um ou mais consumidores. Porém, ainda há problemas que impedem a sua adoção generalizada dos DBaaS. Fornecer serviços em nuvem requer procedimentos sofisticados de gestão por parte do fornecedor para garantir robustez, desempenho, confiabilidade, segurança, elasticidade e qualidade. Portanto, os consumidores esperam que provedores de DBaaS garantam a qualidade do serviço, e lidem com padrões dinâmicos de carga de trabalho e elasticidade, pois é fundamental para garantir que os acordos de nível de serviço (SLA) sejam atendidos. No entanto, prover mecanismos de elasticidade, escalabilidade, qualidade de serviço e disponibilidade em ambientes em nuvem é um grande desafio. Claramente isto é um desafio também na disponibilização dos DBaaS, e para se alcançar essas funcionalidades e princípios é necessário um monitoramento detalhado e preciso. Com isso, esta dissertação tem por objetivo a proposta de um *framework* open-source para o monitoramento de serviços de DBaaS, denominado MyDBaaS, cuja finalidade é possibilitar a criação de soluções de monitoramento personalizáveis e eficientes através de um modelo de programação abrangente e extensível, que disponibiliza desde a definição das métricas, procedimento de coleta, recebimento e armazenamento até mecanismos para consumo das informações coletadas em tempo real.

Palavras-chave: Computação em Nuvem, *Database as a Service*, *Framework* de Aplicação, Monitoramento.

SUMÁRIO

Capítulo 1—Introdução 1

1.1 Motivação 1

1.2 Objetivo do Trabalho 4

1.3 Contribuições 6

1.4 Estrutura da Dissertação 7

Capítulo 2—Monitoramento de Serviços de Banco de Dados em Nuvem 8

2.1 Computação em Nuvem 8

2.1.1 Características 11

2.1.2 Modelos de Serviço 12

2.1.2.1 *Infrastructure as a Service (IaaS)* 13

2.1.2.2 *Platform as a Service (PaaS)* 14

2.1.2.3 *Software as a Service (SaaS)* 15

2.2 *Database as a Service (DBaaS)* 17

2.3 Monitoramento de Serviços em Nuvem 20

2.3.1 Soluções para Monitoramento de Serviços em Nuvem 22

2.4 Requisitos para o Monitoramento de Serviços de Banco de Dados em Nuvem 24

2.5 Trabalhos Relacionados 28

2.5.1 Análise Comparativa entre os Trabalhos Relacionados 30
2.6 Conclusão 31

Capítulo 3—MyDBaaS: Um Framework para o Monitoramento de Serviços de Banco de Dados em Nuvem 32

3.1 Framework de Aplicação 32
3.2 MyDBaaS Framework 33
3.2.1 Modelo do Sistema 33
3.2.2 Arquitetura 35
3.2.3 Definição de Métricas e Recursos 37
3.2.4 Serviço de Monitoramento 40
3.2.5 Serviço de Recebimento das Métricas 49
3.2.6 Serviço de Consumo das Métricas 54
3.2.7 Implementação do Framework 58
3.2.8 Instanciando o Framework 59
3.2.9 Análise Comparativa entre MyDBaaS e Trabalhos Relacionados . 60
3.3 Conclusão 63

Capítulo 4—Estudo de Caso: Aplicação MyDBaaS Monitor 64

4.1 MyDBaaS Monitor 64
4.1.1 Métricas Definidas 65
4.1.2 Coletores Implementados 70
4.1.3 *Receivers* Implementados 76
4.1.4 Gerenciamento dos Recursos 78

4.1.5 Configuração do Monitoramento 83
4.1.6 Monitoramento dos Recursos das Camadas 85
4.1.7 Estudo de Caso 91
4.2 Conclusão 94

Capítulo 5—Conclusão 95

5.1 Oportunidades para Trabalhos Futuros 96

Apêndice A—Classes de Definição das Métricas 103

A.1 Camada Física 103
A.2 Camada Virtual 106
A.3 Camada de Dados 112

Apêndice B—Classes para Definição dos Coletores 121

B.1 Camada Física 121
B.2 Camada Virtual 124
B.3 Camada de Dados 130
B.4 Camada de Carga de Trabalho 140

Apêndice C—Classes de Definição dos Receivers 141

C.1 Camada Virtual 141
C.2 Camada de Dados 143
C.3 Camada de Carga de Trabalho 145

LISTA DE FIGURAS

2.1 Visão geral do acesso a Nuvem Computacional 9

2.2 As principais camadas da Computação em Nuvem 13

2.3 Ilustração do ambiente de um *Database as a Service* (DBaaS). 19

2.4 Ilustração do monitoramento nas diversas camadas de um DBaaS. 25

3.1 Ilustração do ambiente com o modelo de monitoramento. 34

3.2 Arquitetura do *framework* MyDBaaS. 35

3.3 Detalhamento da arquitetura do *framework* MyDBaaS. 36

3.4 Diagrama de classes do módulo *MyDBaaS Common*. 38

3.5 Exemplos de definições de métricas no módulo *MyDBaaS Common* . . . 39

3.6 Diagrama de classes do módulo *MyDBaaS Agent*. 40

3.7 Exemplo da implementação da interface *LoadMetric*. 42

3.8 Exemplo da extensão da classe abstrata *AbstractCollector*. 42

3.9 Exemplo abstrato da implementação do método *run* de um coletor. . . . 43

3.10 Primeiros parâmetros do arquivo de contexto. 43

3.11 Exemplos de parâmetros do arquivo de contexto para os tipos de métricas. 44

3.12 Diagrama de atividade do ciclo de monitoramento de um coletor. 45

3.13 Diagrama de sequência de inicialização do agente de monitoramento. . . 46

3.14 Diagrama de classes do módulo *MyDBaaS Driver*. 47

3.15 Exemplo da extensão da classe abstrata *AbstractWorkloadCollector*. . . . 48

3.16 Diagrama de atividade do fluxo de recebimento de uma métrica. 49

3.17 Diagrama de classe do componente *Controller* para *Receiver*. 50

3.18 Exemplo da extensão da classe abstrata *AbstractReceiver*. 50

3.19 Exemplos da implementação dos métodos de recebimento das métricas. . 51

3.20 Diagrama de classes do componente *Repository*. 52

3.21 Estrutura inicial da base serial histórica. 53

3.22 Diagrama de atividade do fluxo de registro de uma coleta. 54

3.23 Diagrama de classes do módulo *MyDBaaS API*. 55

3.24 Exemplos da instanciação da classe *MyDBaaSClient*. 56

3.25 Exemplos da instânciação da classe *MyMetrics*. 57

3.26 Diagrama de classes do componente *Controller* para *API Receiver*. 58

4.1 Implementação da métrica *WorkloadStatus*. 70

4.2 Implementação do coletor *InformationTableCollector*. 75

4.3 Implementação do *receiver* para métricas de servidor. 77

4.4 Tela para cadastro de um novo DBaaS no ambiente de monitoramento. . 78

4.5 Tela para cadastro de um novo Servidor no ambiente de monitoramento. 79

4.6 Tela para cadastro de uma nova Máquina Virtual no ambiente de monitoramento. 80

4.7 Tela para cadastro de um novo SGBD no ambiente de monitoramento. . 81

4.8 Tela para cadastro de uma nova Instância de Banco de Dados no ambiente de monitoramento. 81

4.9 Tela para visualização dos DBaaS cadastrados no ambiente de monitoramento da aplicação. 82

4.10 Tela para visualização em detalhe de um DBaaS cadastrado. 82

4.11 Tela de configuração do agente de monitoramento para Servidor. 83

4.12 Tela de configuração do agente de monitoramento para Máquina Virtual. 84

4.13 Painel de acompanhamento do monitoramento para servidor. 86

4.14 Painel de acompanhamento do monitoramento para máquina virtual. . . 87

4.15 Painel de acompanhamento do monitoramento para SGBD (Parte 1). . . 88

4.16 Painel de acompanhamento do monitoramento para SGBD (Parte 2). . . 89

4.17 Painel de acompanhamento do monitoramento para instância de banco de dados/carga de trabalho. 90

4.18 Métrica coletada sobre a camada de carga de trabalho. 91

4.19 Métrica coletada sobre a camada física. 92

4.20 Métrica coletada sobre a camada virtual. 92

4.21 Métrica coletada sobre a camada de dados. 93

4.22 Exemplo do armazenamento da métrica *DiskUtilization* na base serial histórica. 94

A.1 Implementação da métrica *HostDomains*. 103

A.2 Implementação da métrica *DomainStatus*. 104

A.3 Implementação da métrica *HostInfo*. 105

A.4 Implementação da métrica *Cpu*. 106

A.5 Implementação da métrica *Memory*. 107

A.6 Implementação da métrica *Network*. 108

A.7 Implementação da métrica *Disk*. 109

A.8 Implementação da métrica *Partition*. 110

A.9 Implementação da métrica *Machine*. 111

A.10 Implementação da métrica *ActiveConnection*. 112

A.11 Implementação da métrica *Size*. 112

A.12 Implementação da métrica *NetworkTraffic*. 113

A.13 Implementação da métrica *ProcessStatus*. 113

A.14 Implementação da métrica *InformationData*. 114

A.15 Implementação da métrica *InformationTable*. 115

A.16 Implementação da métrica *StatementDML*. 116

A.17 Implementação da métrica *StatementTCL*. 117

A.18 Implementação da métrica *StatementDDL*. 118

A.19 Implementação da métrica *StatementDCL*. 119

A.20 Implementação da métrica *DiskUtilization*. 120

B.1 Implementação do coletor *HostDomainsCollector*. 121

B.2 Implementação do coletor *DomainStatusCollector*. 122

B.3 Implementação do coletor *HostInfoCollector*. 123

B.4 Implementação do coletor *CpuCollector*. 124

B.5 Implementação do coletor *MemoryCollector*. 125

B.6 Implementação do coletor *NetworkCollector*. 126

B.7 Implementação do coletor *DiskCollector*. 127

B.8 Implementação do coletor *PartitionCollector*. 128

B.9 Implementação do coletor *MachineCollector*. 129

B.10 Implementação do coletor *ActiveConnectionCollector*. 130

B.11 Implementação do coletor *SizeCollector*. 131

B.12 Implementação do coletor *NetworkTrafficCollector*. 132

B.13 Implementação do coletor *ProcessStatusCollector*. 133

B.14 Implementação do coletor *InformationDataCollector*. 134

B.15 Implementação do coletor *StatementDMLCollector*. 135

B.16 Implementação do coletor *StatementTCLCollector*. 136

B.17 Implementação do coletor *StatementDDLCollector*. 137

B.18 Implementação do coletor *StatementDCLCollector*. 138

B.19 Implementação do coletor *DiskUtilizationCollector*. 139

B.20 Implementação do coletor *WorkloadStatusCollector*. 140

C.1 Implementação do *receiver* para métricas de máquina virtual (Parte 1). . 141

C.2 Implementação do *receiver* para métricas de máquina virtual (Parte 2). . 142

C.3 Implementação do *receiver* para métricas de SGBD/instância de banco de dados (Parte 1). 143

C.4 Implementação do *receiver* para métricas de SGBD/instância de banco de dados (Parte 2). 144

C.5 Implementação do *receiver* para métricas de carga de trabalho. 145

LISTA DE TABELAS

2.1 Requisitos para o monitoramento de serviços de DBaaS. 27

2.2 Análise comparativa dos trabalhos relacionados e requisitos. 30

3.1 Análise comparativa do MyDBaaS e trabalhos relacionados. 62

4.1 Métricas definidas para a camada de servidores. 65

4.2 Métricas definidas para a camada de máquinas virtuais. 67

4.3 Métricas definidas para SGBDs/instâncias de banco de dados. 69

4.4 Métrica definida para a camada de carga de trabalho. 69

4.5 Coletores definidos para a camada de servidores. 71

4.6 Coletores definidos para a camada de máquinas virtuais. 72

4.7 Coletores definidos para a camada de SGBDs/instâncias de banco de dados. 74

4.8 Coletor definido para a camada de carga de trabalho. 74

CAPÍTULO 1

INTRODUÇÃO

Neste capítulo serão apresentadas a justificativa e a motivação para o desenvolvimento deste trabalho, uma definição do problema tratado, assim como as contribuições que se pretende alcançar. Ao final do capítulo, será descrito como está organizada o restante desta dissertação.

1.1 MOTIVAÇÃO

Hoje estamos no meio de uma transformação, o que aconteceu à geração da energia elétrica, há um século, agora está acontecendo com o processamento de informações. A forma como as pessoas utilizam computadores está mudando. Em vez dos dados e aplicativos estarem contidos nos dispositivos de armazenamento dos usuários, eles passam a ser armazenados em inúmeros servidores, e os usuários terão acesso a esse conteúdo através da internet, tornando a utilização dos computadores um serviço, com os benefícios da redução dos custos para os consumidores, dados mais seguros, e sistemas mais atualizados através de um modelo de pagamento baseado no uso [1].

Este serviço de utilidade que tem sido aplicado no contexto da Tecnologia da Informação (TI), gerando essa mudança de paradigma é conhecido como Computação em Nuvem. Essa tendência atual de tecnologia cuja finalidade é fornecer serviços de TI sob demanda, pretende ser global e proporcionar serviços para todos os nichos do mercado, desde usuários residenciais até empresas que terceirizaram todo o seu parque tecnológico.

Uma das razões para o sucesso da computação em nuvem é o papel que tem desempenhado na eliminação do tamanho de uma empresa como um fator crítico para seu sucesso econômico [2]. Alguns exemplos dessa mudança são os data centers que oferecem aos clientes a infraestrutura física necessária para hospedar seus sistemas de informação, incluindo fontes de alimentação redundantes, armazenamento, capacidades

de comunicação de alta largura de banda, monitoramento do ambiente e serviços de segurança. Isso elimina a necessidade das empresas de fazer uma grande despesa de capital na construção de uma infraestrutura para possibilitar a disseminação dos seus produtos em escala global. Segundo Vaquero et al. [3] esse modelo tem sido eficaz, uma vez que permite a uma empresa de qualquer tamanho gerenciar o crescimento do seu produto ou serviço e, ao mesmo tempo, também permite à empresa reduzir seus custos. Durante os últimos anos temos observado uma rápida aceleração da inovação de novos paradigmas de negócios e a computação em nuvem tem desempenhado um papel muito importante neste processo.

A computação em nuvem tem a promessa de fornecer uma enorme onda de desenvolvimento em uma indústria que está se esforçando para crescer. Essa recente solução possui o foco de proporcionar uma economia em grande escala, possibilitando o acesso a diversos recursos computacionais em tempo real, como serviços de aplicações, dispositivos, informações, infraestrutura e armazenamento através de uma rede unificada denominada nuvem, de modo que estes possam ser obtidos de modo dinâmico, elástico e rápido na medida em que forem consumidos [4].

Esse modelo possibilita o aumento ou a diminuição de forma automática dos recursos computacionais, possibilitando o acesso, de modo conveniente e sob demanda, a esses recursos configuráveis que podem ser rapidamente adquiridos e liberados com mínimo esforço gerencial ou interação com provedor de serviços. Uma maneira bastante eficiente de maximizar e flexibilizar os recursos computacionais [5].

A maneira como os serviços baseados em nuvem são cobrados é diferente da maioria dos modelos atuais, o usuário paga por aquilo que utilizar ou pelo tempo de utilização, gerando uma economia e controle melhor dos gastos. Segundo Armbrust et al. [6] uma consequência muito importante desse modelo de faturamento é a redução dos riscos de subutilização e de saturação. Como no modelo *pay-per-use* o usuário só paga por aquilo que consome, reservando somente o necessário, esses riscos são reduzidos. Assim, a utilização de plataformas computacionais terceirizadas é uma saída inteligente para os usuários lidarem com infraestruturas de TI.

A adoção de serviços de computação em nuvem está aumentando exponencialmente, e uma das razões é porque a sua arquitetura salienta os benefícios de serviços compartilhados sobre os produtos isolados. Este uso de serviços compartilhados permite que o foco da organização esteja em seu negócio principal, e que os departamentos de TI

reduzam a lacuna entre a capacidade computacional disponível e à demanda necessária que os sistemas requisitam. A concordância geral sobre o regime de categorização dos três serviços ofertados pela computação em nuvem são os modelos de [4] [5]: *Software as a Service* (SaaS), *Platform as a Service* (PaaS) e *Infrastructure as a Service* (IaaS). SaaS é um modelo de distribuição de software em que os aplicativos são hospedados por um fornecedor ou prestador de serviço e disponibilizados aos clientes através de uma rede, geralmente a Internet. PaaS é um conceito que descreve uma plataforma de computação que é alugada ou entregue como uma solução integrada ou como uma pilha de soluções através de uma conexão com a Internet. Essa pilha pode ser um conjunto de componentes ou subsistemas de software usados para desenvolver um produto ou serviço totalmente funcional. IaaS é um modelo de fornecimento em que uma organização terceiriza o equipamento utilizado para apoiar as operações, incluindo armazenamento e hardware. O prestador de serviços possui o equipamento e é responsável pela hospedagem, execução e manutenção.

Como o conceito da computação em nuvem consiste em fornecer recursos de TI como um serviço, e dentre esses serviços o gerenciamento e armazenamento de dados são componentes críticos na pilha de softwares da nuvem, pois a maioria das aplicações são orientadas a dados [7]. O modelo, conhecido por *Database as a Service* (DBaaS), surge como um novo paradigma de gestão de dados, onde um provedor hospeda e gerencia todo ambiente necessário ao funcionamento dos sistemas de banco de dados e o terceiriza como um serviço para um ou mais consumidores [8].

Os SGBDs são importantes e indispensáveis na maioria dos ambientes de computação. Com o advento da hospedagem e armazenamento na nuvem, a oportunidade de oferecer um SGBD como um serviço terceirizado está ganhando mais força. Segundo [9] um DBaaS oferece uma série de benefícios aos usuários, pois ele dispõe de um ambiente escalável, disponível, rápido e com qualidade de serviço garantida. Por outro lado, o provedor tem que garantir a ilusão de recursos infinitos, sob cargas de trabalho dinâmicas e minimizar os custos operacionais associados a cada usuário [10]. De acordo com [4] um DBaaS, na perspectiva do usuário, tem como principal necessidade uma interface simples que não necessite de ajustes ou administração. Trata-se de uma melhoria em relação às soluções tradicionais que requerem técnicas para provisionar recursos, seleção de SGBDs em nuvem, instalação, configuração e administração. O usuário visa um desempenho satisfatório, expresso em termos de latência e vazão, independente do tamanho da base de dados e das alterações da carga de trabalho. Da perspectiva do provedor, é necessário

atender aos acordos de SLA, apesar da quantidade de dados e alterações na carga de trabalho. O sistema deve ser eficiente na utilização dos recursos, possuir alta disponibilidade e segurança.

Enquanto a infraestrutura de nuvem proporciona um aumento significativo de vantagens e eficiência, ainda há problemas que impedem a sua adoção generalizada [11]. Fornecer serviços de computação em nuvem requer procedimentos sofisticados de gestão por parte do fornecedor para garantir robustez, desempenho, confiabilidade, segurança, elasticidade e qualidade. Portanto, os consumidores esperam que provedores de serviços em nuvem garantam a qualidade do serviço, e lidem com padrões dinâmicos de carga de trabalho e elasticidade, pois é fundamental para garantir que os acordos de nível de serviço (SLA) dos clientes sejam atendidos [7]. Portanto, assim como a computação em nuvem, os DBaaS estão ganhando espaço devido a muitas das mesmas razões que a computação em nuvem. No entanto, prover mecanismos de elasticidade, escalabilidade, qualidade de serviço e disponibilidade em ambientes em nuvem, proporcionando aos usuários a percepção de recursos quase infinitos, é um grande desafio. Claramente isto é um desafio também na disponibilização dos DBaaS, e para se alcançar essas funcionalidades e princípios é necessário um monitoramento detalhado e preciso de todas as camadas e recursos que compõem um DBaaS.

1.2 OBJETIVO DO TRABALHO

O monitoramento de recursos em ambientes de nuvem é fundamental para medir e avaliar continuamente os comportamentos da infraestrutura ou aplicações em termos de performance, confiabilidade, consumo de energia, capacidade de atender SLAs, segurança, etc [12]. Por um lado, é uma necessidade fundamental para o controle e gerenciamento das infraestruturas de hardware e software, por outro lado, fornece informações e indicadores sobre o desempenho da utilização dos serviços. O monitoramento é fundamental para as atividades relacionadas com os objetivos principais de um provedor, portanto, não possuir as informações necessárias em momentos de decisão anularia a utilidade de um serviço, principalmente em um ambiente de nuvem onde o consumidor espera que o mesmo reaja de acordo com as suas necessidades e atenda suas expectativas.

Segundo [13], o monitoramento deve existir nas diversas atividades comuns ao gerenciamento dos serviços em nuvem, tais como capacidade e planejamento de recursos, gerenciamento do data center, gestão de SLAs, faturamento, solução de problemas e

gerenciamento de desempenho. Porém os DBaaS incluem duas novas atividades onde é necessário a existência do monitoramento também: o gerenciamento da camada de dados e o gerenciamento da carga de trabalho. Portanto, o monitoramento deve sempre existir em serviços em nuvem e como relatado, o monitoramento contínuo é necessário sobre as diversas camadas que compõem os serviços, pois abastece os provedores e consumidores com informações estratégicas sobre o real estado do serviço em diversas granularidades. Para o DBaaS não é diferente e o monitoramento deve ser um processo transversal e que abrange todas as camadas de recursos desse modelo de serviço. Esse processo deve ser também horizontal e monitorar toda instância de recurso que exista dentro das camadas.

Atualmente no mercado existem diversas soluções comerciais [14] [15] [16] [17] para monitoramento de serviços em nuvem. Algumas soluções são para o gerenciamento de nuvens e contêm além de outras funcionalidades um módulo especificamente voltado para a atividade de monitoramento. Existem soluções onde a única meta é o monitoramento da nuvem [14] [16], mas como são soluções comerciais e muitas vezes com objetivos específicos não se ajustam totalmente às necessidades de monitoramento que o fornecedor de um serviço em nuvem requer. De fato, cada vez mais a necessidade por um monitoramento personalizado é importante para atender as demandas de qualidade de serviço, e com os serviços de DBaaS não é diferente. Existem atualmente iniciativas que visam prover arcabouços de software (*framework* de aplicação), oferecendo mecanismos de extensibilidade e configurabilidade muito maiores dos que os mecanismos disponíveis em soluções totalmente já implementadas, permitindo criar soluções mais adequadas para diferentes cenários de necessidade.

Na literatura, existem trabalhos [18] [19] [20] [21] que propõem *frameworks* para criação de soluções de monitoramento para serviços em nuvem. Esses trabalhos são importantes pois disponibilizam um arcabouço com estruturas e funcionalidades inicialmente implementadas, as quais aceleram o desenvolvimento das soluções de monitoramento para os serviços em nuvem.

A maior parte desses *frameworks* [18] [20] [21], permite o monitoramento das camadas de recursos físicos e virtuais. Porém, somente o [19] permite parcialmente o monitoramento da camada de dados e da carga de trabalho. No entanto, a definição de métricas customizadas em todas as camadas é uma necessidade fundamental para um processo de monitoramento personalizado. Essa necessidade é parcialmente atendida por [20] e totalmente atendida por [21]. Uma outra necessidade essencial para o monitoramento é permitir configurar o ciclo de monitoramento para cada métrica, em

diferentes granularidades nas diversas camadas de recursos, porém somente o trabalho [18] contempla esta necessidade.

Os ambientes de monitoramento criados através de um framework de aplicação devem permitir que aplicações existentes possam acessar facilmente as métricas coletadas. Neste sentido, o *framework* deve prover uma interface que encapsule a complexidade de comunicação e manipulação das informações monitoradas, de forma a facilitar o acesso ao ambiente de monitoramento pelas aplicações existentes. Até este momento, nenhum trabalho encontrado na literatura contempla tal funcionalidade.

Como é possível verificar na literatura apresentada, as propostas de *framework* existentes não atendem a todas as necessidades uma solução de monitoramento para DBaaS necessita. Desta maneira, este trabalho visa propor um *framework* que atenda a todas as necessidades de monitoramento dos serviços de DBaaS, optando-se por não estender as soluções atuais que não são open-source, diferentemente desta proposta. Assim como, o *framework* proposto neste trabalho visa colaborar com padrões para definição de métricas, procedimentos de coleta e armazenamento das informações monitoradas, que são considerados como questões em aberto pelos autores em [13]. Com base em nossa revisão da literatura, também não há nenhum *framework* com foco no monitoramento para DBaaS que permite a criação de um conjunto de métricas que vão desde infraestrutura física até as cargas de trabalho de bancos de dados.

1.3 CONTRIBUIÇÕES

Dado o objetivo desta dissertação apresentado na seção anterior, as principais contribuições são as seguintes:

1. Proposta de um *framework* de aplicação para instanciação de ferramentas para monitoramento de ambientes de DBaaS.
2. Definição e implementação de um conjunto com 21 coletores de métricas de monitoramento para serviços em nuvem, incluindo camada de dados e cargas de trabalho.
3. Implementação de uma ferramenta baseada no *framework* proposto, para o monitoramento de serviços de banco de dados em nuvem.

4. Definição e implementação de uma API para o consumo das métricas monitoradas, assim como, mecanismos que possibilitam esse consumo de maneira personalizada.

5. Especificação e implementação de uma base de dados histórica para o armazenamento das métricas coletadas nas diversas camadas de recursos.

6. Criação de um driver JDBC para interceptação das cargas de trabalho, possibilitando coletar métricas sobre as consultas enviadas às instâncias de banco de dados.

1.4 ESTRUTURA DA DISSERTAÇÃO

Os próximos capítulos desta dissertação estão estruturados da seguinte forma:

- Capítulo 2: apresenta os principais conceitos ligados ao problema abordado nesse trabalho, apresentando uma contextualização do que é computação em nuvem, *database as a service* (DBaaS) e os requisitos para o monitoramento dos DBaaS.
- Capítulo 3: descreve o MyDBaaS, um *framework* para o monitoramento de ambientes de banco de dados em nuvem cuja finalidade é possibilitar a criação de soluções de monitoramento personalizáveis e eficientes.
- Capítulo 4: apresenta o estudo de caso do MyDBaaS Monitor, uma aplicação web para o gerenciamento do monitoramento de serviços DBaaS cuja finalidade é possibilitar o monitoramento personalizado dos recursos que compõem os serviços.
- Capítulo 5: discute os objetivos alcançados nesta dissertação e as principais dificuldades encontradas durante a sua realização. Também são apresentados alguns tópicos identificados para trabalhos futuros.

CAPÍTULO 2

MONITORAMENTO DE SERVIÇOS DE BANCO DE DADOS EM NUVEM

Este capítulo discute os principais conceitos ligados ao problema abordado nesse trabalho, apresentando uma contextualização do que é computação em nuvem, *database as a service* (DBaaS) e os requisitos para o monitoramento dos DBaaS.

Este capítulo está estruturado da seguinte forma: na seção 2.1 são apresentadas as principais características e modelos de serviço da computação em nuvem, em seguida na seção 2.2 é apresentado o conceito e visão dos DBaaS. Na seção 2.3 são demonstradas as características do monitoramento de serviços em nuvem e em seguida algumas soluções para o monitoramento de serviços em nuvem são apresentadas. Na seção 2.4 são discutidos os requisitos em que uma solução de monitoramento para DBaaS necessita. Por fim, na seção 2.5 são apresentados os trabalhos relacionados e em seguida uma análise comparativa com base nos requisitos levantados na seção anterior é realizada.

2.1 COMPUTAÇÃO EM NUVEM

O termo *Cloud Computing* ou Computação em Nuvem está ganhando maturidade e os grandes nomes da área, como Google, Amazon, Microsoft, IBM, entre outras, estão investindo em ritmo crescente nas pesquisas e serviços que essa tecnologia tem a proporcionar [22]. Porém esse paradigma que promete revolucionar a área da informática, não desperta apenas o interesse desse grupo seleto de empresas, mas também estudantes, especialistas e profissionais da área de TI, assim como pequenas e médias empresas.

O conceito de uma nuvem ainda está mudando mais rápido do que a maioria de nós consegue acompanhar. No entanto, a computação em nuvem tem a promessa de fornecer uma enorme onda de desenvolvimento em uma indústria que está se esforçando para crescer. Essa recente solução possui o foco de proporcionar uma economia em grande escala, possibilitando o acesso a diversos recursos computacionais em tempo real, como

serviços de aplicações, dispositivos, informações, infraestrutura e armazenamento através de uma rede unificada denominada nuvem, de modo que estes possam ser obtidos de modo dinâmico, elástico e rápido na medida em que forem consumidos [4]. Independente de quem os administra e onde estes recursos estejam alocados. A Figura 2.1 ilustra uma nuvem que possui um grande poder e capacidade de processamento, podendo ser acessada a partir de qualquer dispositivo, em qualquer lugar e a qualquer momento, necessitando apenas de uma conexão à Internet.

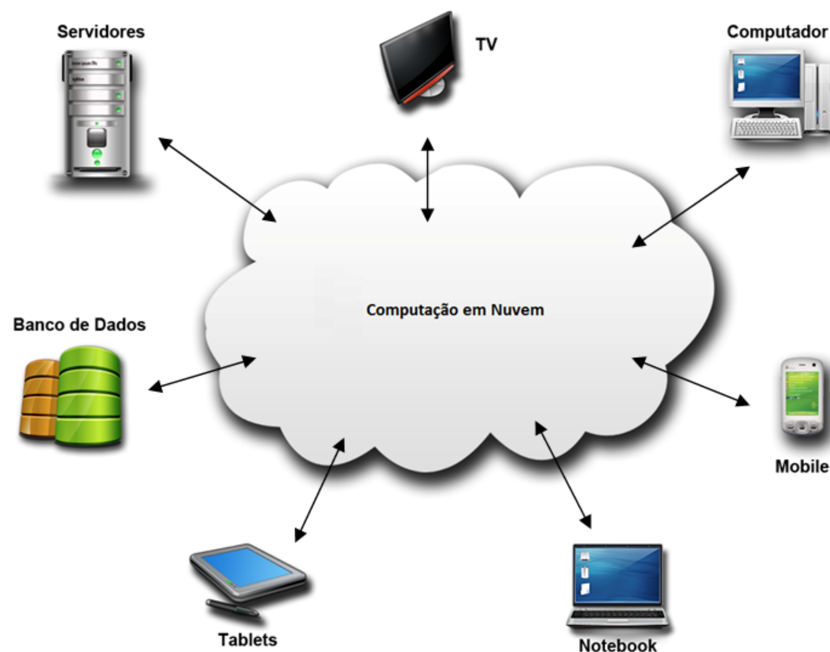


Figura 2.1 Visão geral do acesso a Nuvem Computacional

A nuvem hoje não é mais algo intangível, mas o foco da computação, onde está sendo definido como um novo estilo de computação em que os recursos dinamicamente escaláveis e muitas vezes virtualizados são fornecidos como um serviço através da Internet [23]. Computação em nuvem é uma tendência tecnológica significativa, e há uma grande expectativa na reformulação que está realizando sobre os processos e mercado de TI.

A computação em nuvem possibilita o aumento ou a diminuição de forma automática dos recursos computacionais. Peter Mell e Tim Grance [5] destacam "que esse modelo possibilita acesso, de modo conveniente e sob demanda, a esses recursos configuráveis que podem ser rapidamente adquiridos e liberados com mínimo esforço gerencial ou interação com provedor de serviços. Uma maneira bastante eficiente de maximizar e flexibilizar os recursos computacionais". Além disso, uma nuvem computacional é um

ambiente redundante e resiliente[1] por natureza [11].

O conceito de uma arquitetura que esquematiza a visão de nuvem deve ser vista além de um simples conjunto de servidores que se propõem a hospedar os serviços. Segundo [24], ela deve dispor de uma infraestrutura de gerenciamento que inclua funções como provisionamento de recursos computacionais, balanceamento dinâmico de carga de dados e acesso e monitoração do desempenho. De acordo com [23], nunca uma abordagem para a utilização real foi tão global e completa: não apenas recursos de computação e armazenamento são entregues sob demanda, mas toda a pilha de computação pode ser aproveitada na nuvem.

Com esse paradigma, atualmente os computadores pessoais estão se transformando simplesmente em veículos de acesso à nuvem, que será responsável por hospedar todas as informações relacionadas aos seus usuários, pois as ferramentas, produtos e serviços não ficam enraizados fisicamente nas máquinas dos usuários, mas ficam hospedados entre as nuvens. Ao fornecedor cabem todas as tarefas de desenvolvimento, armazenamento, manutenção, atualização, backup, escalonamento, etc. O usuário não precisa se preocupar com nada disso, apenas com acessar e utilizar. Essa mudança toda não afeta apenas os computadores pessoais, assim como as empresas, que não precisam mais adquirir e gerenciar recursos de TI, isso é precisamente uma grande vantagem da computação em nuvem, permitir à empresa se concentrar no seu *core business*[2], tendo a possibilidade de terceirizar todo o seu parque computacional.

A maneira como os serviços baseados em nuvem são cobrados é diferente, o usuário paga por aquilo que utilizar ou pelo tempo de utilização, gerando uma economia e controle melhor dos gastos. Esse modelo de pagamento é conhecido como *pay-per-use* [4]. Segundo [6] "uma consequência muito importante do modelo pay-per-use é a redução dos riscos de subutilização e de saturação". A subutilização está relacionada ao fato de usar uma quantidade menor de recursos do que a que foi reservada, pagando por recursos que não são utilizados e que, portanto, ficam ociosos. A saturação ocorre quando há um excesso de utilização sobre os recursos reservados, o que pode gerar serviços mais lentos e baixas qualidades de serviço, prejudicando os usuários. Como no modelo *pay-per-use*, o usuário só paga por aquilo que consome, reservando somente o necessário, esses riscos são

[1] Resiliente pode ser definido como a capacidade de um sistema de informação continuar a funcionar corretamente, apesar do mau funcionamento de um ou mais dos seus componentes.

[2] É um termo em inglês que significa a parte central de um negócio ou de uma área de negócios, e que é geralmente definido em função da estratégia dessa empresa para o mercado. Este termo é utilizado habitualmente para definir qual o ponto forte e estratégico da atuação de uma determinada empresa.

reduzidos. Assim, a utilização de plataformas computacionais terceirizadas é uma saída inteligente para os usuários lidarem com infraestruturas de TI.

Após um período tecnológico caracterizado pelo domínio do computador pessoal, está ocorrendo o ressurgimento da centralização, onde não necessitará mais a compra de hardware ou softwares. A computação em nuvem prevê um forte isolamento das aplicações e informações, da infraestrutura e dos mecanismos usados para suportá-las, de modo que os recursos ofertados podem ser dinamicamente alocados de acordo com a demanda [25]. Destarte, refletindo uma forma mais coesa a ideologia de se oferecer ao consumidor não mais a aquisição de um produto de software, mas a aquisição de uma licença de uso desse software, que agora fica hospedado no próprio ambiente da organização [24].

Portanto a Computação em Nuvem é um paradigma que está em evolução, suas definições, problemas, riscos e benefícios estão sendo definidos em debates entre todos os setores afins e essas definições, atributos e características evoluirão com o tempo.

2.1.1 Características

O *National Institute of Standards and Technology* (NIST) [5] descreve que o modelo de nuvem é composto por cinco características essenciais:

1. *On-demand self-service:* o consumidor pode provisionar unilateralmente recursos computacionais, como tempo de processamento no servidor ou armazenamento na rede, conforme necessário automaticamente sem a necessidade de interação humana com os provedores de cada serviço.

2. *Broad network access:* recursos são disponibilizados por meio da rede e acessados através de mecanismos padronizados que possibilitam o uso por plataformas heterogêneas *thin client* ou *thick client*, por exemplo celulares, tablets, notebooks, estações de trabalho, etc).

3. *Resource pooling:* os recursos computacionais do provedor são organizados em um *pool* para servir múltiplos usuários usando um modelo *multi-tenant*, com diferentes recursos físicos e virtuais, dinamicamente atribuídos e ajustados de acordo com a demanda dos usuários. Há um senso de independência de localização em que o cliente geralmente não tem nenhum controle ou conhecimento sobre o local exato

dos recursos fornecidos, mas pode ser capaz de especificar o local em um nível mais alto de abstração (por exemplo, país, estado ou data center). Armazenamento, processamento, memória e largura de banda de rede são exemplos de recursos.

4. *Rapid elasticity:* os recursos podem ser elasticamente provisionados e liberados, em alguns casos automaticamente, escalando rapidamente compatível com a demanda. Para o consumidor, os recursos disponíveis para provisionamento, muitas vezes parece ser ilimitado e podem ser apropriados em qualquer quantidade a qualquer momento.

5. *Measured service:* sistemas em nuvem controlam e otimizam automaticamente o uso dos recursos pelo aproveitamento da capacidade de medição em um certo nível de abstração apropriado para o tipo de serviço (por exemplo, processamento, armazenamento, largura de banda e contas de usuários ativos). Uso de recursos pode ser monitorado, controlado e relatado, proporcionando transparência para o provedor e consumidor do serviço utilizado.

2.1.2 Modelos de Serviço

Como a tecnologia está migrando do modelo tradicional para o modelo em nuvem, às ofertas de serviços evoluem quase que diariamente. A intenção nessa seção é expor qual a perspectiva atualmente dos três principais modelos de serviços e oferecer uma percepção de para onde esses serviços evoluirão no futuro.

A adoção de serviços de computação em nuvem está aumentando exponencialmente, e uma das razões é porque a sua arquitetura salienta os benefícios de serviços compartilhados sobre os produtos isolados. Este uso de serviços compartilhados ajuda para que o foco da organização esteja em seu negócio principal, e permite que os departamentos de TI reduzam a lacuna entre a capacidade computacional disponível e à demanda necessária que os sistemas requisitam. A intenção nessa seção é expor qual a perspectiva atualmente dos três principais modelos de serviços e oferecer uma percepção de para onde esses serviços estão evoluindo.

A concordância geral sobre o regime de categorização dos três serviços ofertados pela computação em nuvem são os modelos de [4] [5]: *Software as a Service*, *Platform as a Service* e *Infrastructure as a Service*, como demonstra a Figura 2.2 abaixo.

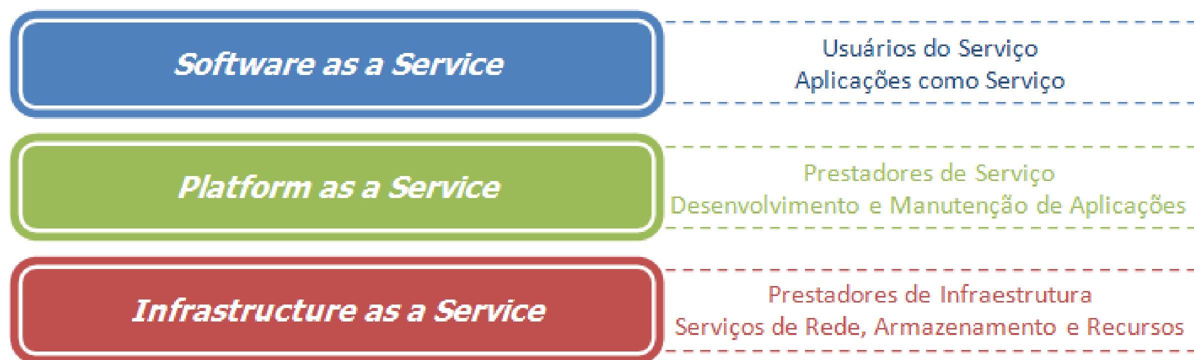


Figura 2.2 As principais camadas da Computação em Nuvem

A computação em nuvem é composta por três atores principais [3]: os prestadores de serviço, os usuários dos serviços e os prestadores de infraestrutura. De acordo com [23] a classificação desses papéis dentro da computação em nuvem é crucial, pois a definição das responsabilidades das partes que estão envolvidas em um solução nesse ambiente é importante. Os prestadores de serviços são aqueles que desenvolvem e deixam os serviços acessíveis aos usuários. Esses serviços, por sua vez, necessitam de uma infraestrutura sobre a qual estarão instalados; essa infraestrutura é fornecida na forma de um serviço pelos prestadores de infraestrutura.

2.1.2.1 *Infrastructure as a Service (IaaS)* A grande maioria das empresas conta com recursos de TI para suprir a necessidade de gerenciamento do negócio, por tanto, mantém uma infraestrutura de processamento, armazenamento, rede, comunicação, aplicações e suporte para garantir que a matéria prima básica, a informação, esteja disponível de forma rápida e segura. Manter esta estrutura vem se tornando um grande desafio em função da velocidade com que os componentes de TI ficam obsoletos e as novas demandas da organização se tornam cada vez mais frequentes [26]. Simples usuários e empresas *start-ups* têm agora acesso a um nível maior de soluções de tecnologias de TI, e a escalabilidade da infraestrutura dinâmica permite aos consumidores se adequar as suas necessidades em um nível mais granular [27].

Infrastructure as a Service é um modelo de computação em nuvem, que facilita a entrega de recursos computacionais como um serviço. IaaS permite a um cliente alugar recursos como um serviço totalmente terceirizado, em vez de comprar servidores, software, espaço para os data centers ou equipamentos de rede. Tem a vantagem de escalabilidade instantânea e fornece uma solução de custo eficaz e flexível. Existe também a entrega

de sistemas operacionais e tecnologia de virtualização para gerenciar os recursos. Os recursos não precisam ser comprados se podemos alugar somente os serviços de que a organização necessita. Com isso, paga-se pela quantidade de processamento, espaço em disco, armazenamento e assim por diante, onde o custo é sobre o que realmente for consumido [28]. Não estamos falando de terceirização ou de locação de infraestrutura, estamos falando na aquisição de serviços de informação, ou seja, estamos falando da aquisição do produto final gerado por toda a infraestrutura de TI e não dos meios para gerá-lo.

O modelo IaaS é semelhante a *Utility Computing*, em que a ideia básica é oferecer serviços de computação da mesma forma, como utilitários. Ou seja, você paga para a quantidade de poder de processamento, espaço em disco, e assim por diante, onde o custo é sobre o que realmente for consumido [29]. O usuário abstrai os detalhes de infraestrutura, incluindo recursos de computação física, localização, particionamento de dados, escalonamento, segurança, backup, e assim por diante [30]. O prestador de serviços possui os equipamentos e é responsável pela instalação, execução e manutenção. Segundo [3], é válido ressaltar que são os prestadores de infraestrutura que, através da virtualização, oferecem esses serviços por demanda aos prestadores de serviços. Alguns exemplos de *Infrastructure as a Service* são os serviços disponibilizados pela *Amazon* através do *Amazon Elastic Compute Cloud*[3] (EC2) e o *HP Cloud*[4].

2.1.2.2 *Platform as a Service (PaaS)* *Platform as a Service* é uma consequência do SaaS, mas é um ambiente de desenvolvimento de aplicativos, não apenas o uso de uma aplicação. As soluções em PaaS diferem das soluções SaaS, porque fornecem uma plataforma de desenvolvimento virtual hospedada na nuvem e acessível através da web. Os fornecedores de soluções PaaS ofertam tanto a plataforma de computação, como toda a pilha de solução necessária para o desenvolvimento e teste de aplicações baseadas no modelo de SaaS e seguida distribui ou implanta suas aplicações na nuvem com facilidade [27].

PaaS é um conceito de entrega de aplicativos, onde os recursos necessários para construir aplicações e serviços não devem ser baixados e instalados, mas são acessíveis através da Internet. O modelo PaaS proporciona um menor custo de entrada para projetistas de aplicações e distribuidores, apoiando o completo ciclo de vida do desenvolvi-

[3]http://aws.amazon.com/ec2/
[4]https://www.hpcloud.com/

mento de softwares, eliminando assim a necessidade de aquisição de recursos de hardware e software [31]. Ao contrário do modelo IaaS, onde há a possibilidade de criar uma instância específica do SO, os desenvolvedores que utilizam o modelo de PaaS estão preocupados apenas com o desenvolvimento baseado na web e, geralmente, não se importam com qual sistema operacional está sendo utilizado [26].

Geograficamente distribuídas, as equipes de desenvolvimento podem trabalhar juntas em projetos de desenvolvimento de aplicações. Esse modelo se propõe a criar uma plataforma para o desenvolvimento de aplicações já baseadas em nuvem, possibilitando a criação e operação das aplicações. Eliminando a complexidade de implantação e configuração da infraestrutura, a plataforma como serviço distingue-se pelo alto nível de abstração que elas fornecem, oferecendo ao usuário uma maneira de implantar suas aplicações em um *pool* aparentemente ilimitado de recursos computacionais. Essa camada é útil porque permite aos programadores freelancers e empresas start-ups a possibilidade de implementar aplicações baseadas na web, sem o custo e a complexidade de compra de servidores e configurações. O benefício de estar no modelo de PaaS é o aumento do número de pessoas que podem desenvolver, manter e implantar aplicações web. Em resumo, PaaS surgi para democratizar o desenvolvimento de aplicações [32]. As plataformas em nuvem fornecem instalações para o apoio ao ciclo de desenvolvimento de aplicação, incluindo a concepção, execução, depuração, testes, implantação, operação e suporte de aplicações web ricas e serviços na internet.

De acordo com [23], em geral os desenvolvedores dispõem de ambientes escaláveis, mas eles têm que aceitar algumas restrições sobre o tipo de software que se pode desenvolver, desde limitações que o ambiente impõe na concepção das aplicações até a utilização de banco de dados NoSQL, ao invés de banco de dados relacionais. Alguns exemplos de *Platform as a Service* podem ser encontrado no serviço ofertado pelo *Google App Engine*[5], que permite o desenvolvimento de aplicações em Java e Python e serviço de armazenamento de dados distribuído que contém um mecanismo de consultas e transações, também podemos encontrar um exemplo de PaaS no *Windows Azure Platform*[6].

2.1.2.3 *Software as a Service (SaaS)*

Com a convergência de várias evoluções tecnológicas, organizacionais e políticas, tais como terceirização, trabalho distribuído de software, o surgimento de ambientes colaborativos baseados na internet e o crescimento

[5]https://developers.google.com/appengine/
[6]http://www.microsoft.com/windowsazure/

do fornecimento global de serviços de TI, presenciamos o surgimento de uma nova era onde software torna-se serviço e serviço torna-se software.O que é oferecido neste modelo são aplicações que rodam na infraestrutura da nuvem e podem ser acessadas por *thin clientes*, como os navegadores [25]. O software passa a ser oferecido como um serviço que pode ser acessado através da Internet e a principal característica desse modelo é que ela separa o proprietário do usuário do software. *Software as a Service*, ou SaaS, é provavelmente o tipo mais comum de desenvolvimento dos serviços em nuvem. Com o SaaS, uma única aplicação é entregue a milhares de usuários que o fornecedor atende. Os clientes não pagam por possuir o software, mas sim, por utilizá-lo [33].

Ao contrário do método tradicional de compra e instalação de software (geralmente envolvendo uma despesa de capital ou uma taxa de licenciamento), o cliente aluga o SaaS através de um modelo de despesa operacional, ou seja, o modelo conhecido como pay-per-use, ou contrato de assinatura. O modelo de licenciamento pay-per-use é também conhecido como o licenciamento on-demand, as aplicações entregues através do modelo de SaaS são faturadas em um uso previsto com base período de tempo, ao invés de pagar os custos iniciais comum de licenciamento tradicional [34].

No modelo tradicional, o cliente possuía diversas áreas com que se preocupar e poderia facilmente ter grandes gargalos e a alta subutilização dos recursos, pois necessitava a compatibilidade com hardware, softwares e sistemas operacionais, além dos custos adicionais com a manutenção, suporte e atualização. O SaaS permite aos clientes obter os mesmos benefícios de softwares tradicionais comercialmente licenciados, internamente operados sem a complexidade associada à instalação, gestão, apoio, licenciamento e alto custo inicial [26]. Nesse modelo, cliente não compra o software, mas sim aluga-o para uso em uma assinatura. Em alguns casos, o serviço é gratuito para uso limitado. Cada usuário é chamado de inquilino, e esse tipo de disposição é chamado de arquitetura *multi-tenancy* [33].

A diferença mais importante entre o modelo de arquitetura de software tradicional e do modelo SaaS é o número de inquilinos que o aplicativo suporta. O modelo tradicional de software é um modelo, isolado de um único inquilino, o que significa que um cliente compra um software e instala-o. É executada apenas a aplicação específica e só para esse. O modelo SaaS é um modelo de arquitetura *multi-tenancy*, o que significa que a infraestrutura física de hardware do servidor é compartilhada entre vários clientes diferentes, mas logicamente é única para cada cliente [32]. Uma aplicação *multi-tenancy* basea-se no servidor que suporta a implantação de vários clientes em uma instância única

de software. Os servidores do fornecedor estão praticamente divididos de forma que cada usuário utiliza com uma instância de aplicativo virtual personalizada [23]. Esse recurso tem vantagens óbvias para o fornecedor de SaaS que, de alguma forma são transferidas para o usuário final: suporte para mais clientes em menos componentes de hardware e, mais rápido e mais simples lançamentos de atualizações de aplicativos e patches de segurança [29]. Hurwitz et al. [28] apresenta algumas características que definem mais especificamente o modelo de *Software as a Service*.

O processo para aderir esse modelo é rápido, pois uma série de atividades que não agregam valor, como instalação e configuração, simplesmente deixam de existir. Além disso, os produtos SaaS usam interfaces baseadas em browsers, mais intuitivas e fáceis de usar que as interfaces usadas nos softwares tradicionais. Segundo Salesforce[7], os SaaS são mais escaláveis, mais seguros e mais confiáveis do que a maioria dos aplicativos. Por fim, como você recebe todos os upgrades, seus aplicativos obtêm segurança, melhorias de desempenho e novos recursos automaticamente. Os usuários das aplicações baseadas nesse modelo, não se importam onde a aplicação esteja hospedada, que tipo de sistema operacional ele utiliza, ou se é desenvolvida em PHP, Java, Ruby, Python ou qualquer outra tecnologia. Alguns exemplos de *Software as a Service* podem ser encontrados nos diversos serviços ofertados pelo Google Apps[8], o *Customer Relationship Management* (CRM) on-line da Salesforce[9] e o serviço da Apple conhecido como iCloud[10].

2.2 DATABASE AS A SERVICE (DBaaS)

Para a maioria das empresas/usuários, o armazenamento é o mais importante e o mais caro dos recursos de TI dentro de sua infraestrutura. Do ponto de vista comercial, os conceitos de produção em massa e de gestão da qualidade levaram à necessidade de armazenamento de dados para um novo nível. Agora, os dados são recolhidos para efeitos de análise e previsão, e essa análise é usada para construir a credibilidade de um negócio e melhorar suas práticas gerenciais. Porém, as soluções de banco de dados utilizadas hoje estão se tornando cada vez mais caras e pouco práticas, e as dificuldades com gerenciamento, manutenção e custos se tornam maiores e mais complicadas.

[7]Disponível em: http://www.salesforce.com/br/cloudcomputing/
[8]http://www.google.com/enterprise/apps/business/
[9]http://www.salesforce.com/
[10]https://www.icloud.com/

O conceito da computação em nuvem consiste em fornecer recursos de TI como um serviço, e dentre esses serviços o gerenciamento e armazenamento de dados são componentes críticos na pilha de softwares da nuvem, pois a maioria das aplicações são orientadas a dados [7]. Os acelerados progressos em tecnologias de rede e Internet têm colaborado para o surgimento do modelo de PaaS, e os bancos de dados em nuvem herdam todas as vantagens desse modelo, permitindo às organizações migrarem o gerenciamento de dados para soluções fornecidas por prestadores de serviço. Sem a necessidade de implementar e gerenciar por conta própria, aliviando a necessidade das organizações de comprar hardware e software caros, ou lidar com atualizações e contratações de profissionais para tarefas administrativas e de manutenção [4].

Database as a Service (DBaaS) é uma tecnologia onde um provedor hospeda e gerencia todo ambiente necessário ao funcionamento dos sistemas de banco de dados e o terceiriza como um serviço para um ou mais consumidores [8]. Na literatura, a visão de um DBaaS não possui um consenso e de acordo com [35], um serviço em nuvem pode ser modelado em sete camadas: (1) instalações: infraestrutura que compreende os data centers; (2) rede: nesta camada são considerados os links de rede e os caminhos tanto na nuvem quanto entre a nuvem e o usuário; (3) hardware: componentes físicos da computação e equipamentos de rede; (4) sistema operacional: são os componentes de software que formam o sistema operacional, tanto no servidor (sistema operacional em execução na máquina física) quanto no usuário (sistema operacional em execução na máquina virtual); (5) middleware: camada de software entre o sistema operacional e os aplicativos do usuário (tipicamente presente nos serviços em nuvem os modelos SaaS e PaaS); (6) aplicações: aplicações executadas pelos usuários dentro da nuvem; (7) usuários: usuário final do serviço em nuvem e as aplicações executadas fora da nuvem.

Com isso, nesse trabalho consideramos que um DBaaS é baseado em uma infraestrutura como um serviço semelhante a provida pelo OpenNebula [17] ou Amazon [36], onde caracterizamos um DBaaS de acordo com a Figura 2.3, como uma especialização das camadas listadas acima, sendo composto por uma camada de máquinas físicas, denominadas servidores, onde são executados os *hypervisors*[11], criando o ambiente virtualizado. Este ambiente provê máquinas virtuais (VM) configuradas nas quais são hospedados e executados os SGBDs. Cada VM pode possuir mais de um SGBD que pode conter várias instâncias de banco de dados. De acordo com esta visão da organização das camadas que

[11] *Hypervisor* é um pedaço de software, firmware ou hardware que cria e executa máquinas virtuais. O *hypervisor* controla o processador do sistema, memória e outros recursos para alocar o que cada máquina virtual requer.

compõem o serviço, o provedor é responsável pelo gerenciamento, tarefas administrativas e manutenção de toda a pilha de recursos necessária para o funcionamento do serviço.

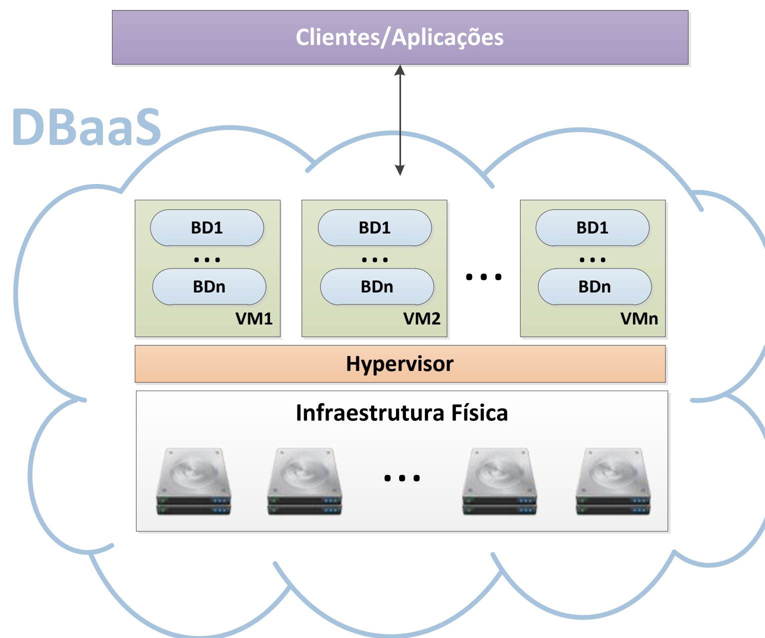


Figura 2.3 Ilustração do ambiente de um *Database as a Service* (DBaaS).

Segundo [9] um DBaaS oferece uma série de benefícios aos usuários, pois ele dispõe de um ambiente escalável, disponível, rápido e com qualidade de serviço garantida pelos contratos de SLA. Por outro lado, o provedor tem que garantir a ilusão de recursos infinitos, sob cargas de trabalho dinâmicas e minimizar os custos operacionais associados a cada usuário [10]. De acordo com [4], um DBaaS na perspectiva do usuário tem como principal necessidade uma interface simples que não necessite de ajustes ou administração. Trata-se de uma melhoria em relação às soluções tradicionais que requerem técnicas para provisionar recursos, seleção de SGBDs em nuvem, instalação, configuração e administração. O usuário deseja um desempenho satisfatório, expresso em termos de latência e vazão, independente do tamanho da base de dados e das alterações da carga de trabalho. Da perspectiva do provedor, é necessário atender aos acordos de SLA, apesar da quantidade de dados e alterações na carga de trabalho. O sistema deve ser eficiente na utilização dos recursos, possuir alta disponibilidade e segurança.

Portanto, assim como a computação em nuvem os DBaaS estão ganhando espaço devido a muitas das mesmas razões que a computação em nuvem. No entanto, prover mecanismos de elasticidade, escalabilidade, qualidade de serviço e disponibilidade em ambientes em nuvem, proporcionando aos usuários a percepção de recursos quase infinitos, é

um grande desafio. Claramente isto é um desafio também na disponibilização dos DBaaS, e para se alcançar essas funcionalidades e princípios é necessário um monitoramento detalhado e preciso de todas as camadas e recursos que compõem um DBaaS.

2.3 MONITORAMENTO DE SERVIÇOS EM NUVEM

Segundo [13] essas camadas devem ser vistas como onde as soluções de monitoramento devem agir. Portanto, o monitoramento de recursos em ambientes de nuvem é fundamental para medir e avaliar continuamente os comportamentos da infraestrutura ou aplicações em termos de performance, confiabilidade, consumo de energia, capacidade de atender SLAs, segurança, etc [12]. Por um lado, é uma necessidade fundamental para o controle e gerenciamento das infraestruturas de hardware e software, por outro lado, fornece informações e indicadores sobre o desempenho da utilização dos serviços.

O monitoramento é fundamental para as atividades relacionadas com os objetivos principais de um provedor, portanto, não possuir as informações necessárias em momentos de decisão anularia a utilidade de um serviço, principalmente em um ambiente de nuvem onde o consumidor espera que o mesmo reaja de acordo com as suas necessidades e atenda suas expectativas. Em [13] os autores analisam e destacam o papel do monitoramento nas diversas atividades de gerenciamento da infraestrutura em nuvem, apresentadas abaixo:

1. **Capacidade e planejamento de recursos:** os provedores de serviços em nuvem ao oferecerem garantias em termos de QoS especificados em SLAs, eles passam a ser responsáveis pelo planejamento da capacidade dos recursos para que os usuários/aplicações não se preocupem com isso. Portanto, o monitoramento torna-se essencial para os provedores de serviços em nuvem prever e acompanhar a evolução de todos os parâmetros envolvidos nesse processo, a fim de planejar adequadamente sua infraestrutura e recursos.

2. **Capacidade e gestão de recursos:** a virtualização tornou-se um componente chave para implementar ambientes de computação em nuvem. Escondendo a grande heterogeneidade de recursos físicos, a tecnologia de virtualização introduziu um outro nível de complexidade para o provedor de serviços, tendo que gerir os recursos físicos e virtualizados. Assim, com essa complexidade o monitoramento deve capturar com precisão o estado dessas camadas, sendo necessário para lidar com a volatilidade apresentada na virtualização.

3. **Gerenciamento do data center:** serviços em nuvem são fornecidos através de grandes centros de dados, cuja gestão é uma atividade muito importante. Essa atividade faz parte da gestão de recursos, porém implica em duas tarefas fundamentais: (a) monitoramento, que mantém o controle desejado de métricas sobre hardware e software, (b) análise dos dados, que processa essas métricas para inferir estados do sistema, provisionamento de recursos, solução de problemas ou outras decisões.

4. **Gestão de SLA:** o monitoramento é obrigatório e fundamental para o cumprimento dos acordos de SLA, portanto o monitoramento permite que os provedores de serviços em nuvem formulem SLAs mais realistas e dinâmicos.

5. **Faturamento:** uma das características essenciais dos serviços em nuvem é a oferta de "serviços medidos", que permite ao usuário pagar proporcionalmente a utilização do serviço em diferentes granularidades. Para proporcionar esses modelos de custo, o monitoramento é necessário, tanto do lado do fornecedor para o faturamento, quanto do lado do consumidor para verificar o seu próprio uso e comparar diferentes fornecedores. É importante destacar que não é um processo trivial porque necessita que o monitoramento seja realizado em diferentes granularidades dos recursos utilizados.

6. **Solução de problemas:** a infraestrutura complexa de uma nuvem representa um grande desafio para a solução de problemas, pois quando um problema acontece deve ser procurado nos vários componentes possíveis e cada um deles é composto de várias camadas. Portanto, o monitoramento precisa ser abrangente, confiável e em tempo hábil para que o provedor do serviço possa localizar o problema dentro da infraestrutura e analisar se o mesmo é uma falha por sua parte ou do consumidor.

7. **Gerenciamento de desempenho:** se um consumidor adota uma nuvem pública para hospedar um serviço crítico, uma aplicação científica, a variabilidade e disponibilidade do desempenho tornam-se uma preocupação. Portanto, da perspectiva do consumidor, monitorar o desempenho é necessário para adaptar-se às mudanças ou aplicar medidas corretivas. O monitoramento nesse ponto é importante, uma vez que pode melhorar consideravelmente o desempenho das aplicações reais do consumidor.

2.3.1 Soluções para Monitoramento de Serviços em Nuvem

Atualmente no mercado existem diversas soluções comerciais para monitoramento de serviços em nuvem, algumas soluções são para o gerenciamento de nuvens e contêm além de outras funcionalidades um módulo especificamente voltado para a atividade de monitoramento e outras soluções onde a única meta é o monitoramento da nuvem. Nesta seção apresentamos algumas das mais conhecidas atualmente.

CloudWatch

CloudWatch [14] é a solução da Amazon que fornece monitoramento para os recursos da nuvem AWS[12] e para as aplicações dos clientes hospedadas em sua variedade de serviços. Desenvolvedores e administradores de sistema podem usá-lo para coletar e monitorar métricas, e reagir imediatamente para manter seus aplicativos e empresas em funcionamento. As informações coletadas são principalmente relacionadas com as plataformas virtuais. CloudWatch reúne vários tipos de informações de monitoramento e as armazena por duas semanas. Em relação a estes dados, os usuários podem construir *plots*, estatísticas, indicadores, comportamentos temporais e alarmes de estado. De acordo com [13] o CloudWatch não fornece informações sobre monitoramento da infraestrutura física, e a forma como os dados de monitoramento são recolhidos, coletados e analisados não é transparente e acessível. A Amazon cobra por esse serviço de monitoramento de forma separada, sendo gratuito para ciclos com amostragens de cinco minutos para recursos básicos. Para ciclos de um minuto e mais métricas disponíveis passa a ser taxado com base na quantidade de recursos habilitados [14].

AzureWatch

AzureWatch [15] é uma solução baseada em nuvem dedicada ao monitoramento avançado e escalável para produtos baseadas no Windows Azure[13]. Possui um conjunto de métricas que podem ser monitoradas nos serviços que vão desde máquinas virtuais, armazenamento, banco de dados, aplicações web e outros serviços de mais alto nível que a plataforma fornece. O AzureWatch permite o usuário especificar regras com base em alertas criados sobre as métricas monitoradas para gerar notificações de problemas, realizar ajuste de *scale up* ou *scale down* dinamicamente, assim como, possibilita o acompanhamento do estado dos serviços através de gráficos e relatórios. Por ser um serviço

[12] http://aws.amazon.com
[13] http://www.windowsazure.com

comercial, o AzureWatch cobra com base na quantidade de tempo em que os recursos habilitados foram monitorados, ou seja, por cada hora de monitoramento é cobrado um valor monetário por recurso. O AzureWatch é desenvolvido sobre a SDK do Windows Azure e com isso limita-se as métricas disponíveis pela mesma.

New Relic

New Relic [16] é um *software-as-a-service* para gestão de desempenho e monitoramento, baseado em uma arquitetura de agentes. Através desses agentes disponibilizados pelo serviço, que são instalados manualmente, é possível monitorar servidores, máquinas virtuais, aplicações web em diversas linguagens e aplicações mobile em iOS ou Android. Possui diversos plugins que aumentam a capacidade de monitoramento da pilha de tecnologias open-source existentes, possibilitando também a criação de novos plugins. O New Relic disponibiliza uma API com *bindings* em diversas linguagens, o que permite consultar as métricas monitoradas pelos agentes e utilizar funções relacionadas a gestão de desempenho. Assim como os serviços anteriores, o New Relic também é comercial e possui um modelo de cobrança. Existem perfis que vão de um conjunto limitado de funcionalidades até o acesso total delas, esses perfis também possuem a característica que informa a quantidade de tempo que os dados coletados pelos agentes estarão disponíveis. É importante destacar que independente do perfil, o ciclo de monitoramento das métricas é a cada minuto.

OpenNebula

OpenNebula [17] é um kit de ferramentas para computação em nuvem open-source para gerenciamento de infraestruturas distribuídas e heterogêneas. O OpenNebula permite a virtualização dessas infraestruturas físicas para construir implementações privadas, públicas e híbridas de *infrastructure-as-a-service*. O OpenNebula gerencia o armazenamento, rede, virtualização, monitoramento e segurança a fim de implantar serviços em múltiplas camadas, combinando recursos dos data centers com recursos remotos em nuvem, de acordo com as políticas de alocação [37]. O monitoramento é realizado através do módulo *Information Manager* [38], sendo encarregado pelo monitoramento dos servidores. É composto por diversos sensores, cada um é responsável por um diferente aspecto a ser monitorado (CPU, memória, hostnames e outros). Além disso, existem sensores para coletar informações sobre os diferentes *hypervisors* que a infraestrutura possa utilizar. Esses sensores utilizam conexões SSH para acessar e coletar os dados de monitoramento da camada física, mas o OpenNebula também possui integração com a ferramenta Gan-

glia para obter mais informações de monitoramento da camada física e virtual. Porém requer um trabalho maior de configuração por parte do administrador do sistema [39].

Existem mais soluções para o monitoramento de serviços em nuvem que foram detalhadas em [13], onde os autores as classificam e explicam seus objetivos. Essas soluções se demonstram poderosas, mas, como são soluções comerciais e muitas vezes com objetivos específicos, não se ajustam totalmente as necessidades de monitoramento que o fornecedor de um serviço em nuvem requer. De fato, cada vez mais a necessidade por um monitoramento personalizado é importante para atender as demandas de qualidade de serviço, e com os serviços de banco de dados em nuvem não é diferente.

2.4 REQUISITOS PARA O MONITORAMENTO DE SERVIÇOS DE BANCO DE DADOS EM NUVEM

As camadas e atividades apresentadas na seção 2.3, são comuns a quase todos os serviços em nuvem, porém os DBaaS incluem a camada de dados onde é necessário a existência do monitoramento de novas atividades. Nessa camada identificamos duas novas atividades:

8. **Gerenciamento da camada de dados:** os provedores ao encapsularem a complexidade inerente a implantação e ao gerenciamento da camada de dados, passam a ser os responsáveis por toda a administração e manutenção dos SGBDs que compõem o serviço. Portanto, o monitoramento torna-se essencial para que os provedores do serviço possam acompanhar a utilização dos recursos pelos diversos SGBDs, assim como seus estados e estatísticas. Também sendo importante o monitoramento sobre as instâncias de banco de dados existentes dentro dos SGBDs, a fim de acompanhar a evolução dos esquemas. Com isso, o monitoramento dessa atividade é imprescindível para que os provedores possam criar estratégias para garantir a qualidade de serviço - como replicação, elasticidade, migração de dados ou outras tomadas de decisões.

9. **Gerenciamento da carga de trabalho:** o monitoramento das cargas de trabalho enviadas ao serviço de banco de dados em nuvem é fundamental para o controle dos SLOs vinculados as consultas, assim como, permite que os provedores do serviço possam inferir e planejar adequadamente os recursos de acordo com as cargas.

Portanto, o monitoramento deve sempre existir em serviços em nuvem e como relatado nesse capítulo, o monitoramento contínuo é necessário sobre as diversas camadas que compõem os serviços, pois abastece os provedores e consumidores com informações estratégicas sobre o real estado do serviço em diversas granularidades. Nos DBaaS não é diferente, com base na visão proposta na seção 2.2 a Figura 2.4 demonstra que o monitoramento de um DBaaS é um processo transversal e abrange a camada física (1), virtualização (2), a camada virtual (3), e principalmente a nova camada que aparece na disponibilização desse serviço, a camada de dados (4) e as cargas de trabalho específicas nessa camada (5). Esse processo deve ser também horizontal e monitorar toda instância de recurso que exista dentro das camadas, como por exemplo na camada física todos os servidores e *hypervisors* existentes dentro dos servidores, na camada virtual todas as máquinas virtuais existentes, na camada de dados todos os SGBDs e instâncias de banco de dados e nas cargas de trabalho interceptar as consultas enviadas.

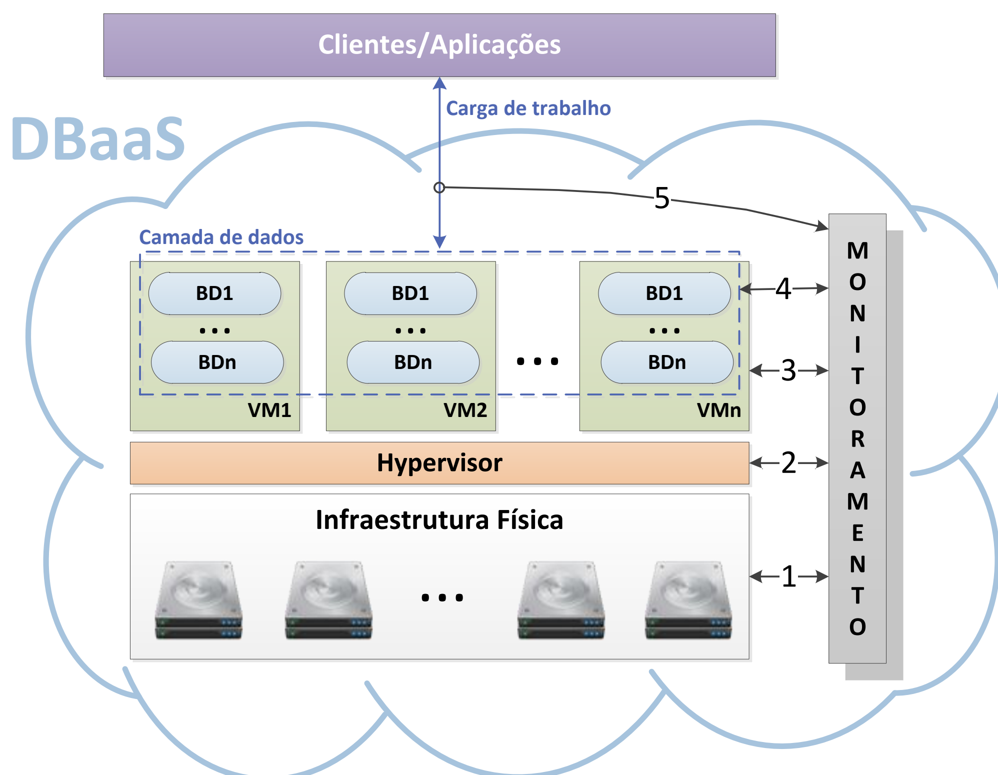


Figura 2.4 Ilustração do monitoramento nas diversas camadas de um DBaaS.

A partir dessas definições de camadas e atividades envolvidas em um DBaaS, levantamos uma lista de requisitos que são comuns e que devem ser atendidos em implementações de soluções de monitoramento para esse tipo de serviço:

Requisito (I): Com a grande quantidade de camadas e a heterogeneidade de recursos existentes nessas camadas, a solução deve permitir a definição de diferentes métricas de acordo com as camadas e conforme a necessidade.

Requisitos (II) e (III): Com a necessidade de capturar com precisão o estado dos recursos físicos a fim de planejar adequadamente a infraestrutura e acompanhar a volatilidade dos recursos virtuais, a solução deve permitir o monitoramento de métricas sobre cada recurso físico e virtual que compõem o serviço.

Requisito (IV): Como o DBaaS é um serviço especializado em banco de dados, existe a grande necessidade do acompanhamento de como os SGBDs e instâncias de banco de dados estão reagindo frente as cargas de trabalho dos diferentes clientes que chegam ao serviço, com isso, a solução deve permitir o monitoramento de métricas em ambos os níveis da camada de dados.

Requisito (V): Com a necessidade da formulação de SLAs mais realistas com a camada de dados e o planejamento mais adequado dos seus recursos em função das cargas de trabalho recebidas, a solução deve permitir o monitoramento de métricas no nível das cargas de trabalho (consultas) enviadas as instâncias de banco de dados existentes no DBaaS.

Requisito (VI): Com a necessidade de analisar os dados sobre as métricas coletadas a fim de inferir o provisionamento de recursos, estratégias de replicação e elasticidade, transferência de dados ou outras decisões, a solução deve permitir o armazenamento das métricas coletadas em função dos recursos em uma base serial histórica.

Requisito (VII): Com a existência de diferentes tipos de recursos com diferentes configurações, é muito importante um acompanhamento mais específico e preciso de seus estados pelo monitoramento, com isso, a solução deve permitir a personalização dos ciclos de monitoramento em diferentes granularidades para as métricas nos diferentes tipos de recursos nas diversas camadas.

Requisito (VIII): Com a necessidade de utilizar as métricas que estão sendo coletadas nas diversas camadas de forma fácil e transparente, a solução deve permitir o acesso ao ambiente de monitoramento através de uma API que encapsule a complexidade da comunicação e manipulação das informações trocadas, elevando o consumo das métricas para o modelo de programação.

Requisito (IX): Para gerenciar os diversos SLAs existentes no serviço, proporcionar modelos de custo/faturação, criar novas soluções gerenciais ou outras necessidades, a API da solução deve possibilitar o consumo das métricas conforme a necessidade em função do tempo ou da quantidade de coletas existentes.

Em resumo, os requisitos são listados na Tabela 2.1.

Requisito	Descrição
I	Permitir a definição de diferentes métricas nas camadas.
II	Permitir o monitoramento da camada de recursos físicos.
III	Permitir o monitoramento da camada de recursos virtuais.
IV	Permitir o monitoramento da camada de dados.
V	Monitorar métricas de carga de trabalho relacionadas a banco de dados.
VI	Armazenar as métricas coletadas em uma base serial histórica.
VII	Permitir configurar o conjunto de métricas e os ciclos de monitoramento das métricas em diferentes granularidades nas diversas camadas de recursos.
VIII	Fornecer fácil acesso ao ambiente de monitoramento através de uma API que encapsule a complexidade da comunicação e manipulação das informações trocadas.
IX	Possibilitar o consumo das métricas em função do tempo e quantidade de coletas.

Tabela 2.1 Requisitos para o monitoramento de serviços de DBaaS.

Os requisitos elencados anteriormente são importantes para o desenvolvimento de soluções de monitoramento para DBaaS, porém somente a descrição dos requisitos não ajudam em como as soluções devem ser implementadas. Desta forma, é fundamental que tais requisitos sejam traduzidos em boas práticas para implementação das soluções de monitoramento. Uma possível solução é adotar a estratégia comum na área de Engenharia de Software de arcabouços de software, os quais facilitam a implementação de soluções que devem seguir requisitos de um determinado domínio de aplicação. A utilização dessa estratégia oferece mecanismos de extensibilidade e configurabilidade muito maiores dos que os mecanismos disponíveis em soluções totalmente já implementadas, permitindo criar soluções mais adequadas para diferentes cenários de necessidade.

Neste trabalho, defendemos a criação de um *framework* de aplicação que encapsule os requisitos elencados para o monitoramento de DBaaS. Na próxima seção são apresentados os trabalhos relacionados com *framework* de aplicação para monitoramento em nuvem. Em seguida, tais trabalhos são analisados diante dos requisitos elencados anteriormente.

2.5 TRABALHOS RELACIONADOS

Na literatura, existem alguns trabalhos que propõem *frameworks* para criação de soluções para o monitoramento de serviços em nuvem. Esses trabalhos são importantes pois disponibilizam um arcabouço com estruturas e funcionalidades inicialmente implementadas, as quais aceleram o desenvolvimento das soluções de monitoramento para os serviços em nuvem.

Os autores de [18] propõem um *framework* para monitoramento orientado a serviços com REST e Nagios. Através de uma arquitetura baseada em provedor e consumidor, onde o consumidor representa entidades que irão consumir os dados monitorados e o provedor é representado por um serviço que existirá dentro dos recursos do ambiente e é responsável pelo monitoramento da camada física e virtual. Para realizar o processo de monitoramento dos recursos os autores limitam-se a utilização da ferramenta Nagios e suas métricas para ambas as camadas, mas possibilita a configuração dos ciclos de monitoramento em diferentes granularidades. O *framework* proposto pelos autores não contempla o monitoramento para camada de dados e para as cargas de trabalho específicas. As métricas monitoradas são enviadas pelos provedores a um gerenciador que as armazena em uma base histórica, porém o modelo proposto não possibilita o armazenamento de métricas mais elaboradas. Uma interface RESTful para consumo das métricas é disponibilizada pelo *framework*, onde os consumidores realizam requisições HTTP através de URIs definidas. Porém, os autores deixam a cargo do consumidor toda a complexidade de manipulação dos dados retornados em formato XML, assim como não possibilitam que o consumo das métricas seja realizado com filtros.

Um *framework* para medição da qualidade de serviços de banco de dados em nuvem em tempo de execução é proposto em [19]. Os autores propõem uma estratégia de monitoramento e análise para um conjunto definido de métricas relacionadas a qualidade para o gerenciamento de DBaaS. O *framework* é baseado no Yahoo! Cloud Serving Benchmark (YCSB) e a contribuição proposta é a integração de um sistema de configuração

de serviços de banco de dados em nuvem com o benchmarking. A métricas definidas são relacionadas a desempenho (vazão e tempo de resposta) utilizadas para medir a qualidade da elasticidade, assim como métricas para medir a consistência e confiabilidade - não informando a possibilidade da definição de novas métricas ou do monitoramento em camadas inferiores. O artigo não especifica em que frequência o ciclo de monitoramento das métricas é realizado ou se é possível coletar em diferentes granularidades. O *framework* não contempla uma API para acesso as métricas definidas, assim como não é informado a existência de uma base para o armazenamento das mesmas.

No trabalho [20], os autores apresentam o *framework* LoM2HiS para gerenciar mapeamentos entre métricas de baixo nível coletadas nos recursos com os parâmetros de alto nível dos SLAs. O LoM2HiS utiliza uma arquitetura dividida em componentes, onde um agente de monitoramento existirá em cada recurso da infraestrutura e são responsáveis por monitorar a camada física e virtual. As métricas coletadas pelos agentes são acessadas por um controlador que as trata e envia para um componente responsável por monitorar em tempo de execução os acordos de SLA definidos entre provedor e consumidor com base nas métricas coletadas, gerando notificações caso ameaças de violação dos acordos surjam. O *framework* permite a definição de SLAs simples e complexos, porém que se limitam as métricas monitoradas pela ferramenta Ganglia. As métricas coletadas são armazenadas em arquivos XML que são utilizados nas trocas de mensagens entre os componentes, porém as métricas não são armazenadas em uma base a fim de serem utilizadas posteriormente.

O trabalho [21] apresenta um *framework* baseado em um modelo de execução para o monitoramento em nuvem (RMCM), que é capaz de alcançar um equilíbrio entre a sobrecarga no tempo de execução e capacidade de monitoramento via gestão adaptativa do monitoramento da infraestrutura. O *framework* utiliza uma arquitetura agente-servidor onde o processo de monitoramento pode ser realizado sobre uma hierarquia de entidades que abrange a camada física e virtual. Um agente é implantado nos recursos dessas camadas afim de monitorar os serviços e aplicações em execução, porém o agente possui um conjunto definido de possibilidades para o monitoramento, mas possibilita a definição de novas métricas e a configuração individual dos agentes. O *framework* possibilita o monitoramento sobre bancos de dados, porém não possui o refinamento necessário de monitoramento para camada de dados - assim como não possibilita o monitoramento sobre as cargas de trabalho específicas de banco de dados. As informações coletadas são enviadas a uma base de dados localizada no centro de monitoramento (servidor), mas

o *framework* não disponibiliza uma forma fácil e transparente de acesso ao ambiente de monitoramento, e para consumir as informações coletadas é necessário realizar consultas diretamente a base de dados.

2.5.1 Análise Comparativa entre os Trabalhos Relacionados

Com base nos trabalhos apresentados anteriormente, a Tabela 2.2 apresenta uma análise comparativa dos trabalhos relacionados com base nos requisitos levantados na seção 2.4.

Req.	**Descrição**	**[18]**	**[19]**	**[20]**	**[21]**
I	Permitir a definição de diferentes métricas nas camadas.	Não	Não	Parcial	Sim
II	Permitir o monitoramento da camada de recursos físicos.	Sim	Não	Sim	Sim
III	Permitir o monitoramento da camada de recursos virtuais.	Sim	Não	Sim	Sim
IV	Permitir o monitoramento da camada do dados.	Não	Parcial	Não	Não
V	Monitorar métricas de carga de trabalho relacionadas a banco de dados.	Não	Sim	Não	Não
VI	Armazenar as métricas coletadas em uma base serial histórica.	Sim	N/A	Não	Parcial
VII	Permitir configurar o conjunto de métricas e os ciclos de monitoramento das métricas em diferentes granularidades nas diversas camadas de recursos.	Sim	N/A	Não	Parcial
VIII	Fornecer fácil acesso ao ambiente de monitoramento através de uma API que encapsule a complexidade da comunicação e manipulação das informações trocadas.	Parcial	Não	Não	Não
IX	Possibilitar o consumo das métricas em função do tempo e quantidade de coletas.	Não	Não	Não	Não

Tabela 2.2 Análise comparativa dos trabalhos relacionados e requisitos.

Como é possível verificar na tabela apresentada, as propostas de *framework* existentes não atendem a todos requisitos que uma solução de monitoramento para DBaaS necessita. Desta maneira, este trabalho visa propor um *framework* que atenda a todos os requisitos levantados neste capítulo, optando-se por não estender as soluções atuais que não são open-source, diferentemente da nossa proposta. Assim como, o *framework* proposto neste trabalho visa colaborar com padrões para definição de métricas, procedimentos de coleta e armazenamento das informações monitoradas, que são considerados como questões em aberto pelos autores em [13]. Com base em nossa revisão da literatura, também não há nenhum *framework* com foco no monitoramento para DBaaS que permite a criação de um conjunto de métricas que vão desde infraestrutura física até as cargas de trabalho de bancos de dados.

2.6 CONCLUSÃO

Este capítulo apresentou uma introdução ao paradigma da computação em nuvem, destacando as principais características e modelos de serviço. Foram abordadas também as características dos serviços DBaaS, apresentando os principais aspectos envolvidos no monitoramento desses serviços. Algumas soluções existentes para monitoramento de serviços em nuvem também foram demonstradas, mas como soluções comerciais não são adaptáveis e extensíveis, como também não possuem o foco para o monitoramento de serviços DBaaS. Com as necessidades de monitoramento que um DBaaS precisa, esse capítulo apresentou uma lista de requisitos que as soluções de monitoramento para DBaaS devem possuir como base. Por fim, os principais trabalhos relacionados com o tema desta dissertação foram analisados com base nos requisitos elencados e foi verificado que não atendem a todos os requisitos que uma solução de monitoramento para DBaaS necessita.

CAPÍTULO 3

MyDBaaS: UM FRAMEWORK PARA O MONITORAMENTO DE SERVIÇOS DE BANCO DE DADOS EM NUVEM

Este capítulo descreve o MyDBaaS[1], um *framework* para o monitoramento de ambientes de banco de dados em nuvem cuja finalidade é possibilitar a criação de soluções de monitoramento personalizáveis e eficientes. O capítulo está estruturado da seguinte forma: na seção 4.1 é apresentado o conceito de *framework* de aplicação, em seguida na seção 4.2 é apresentado o *framework* MyDBaaS, onde são detalhados o modelo conceitual do monitoramento, a arquitetura, os diversos serviços existentes dentro do *framework* e informações sobre sua implementação e instanciação. Por fim, uma conclusão é realizada.

3.1 FRAMEWORK DE APLICAÇÃO

De acordo com [40], [41] e [42] um *framework* pode ser definido sob os aspectos de estrutura e propósito. Quanto à estrutura, um *framework* é a reutilização do design de parte ou totalidade de um sistema representado por classes abstratas e pela forma como instâncias destas classes interagem. Quanto ao propósito, um *framework* é o esqueleto de uma aplicação que pode ser personalizado por um desenvolvedor de aplicações. Em outras palavras, o propósito do *framework* é extrair e disponibilizar a essência de um determinado domínio de aplicações, de modo que o desenvolvedor possa partir de um patamar superior de implementação de uma solução [43] [44].

Segundo [45] e [46] um *framework* consiste de *frozen spots* e *hot spots*. Os *hot spots* de acordo com [47] são partes que baseiam-se em mecanismos de herança, os recursos existentes nessas partes são reutilizados e estendidos a partir da herança de classes abstratas/interfaces do *framework* e sobrecarga de métodos pré-definidos. Portanto essas partes são os pontos flexíveis onde os desenvolvedores usam para adicionar o seu código

[1] Site do *framework*: http://mydbaasmonitor.com/framework.html

para gerar as funcionalidades específicas da aplicação. Os *frozen spots* são partes fixas do *framework* como classes, serviços e funcionalidades já implementados que não são mutáveis e constituem o núcleo do *framework*. Segundo [46] os *frozen spots* definem a arquitetura geral de uma aplicação, isto é, seus componentes básicos e as relações entre eles que permanecem inalterados em qualquer instanciação do *framework*, e normalmente são responsáveis por invocarem os *hot spots* implementados.

3.2 MYDBAAS FRAMEWORK

Os requisitos elencados no Capítulo 2, serviram de base para a especificação deste *framework* de aplicação que permite o monitoramento de serviços em nuvem, mais especificamente para DBaaS. O MyDBaaS oferece uma programação abrangente e um modelo extensível para o desenvolvimento de estratégias para o monitoramento de serviços DBaaS, disponibilizando um conjunto de *frozen spots* que tornam a utilização fácil e transparente, e também um conjunto de *hots pots* que tornam sua estrutura extensível e flexível conforme necessário. O monitoramento de recursos na maior parte dos casos não é o objetivo final, mas um meio para alcançá-lo. Com isso o objetivo do *framework* é tornar fácil o desenvolvimento do monitoramento possibilitando o programador escrever menos código, porém mais objetivo com a sua necessidade. O processo de monitoramento baseia-se na paralelização de atividades, onde uma atividade se refere a coleta de uma métrica. Com isso as métricas são coletadas simultaneamente e independentemente uma das outras, possibilitando coletas em diferentes ciclos de monitoramento. O *framework* automatiza diversas funcionalidades como o gerenciamento da base serial histórica, envio e recebimento das métricas coletadas, consumo das métricas através da API e outras.

3.2.1 Modelo do Sistema

Com base no ambiente discutido no capítulo anterior, a Figura 3.1 apresenta a criação da camada de monitoramento pelo *framework* proposto sobre os recursos que compõem o serviço DBaaS, onde são coletadas simultaneamente as métricas sobre os servidores, máquinas virtuais, SGBDs, instâncias de banco de dados e cargas de trabalho. Nessa camada criada pelo *framework* o processo de monitoramento é realizado através de um monitor que existirá dentro de cada servidor e máquina virtual, possibilitando o monitoramento individual dos recursos. Nos servidores o monitor possui as funcionalidades

de coletar métricas sobre o estado da própria máquina física, *hypervisor* e informações externas das máquinas virtuais nela hospedadas. Nas máquinas virtuais o monitor possui as funcionalidades de coletar métricas sobre o estado interno da VM, métricas sobre os SGBDs hospedados na VM e métricas sobre as instâncias de banco de dados existentes. O monitor também existirá na camada dos clientes/aplicações para interceptar as cargas de trabalho enviadas as instâncias de banco de dados e poder coletar as métricas relacionadas. Para gerenciar o processo de monitoramento existe um coordenador, que dispõe das funcionalidades para coordenar os diversos monitores, para gerenciar os recursos no ambiente de monitoramento, para receber as métricas coletadas e armazenar em uma base. Com o processo de monitoramento em execução o *framework* dispõe de mais uma funcionalidade que possibilita o consumo das métricas coletadas em tempo real através de uma API. Esse conjunto de funcionalidades está dividida entre os módulos da arquitetura do *framework*, e são o que compõem os *frozen spots* e *hot spots*. Em seguida a arquitetura e as funcionalidades do *framework* são melhores detalhadas.

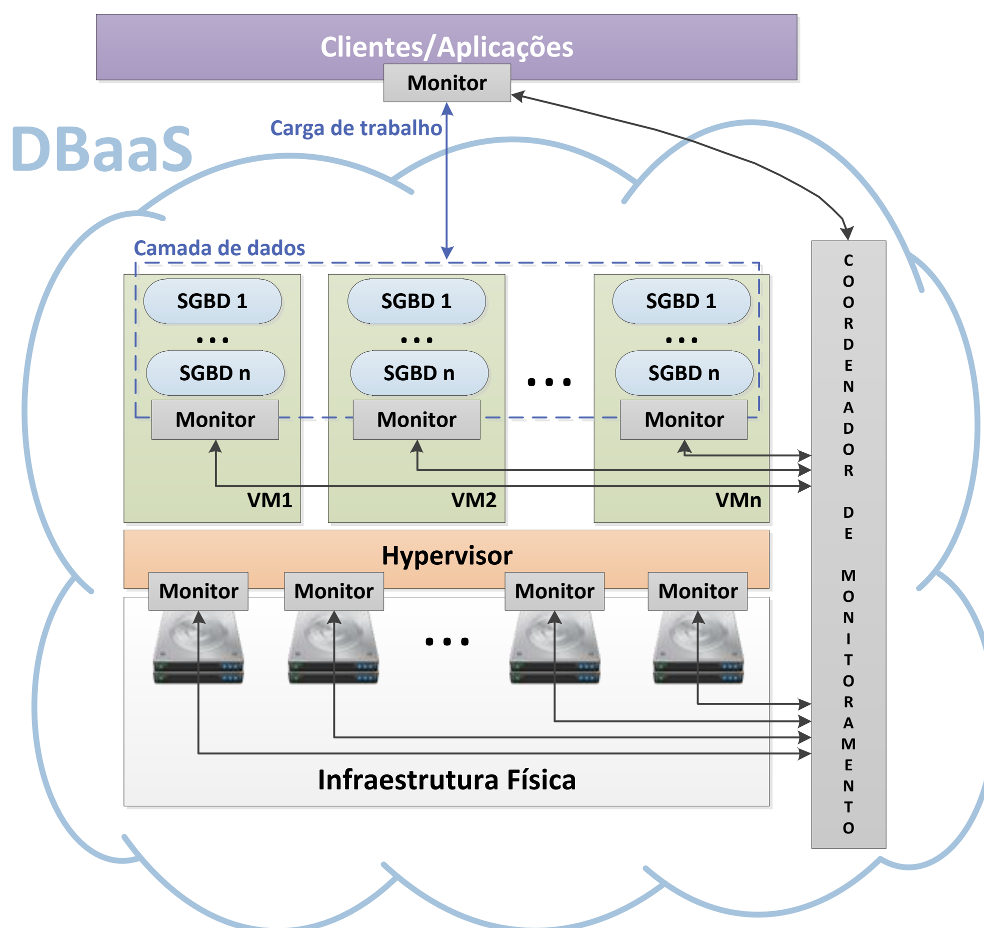


Figura 3.1 Ilustração do ambiente com o modelo de monitoramento.

3.2.2 Arquitetura

O *framework* MyDBaaS possui uma arquitetura baseada em módulos separados de acordo com suas funcionalidades: *MyDBaaS Core*, *MyDBaaS Agent*, *MyDBaaS Common*, *MyDBaaS API* e *MyDBaaS Driver*. A utilização de uma estratégia de modularização foi adotada, pois permite o desenvolvimento mais independente e extensível, assim como uma maior reutilização de código. Para cada módulo foram especificados componentes que interagem entre si e declaram padrões e procedimentos para a realização das respectivas funções. Com base no modelo utilizado, uma visão geral da arquitetura do *framework* pode ser observada na Figura 3.2. O módulo *MyDBaaS Common* não está presente nessa visão porque está intrínseco nos outros módulos, mas será melhor discutido adiante.

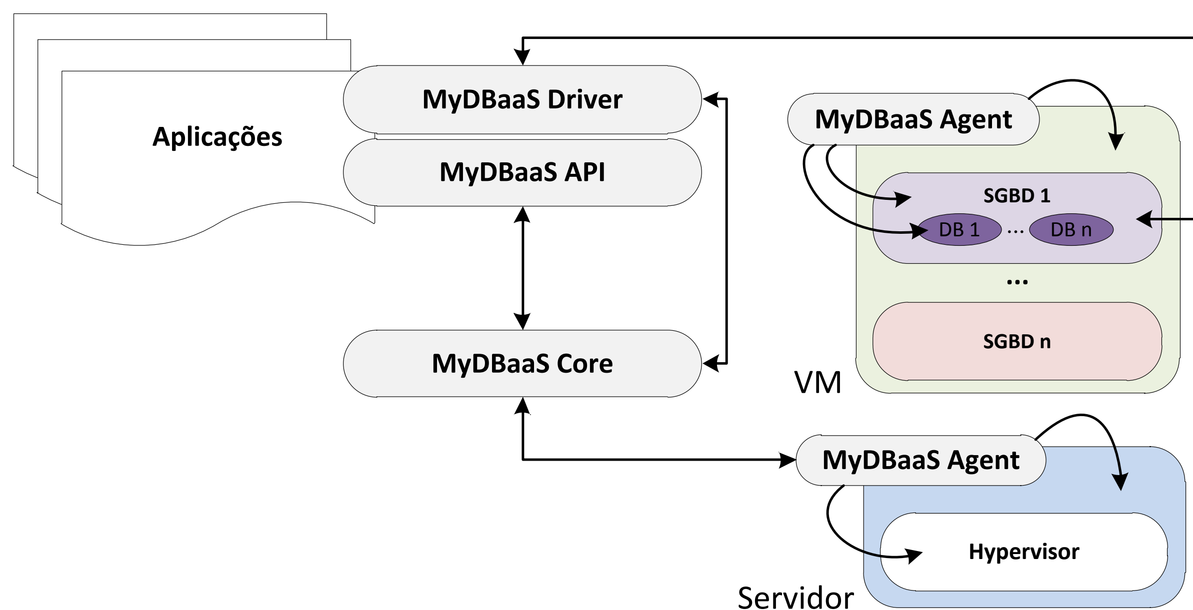


Figura 3.2 Arquitetura do *framework* MyDBaaS.

O *MyDBaaS Core* é o módulo central da arquitetura (coordenador), responsável por gerenciar os recursos cadastrados no ambiente de monitoramento, coordenar os agentes de monitoramento e manter a base serial histórica sobre métricas coletadas. O *MyDBaaS Agent* é o módulo que está presente nas VMs e servidores da nuvem (monitor), e é responsável por interagir com ambos, assim como os SGBDs e bancos de dados existentes em cada VM. Especificamente, o agente monitora, coleta e envia as diversas métricas para o *MyDBaaS Core*. O *MyDBaaS Common* é o mais simples, mas não menos importante, e é responsável por conter tudo o que é comum a todos os outros módulos,

como a especificação das métricas e tipos de recursos. O *MyDBaaS API* é o módulo que permite o acesso externo ao ambiente de monitoramento e responsável por prover um conjunto de rotinas que possibilita o consumo das métricas coletadas e acesso aos recursos cadastrados. O último módulo é o *MyDBaaS Driver* que é responsável por permitir a coleta de métricas no nível das cargas de trabalho enviadas aos SGBDs.

Cada módulo é composto por um conjunto de componentes, como detalhado na Figura 3.3. O módulo *MyDBaaS Core* é composto por quatro componentes: *Controller*, *MonitoringCoordinator*, *Repository* e *Serial Base*. O *Controller* é responsável pelo recebimento e gerenciamento das requisições enviadas pelos agentes de monitoramento e através da *API* e *Driver*. Também provê a estrutura para criação de novos controladores. O *MonitoringCoordinator* é o componente cuja função é acessar os recursos e configurar os agentes de monitoramento automaticamente. *Repository* é o principal componente, disponibilizando funcionalidades transparentes e automáticas para criação e manutenção da base serial histórica e para o armazenamento das métricas coletadas e recursos cadastrados. O componente *Serial Base* tem como função armazenar as informações geradas pelo *framework*, criando um histórico das coletas em função do tempo.

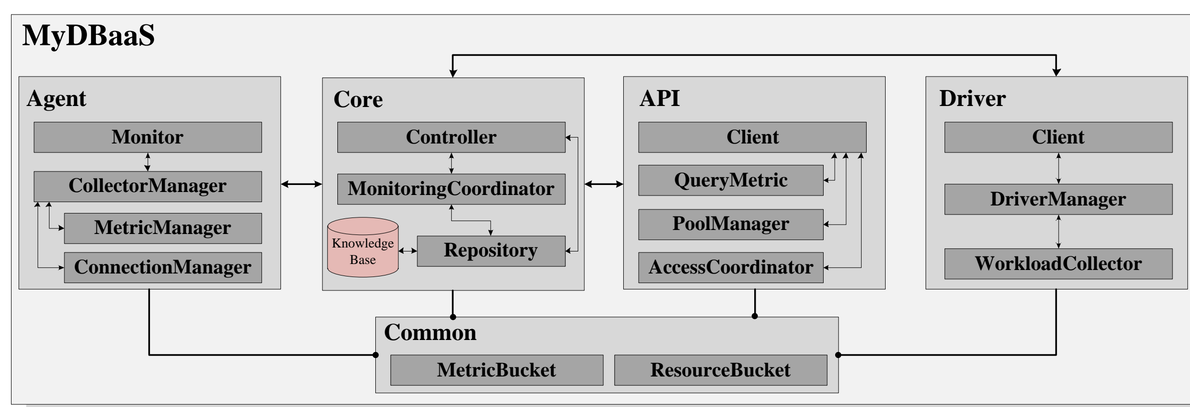


Figura 3.3 Detalhamento da arquitetura do *framework* MyDBaaS.

O módulo *MyDBaaS Agent* é composto de quatro componentes: *Monitor*, *CollectorManager*, *MetricManager* e *ConnectionManager*. O componente *Monitor* provê as funções para carregar as configurações do agente de monitoramento automaticamente, assim como métodos que possibilitam o agente autoconfigurar-se. O *CollectorManager* é o componente que provê a infraestrutura para coleta das métricas. Ele possibilita que as métricas sejam coletadas de forma independente e em ciclos diferentes. Também disponibiliza métodos dinâmicos e automáticos para o envio das métricas coletadas para o módulo *Core*. O *MetricManager* é responsável por carregar as configurações sobre as

métricas que o agente de monitoramento deverá utilizar, como ciclo de monitoramento, URL de envio e outras. O *ConnectionManager* é encarregado de prover a funcionalidade de acessar os bancos de dados e SGBDs automaticamente quando o agente precisar monitorar uma métrica nesses níveis. O módulo *MyDBaaS Common* é composto por dois componentes: *ResourceBucket* e *MetricBucket*. Ambos os componentes são responsáveis por disponibilizar o padrão para criação da representação de novas métricas e recursos.

O módulo *MyDBaaS API* é composto de quatro componentes: *Client*, *PoolManager*, *QueryMetric* e *AccessCoordinator*. O componente *Client* é responsável pela comunicação, fornecendo a estrutura para gerenciar a conexão com os componentes do módulo *MyDBaaS Core*. Ele também disponibiliza o acesso aos *pools* de recursos cadastrados no ambiente de monitoramento. O *PoolManager* é o componente que gerencia os diversos pools de recursos, permitindo o gerenciamento de cada recursos como, atualização, adição ou consulta. Ele também provê a estrutura para criação de novos gerenciadores. O *QueryMetric* é o principal componente desse módulo. Ele possibilita o consumo das métricas que estão sendo monitoradas e coletadas nas diversas camadas de recursos de forma automática e transparente. Esse consumo é feito através de métodos que permitem recuperar as métricas de diversas formas conforme a necessidade. O *AccessCoordinator* permite o gerenciamento dos agentes de monitoramento, possibilitando a parada do agente ou a inicialização de um novo agente. O módulo *MyDBaaS Driver* é composto de três componentes: *Client*, *DriverManager* e *QueryCollector*. O componente *Client*, assim como no módulo *MyDBaaS API*, é responsável pela comunicação e também disponibiliza o acesso aos SGBDs cadastrados no ambiente de monitoramento. O *DriverManager* é o componente que cria as conexões com os SGBDs e instâncias de banco de dados, e permite a execução das cargas de trabalho (consultas) nesses recursos. Ao executar uma consulta o componente *WorkloadCollector* é acionado, esse componente é responsável por coletar as métricas referentes a cargas de trabalho e enviar ao módulo *MyDBaaS Core* para serem armazenadas.

3.2.3 Definição de Métricas e Recursos

A definição das métricas e recursos são feitas através do módulo *MyDBaaS Common*. Esse módulo é constituído apenas por *hot spots*, os quais os desenvolvedores utilizam para criar a definição das métricas e recursos desejáveis, que serão invocados pelo *framework* através dos *frozen spots* nos outros módulos. Com base nesses pontos extensíveis é possível criar as

definições necessárias dos tipos de recursos e métricas. Essas definições são feitas através da implementação de classes que herdam os *hot spots* do módulo *Common*. O *framework* possui quatro definições de recursos já implementadas: servidor, máquina virtual, SGBD e instância de banco de dados. Alguns exemplos de métricas que podem ser definidas para esses recursos: para servidor e VM é possível definir métricas como CPU, memória, disco, rede e/ou informações de sistema operacional. Para servidor também podem ser definidas métricas sobre *hypervisor*, como por exemplo a quantidade de VMs hospedadas ou seus estados. Para SGBD e instância de banco de dados é possível definir métricas como CPU e memória utilizadas, quantidade de conexões abertas, tamanho da base, tempo de resposta, vazão, entre outras. É importante destacar que essas métricas são exemplos e que o *framework* não se limita somente a elas, mas a qualquer métrica que o desenvolvedor deseje implementar.

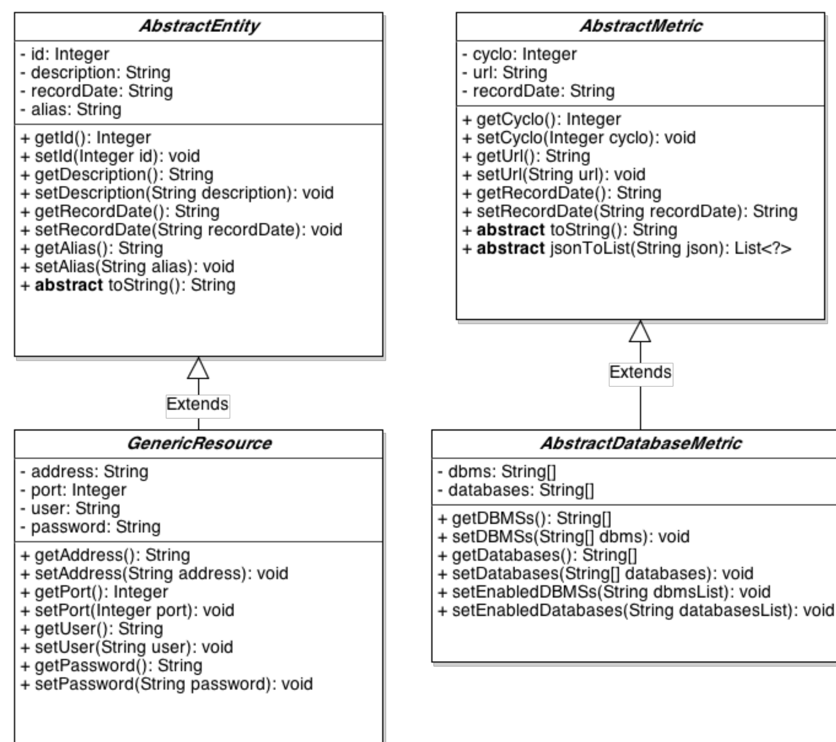


Figura 3.4 Diagrama de classes do módulo *MyDBaaS Common*.

A Figura 3.4 apresenta os *hot spots*, que são quatro classes abstratas requeridas para implementar as métricas e recursos que serão utilizados pelo *framework*. As classes *AbstractEntity* e *GenericResource* devem ser herdadas pelas implementações de novos tipos de recursos. A classe *AbstractDatabaseMetric* deve ser herdada pelas imple-

mentações de métricas para SGBD e instância de banco de dados. E por último a classe *AbstractMetric* que deve ser herdada por qualquer métrica que não seja para camada de dados, como por exemplo para servidor ou VM. O objetivo dessas classes é desacoplar o conjunto de atributos e métodos comuns aos diversos tipos de recursos e métricas, mas também permitir a identificação e utilização dos mesmos no ciclo de instanciação e execução do *framework*. A Figura 3.5 apresenta dois exemplos de definições de métricas.

```
public class Cpu extends AbstractMetric {

    private double cpuUser;
    private double cpuSystem;
    private double cpuNice;
    private double cpuWait;
    private double cpuIdle;
    private double cpuCombined;

    //Getters and Setters
}

public class Size extends AbstractDatabaseMetric {

    private double sizeUsed;

    //Getters e Setters
}
```

Figura 3.5 Exemplos de definições de métricas no módulo *MyDBaaS Common*

Uma métrica é uma entidade que pode ser avaliada. Cada métrica é composta por uma ou mais medidas que serão coletadas durante o monitoramento. Consequentemente, as métricas devem ser definidas e essas medidas identificadas. Essas medidas são representadas pelos atributos da classe, mas a classe pode conter outros atributos que não representam medidas. Então, para identificar quem são as medidas basta colocar o nome da classe na frente dos atributos que representam medidas. Esse padrão de nomenclatura é necessário porque permite a identificação da métrica e de suas medidas dentro de todo o *framework*, e isso é importante pois os *frozen spots* utilizam esse padrão de identificação para automatizar os serviços. Esse processo de definição e identificação das medidas é igual para toda métrica independente de qual camada ela pertence.

Ao estender os *hot spots* para definição de uma métrica dois métodos são herdados, o primeiro é o *toString()* e na sua implementação ele deve retornar o tipo de camada da métrica (*host*, *machine* e *database*). O segundo é o *jsonToList()* e na sua implementação ao receber um JSON ele deve converter para uma lista de objetos da métrica.

3.2.4 Serviço de Monitoramento

O serviço de monitoramento é realizado pelo módulo *MyDBaaS Agent*, que gerencia as métricas que devem ser coletadas. Este serviço monitora o estado sobre cada servidor, VM, SGBD e instância de banco de dados. Essas informações são coletadas e enviadas ao *MyDBaaS Core* para serem armazenadas e utilizadas posteriormente para controle de SLAs, garantia da qualidade de serviço, estratégias de elasticidade e replicação ou para outras tomadas de decisões.

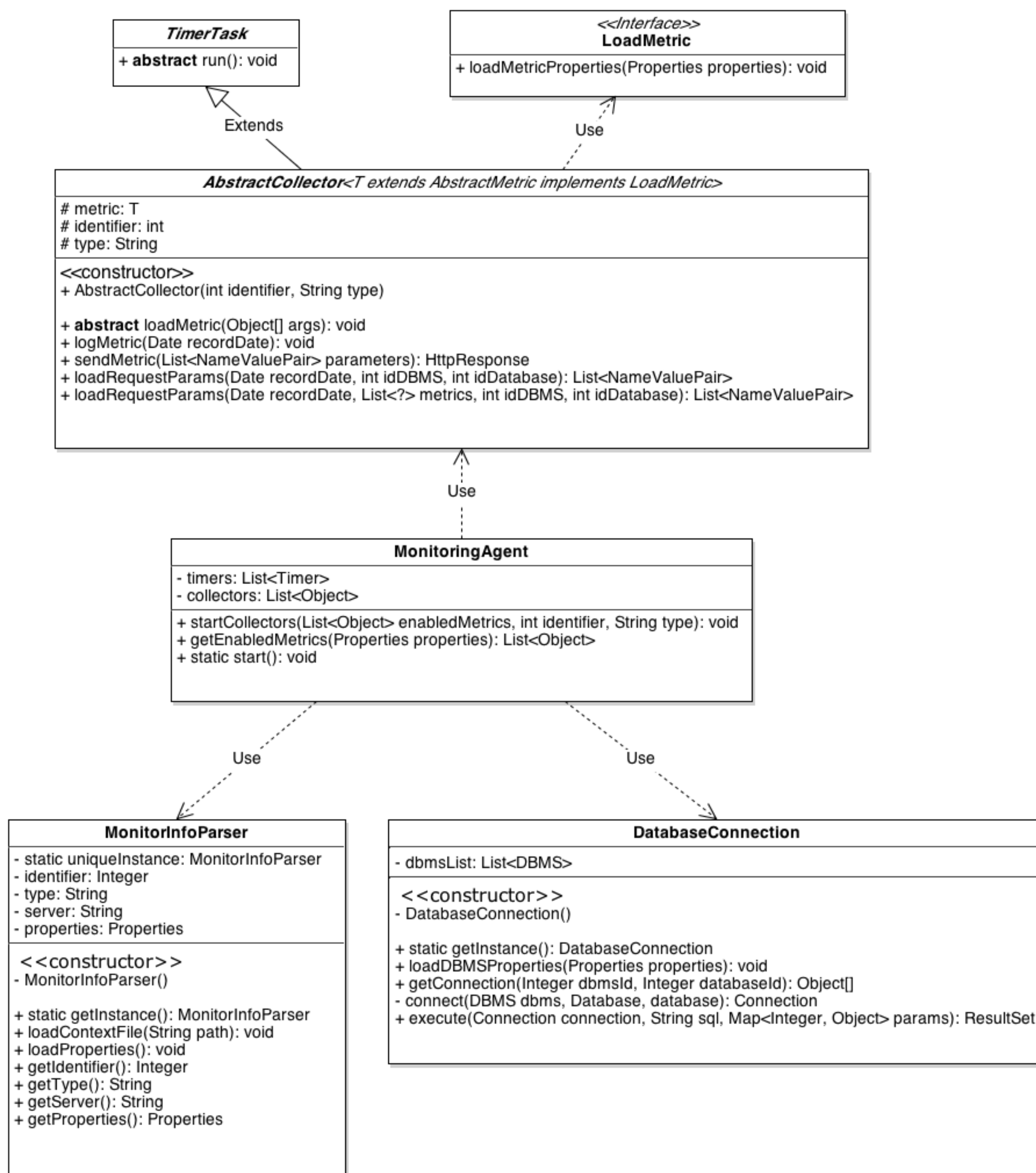


Figura 3.6 Diagrama de classes do módulo *MyDBaaS Agent*.

A criação do serviço de monitoramento é feita através da extensão de alguns *hot spots* e implementações de métodos herdados, que serão utilizados pelos *frozen spots* existentes dentro do módulo. Com essa estrutura disponibilizada o serviço de monitoramento das métricas é realizado de maneira independente e transparente uma das outras, o que possibilita que a coleta das métricas possuam diferentes ciclos de monitoramento.

Após a definição das métricas é necessário implementar seus coletores. Os coletores são motores que trabalham paralelamente realizando a coleta das métricas ativas. É importante destacar que além dos coletores poderem possuir diferentes ciclos de monitoramento, os agentes de monitoramento podem possuir diferentes conjuntos de métricas ativas a serem monitoradas. Na Figura 3.6 é possível visualizar os *hot spots* e *frozen spots* existentes no módulo *MyDBaaS Agent*. A classe *AbstractColletor* e a interface *LoadMetric* são os dois *hot spots* do módulo. A interface *LoadMetric* deve ser implementada utilizando as métricas que foram definidas no módulo *MyDBaaS Common*, esse *hot spot* é responsável pela estrutura que deve ser implementada para carregar as configurações que o agente precisa conhecer sobre a métrica. Um exemplo da implementação da interface *LoadMetric* é demonstrado na Figura 3.7. Essa implementação é realizada através da criação de uma classe dentro do componente *MetricManager* do módulo *MyDBaaS Agent*. A nova classe deve estender a métrica que ela se refere e implementar a interface. Por padrão de nomenclatura a classe deve ter o nome da métrica concatenado a palavra *Metric*. Com a classe criada é importante utilizar o padrão *Singleton*, isso é necessário porque cada métrica possui apenas um objeto instanciado enquanto o agente estiver ativo. Assim, menos memória será utilizada e menor será o *overhead*. No método *loadMetricProperties()* são onde as propriedades de configuração da métrica, como URL e ciclo, são carregadas. O desenvolvedor pode configurar novas propriedades caso necessário.

A classe abstrata *AbstractColletor* é o *hot spot* que deve ser estendido pela classe que define um coletor. Ela disponibiliza os métodos que deverão ser implementados por cada coletor, como qual processo será realizado para coletar a determinada métrica e qual a sequência de passos que deverá ser executada. Ela também disponibiliza funções que automatizam e tornam transparente para o desenvolvedor os processos de criação dos parâmetros da requisição, de envio da métrica ao módulo *MyDBaaS Core* e de efetuar o log caso o envio não seja bem-sucedido. Assim como a definição das métricas, para definir os coletores é preciso seguir um padrão de nomenclatura. A Figura 3.8 apresenta um exemplo da implementação dessa classe abstrata. Essa implementação é realizada

```
public class SizeMetric extends Size implements LoadMetric {

    private static SizeMetric uniqueInstance;

    private SizeMetric() {}

    public static SizeMetric getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new SizeMetric();
        }
        return uniqueInstance;
    }

    @Override
    public void loadMetricProperties(Properties properties) {
        // TODO Auto-generated method stub
    }
}
```

Figura 3.7 Exemplo da implementação da interface *LoadMetric*.

através da criação de uma classe dentro do componente *CollectorManager* do módulo *MyDBaaS Agent*. A nova classe deve estender a classe abstrata e passar como parâmetro uma classe que implementou a interface *LoadMetric*, como demonstrado anteriormente. Por padrão de nomenclatura o novo coletor deve ter como nome o nome da métrica que ele se refere concatenado a palavra *Collector*. Com a classe criada três métodos são herdados, o primeiro é o construtor padrão dos coletores que será utilizado pelo componente *Monitor* para instanciar o coletor. O segundo é o método *loadMetric()* que é responsável por coletar as medidas da métrica, onde deve ser implementado como a nova métrica deverá ser coletada - podendo utilizar chamadas a outros aplicativos, bibliotecas ou serviços. O terceiro é o método *run()* responsável pelo fluxo que o coletor realizará durante o ciclo de monitoramento.

```
public class SizeCollector extends AbstractCollector<SizeMetric> {

    public SizeCollector(int identifier, String type) {
        super(identifier, type);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void loadMetric(Object[] args) throws Exception {
        // TODO Auto-generated method stub
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
    }
}
```

Figura 3.8 Exemplo da extensão da classe abstrata *AbstractCollector*.

O método *run()* é invocado sempre que o coletor entra em um novo ciclo de monitoramento, a implementação desse método deve conter as chamadas para as atividades que deverão ser executadas durante o processo de monitoramento. A Figura 3.9 apresenta um exemplo da implementação desse método - por padrão três chamadas devem ser realizadas: a primeira chamada é o método *loadMetric()* que coletará a métrica, a segunda chamada é o método *loadRequestParams()* que montará a requisição da métrica coletada e a terceira chamada é o método *sendMetric()* que enviará ao *receiver* (seção 3.2.5) responsável a requisição com a nova coleta para ser armazenada na base serial histórica.

```
@Override
public void run() {
    this.loadMetric(new Object[] {...});

    List<NameValuePair> parameters = null;
    parameters = this.loadRequestParams(...);

    HttpResponse response;
    response = this.sendMetric(parameters);
}
```

Figura 3.9 Exemplo abstrato da implementação do método *run* de um coletor.

Arquivo de Contexto

O arquivo de contexto é responsável por conter as propriedades de configuração que um agente de monitoramento deverá executar em um recurso. Este arquivo é carregado pelo agente de monitoramento que a partir das informações contidas nele, configura-se automaticamente. As informações são representadas em uma estrutura de parâmetros e valores, na forma *nome* = *valor*. Essas informações são compostas por URL de acesso ao módulo *MyDBaaS Core*, identificador do recurso, tipo do recurso e as configurações do conjunto de métricas que deverão ser coletadas. A Figura 3.10 apresenta os três primeiros parâmetros que são fixos em todos os arquivos de contexto.

```
# Unique code to identify the resource on the server
identifier =

# Resource type
type =

# Server URL
server =
```

Figura 3.10 Primeiros parâmetros do arquivo de contexto.

O parâmetro *identifier* deve conter como valor o código único que identifica o recurso na base serial histórica, o parâmetro *type* deve conter como valor o tipo do recurso onde o agente de monitoramento está (*machine* ou *host*) e por último o parâmetro *server* deve conter como valor a URL padrão de acesso ao módulo *MyDBaaS Core*. A Figura 3.11 apresenta exemplos da definição dos parâmetros para o conjunto de métricas que serão coletadas por um agente de monitoramento.

```
# Example: metric of virtual machine
cpu.url =
cpu.cycle =

# Example: metric of host
hostDomains.url =
hostDomains.cycle =

# Example: metric of DBMS/database instance
size.url =
size.cycle =
size.dbms =
size.databases =
```

Figura 3.11 Exemplos de parâmetros do arquivo de contexto para os tipos de métricas.

As métricas para as camadas de máquina virtual e servidor seguem o mesmo padrão. Cada métrica deverá conter dois parâmetros no arquivo de contexto: *.url* e *.cycle*. Esses nomes devem ser concatenados com o nome da classe que foi criada para representar a métrica. O primeiro parâmetro deve conter como valor a URL que identifica o *receiver* da determinada métrica, o valor deste parâmetro é concatenado com o parâmetro *server* para formar a URL final de acesso. O segundo deve conter como valor o ciclo de monitoramento da métrica em segundos, que representa a janela de tempo em que o agente deverá efetuar a coleta. No caso das métricas para as camadas de SGBD e instância de banco de dados é possível adicionar dois novos parâmetros: *.dbms* e *.databases*, eles também devem ser concatenados com o nome da classe que define a métrica. A presença desses parâmetros indica que a métrica será coletadas nessas respectivas camadas. Esses parâmetros devem conter como valores os códigos que identificam os SGBDs/instâncias de banco de dados que deverão ser monitorados, os códigos devem ser separados por vírgula.

Inicialização do Agente de Monitoramento

Como destacado anteriormente os coletores das métricas são independentes uns dos outros e funcionam paralelamente. Uma vez que o agente de monitoramento foi ini-

cializado, um coletor foi disparado para cada métrica como uma atividade agendada à ser executada repetidamente em um espaço de tempo definido pelo ciclo de monitoramento da métrica. Apesar do paralelismo existir para garantir que um coletor não interfira na execução do outro, o fluxo realizado por eles durante o processo de coleta das métricas é o mesmo. A Figura 3.12 apresenta o diagrama de atividade do coletor durante o fluxo do ciclo de monitoramento. Logo após o disparo de um coletor, o mesmo efetua um primeiro ciclo de monitoramento. O fluxo é composto pela inicialização do coletor que coleta os dados da métrica, cria a requisição da nova coleta e realiza o envio da mesma ao módulo *MyDBaaS Core*. Uma resposta é retornada e caso o envio não seja bem-sucedido o coletor cria um log da coleta efetuada para que a mesma não seja perdida. E por fim, o coletor para e aguarda a chegada do próximo ciclo de monitoramento.

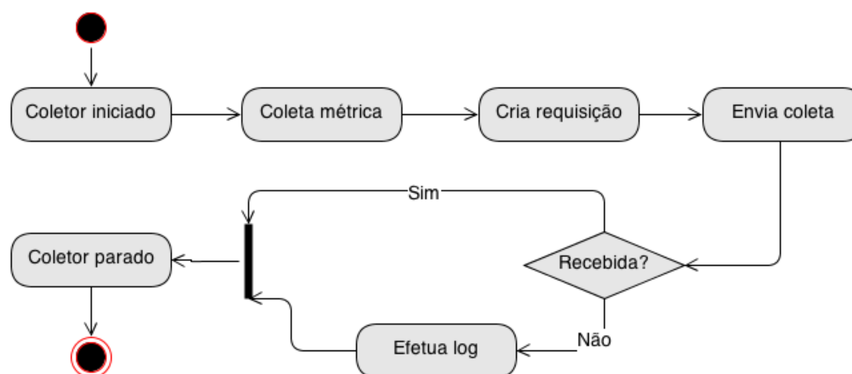


Figura 3.12 Diagrama de atividade do ciclo de monitoramento de um coletor.

A Figura 3.13 apresenta o diagrama de sequência da inicialização do agente de monitoramento. Após a inicialização o agente realiza um processo de autoconfiguração com base nas informações passadas como entrada. Essas informações de configuração são passadas através de um arquivo de contexto que contém quais métricas devem ser coletadas, quais seus ciclos de monitoramento e os dados de acesso ao módulo *MyDBaaS Core*. O processo de autoconfiguração é realizado pelos *frozen spots* existentes no componente *Monitor*. A classe *MonitoringAgent* é o centro do agente de monitoramento, ela é responsável por disparar as várias tarefas de autoconfiguração. O primeiro passo é receber o arquivo de contexto e carregar as informações dos dados de acesso ao módulo *MyDBaaS Core* através da classe *MonitorInfoParser* que possui os serviços já implementados para gerenciar o arquivo de contexto e carregar as informações. Com o arquivo de contexto carregado uma segunda classe entre em ação, a *DatabaseConnection*, que é responsável por gerenciar todo o acesso necessário entre o agente de monitoramento e os SGBDs e

instâncias de banco de dados. Essa classe também fornece serviços já implementados que tornam transparente aos coletores do agente de monitoramento a criação de conexões com os recursos e execução de consultas sobre métricas. Após a classe *MonitoringAgent* realizar essas duas chamadas externas ela executa os dois últimos serviços do processo de autoconfiguração, um que instância uma lista com as métricas que foram ativas para o agente de monitoramento e o segundo que a partir dessa lista inicializa os coletores definidos para cada métrica.

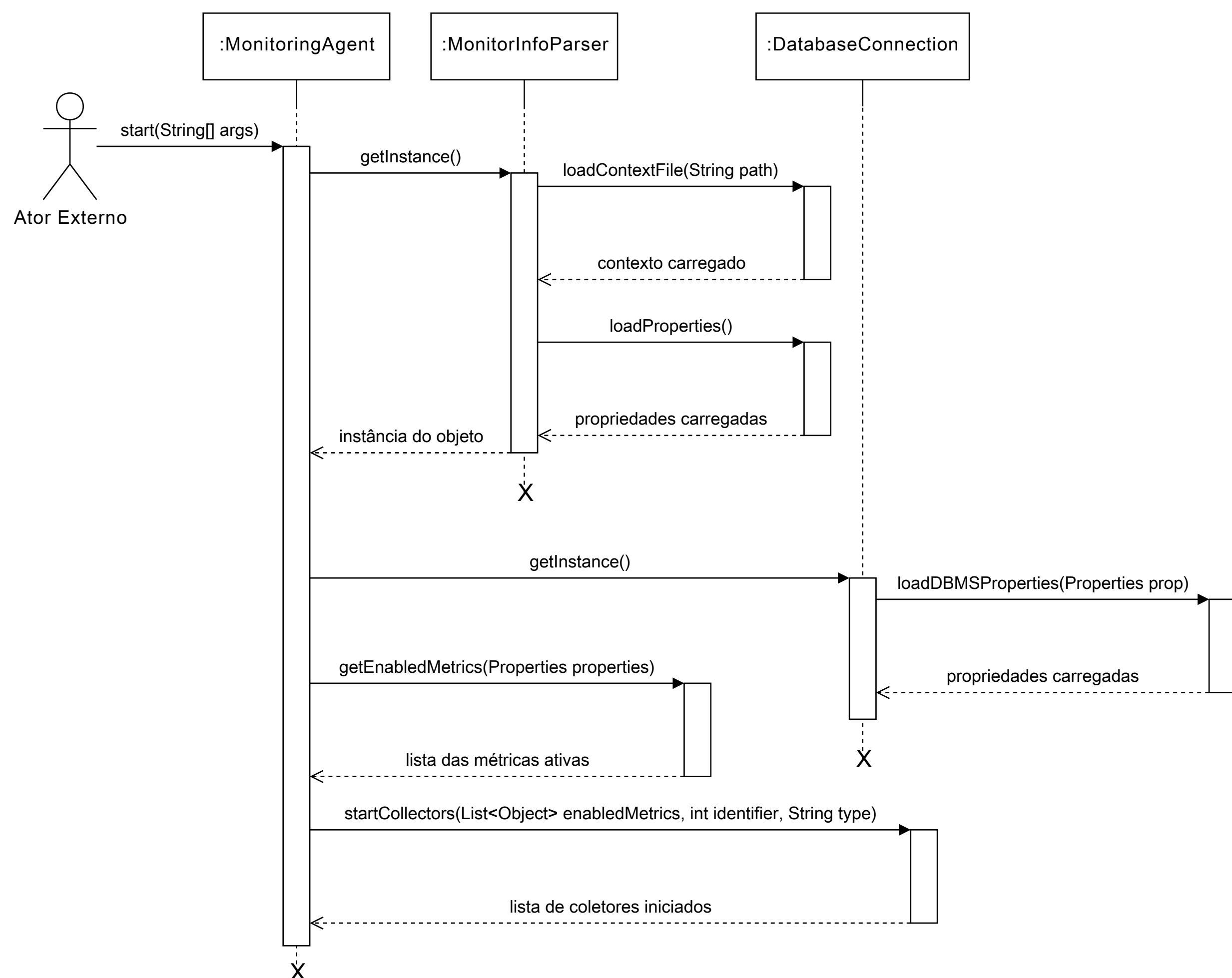


Figura 3.13 Diagrama de sequência de inicialização do agente de monitoramento.

Monitoramento da Carga de Trabalho

O monitoramento da carga de trabalho é realizada pelo módulo *MyDBaaS Driver*, que possibilitada a interceptação das consultas enviadas aos SGBDs presentes no ambiente de monitoramento. Com isso, o serviço permite que métricas possam ser coletadas nesse

nível de informação. Assim como as métricas coletadas nos outros níveis, as métricas sobre carga de trabalho após coletadas também são enviadas ao módulo *MyDBaaS Core* para serem armazenadas na base serial histórica.

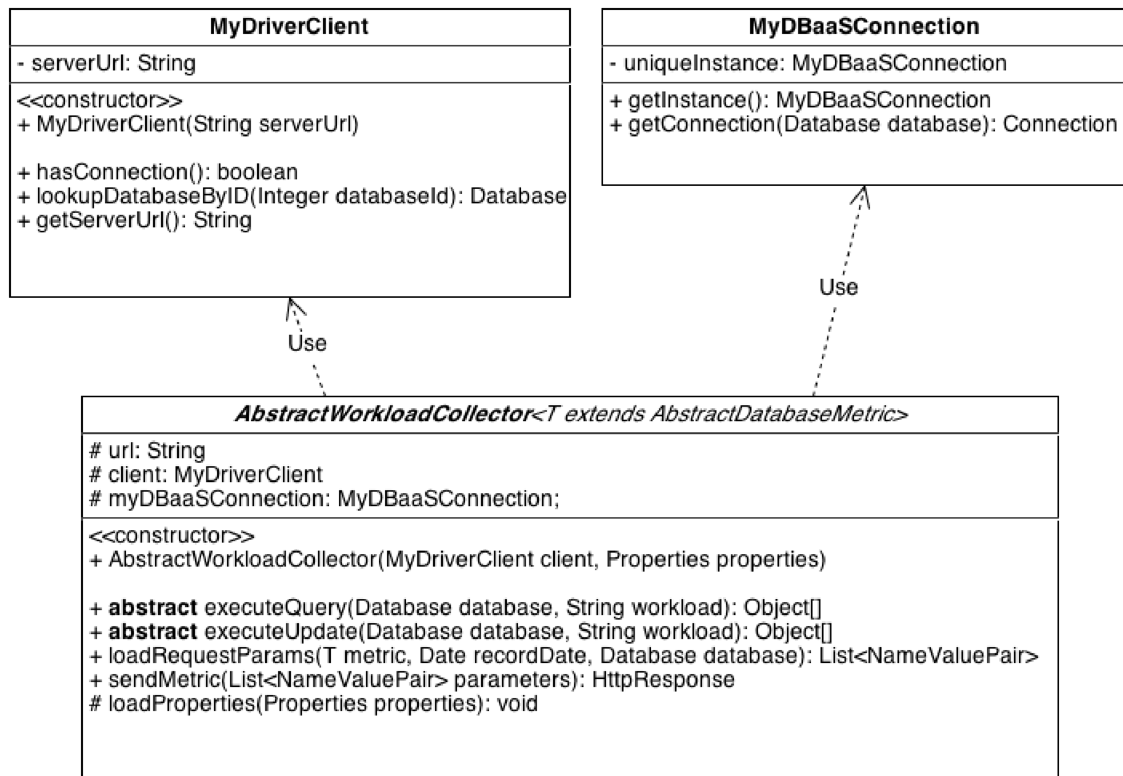


Figura 3.14 Diagrama de classes do módulo *MyDBaaS Driver*.

Na Figura 3.14 apresenta o diagrama das classes e métodos que compõem o módulo *MyDBaaS Driver*. As classes *MyDriverClient* e *MyDBaaSConnection* são os *frozen spots* disponibilizados pelo módulo. A classe *MyDriverClient* representa o componente *Client*, sendo responsável por criar a conexão com o módulo *MyDBaaS Core* e os métodos para testar a conexão e consultar as instâncias de banco de dados disponíveis no ambiente de monitoramento. A classe *MyDBaaSConnection* representa o componente *DriverManager*, sendo responsável por disponibilizar a conexão com a camada de dados. A classe abstrata *AbstractWorkloadCollector* é o *hot spot* do módulo, é responsável por prover a estrutura para implementação do componente *WorkloadCollector*. Essa classe disponibiliza os métodos que deverão ser implementados para realização da coleta da métrica referente a carga de trabalho. Ela também disponibiliza as funções que tornam automáticos os processos de criação dos parâmetros da requisição e de envio da métrica coletada ao *MyDBaaS Core*. A Figura 3.15 apresenta um exemplo da implementação

dessa classe abstrata.

```
public class WorkloadStatusCollector extends AbstractWorkloadCollector<WorkloadStatus> {

    public WorkloadStatusCollector(MyDriverClient client, Properties properties) {
        super(client, properties);
        // TODO Auto-generated constructor stub
    }

    @Override
    public Object[] executeQuery(Database database, String workload)
            throws ClassNotFoundException, SQLException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Object[] executeUpdate(Database database, String workload)
            throws ClassNotFoundException, SQLException {
        // TODO Auto-generated method stub
        return null;
    }
}
```

Figura 3.15 Exemplo da extensão da classe abstrata *AbstractWorkloadCollector*.

Essa implementação é realizada através da criação de uma classe que deve estender a classe abstrata e passar como parâmetro uma classe de métrica definida para carga de trabalho que implementou a classe abstrata *AbstractDatabaseMetric*, como demonstrado anteriormente. Por padrão de nomenclatura essa nova classe deve ter como nome o nome da métrica que ele se refere concatenado a palavra *Collector*. Com a classe criada três métodos são herdados, o primeiro é o construtor padrão para o componente *WorkloadColletor*. O segundo método é o *executeQuery()*, onde deve ser implementado como a métrica deverá ser coletada para cargas de trabalho de consultas do tipo *SELECT*. O terceiro método é o *executeUpdate()*, onde deve ser implementado como a métrica deverá ser coletada para cargas de trabalho de consultas do tipo *INSERT*, *UPDATE* e *DELETE*. Esses dois últimos devem ser utilizados para disparar as cargas de trabalho à camada de dados, eles recebem por parâmetro qual a consulta e em qual instância de banco de dados ela deve ser executada. Com isso, na implementação desses métodos é possível coletar a métrica sobre carga de trabalho pela interceptação do que está sendo enviado para camada de dados. Com a métrica coletada basta invocar as funções *loadRequestParams()* e *sendMetric()* para enviar a coleta da métrica para ser armazenada na base serial histórica.

3.2.5 Serviço de Recebimento das Métricas

O serviço de recebimento das métricas coletadas pelos agentes de monitoramento é realizado pelo módulo *MyDBaaS Core*, mais especificamente pelo componente *Controller*. Como destacado anteriormente esse componente é responsável por gerenciar todas as requisições enviadas pelos agentes de monitoramento e pelos módulos *MyDBaaS API* e *MyDBaaS Driver*. Essas requisições não são somente sobre o contexto das métricas, mas também podem ser sobre os recursos. Para tratar do recebimento das métricas coletadas pelos agentes de monitoramento existe um subcomponente dentro do componente *Controller* chamado de *ReceiverManager*. Esse subcomponente pode ser composto por vários *receivers*.

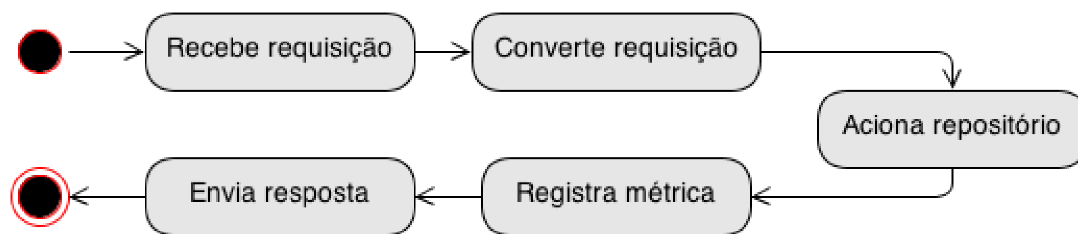


Figura 3.16 Diagrama de atividade do fluxo de recebimento de uma métrica.

Um *receiver* pode ser definido como um recepcionista que foi implementado para receber requisições de um conjunto de métricas de uma determinada camada e direcioná-las ao componente *Repository* para serem armazenadas na base serial histórica. O fluxo de recebimento das métricas realizado por um *receiver* é simples e transparente, e pode ser visto na Figura 3.16. Depois que o *framework* é instanciado os recepcionistas que foram implementados ficam atentos a chegada de novas requisições, quando uma requisição é enviada o *receiver* responsável a recebe, depois converte os parâmetros da requisição em um objeto da classe que define a métrica recebida, feito isso ele aciona o componente *Repository* e passa a nova métrica coletada para que seja salva na base serial histórica, e por fim envia uma resposta confirmando o recebimento.

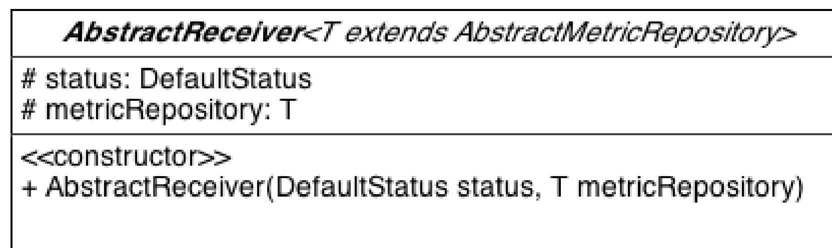


Figura 3.17 Diagrama de classe do componente *Controller* para *Receiver*.

A criação de um *receiver* é feita através da extensão do *hotspot* apresentado na Figura 3.17, que é a classe abstrata *AbstractReceiver*. Ela disponibiliza o acesso ao componente *Repository* e também a funcionalidade que possibilita o envio de uma resposta para quem realizou a requisição. A Figura 3.18 apresenta um exemplo da implementação dessa classe abstrata. Essa implementação é realizada através da criação de uma classe dentro do subcomponente *ReceiverManager*. Esta classe deve estender a classe abstrata e passar como parâmetro uma classe que implementou a classe abstrata *AbstractMetricRepository* (apresentada na subseção a seguir), com isso o construtor padrão dos *receivers* é herdado. Com a classe implementada é necessário utilizar as anotações *@Resource*[2] e *@Path*[3] do *framework* VRaptor - informações sobre esse *framework* são apresentadas na seção 3.2.7. Na anotação *@Path* é necessário passar como parâmetro o valor da URL, esse valor por padrão de nomenclatura deve ser representado pelo tipo de métrica a qual o *receiver* é responsável por gerenciar o recebimento.

```
@Resource
@Path("/storage")
public class StorageReceiverController extends AbstractReceiver<MetricRepository> {

    public StorageReceiverController(DefaultStatus status, MetricRepository metricRepository) {
        super(status, metricRepository);
        // TODO Auto-generated constructor stub
    }
}
```

Figura 3.18 Exemplo da extensão da classe abstrata *AbstractReceiver*.

Como destacado anteriormente um *receiver* recebe requisições de um conjunto de métricas, portanto para cada métrica é necessário criar um método que será responsável pelo recebimento da coleta da determinada métrica e por padrão de nomenclatura o

[2]Essa anotação disponibiliza o acesso ao *receiver* pelos agentes de monitoramento, a utilização dela torna todos os métodos públicos da classe acessíveis através de chamadas HTTP as URIs especificadas.
[3]Essa anotação permite a customização da URL de acesso ao *receiver*.

nome do método deve possuir o mesmo nome da métrica que ele representa. A Figura 3.19 apresenta exemplos da implementação desses métodos para os diferentes tipos de métricas. Esses métodos possuem variações de acordo com a camada que as métricas pertencem. Por padrão todos os métodos devem conter um parâmetro que representa qual métrica o método é responsável por receber, o nome desse parâmetro deve ser *metric*. O segundo parâmetro é uma string que representa o exato momento em que a métrica foi coletada, o nome desse parâmetro deve ser *recordData*. O último parâmetro é o recurso a que a métrica pertence e deve ser do tipo inteiro. O nome desse parâmetro varia com o tipo da métrica, se a métrica é sobre máquina virtual o nome deve ser *machine* e se for sobre servidor o nome deve ser *host*. Caso a métrica seja referente a SGBD/instância de banco de dados é necessário adicionar dois campos inteiros, *dbms* e *database*, isso acontece porque esse tipo de métrica pode ser coletada em ambos os níveis de recursos.

```
@Post("/memory")
public void memory(Memory metric, int machine, String recordDate) {
    if (repository.saveMetric(metric, recordDate, machine, 0, 0, 0)) {
        status.accepted();
    }
}
@Post("/hostdomains")
public void hostDomains(HostDomains metric, int host, String recordDate) {
    if (repository.saveMetric(metric, recordDate, 0, host, 0, 0)) {
        status.accepted();
    }
}
@Post("/size")
public void size(Size metric, int dbms, int database, String recordDate) {
    if (repository.saveMetric(metric, recordDate, 0, 0, dbms, database)) {
        status.accepted();
    }
}
```

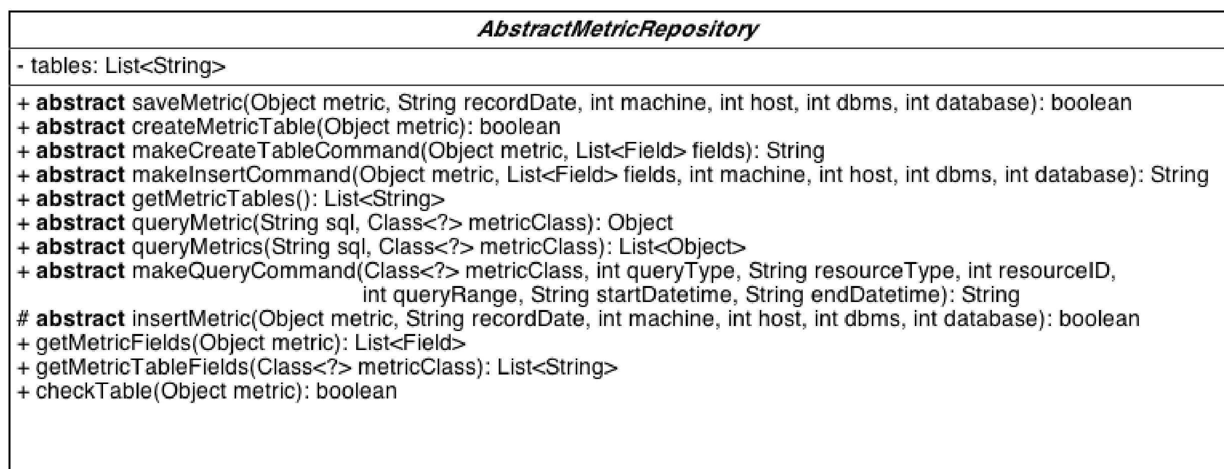
Figura 3.19 Exemplos da implementação dos métodos de recebimento das métricas.

Com os métodos responsáveis pelo recebimento das métricas criados, é necessário anotá-los com a anotação *@Post*[4] e o valor do parâmetro por padrão do *framework* deve ser o nome da métrica a qual o método é responsável. A implementação dos métodos é simples, basta chamar o objeto herdado do componente *Repository* e utilizar o método *saveMetric()* para armazenar a métrica recebida na base serial histórica. Após o armazenamento da métrica é necessário avisar ao coletor que enviou a requisição que a mesma foi aceita, para isso basta utilizar o método *accepted()*.

[4]Essa anotação assim como o *@Path*, permite a customização da URL de acesso ao serviço, mas limita esse acesso somente via requisições HTTP do tipo POST.

Base Serial Histórica

As métricas enviadas são armazenadas na base serial histórica dentro do módulo *MyDBaaS Core*. Essa base é gerenciada pelo componente *Repository* e continuamente atualizada em função do tempo e de novas coletas. Esse componente é responsável por tornar totalmente transparente ao desenvolvedor o acesso a base serial histórica. Ele provê os métodos para salvar e consultar dados na base serial histórica independentemente de qual seja a métrica. Isso é possível porque as métricas são definidas a partir do módulo *Common* como explicado anteriormente e também pelo padrão de nomenclaturas utilizado nas definições do *framework*.



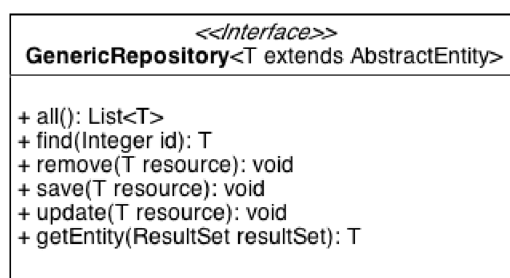


Figura 3.20 Diagrama de classes do componente *Repository*.

O componente *Repository* disponibiliza dois *hot spots* que podem ser estendidos para gerar implementações de novas versões desse componente, como representado na Figura 3.20. O *framework* já possui uma versão implementada do *Repository* para criação e gerenciamento da base serial histórica em SGBDs relacionais. É importante destacar que esse componente pode utilizar banco de dados não relacionais, como MongoDB[5],

[5]http://www.mongodb.org

Cassandra[6] ou outros. Essa instanciação já disponibilizada desse componente é composta por uma extensão da classe abstrata *AbstractMetricRepository* que é responsável por toda persistência das métricas e uma extensão da interface *GenericRepository* para cada definição de recurso já existentes no *framework*, que são responsáveis por toda persistência dos recursos.

A versão já implementada para SGBDs relacionais do *Repository* possui uma estrutura inicial de tabelas quando o *framework* é instanciado. Essas tabelas iniciais representam as camadas de recursos já definidas e suas relações, como discutido na seção 3.2.1. Com isso, uma tabela é criada para representar e salvar as informações sobre servidores, máquinas virtuais, SGBDs e instâncias de banco de dados que serão monitorados, como apresentado na Figura 3.21. A estrutura é expandida de acordo com as métricas definidas e coletadas, então para cada métrica uma tabela é criada conforme a sua definição e relacionada a tabela de recurso em que a mesma é coletada.

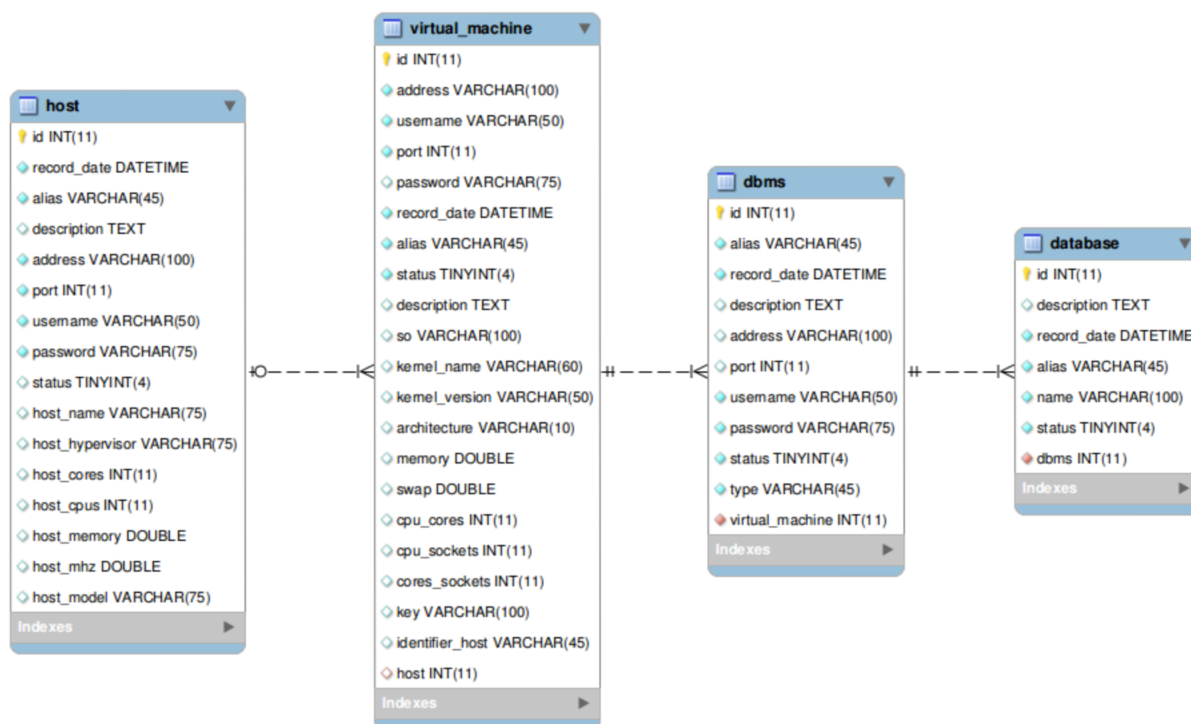


Figura 3.21 Estrutura inicial da base serial histórica.

A Figura 3.22 apresenta uma abstração do processo que um *Repository* efetua quando é requisitado para registrar uma nova métrica coletada. Ao receber uma métrica

[6]http://cassandra.apache.org

é verificado se a tabela que armazena aquele tipo de dado existe na base serial histórica. Essa verificação não acessa a base toda vez que uma nova métrica chega, o *Repository* mantém em memória uma lista com o nome das tabelas existentes sobre métricas, tornando o acesso mais rápido. Caso a tabela não exista o componente cria a tabela que irá representar a métrica na base serial histórica. Por fim a métrica é salva e uma confirmação é retornada para o *receiver*.

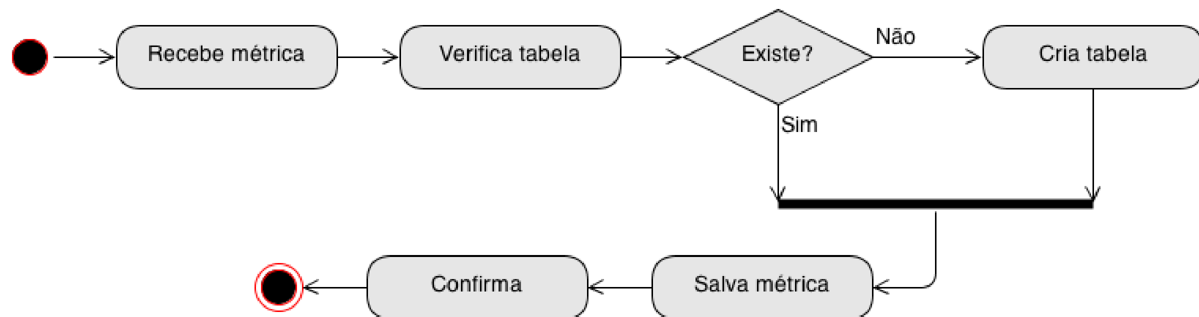


Figura 3.22 Diagrama de atividade do fluxo de registro de uma coleta.

3.2.6 Serviço de Consumo das Métricas

O serviço de consumo de métricas é responsável por permitir que aplicações possam consultar as métricas que estão sendo coletadas sobre os recursos cadastrados no ambiente de monitoramento e utilizá-las para diversos objetivos, tais como: controle de SLAs, gerenciamento da qualidade de serviço, estratégias de replicação e elasticidade, ou outras decisões que sejam necessárias. O serviço permite realizar o consumo das métricas de diversas maneiras pelas aplicações, possibilitando um consumo flexível, como por exemplo: consultar a última coleta de uma métrica, consultar todas as coletas, consultar as coletas a partir de uma data ou até uma data ou consultar as coletas entre duas datas. Esse serviço é realizado pelos módulos *MyDBaaS API* e *MyDBaaS Core*, que terão as suas participações melhores detalhadas em seguida.

MyDBaaS API

O módulo *MyDBaaS API* é uma interface de programação, cuja a função é estar presente dentro das aplicações e possibilitar que os desenvolvedores construam soluções que interagem diretamente com o ambiente de monitoramento criado pelo *framework*. Este módulo possibilita o acesso transparente a todos os dados das métricas que estão sendo monitoradas em tempo real, assim como ao histórico de tudo que já foi coletado.

É importante destacar que o módulo *MyDBaaS API* além de possibilitar o consumo das métricas, permite o gerenciamento dos recursos cadastrados no ambiente de monitoramento.

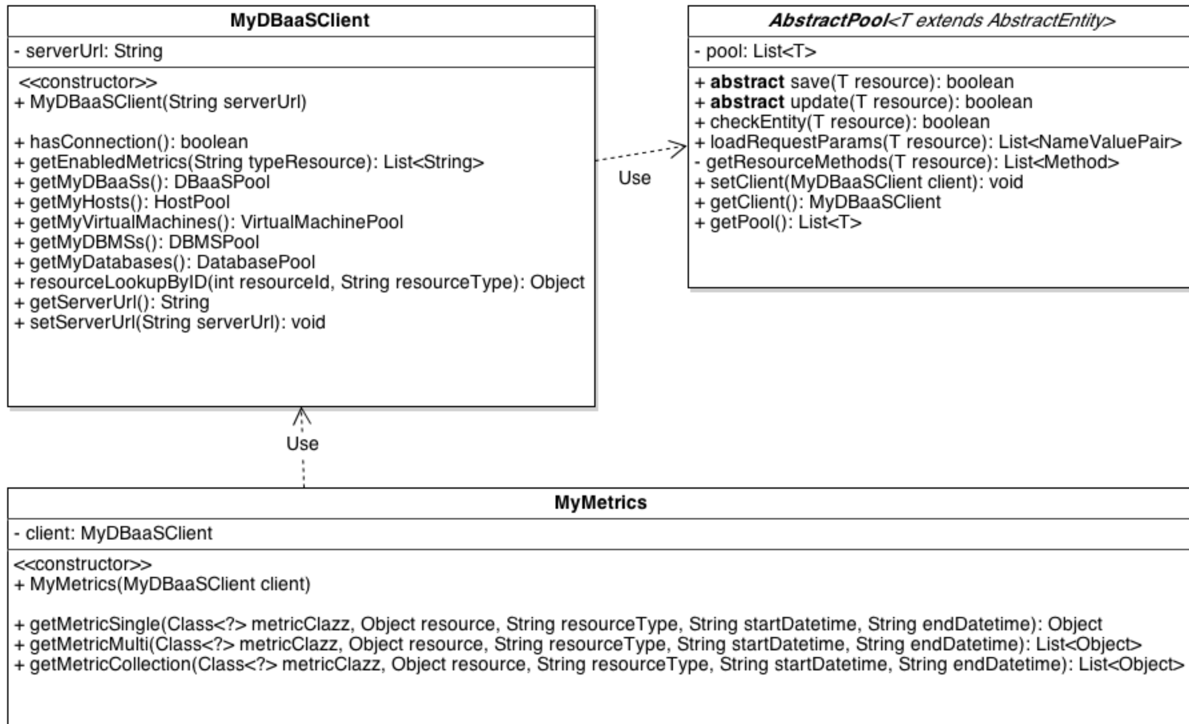


Figura 3.23 Diagrama de classes do módulo *MyDBaaS API*.

A Figura 3.23 apresenta o diagrama das classes e métodos que compõem o módulo *MyDBaaS API*. As classes *MyDBaaSClient* e *MyMetrics* são os *frozen spots* disponibilizados pelo módulo. A classe *MyDBaaSClient* é responsável por criar a conexão com o módulo *MyDBaaS Core* e provê os primeiros métodos de acesso ao ambiente de monitoramento. Esses métodos são para consultar as listas de um determinado tipo de recurso ou consultar diretamente um recurso pelo seu código. Os métodos que consultam as listas dos recursos cadastrados retornam um objeto do tipo *Pool*, que são classes que implementam a classe abstrata *AbstractPool*, o qual é *hot spot* do módulo. É através dessas classes que é possível adicionar um novo recurso ao ambiente ou atualizar os existentes. O *framework* já possui implementado nesse módulo as classes *DBaaSPool*, *HostPool*, *VirtualMachinePool*, *DBMSPool* e *DatabasePool*.

Alguns exemplos da utilização da classe *MyDBaaSClient* são demonstrados na Figura 3.24. O exemplo (1) apresenta a criação da conexão com o módulo *MyDBaaS Core* através da instanciação da classe com o endereço do servidor. O exemplo (2)

apresenta uma consulta que retorna exatamente o recurso de código *10* do tipo *machine*. E por fim, o exemplo (3) apresenta uma consulta ao ambiente de monitoramento que retorna o *pool* de todos os SGBDs existentes.

```
//(1)
MyDBaaSClient client = new MyDBaaSClient("localhost:8080/mydbaas");

//(2)
VirtualMachine virtualMachine = (VirtualMachine) client.resourceLookupByID(10, "machine");

//(3)
DBMSPool pool = client.getMyDBMSs();
```

Figura 3.24 Exemplos da instanciação da classe *MyDBaaSClient*.

O consumo das métricas é realizado através da classe *MyMetrics* também apresentada na Figura 3.23. Essa classe disponibiliza métodos já implementados, os quais automatizam as consultas sobre as métricas monitoradas. Esse métodos possibilitam utilizar diversas combinações para realizar as consultas. A Figura 3.25 demonstra alguns exemplos dessas consultas. O exemplo (4) demonstra a instanciação da classe *MyMetrics*, e para isso é preciso passar como parâmetro um objeto instanciado da classe *MyDBaaSClient*. Com o objeto instanciado é possível iniciar o consumo das métricas, como mostra os exemplos a seguir. O exemplo (5) apresenta a utilização do método *getMetricSingle()*, que é responsável por consultar a última coleta de uma métrica sobre um determinado recurso. Algumas métricas podem gerar mais de uma coleta por ciclo, por exemplo, se uma máquina virtual possuir mais de um núcleo por processador. O agente irá coletar a métrica sobre CPU para cada núcleo. O exemplo (6) apresenta a utilização do método *getMetricMulti()*, que é responsável pelas consultas desse tipo de métrica. Por fim, o exemplo (7) apresenta a utilização do método *getMetricCollection()*, que é responsável por consultar coleções de uma métrica sobre um determinado recurso. É importante destacar que todos os métodos possuem filtros que combinam datas, o que possibilita fazer consultas de métricas a partir de uma data, ou até uma data, ou entre duas datas.

```
//(4)
MyMetrics myMetrics = new MyMetrics(client);

//(5)
Memory memory = (Memory) myMetrics.getMetricSingle(Memory.class, virtualMachine, MyMetrics.RESOURCE_TYPE_VM, null, null);
ActiveConnection activeConnection = (ActiveConnection) myMetrics.getMetricSingle(ActiveConnection.class, dbms, MyMetrics.RESOURCE_TYPE_DBMS, null, null);

//(6)
List<Object> cpus = myMetrics.getMetricMulti(Cpu.class, virtualMachine, MyMetrics.RESOURCE_TYPE_VM, null, null);

//(7)
List<Object> sizes1 = myMetrics.getMetricCollection(Size.class, database, MyMetrics.RESOURCE_TYPE_DATABASE, "15-06-2013", null);
List<Object> sizes2 = myMetrics.getMetricCollection(Size.class, database, MyMetrics.RESOURCE_TYPE_DATABASE, null, "15-06-2013");
List<Object> sizes3 = myMetrics.getMetricCollection(Size.class, database, MyMetrics.RESOURCE_TYPE_DATABASE, "15-06-2013 15:30:00", "11-08-2013");
```

Figura 3.25 Exemplos da instânciação da classe *MyMetrics*.

A utilização do consumo de métricas pelo módulo *MyDBaaS API* irá funcionar para qualquer métrica que foi definida seguindo a especificação da seção 3.2.3. Mais detalhes técnicos e exemplos da utilização desse módulo podem ser vistos no site do *framework* na parte referente a API[7].

MyDBaaS Core

A responsabilidade do módulo *MyDBaaS Core* no serviço de consumo das métricas é gerenciar as requisições enviadas pelo módulo *MyDBaaS API*. Esse gerenciamento é feito através do componente *Controller*. Para tratar esse tipo de requisição existe um subcomponente dentro do *Controller* chamado *API Receiver*. Esse subcomponente já possui uma versão totalmente implementada e funcional dos *hot spots* apresentados na Figura 3.26.

A classe abstrata *AbstractMetricQuery* é responsável por provê a estrutura para a implementação da classe que irá gerenciar as requisições que consultam as métricas do ambiente de monitoramento. Ao estender esse *hot spot* ele automaticamente disponibiliza o acesso ao *Repository* e ao objeto que cria as respostas das requisições em JSON. O *Repository* nesse momento torna transparente ao desenvolvedor a consulta do consumo da métrica que está sendo requisitada, basta invocar os métodos disponibilizados pelo componente, pois toda a camada de acesso a base serial histórica é abstraída por eles. A interface *InterfacePoolQuery* e a classe abstrata *AbstractPoolQuery* são responsáveis por provê a estrutura para a implementação da classe que deverá tratar as requisições que consultam os *pools* de recursos cadastrados no ambiente. A interface *InterfaceResourceManager* e a classe abstrata *AbstractResourceManager* são responsáveis por provê a estrutura para a implementação da classe que deverá tratar as requisições de gerenciamento dos recursos, como adicionar ou atualizar. Quando esses quatro *hot spots* são implementados, as classe que os herdam passam a ter acesso também ao objeto que cria as respostas das requisições em formato JSON.

[7]http://mydbaasmonitor.com/api.html

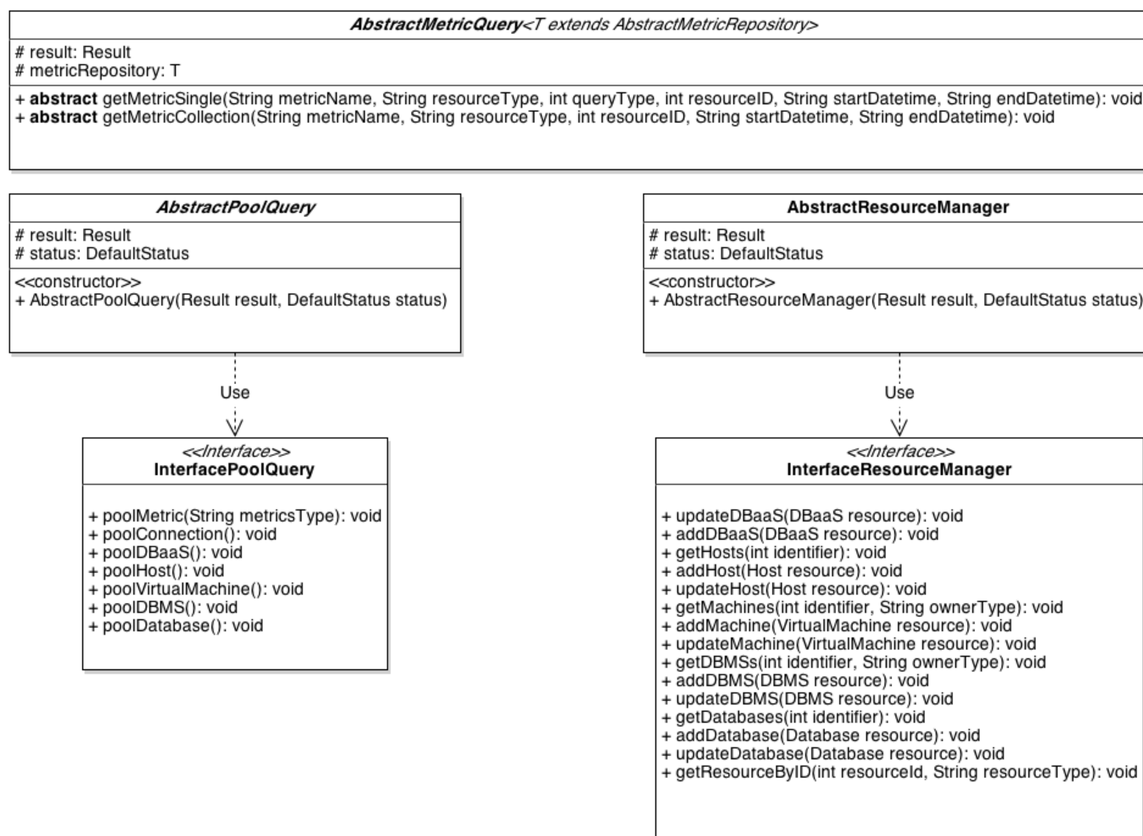


Figura 3.26 Diagrama de classes do componente *Controller* para *API Receiver*.

3.2.7 Implementação do Framework

O *framework* MyDBaaS foi totalmente implementado em Java. O módulo *MyDBaaS Core* é baseado no *framework* VRaptor[8], o que possibilita o desenvolvimento simples e transparente do componente *Controller*. Como esse componente é composto por subcomponentes que são orientados a serviço, o VRaptor agiliza o desenvolvimento desse ambiente RESTful[9]. O VRaptor foi escolhido por ser um *framework* brasileiro de fácil utilização que atinge alto nível de produtividade para Web, ser open-source, ser extensível e possuir uma comunidade ativa. O módulo *MyDBaaS Common* é puramente Java, pois somente são feitas as definições das classes sobre as métricas e recursos.

Os módulos *MyDBaaS Agent*, *MyDBaaS API* e *MyDBaaS Driver* são baseados

[8]http://vraptor.caelum.com.br

[9]É normalmente utilizado para se referir a serviços web executados sobre a arquitetura *Representational State Transfer* (REST) [48].

no Apache HttpComponents[10], que provê um conjunto de componentes de baixo nível para manipulação do protocolo HTTP em Java. Isso é necessário porque esses módulos não são Web e com isso precisam enviar as diversas requisições ao módulo *MyDBaaS Core*, como também saber tratar as respostas retornadas. O procoloto HTTP foi escolhido como forma de comunicação porque utiliza um modelo cliente-servidor, baseando-se no paradigma de requisição e resposta. Também possibilitando uma escalabilidade da arquitetura do *framework*. As requisições feitas pelo *MyDBaaS API* para consultar métricas e recursos da base serial histórica são retornadas pelo *MyDBaaS Core* no formato JSON[11]. Esse formato foi escolhido porque é leve e mundialmente difundido, mas também pelo fato do VRaptor manipulá-lo de maneira transparente. Para que os módulos possam traduzir as respostas JSON em objetos foi utilizado a biblioteca Google Gson[12]. O MyDBaaS é uma iniciativa open-source e possui um repositório[13] com o seu código fonte aberto para contribuições e extensões. Também se encontra disponibilizado o site[14] do *framework* com mais detalhes e atualizações constantes.

3.2.8 Instanciando o Framework

Para instanciar o *framework* é necessário que alguns softwares estejam instalados e configurados, como servidor de aplicação (*servlet container*) e o *framework* VRaptor. Para otimizar a utilização do *framework* proposto, existe um projeto pré-configurado com a estrutura inicial do MyDBaaS e as dependências necessárias. O MyDBaaS possui dois repositórios públicos, o primeiro[15] é referente ao projeto pré-configurado e o segundo[16] contém todo o código fonte do *framework*. O projeto pré-configurado é totalmente funcional e baseado na IDE Eclipse[17], mas também é facilmente importado em outras IDEs como NetBeans[18] e IntelliJ IDEA[19].

Com o projeto importado na IDE é possível estender todos os *hot spots* e configurar os *frozen spots* apresentados durante esse capítulo, criando um ambiente de mon-

[10]http://hc.apache.org
[11]http://json.org
[12]https://code.google.com/p/google-gson
[13]http://github.com/araujodavid/mydbaas-framework
[14]http://mydbaasmonitor.com/framework.html
[15]http://github.com/araujodavid/mydbaas-starting-project
[16]http://github.com/araujodavid/mydbaas-framework
[17]http://www.eclipse.org
[18]http://netbeans.org/kb/docs/java/import-eclipse.html
[19]http://www.jetbrains.com/idea/webhelp/importing-eclipse-project-to-intellij-idea.html

itoramento de acordo com as demandas necessárias. O componente *Repository* como destacado na seção 3.2.5 já possui uma versão implementada e funcional para criação da base serial histórica em SGBDs relacionais e nesse projeto está configurado para utilizar o MySQL[20], porém pode ser facilmente alterado para outro SGBD relacional. Com o *framework* importado e configurado, para criar o ambiente de monitoramento desejável basta seguir os passos:

1. Definir as métricas como apresentado na seção 3.2.3.
2. Implementar os coletores das métricas como apresentado na seção 3.2.4.
3. Implementar os *receivers* das métricas como apresentado na seção 3.2.5.
4. Configurar os arquivos de contexto para os agentes de monitoramento como apresentado na seção 3.2.4 no item *Arquivo de Contexto*.
5. Inicializar os agentes de monitoramento como apresentado na seção 3.2.4 no item *Inicialização do Agente de Monitoramento*.
6. Por fim, com o ambiente de monitoramento criado é possível consumir as métricas como apresentado na seção 3.2.6.

3.2.9 Análise Comparativa entre MyDBaaS e Trabalhos Relacionados

Na seção 2.5 do capítulo anterior, os trabalhos relacionados foram analisados com base nos requisitos levantados como demonstrado na Tabela 2.2. Nesta seção recuperamos essa tabela comparativa e adicionamos o *framework* MyDBaaS para uma nova análise, agora entre o *framework* proposto e os trabalho relacionados como apresentado na nova Tabela 3.1.

O *framework* MyDBaaS foi concebido com base nos requisitos elencados na seção 2.4, diferentemente dos trabalhos [18], [19], [20] e [21]. Permitir que métricas sejam definidas é um conceito muito importante para criação de um monitoramento adequado a necessidade do serviço. Os trabalhos [20] e [21] possibilitam essa definição, mas o [20] limita-se ao que a ferramenta Ganglia consegue monitorar. Porém em ambos a definição dessas métricas somente são sobre as camadas física e virtual. Os trabalhos [18] e [19]

[20]http://www.mysql.com

não contemplam esse conceito e utilizam um conjunto fixo de métricas. O MyDBaaS diferente deles, permite a definição de métricas de acordo com a necessidade do monitoramento tanto na camada física, virtual, dados e carga de trabalho - ainda possibilita a definição de novos recursos que podem representar novas camadas. Com relação ao monitoramento das camadas física, virtual e de dados o MyDBaaS é o único que possibilita o monitoramento total das três. Com base nas duas primeiras camadas os trabalhos [18], [20] e [21] contemplam o monitoramento, mas são fixos porque para monitorar as métricas dessas camadas se apoiam em ferramentas externas não possibilitando a alteração. No sentido contrário, o MyDBaaS permite que para cada métrica, independente da camada que pertence, seja definido como ela será coletada - possibilitando utilizar chamadas a ferramentas, APIs, bibliotecas ou serviços externos. Para o monitoramento da camada de dados e carga de trabalho relacionadas a banco de dados, somente o trabalho [19] propõem um movimento nesse sentido. Porém o seu monitoramento é restrito com relação a SGBD/instâncias de banco de dados, focando mais em métricas sobre as cargas de trabalho. O MyDBaaS possibilita que métricas sejam definidas para o monitoramento tanto de SGBDs, instâncias de banco de dados e para as cargas de trabalho.

Os *frameworks* propostos em [19] e [20], possuem um objetivo além do monitoramento das camadas. Em [19] o *framework* utiliza o monitoramento para coletar métricas e mensurar a qualidade dos sistemas de banco de dados em nuvem e em [20] o *framework* utiliza o monitoramento para coletar métricas nos recursos das camadas para acompanhar e gerenciar os acordos de SLA. O MyDBaaS não restringe a utilização das métricas que estão sendo monitoradas, para isso, uma base serial histórica é criada para armazenar as métricas coletadas e uma API de fácil utilização é disponibilizada para consumo das métricas. Com isso, o ambiente de monitoramento criando pelo MyDBaaS pode ser utilizado para diversas estratégias, como replicação, elasticidade, entre outros além também das apresentadas por [19] e [20]. O trabalho [18] também possui essa abordagem de criação de uma base serial histórica e disponibilização de uma API de acesso, porém a estrutura desta base é fixa e não se adapta a definição de novas métricas e a API não encapsula a complexidade da comunicação e manipulação das informações requisitadas. O MyDBaaS também permite o consumo das métricas de forma transparente através de mecanismos em função do tempo e quantidade de coletas realizadas possibilitando consultas de acordo com a necessidade, diferente dos trabalhos relacionados.

Req.	Descrição	[18]	[19]	[20]	[21]	MyDBaaS
I	Permitir a definição de diferentes métricas nas camadas.	Não	Não	Parcial	Sim	Sim
II	Permitir o monitoramento da camada de recursos físicos.	Sim	Não	Sim	Sim	Sim
III	Permitir o monitoramento da camada de recursos virtuais.	Sim	Não	Sim	Sim	Sim
IV	Permitir o monitoramento da camada de dados.	Não	Parcial	Não	Não	Sim
V	Monitorar métricas de carga de trabalho relacionadas a banco de dados.	Não	Sim	Não	Não	Sim
VI	Armazenar as métricas coletadas em uma base serial histórica.	Sim	N/A	Não	Parcial	Sim
VII	Permitir configurar o conjunto de métricas e os ciclos de monitoramento das métricas em diferentes granularidades nas diversas camadas de recursos.	Sim	N/A	Não	Parcial	Sim
VIII	Fornecer fácil acesso ao ambiente de monitoramento através de uma API que encapsule a complexidade da comunicação e manipulação das informações trocadas.	Parcial	Não	Não	Não	Sim
IX	Possibilitar o consumo das métricas em função do tempo e quantidade de coletas.	Não	Não	Não	Não	Sim

Tabela 3.1 Análise comparativa do MyDBaaS e trabalhos relacionados.

3.3 CONCLUSÃO

Esse capítulo apresentou o MyDBaaS, um *framework* para o monitoramento de ambientes de banco de dados em nuvem, cuja finalidade é possibilitar a criação de soluções de monitoramento personalizáveis e eficientes. Foram destacados quais os principais conceitos e características do MyDBaaS, sua arquitetura e uma explanação detalhada dos seus componentes também foram apresentadas. Uma especificação sobre os serviços existentes dentro do *framework* também foi apresentada, e um passo-a-passo para instanciação do *framework* foi demonstrada. Por fim, um comparativo com o *framework* proposto e os trabalhos relacionados foi realizado.

CAPÍTULO 4

ESTUDO DE CASO: APLICAÇÃO MyDBaaS MONITOR

Esse capítulo descreve o MyDBaaS Monitor[1], uma aplicação web para o gerenciamento do monitoramento de serviços DBaaS cuja finalidade é possibilitar o monitoramento personalizado dos recursos que compõem os serviços. O capítulo está estruturado da seguinte forma: na seção 4.1.1 são apresentadas todas as métricas que foram definidas para a aplicação. Na seção 4.1.2 são apresentados os coletores implementados para cada métrica e na seção 4.1.3 são apresentados os *receivers* implementados para recebimento das métricas coletadas. Na seção 4.1.4 o processo de gerenciamento dos recursos pela aplicação no ambiente de monitoramento é discutido e em seguida na seção 4.1.5 é apresentado o processo de configuração do monitoramento para os recursos. Na seção 4.1.6 são apresentados as telas disponibilizadas pela aplicação para acompanhamento dos recursos monitorados através de gráficos, em seguida na seção 4.1.7 é apresentado um experimento com a aplicação. Por fim, uma conclusão é realizada.

4.1 MyDBaaS MONITOR

O MyDBaaS Monitor é uma aplicação web open-source[2] para o gerenciamento do monitoramento de serviços DBaaS, possibilitando o monitoramento e acompanhamento de forma transparente e automática dos diversos recursos que compõem esses serviços. O MyDBaaS Monitor foi totalmente baseado no *framework* MyDBaaS como back-end e utiliza as tecnologias Twitter Bootstrap[3] e Highcharts[4] para criação das interfaces gráficas para interação do usuário. Esta aplicação permite o usuário cadastrar os recursos existentes em cada camada do seu serviço DBaaS através de interfaces web, assim como,

[1] Site da aplicação: http://mydbaasmonitor.com
[2] Repositório: http://github.com/araujodavid/mydbaas-monitor
[3] *Framework* para front-end: http://getbootstrap.com/2.3.2
[4] Biblioteca de gráficos: http://www.highcharts.com

realize a configuração do monitoramento de forma fácil e também acompanhe em tempo real o monitoramento executado sobre cada recurso através de gráficos dinâmicos.

4.1.1 Métricas Definidas

Para implementação dessa aplicação foram definidas métricas nas diversas camadas de recursos de acordo com a seção 3.2.3. Para a camada física foram definidas as métricas *DomainStatus*, *HostDomains* e *HostInfo*. Para a camada virtual foram definidas as métricas *Cpu*, *Memory*, *Network*, *Disk*, *Partition* e *Machine*, as cinco primeiras métricas dessa camada também podem ser utilizadas na camada de servidores. Para a camada de dados (SGBDs e instâncias de banco de dados) foram definidas as métricas *ActiveConnection*, *Size*, *NetworkTraffic*, *ProcessStatus*, *InformationData*, *InformationTable*, *StatementDML*, *StatementTCL*, *StatementDDL*, *StatementDCL* e *DiskUtilization*. Por fim, para a camada de carga de trabalho foi definida a métrica *WorkloadStatus*. As descrições e medidas coletadas por essas métricas podem ser vistas nas Tabelas 4.1, 4.2, 4.3 e 4.4.

Métrica	**Descrição**	**Medidas**
HostDomains	Essa métrica tem como objetivo coletar os estados das VMs hospedadas no servidor.	• Quantidade de VMs ativas • Quantidade de VMs inativas
DomainStatus	Essa métrica tem como objetivo coletar a quantidade de CPU e memória que os processos das VMs hospedadas estão consumindo no servidor.	• Identificação da VM • Porcentagem de CPU utilizada • Porcentagem de memória utilizada
HostInfo	Essa métrica tem como objetivo coletar as informações da configuração física do servidor.	• Identificação do servidor • Hypervisor utilizado • Memória total • Quantidade de CPUs • Quantidade de cores • Frequência da CPU • Modelo da CPU

Tabela 4.1 Métricas definidas para a camada de servidores.

Métrica	**Descrição**	**Medidas**
Cpu	Essa métrica tem como objetivo coletar a utilização da CPU dentro da VM.	• Porcentagem CPU User • Porcentagem CPU System • Porcentagem CPU Nice • Porcentagem CPU Wait • Porcentagem CPU Idle • Porcentagem Total
Memory	Essa métrica tem como objetivo coletar a utilização da memória dentro da VM.	• Swap utilizada • Swap livre • Porcentagem swap utilizada • Porcentagem swap livre • Memória utilizada • Memória livre • Porcentagem memória utilizada • Porcentagem memória livre • Buffer cache utilizado • Buffer cache livre
Network	Essa métrica tem como objetivo coletar o tráfego de rede da VM.	• Bytes enviados • Bytes recebidos • Pacotes enviados • Pacotes recebidos
Disk	Essa métrica tem como objetivo coletar a utilização total do disco da VM.	• Bytes lidos • Bytes escritos • Quantidade de leituras • Quantidade de escritas • Espaço livre • Espaço usado • Espaço total • Porcentagem de utilização

Partition	Essa métrica tem como objetivo coletar a utilização por partição do disco da VM.	• Nome do diretório • Nome do dispositivo • Bytes lidos • Bytes escritos • Quantidade de leituras • Quantidade de escritas • Espaço livre • Espaço usado • Espaço total • Porcentagem de utilização
Machine	Essa métrica tem como objetivo coletar as informações da configuração física/lógica da VM.	• Sistema operacional • Modelo do kernel • Versão do kernel • Arquitetura • Memória total • Swap total • Total de CPUs • Total de cores • Cores por CPU

Tabela 4.2 Métricas definidas para a camada de máquinas virtuais.

Métrica	**Descrição**	**Medidas**
ActiveConnection	Essa métrica tem como objetivo coletar a quantidade total de conexões abertas do SGBD ou instância de banco de dados.	• Conexões abertas
Size	Essa métrica tem como objetivo coletar o total utilizado em disco do SGBD ou instância de banco de dados.	• Total utilizado
NetworkTraffic	Essa métrica tem como objetivo coletar o tráfego de rede do SGBD.	• Bytes recebidos • Bytes enviados

ProcessStatus	Essa métrica tem como objetivo coletar o total de CPU e memória utilizados pelo SGBD.	• Porcentagem CPU utilizada • Porcentagem memória utilizada
InformationData	Essa métrica tem como objetivo coletar a totalidade de objetos do SGBD.	• Total de bancos de dados • Total de tabelas • Total de índices • Total de triggers • Total de views • Total de rotinas
InformationTable	Essa métrica tem como objetivo coletar informações sobre as tabelas existentes da instância de banco de dados.	• Nome da tabela • Tamanho total dos dados • Tamanho total dos índices • Quantidade de tuplas • Tamanho médio de uma tupla • Tamanho total da tabela
StatementDML	Essa métrica tem como objetivo coletar a totalidade de comandos DML executados no SGBD.	• Inserts executados • Updates executados • Selects executados • Commits executados • Deletes executados • Rollbacks executados • Savepoints executados
StatementTCL	Essa métrica tem como objetivo coletar a totalidade de comandos TCL executados no SGBD.	
StatementDDL	Essa métrica tem como objetivo coletar a totalidade de comandos DDL executados no SGBD.	• Create executados • Alter executados • Drop executados • Truncate executados • Rename executados
StatementDCL	Essa métrica tem como objetivo coletar a totalidade de comandos DCL executados no SGBD.	• Grant executados • Revoke executados

DiskUtilization	Essa métrica tem como objetivo coletar o acesso ao disco pelo SGBD.	• Qnt. de leituras físicas • Qnt. de leituras lógicas • Qnt. de leituras pendentes • Qnt. de escritas pendentes • Bytes lidos • Bytes escritos • Qnt. de leituras em página • Qnt. de escritas em página • Qnt. de leituras físicas em bloco do disco • Qnt. de escritas físicas em bloco do disco

Tabela 4.3 Métricas definidas para SGBDs/instâncias de banco de dados.

Métrica	**Descrição**	**Medidas**
WorkloadStatus	Essa métrica tem como objetivo coletar informações sobre uma consulta disparada para uma instância de banco de dados	• Consulta executada • Seletividade • Tempo de resposta • Throughput

Tabela 4.4 Métrica definida para a camada de carga de trabalho.

Como descrita na Tabela 4.4, apresentamos na Figura 4.1 o exemplo da classe que implementa a métrica *WorkloadStatus* para a camada de carga de trabalho. As outras métricas descritas nas tabelas 4.1, 4.2 e 4.3 podem ser vistas no Apêndice **A** através das Figuras A.1, A.2 e A.3 que apresentam as implementações das três métricas da camada física respectivamente, das Figuras A.4, A.5, A.6, A.7, A.8 e A.9 que apresentam as implementações das seis métricas da camada virtual respectivamente e as Figuras A.10, A.11, A.12, A.13, A.14, A.15, A.16, A.17, A.18, A.19 e A.20 que apresentam as implementações das onze métricas da camada de dados respectivamente.

```java
public class WorkloadStatus extends AbstractDatabaseMetric {

    private String workloadStatusQuery;
    private long workloadStatusSelectivity;
    private double workloadStatusResponseTime;
    private double workloadStatusThroughput;

    public String getWorkloadStatusQuery() {
        return workloadStatusQuery;
    }

    public void setWorkloadStatusQuery(String workloadStatusQuery) {
        this.workloadStatusQuery = workloadStatusQuery;
    }

    public long getWorkloadStatusSelectivity() {
        return workloadStatusSelectivity;
    }

    public void setWorkloadStatusSelectivity(long workloadStatusSelectivity) {
        this.workloadStatusSelectivity = workloadStatusSelectivity;
    }

    public double getWorkloadStatusResponseTime() {
        return workloadStatusResponseTime;
    }

    public void setWorkloadStatusResponseTime(double workloadStatusResponseTime) {
        this.workloadStatusResponseTime = workloadStatusResponseTime;
    }

    public double getWorkloadStatusThroughput() {
        return workloadStatusThroughput;
    }

    public void setWorkloadStatusThroughput(double workloadStatusThroughput) {
        this.workloadStatusThroughput = workloadStatusThroughput;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<WorkloadStatus> jsonToList(String json) {
        Gson gson = new Gson();
        List<WorkloadStatus> workloadStatusList = gson.fromJson(json, new TypeToken<List<WorkloadStatus>>(){}.getType());
        return workloadStatusList;
    }
}
```

Figura 4.1 Implementação da métrica *WorkloadStatus*.

4.1.2 Coletores Implementados

Como apresentados na seção 3.2.4 para cada métrica definida é necessário implementar um coletor, o qual é responsável pelo processo de coleta da métrica e de envio para que seja armazenada na base serial histórica. Com isso, para cada métrica definida na seção 4.1.1 foi implementado um coletor conforme a seção 3.2.4. Os coletores das métricas da camada física foram implementados utilizando a biblioteca *Libvirt*[5], que é uma API open-source para o gerenciamento de plataformas de virtualização. Os coletores das métricas

[5] http://libvirt.org

da camada virtual foram implementados com base no *Hyperic SIGAR*[6], que é uma API multiplataforma para coleta de informações sobre o sistema. Os coletores das métricas da camada de dados são em sua maioria consultas executadas sobre os contextos dos SGBDs ou das instâncias de banco de dados. É importante destacar que os coletores das métricas da camada de dados foram inicialmente implementados para os SGBDs MySQL[7] e PostgreSQL[8], mas como a aplicação é open-source e baseada no *framework* MyDBaaS, esses coletores podem ser facilmente estendidos a outros SGBDs caso necessário. Alguns coletores foram implementados em ambos os níveis da camada para ambos SGBDs, mas outros somente funcionam para o MySQL. O coletor da métrica da camada de carga de trabalho foi implementado encapsulando os JDBCs de ambos os SGBDs para poder interceptar as consultas enviadas para as instâncias de banco de dados.

Com isso, o *framework* se demostrou flexível e genérico pois não limitou a utilização das tecnologias nos coletores implementados. As características e metodologias de coleta de cada coletor podem ser vistas nas Tabelas 4.5, 4.6, 4.7 e 4.8.

Coletor	**Metodologia de Coleta**
HostDomainsCollector	O coletor utiliza a API do *Libvirt* que se conecta ao *hypervisor* utilizado pelo servidor e através da classe *Connect* disponibilizada pela API, é possível verificar quantas VMs estão ativas e inativas no momento.
DomainStatusCollector	O coletor utiliza a API do *Libvirt* que se conecta ao *hypervisor* e através da classe *Domain* disponibilizada pela API, é possível acessar o código do processo de cada VM hospedada no servidor e a partir desse código é verificado o quanto de CPU e memória cada VM está consumindo do servidor no momento.
HostInfoCollector	O coletor utiliza a API do *Libvirt* que se conecta ao *hypervisor* e através da classe *NodeInfo* disponibilizada pela API, é possível verificar as informações de configuração sobre o *hypervisor* e servidor.

Tabela 4.5 Coletores definidos para a camada de servidores.

[6]http://www.hyperic.com/products/sigar
[7]http://www.mysql.com
[8]http://www.postgresql.org

Coletor	**Metodologia de Coleta**
CpuCollector	O coletor utiliza a API do *Hyperic SIGAR* e através da classe *Sigar* acessa a lista de CPUs existentes na máquina. E para cada CPU o coletor utiliza a classe *CpuPerc* para coletar as medidas da métrica no momento.
MemoryCollector	O coletor utiliza a API do *Hyperic SIGAR* e a partir da classe *Sigar*, acessa a memória e a área swap da máquina através das classes *Mem* e *Swap* para coletar as medidas da métrica no momento.
NetworkCollector	O coletor utiliza a API do *Hyperic SIGAR* e através da classe *Sigar* acessa a lista de interfaces de rede existentes na máquina. A partir de cada interface o coletor utiliza a classe *NetInterfaceStat* para coletar os valores da interface e realizar um somatório totalizando as medidas da métrica no momento.
DiskCollector	O coletor utiliza a API do *Hyperic SIGAR* e através da classe *Sigar* acessa a lista de sistemas de arquivos existentes na máquina. A partir de cada sistema de arquivos o coletor utiliza a classe *FileSystemUsage* para coletar os valores da partição e realizar um somatório totalizando as medidas da métrica no momento.
PartitionCollector	O coletor utiliza a API do *Hyperic SIGAR* que através da classe *Sigar* acessa a lista de sistemas de arquivos existentes e a partir de cada partição, o coletor utiliza a classe *FileSystemUsage* para coletar as medidas da métrica no momento.
MachineCollector	O coletor utiliza a API do *Hyperic SIGAR* e a partir da classe *Sigar*, acessa informações do sistema operacional e configurações de CPU e memória através das classes *OperatingSystem*, *Mem*, *Swap* e *CpuInfo*.

Tabela 4.6 Coletores definidos para a camada de máquinas virtuais.

Coletor	**Metodologia de Coleta**
ActiveConnectionCollector	MySQL: select count(*) as connections from information_schema.processlist; select count(*) as connections from information_schema.processlist where db = database(); PostgreSQL: select count(*) as connections from pg_stat_activity; select count(*) as connections from pg_stat_activity where datname = ?;
SizeCollector	MySQL: select sum(data_length+index_length)/1024/1024 as size from information_schema.tables; select sum(data_length+index_length)/1024/1024 as size from information_schema.tables where table_schema = database(); PostgreSQL: select (sum(pg_database_size(pg_database.datname))/1024/1024) as size from pg_database; select (pg_database_size(?)/1024/1024) as size;
NetworkTrafficCollector	MySQL: show global status where variable_name in ('Bytes_received', 'Bytes_sent');
ProcessStatusCollector	O coletor utiliza comandos *shell script* para coletar o quanto de CPU e memória um SGBD está consumindo na máquina. Para o MySQL, o coletor recupera o código do processo (PID) do SGBD no sistema através do nome *mysqld* que é o padrão do serviço, com o PID é possível coletar a quantidade de CPU e memória utilizadas no momento. Para o PostgreSQL, o coletor utiliza o usuário padrão *postgres* criado pelo SGBD no sistema e através dele é coletado a quantidade de CPU e memória utilizadas pelos processos do usuário no momento.
InformationDataCollector	MySQL: select (select count(*) from information_schema.schemata) as amount_db, (select count(*) from information_schema.tables) as amount_tables, (select count(*) from information_schema.statistics) as amount_index, (select count(*) from information_schema.triggers) as amount_trigger, (select count(*) from information_schema.views) as amount_views, (select count(*) from information_schema.routines) as amount_routines from dual;

InformationTableCollector	MySQL: select concat(table_schema, '.', table_name) as table_name, table_rows, round(avg_row_length/1024, 2) as row_average, round(data_length/(1024*1024), 2) as data_length, round(index_length/(1024*1024), 2) as index_length, round(round(data_length + index_length)/(1024*1024), 2) as total_size from information_schema.tables where table_schema = database() order by data_length desc;
StatementDMLCollector	MySQL: show global status where variable_name in ('Com_insert', 'Com_select', 'Com_update', 'Com_delete');
StatementTCLCollector	MySQL: show global status where variable_name in ('Com_commit', 'Com_rollback', 'Com_savepoint');
StatementDDLCollector	MySQL: show global status where variable_name in ('Com_create_table', 'Com_alter_table', 'Com_drop_table', 'Com_truncate', 'Com_rename_table');
StatementDCLCollector	MySQL: show global status where variable_name in ('Com_grant', 'Com_revoke');
DiskUtilizationCollector	MySQL: show global status where variable_name in ('Innodb_buffer_pool_reads', 'Innodb_buffer_pool_read_requests', 'Innodb_data_pending_reads', 'Innodb_data_pending_writes', 'Innodb_data_read', 'Innodb_data_written', 'Innodb_pages_read', 'Innodb_pages_written', 'Key_reads', 'Key_writes');

Tabela 4.7 Coletores definidos para a camada de SGBDs/instâncias de banco de dados.

Coletor	**Metodologia de Coleta**
WorkloadStatusCollector	O coletor utiliza os JDBCs dos SGBDs para interceptar as consultas enviadas para as instâncias de banco de dados. O coletor possui agora a responsabilidade de criar a conexão com a instância e executar as cargas de trabalho. Ao interceptar uma consulta o coletor consegue saber quando o processo iniciou e terminou, e calcular o tempo de resposta total. Também consegue coletar a seletividade da consulta através da classe *ResultSet*. Com essas duas informações o coletor também consegue coletar o *throughput* de cada execução, calculando a divisão da seletividade pelo tempo de resposta total.

Tabela 4.8 Coletor definido para a camada de carga de trabalho.

```java
public class InformationTableCollector extends AbstractCollector<InformationTableMetric> {

    private List<InformationTable> informationTableMetrics;

    public InformationTableCollector(int identifier, String type) {
        super(identifier, type);
        this.informationTableMetrics = new ArrayList<InformationTable>();
    }

    @Override
    public void loadMetric(Object[] args) throws NumberFormatException, ClassNotFoundException, SQLException  {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            String sql = "select concat(table_schema, '.', table_name) 'table_name', table_rows, " +
                         "round(`avg_row_length` / 1024 , 2) row_average, " +
                         "round(data_length / (1024*1024) , 2) data_length,  " +
                         "round(index_length / (1024*1024) , 2) index_length, " +
                         "round(round(data_length + index_length) / (1024*1024) , 2) total_size " +
                         "from information_schema.tables where table_schema = database() " +
                         "order by data_length desc;";
            resultSet = databaseConnection.executeSQL(connection, sql, null);
        }
        //Dealing with the return of the query and inserting the data in the object of the metric
        while (resultSet != null && resultSet.next()) {
            InformationTable informationTable = new InformationTable();
            informationTable.setInformationTableName(resultSet.getString("table_name"));
            informationTable.setInformationTableAmountRows(resultSet.getLong("table_rows"));
            informationTable.setInformationTableRowAverage(resultSet.getDouble("row_average"));
            informationTable.setInformationTableDataLength(resultSet.getDouble("data_length"));
            informationTable.setInformationTableIndexLength(resultSet.getDouble("index_length"));
            informationTable.setInformationTableTotalSize(resultSet.getDouble("total_size"));
            this.informationTableMetrics.add(informationTable);
        }
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = InformationTableMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDatabases().length > 0) {
            for (String database : this.metric.getDatabases()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {database});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the InformationTable metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), this.informationTableMetrics, 0, Integer.parseInt(database));
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("InformationTable request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        //Release any native resources associated with this sigar instance
        this.informationTableMetrics.clear();
    }
}
```

Figura 4.2 Implementação do coletor *InformationTableCollector*.

Como demonstrado nas tabelas 4.5, 4.6, 4.7 e 4.8, o destaque principal foi com relação a metodologia de coleta de cada coletor implementado para as métricas, mas os coletores não possuem somente essa funcionalidade. Eles também são responsáveis por criar a requisição com os parâmetros da métrica coletada e pelo envio ao servidor, porém essas funcionalidades são iguais a todos coletores e por isso não participaram das tabelas. A Figura 4.2 apresenta a classe que implementa o coletor da métrica *InformationTable*, nela foi definida a metodologia de coleta no método *loadMetric()* e no método *run()* a execução da metodologia para cada instância de banco de dados que deverá ser monitorada.

Os outros coletores implementados podem ser visto no Apêndice **B**. As Figuras B.1, B.2 e B.3 apresentam os coletores para as métricas da camada física. As Figuras B.4, B.5, B.6, B.7, B.8 e B.9 apresentam os coletores para as métricas da camada virtual. As Figuras B.10, B.11, B.12, B.13, B.14, B.15, B.16, B.17, B.18 e B.19 apresentam os coletores para as métricas da camada de dados. Por fim, a Figura B.20 apresenta o coletor da métrica da camada de carga de trabalho.

4.1.3 Receivers Implementados

Com o conjunto de métricas definidas na seção 4.1.1, foram implementados quatro *receivers* como apresentado na seção 3.2.5. Um *receiver* para cada camada em que as métricas são coletadas, ou seja, existe um *receiver* para receber as métricas referentes aos servidores, um *receiver* para receber as métricas referentes as máquinas virtuais, um *receiver* para receber as métricas referentes aos SGBDs/instâncias de banco de dados e um *receiver* para receber as métricas de carga de trabalho. A Figura 4.3 apresenta um exemplo da implementação de um *receiver*, a classe *HostReceiverController* que é o *receiver* das métricas coletadas pelos agentes de monitoramento na camada de servidores. Os outros *receivers* implementados para a aplicação podem ser visto no Apêndice **C**, as Figuras C.1 e C.2 apresentam a implementação da classe *MachineReceiverController*, que é o *receiver* das métricas coletadas pelos agentes de monitoramento na camada de máquinas virtuais. As Figuras C.3 e C.4 apresentam a implementação da classe *StorageReceiverController*, que é o *receiver* das métricas coletadas pelos agentes de monitoramento na camada de SGBDs/instâncias de banco de dados. Por fim, a Figura C.5 apresenta a classe *WorkloadReceiverController* que é o *receiver* da métrica coletada pelo módulo *MyDBaaS Driver* na camada de carga de trabalho enviadas as instâncias de banco de dados.

```java
@Resource
@Path("/host")
public class HostReceiverController extends AbstractReceiver<MetricRepository> {

    private HostRepository hostRepository;

    public HostReceiverController(DefaultStatus status, MetricRepository repository,
            HostRepository hostRepository) {
        super(status, repository);
        this.hostRepository = hostRepository;
    }

    @Post("/info")
    public void information(HostInfo metric, int host) {
        if (hostRepository.updateHostInformation(metric, host)) {
            status.accepted();
        }
    }

    @Post("/domainstatus")
    public void domainStatus(List<DomainStatus> metric, int host, String recordDate) {
        for (DomainStatus domainStatus : metric) {
            try {
                if (repository.saveMetric(domainStatus, recordDate, 0, host, 0, 0)) {
                    status.accepted();
                }
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }

    @Post("/hostdomains")
    public void hostDomains(HostDomains metric, int host, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, host, 0, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 4.3 Implementação do *receiver* para métricas de servidor.

O método *domainStatus()* da classe *HostReceiverController*, os métodos *cpu()* e *partition()* da classe *MachineReceiverController* (Figura C.1) e o método *informationTable()* da classe *StorageReceiverController* (Figura C.4) como é possível visualizar, possuem uma estrutura diferente dos demais. Em vez de receberem um único objeto da métrica a qual são responsáveis, eles recebem uma lista de objetos da métrica - isso é possível desde que o nome do parâmetro permaneça *metric* como especificado na seção 3.2.5. Isso não implica em nenhuma mudança na estrutura do *framework* durante a sua instanciação.

4.1.4 Gerenciamento dos Recursos

Para a concepção da aplicação MyDBaaS Monitor definimos o recurso DBaaS, mas não para fins de monitoramento. Como explicado na seção 4.1, a aplicação possibilita o monitoramento de múltiplos serviços de DBaaS simultaneamente, e isso implica em diversos servidores, máquinas virtuais, SGBDs e instâncias de banco de dados no ambiente de monitoramento. Portanto, o recurso DBaaS foi definido com o objetivo de poder agrupar e organizar os demais recursos das camadas por serviço de DBaaS.

Como informado na seção 3.2.3, o *framework* já possui a definição para os recursos: servidor, VM, SGBD e instância de banco de dados, que compõem as camadas física, virtual e de dados respectivamente. Com base nessas definições, a aplicação MyDBaaS Monitor possui um módulo de gerenciamento para manutenção desses recursos no ambiente de monitoramento, e por consequência na base serial histórica que é gerada ao longo do tempo. Com isso, na aplicação esses recursos são cadastrados como partes que compõem um DBaaS, ou seja, o usuário deve registrar o serviço de DBaaS e a partir desse ambiente cadastrar a pilha de recursos das camadas que o compõem. Com base na visão de um serviço de DBaaS apresentada na seção 2.2, o módulo de gerenciamento dos recursos foi projetado sobre o mesmo conceito, tratando o serviço como um conjunto de servidores físicos que podem possuir diversas VMs, onde cada VM pode possuir diversos SGBDs, que por sua vez podem possuir diversas instâncias de banco de dados. Assim, a aplicação permite que o usuário cadastre um DBaaS através da tela apresentada na Figura 4.4. O usuário precisa informar um nome para o serviço, a fim de ajudar na identificação do mesmo, e uma descrição caso necessário.

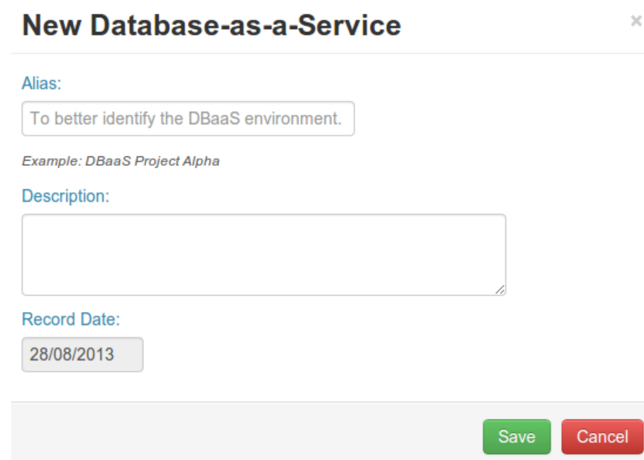


Figura 4.4 Tela para cadastro de um novo DBaaS no ambiente de monitoramento.

A partir do DBaaS cadastrado, o usuário pode cadastrar os recursos físicos, virtuais e de dados que pertencem ao serviço. O registro de um servidor é realizado através da tela apresentada na Figura 4.5, onde o usuário informa qual o serviço DBaaS o servidor pertence e quais as informações de acesso.

New Host

Environment DBaaS:
Select one
Alias:
To better identify the resource
Example: Cluster Project X
Address:
Access address
Example: 127.0.0.1
Username:
Make sure that the user is **root**.
Port:
e.g. 22
Enter the **SSH** port.
Password:
Root password
Confirm the password
Description:
Record Date:
28/08/2013
Save Cancel

Figura 4.5 Tela para cadastro de um novo Servidor no ambiente de monitoramento.

Para registrar as VMs, o usuário informa a qual serviço DBaaS e qual servidor ela pertence, assim como as informações de acesso. É importante destacar que uma VM pode ser registrada sem pertencer a um servidor, caso a mesma faça parte de uma nuvem pública. O registro de uma VM é realizado através da tela apresentada na Figura 4.6.

New Virtual Machine

Environment DBaaS:
Select one
Host:
Without Host
Alias:
To better identify the resource
Example: VM Project X
Address:
Access address
Example: 127.0.0.1
Username:
Make sure that the user is root.
Port:
e.g. 22
Enter the SSH port.
Password:
Root password
Confirm the password
Description:
Record Date:
28/08/2013
Save Cancel

Figura 4.6 Tela para cadastro de uma nova Máquina Virtual no ambiente de monitoramento.

Com as VMs cadastradas, é possível registrar os SGBDs e instâncias de banco de dados que compõem a camada de dados do serviço. Para registrar um SGBD o usuário deve informar a qual VM ele pertence e as informações de conexão, como apresenta a tela da Figura 4.7. As instâncias de banco de dados são registradas através da tela apresentada na Figura 4.8, onde o usuário informa qual SGBD a instância pertence e qual nome do esquema. Os dados de acesso informados durante o cadastro dos servidores e VMs são utilizados pela aplicação para acessar automaticamente os recursos para configurar e inicializar os agentes de monitoramento, a configuração do agente de monitoramento para o recurso é melhor apresentada na seção 4.1.5. Os dados de conexão informados durante o cadastro dos SGBDs são utilizados pelos agentes de monitoramento para recolher possíveis métricas nesse nível, assim como sobre as instâncias de banco de dados.

Figura 4.7 Tela para cadastro de um novo SGBD no ambiente de monitoramento.

Figura 4.8 Tela para cadastro de uma nova Instância de Banco de Dados no ambiente de monitoramento.

Após o cadastro dos recursos, é possível visualizar e gerenciar os DBaaS e suas pilhas de recursos. A Figura 4.9 apresenta a tela da aplicação onde é possível visualizar

todos os DBaaS registrados, essa tela também destaca os três serviços DBaaS que possuem mais recursos cadastrados e em monitoramento. A partir dessa tela, é possível entrar no detalhe de um determinado DBaaS, como apresentado na Figura 4.10, onde é possível visualizar as informações de cadastro do DBaaS e o detalhe dos seus recursos cadastrados nas diversas camadas. Mais *screenshots* podem ser vistos no site[9] da aplicação, assim como, mais detalhes e funcionalidades.

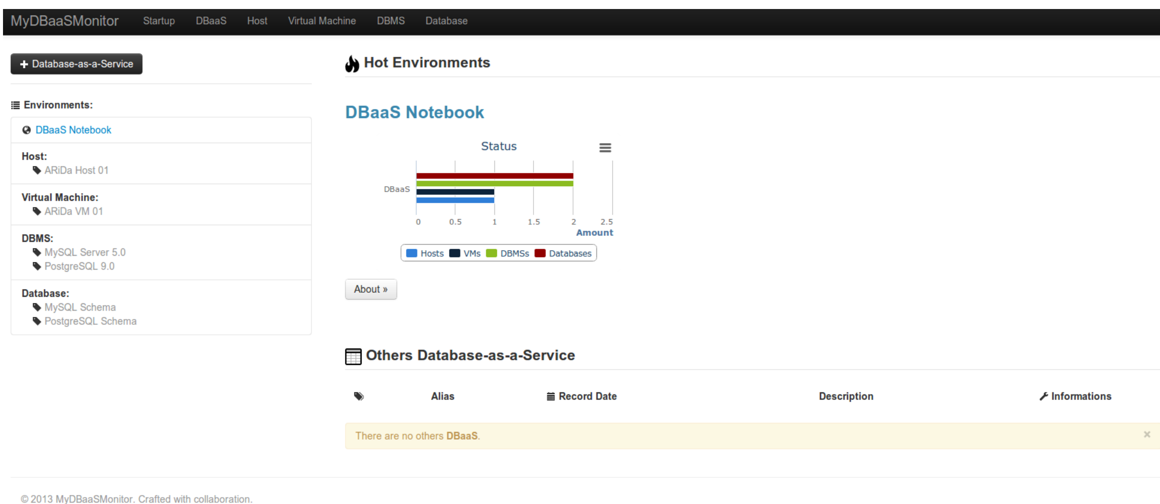


Figura 4.9 Tela para visualização dos DBaaS cadastrados no ambiente de monitoramento da aplicação.

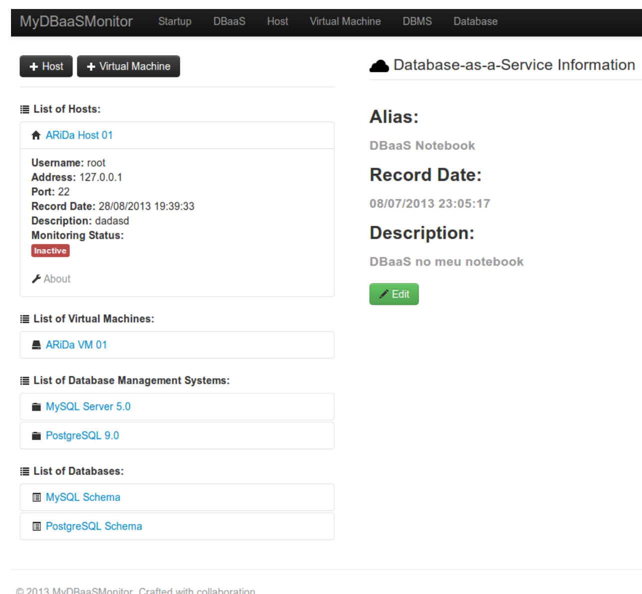


Figura 4.10 Tela para visualização em detalhe de um DBaaS cadastrado.

[9]http://mydbaasmonitor.com/screenshots.html

4.1.5 Configuração do Monitoramento

A configuração do monitoramento dos recursos na aplicação acontece em dois momentos, com a configuração do agente de monitoramento para execução em servidores e com a configuração do agente de monitoramento para execução em máquinas virtuais. O processo de configuração acontece a partir da seleção das métricas desejadas para serem coletadas sobre um determinado tipo de recurso e pela informação do ciclo de monitoramento em que cada métrica deve ser coletada. A Figura 4.11 apresenta a tela de configuração do agente de monitoramento para os servidores. Com base nas métricas definidas na seção 4.1.1, a tela disponibiliza para o usuário as métricas para camada física, assim como, as métricas para camada virtual que também podem ser coletadas nos servidores. Para selecionar as métricas desejadas, o usuário basta informar o ciclo de monitoramento nas métricas que deseja coletar, as que possuírem o ciclo vazio ou zero não serão coletadas pelo agente. Como exemplo, a figura demonstra que as métricas *Cpu*, *Memory* e *Network* serão monitoradas em ciclos de cinco segundos, já a métrica *Disk* possuirá o ciclo de quinze segundos e a métrica *Partition* não será coletada, pois nenhum valor foi informado. Por fim, as métricas da camada física serão coletadas em 20 e 10 segundos respectivamente.

Figura 4.11 Tela de configuração do agente de monitoramento para Servidor.

Como os recursos da camada de dados estão hospedados nas VMs, é da respon-

sabilidade do agente de monitoramento executado sobre as VMs monitorar também as métricas definidas para os SGBDs e instâncias de banco de dados. A Figura 4.12 apresenta a tela de configuração do agente de monitoramento para as máquinas virtuais, e por consequência para os SGBDs/instâncias de banco de dados existentes na VM. Também como definidas na seção 4.1.1, a tela disponibiliza para o usuário as métricas para a camada virtual, assim como, as métricas para a camada de dados que também serão coletadas pelo agente de monitoramento. A seleção das métricas que serão coletadas é semelhante a tela de configuração dos servidores, porém nas métricas da camada de dados é necessário selecionar em quais os recursos a métrica deverá ser coletada.

Setup Agent
Resource: ARiDa VM 01
1. If the cycle of a metric is zero or empty it is considered disabled.
2. When you set the cycle of a metric of DBMS or database check whether you really selected some resource.
Virtual Machine Metrics:
Cpu
Cycle in seconds
Memory
Cycle in seconds
Network
Cycle in seconds
Disk
Cycle in seconds
Partition
Cycle in seconds
DBMS and Database Metrics:
Active Connections
Cycle in seconds
DBMSs Available:
DBMS 01
Databases Available:
Screencast Schema
teste
Size
Cycle in seconds
DBMSs Available:
DBMS 01
Databases Available:
Screencast Schema
teste
Process Status
Cycle in seconds
DBMSs Available:
DBMS 01
Network Traffic
Cycle in seconds
DBMSs Available:
DBMS 01
Information Data
Cycle in seconds
DBMSs Available:
DBMS 01
Information Table
Cycle in seconds
Databases Available:
Screencast Schema
teste
Statement DML
Cycle in seconds
DBMSs Available:
DBMS 01
Statement TCL
Cycle in seconds
DBMSs Available:
DBMS 01
Statement DDL
Cycle in seconds
DBMSs Available:
DBMS 01
Statement DCL
Cycle in seconds
DBMSs Available:
DBMS 01
Disk Utilization
Cycle in seconds
DBMSs Available:
DBMS 01
Start
Cancel

Figura 4.12 Tela de configuração do agente de monitoramento para Máquina Virtual.

Após a confirmação da configuração de monitoramento, o sistema cria automaticamente o arquivo de contexto conforme apresentado na seção 3.2.4, acessa o recurso via SSH para enviar o agente de monitoramento e o arquivo de contexto com a configuração selecionada. Com os arquivos enviados, a aplicação inicializa o agente de monitoramento, que automaticamente dispara os coletores com base nas métricas e ciclos de monitoramento informados no arquivo de contexto. Esse processo de configuração pode ser refeito caso exista a necessidade de executar o monitoramento com uma configuração diferente. Por fim, o monitoramento é inicializado por completo e o usuário pode acompanhar as métricas que configurou conforme apresentado na seção a seguir.

4.1.6 Monitoramento dos Recursos das Camadas

A aplicação MyDBaaS Monitor além de criar a base serial histórica disponibilizada pela estrutura do *framework* MyDBaaS, possui interfaces visuais que permitem o usuário acompanhar o processo de monitoramento sobre os recursos e para visualização das informações de cadastro. Para cada tipo de recurso definido nas camadas que compõem o serviço DBaaS, a aplicação disponibiliza uma interface web implementada com base nas métricas que foram definidas para aquele tipo de recurso. Essas interfaces web são compostas por gráficos dinâmicos e em tempo real, que apresentam as medidas que estão sendo coletadas no momento pelos agentes de monitoramento configurados.

Como destacado, os gráficos foram criados com base nas métricas definidas anteriormente. Porém, algumas métricas possuem diversas medidas como apresentado na seção 4.1.1, com isso, alguns gráficos são derivados da mesma métrica. A Figura 4.13 apresenta a interface web para o acompanhamento do monitoramento sobre um servidor. São disponibilizados dez gráficos, que por ordem apresentam: quantidade ativa e inativa de máquinas virtuais hospedadas, porcentagem de utilização de CPU e memória, quantidade de *bytes* e pacotes enviados/recebidos pelo servidor, quantidade de requisições e *bytes* lidos/escritos no disco do servidor e o estado de utilização do armazenamento do servidor. A Figura 4.14 apresenta a interface web para o acompanhamento do monitoramento sobre uma máquina virtual. Também são disponibilizados dez gráficos: porcentagem de utilização de CPU e memória, porcentagem de utilização de CPU e memória pelo processo da VM dentro do servidor, quantidade de *bytes* e pacotes enviados/recebidos pela VM, quantidade de requisições e *bytes* lidos/escritos no disco da VM e o estado de utilização do armazenamento da VM.

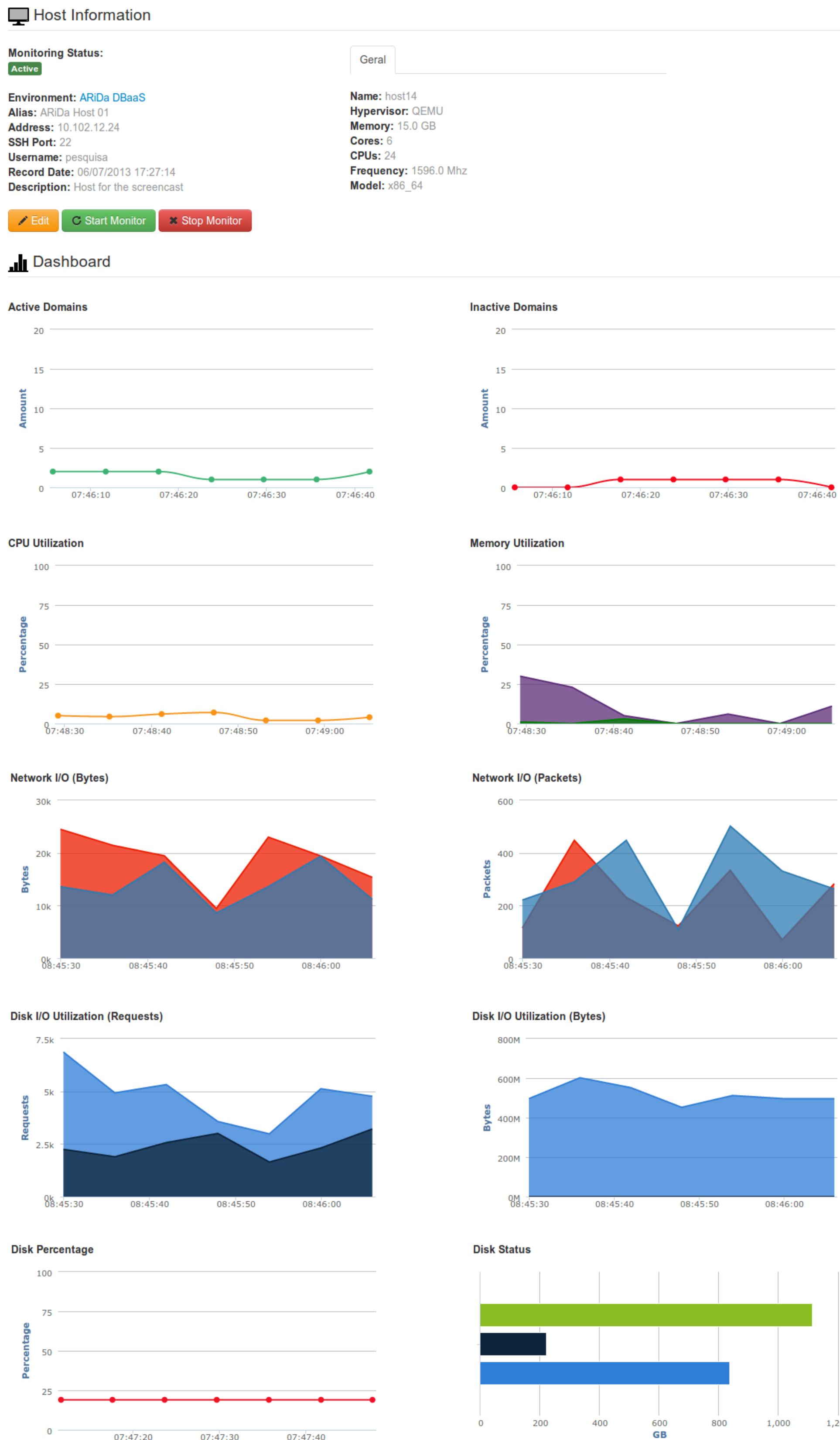


Figura 4.13 Painel de acompanhamento do monitoramento para servidor.

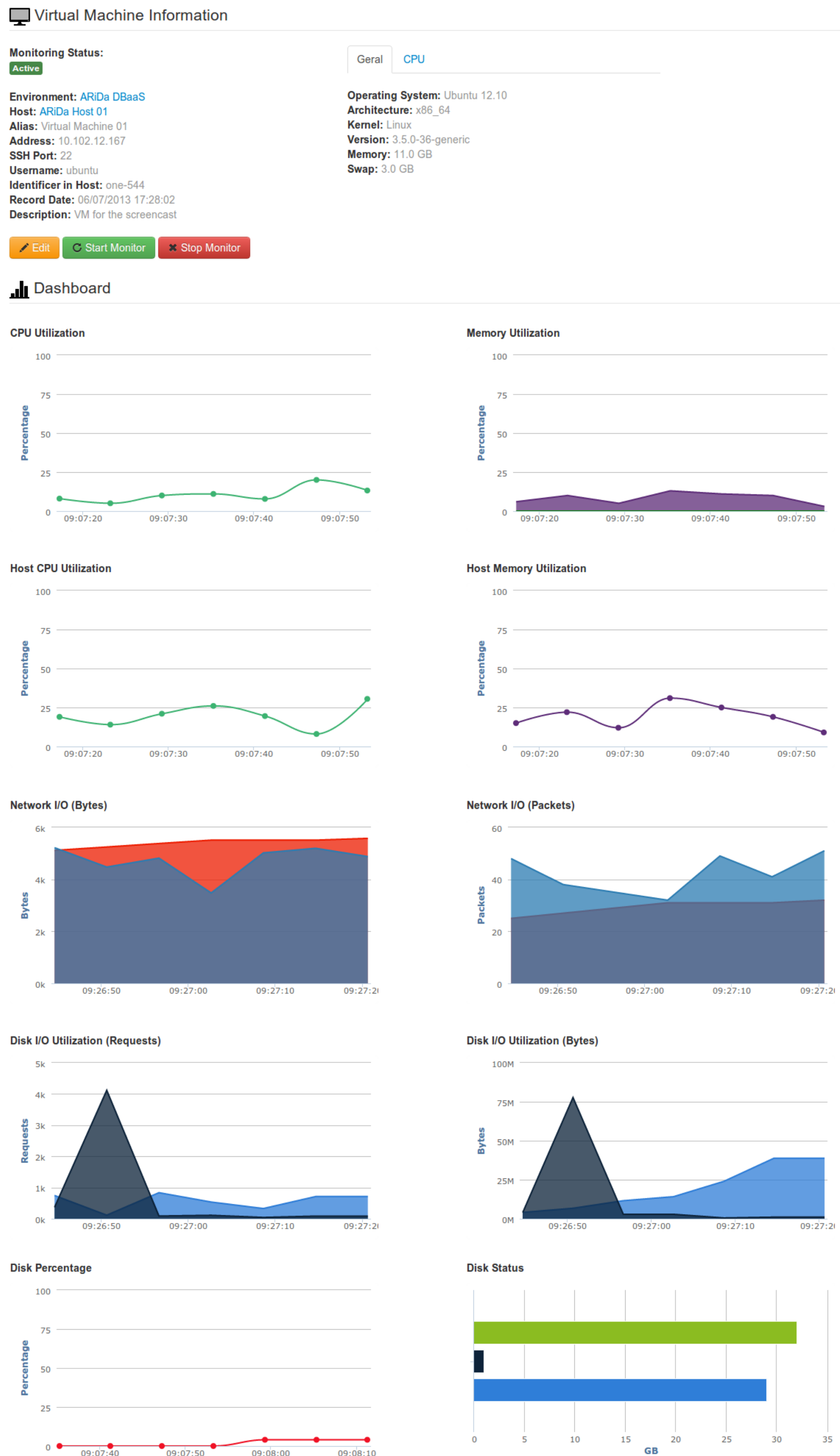


Figura 4.14 Painel de acompanhamento do monitoramento para máquina virtual.

As Figuras 4.15 e 4.16 apresentam a interface web para o acompanhamento do monitoramento sobre um SGBD, com base nas métricas definidas são disponibilizados dezoito gráficos: quantidade dos principais objetos existentes dentro do SGBD, quantidade de CPU e memória que os processos do SGBD estão utilizando na VM, quantidade total de conexões ativas, quantidade total de armazenamento utilizada, quantidade de *bytes* enviados/recebidos pelo SGBD, quantidade de requisições de leituras físicas/lógicas ao disco, quantidade de *bytes* lidos/escritos no disco, quantidade de páginas lidas/escritas, quantidade de leituras/escritas em blocos do disco, quantidade de leituras/escritas pendentes para acesso ao disco, quantidade de comandos DML executados, quantidade de comandos TCL executados, quantidade de comandos DDL executados e quantidade de comandos DCL executados.

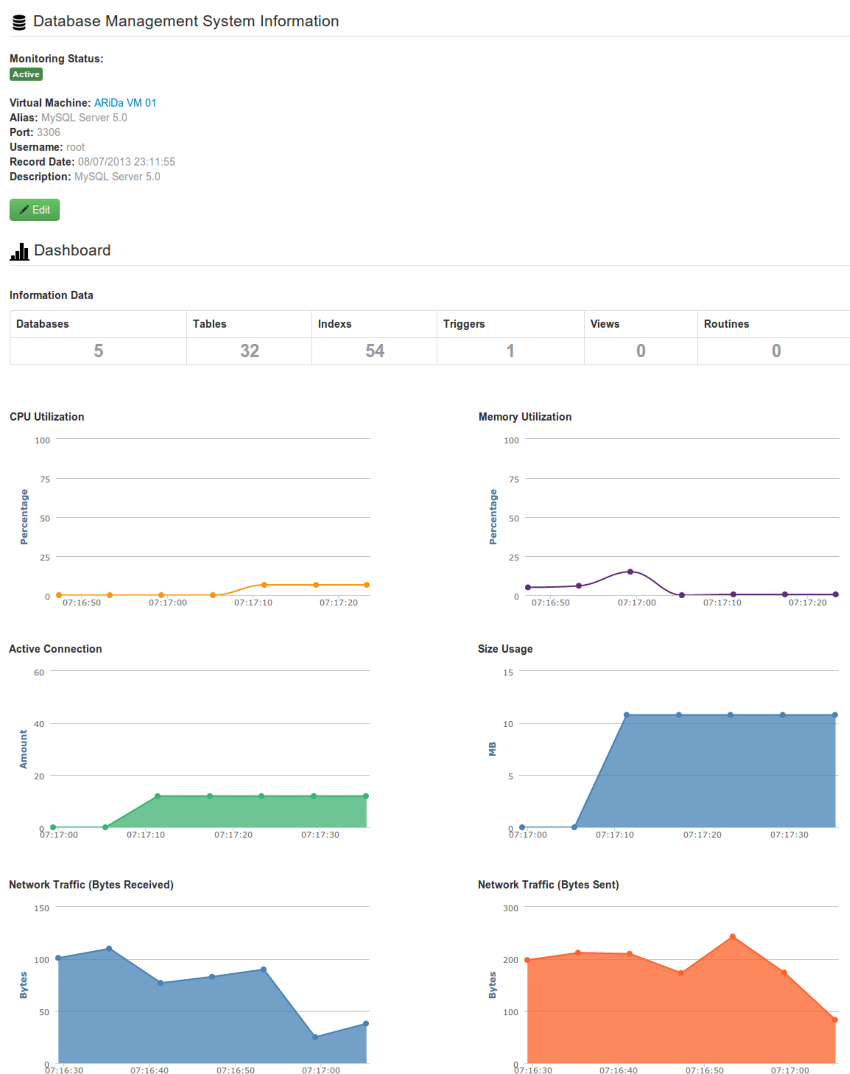


Figura 4.15 Painel de acompanhamento do monitoramento para SGBD (Parte 1).

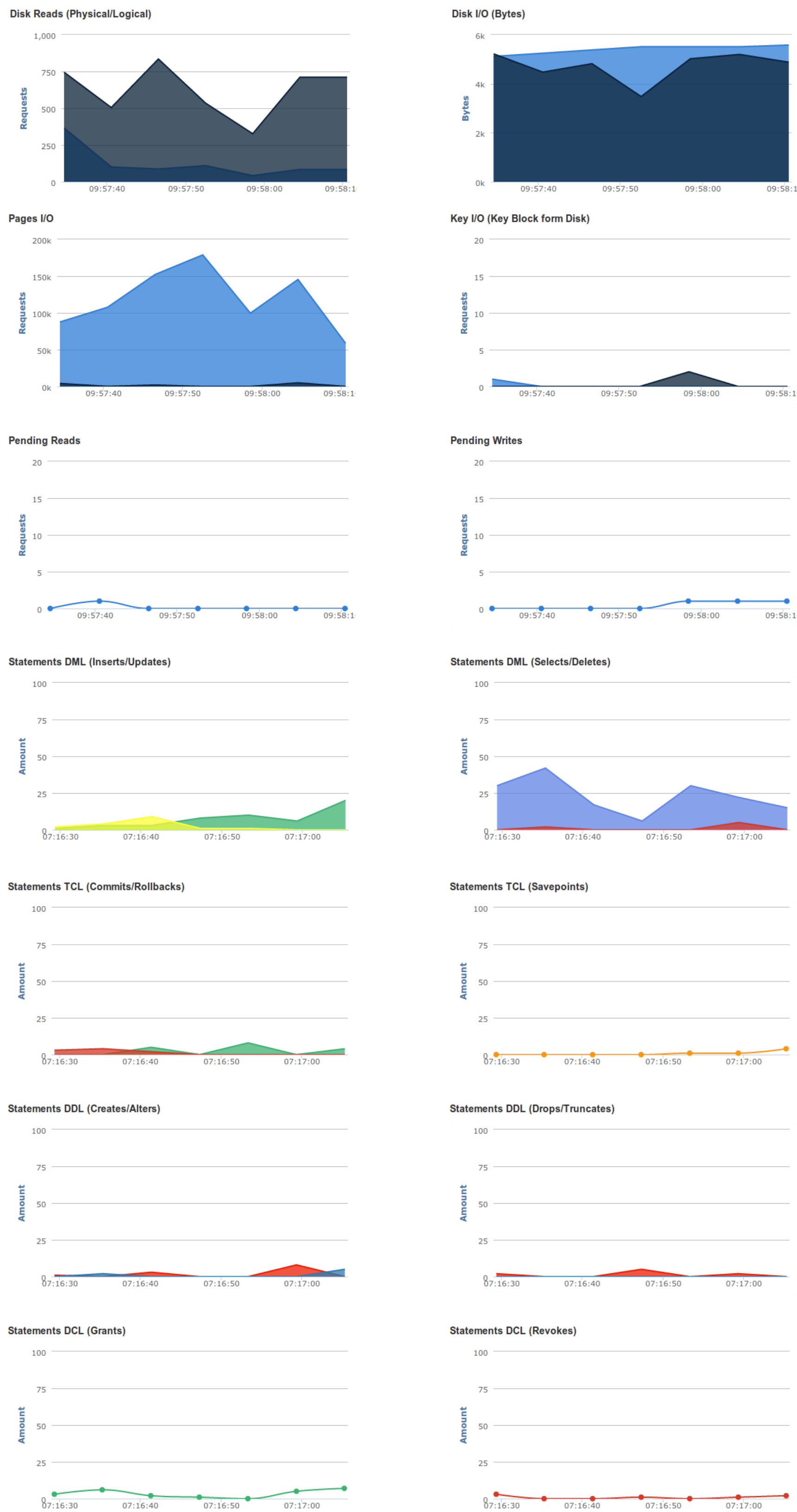


Figura 4.16 Painel de acompanhamento do monitoramento para SGBD (Parte 2).

Por fim, a Figura 4.17 apresenta a interface web para o acompanhamento do monitoramento sobre uma instância de banco de dados, mas também é apresentado o monitoramento sobre as cargas de trabalho enviadas à instância. São disponibilizados para visualização um gráfico da quantidade de conexões abertas com a instância de banco de dados e um com a quantidade total de armazenamento utilizada somente pela instância. Também são disponibilizadas uma tabela que apresenta o estado das tabelas existentes na instância e uma tabela que apresenta o estado das cargas de trabalho executadas.

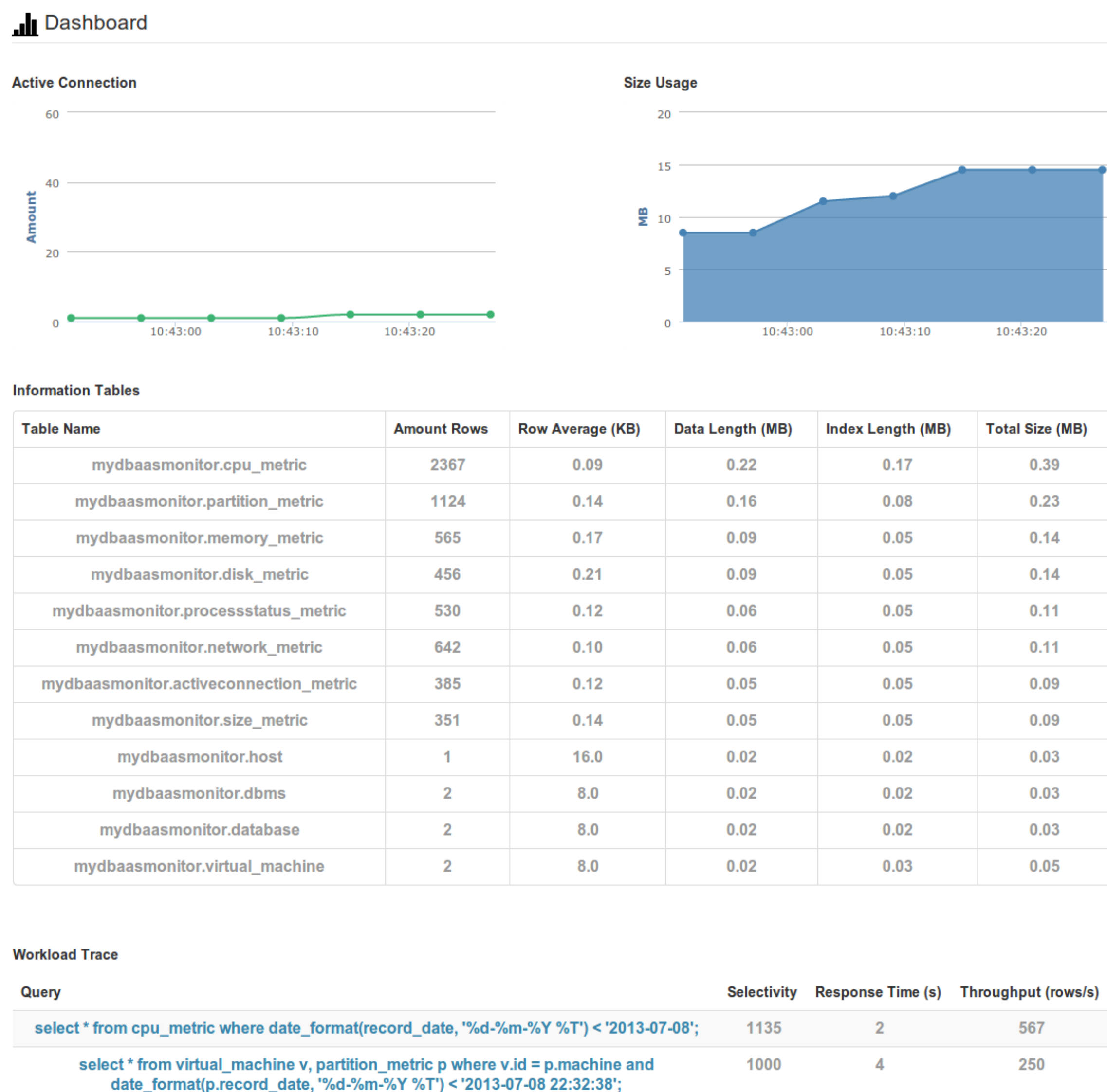


Table Name	Amount Rows	Row Average (KB)	Data Length (MB)	Index Length (MB)	Total Size (MB)
mydbaasmonitor.cpu_metric	2367	0.09	0.22	0.17	0.39
mydbaasmonitor.partition_metric	1124	0.14	0.16	0.08	0.23
mydbaasmonitor.memory_metric	565	0.17	0.09	0.05	0.14
mydbaasmonitor.disk_metric	456	0.21	0.09	0.05	0.14
mydbaasmonitor.processstatus_metric	530	0.12	0.06	0.05	0.11
mydbaasmonitor.network_metric	642	0.10	0.06	0.05	0.11
mydbaasmonitor.activeconnection_metric	385	0.12	0.05	0.05	0.09
mydbaasmonitor.size_metric	351	0.14	0.05	0.05	0.09
mydbaasmonitor.host	1	16.0	0.02	0.02	0.03
mydbaasmonitor.dbms	2	8.0	0.02	0.02	0.03
mydbaasmonitor.database	2	8.0	0.02	0.02	0.03
mydbaasmonitor.virtual_machine	2	8.0	0.02	0.03	0.05

Query	Selectivity	Response Time (s)	Throughput (rows/s)
select * from cpu_metric where date_format(record_date, '%d-%m-%Y %T') < '2013-07-08';	1135	2	567
select * from virtual_machine v, partition_metric p where v.id = p.machine and date_format(p.record_date, '%d-%m-%Y %T') < '2013-07-08 22:32:38';	1000	4	250

Figura 4.17 Painel de acompanhamento do monitoramento para instância de banco de dados/carga de trabalho.

4.1.7 Estudo de Caso

O estudo de caso apresentado nesta seção tem como objetivo demonstrar a abrangência da estrutura de monitoramento, fornecida pelo *framework* MyDBaaS e implementada pela aplicação MyDBaaS Monitor, sobre as camadas de recursos que compõem um serviço DBaaS. Para execução deste estudo de caso foi utilizado o TPC-H *Benchmark* [49], que é um *benchmark* de suporte a decisões que consiste em um conjunto de consultas *ad-hoc* orientadas a negócios. Neste sentido, escolhemos três consultas aleatórias desse conjunto disponibilizado para serem executadas em série sobre uma instância de banco de dados de um DBaaS cadastrado dentro do ambiente de monitoramento da aplicação. A execução das três consultas foi realizada utilizado o *MyDBaaS Driver*, o que possibilitou o monitoramento da camada de carga de trabalho. Para as demais camadas, foram monitoradas métricas referentes à utilização do disco, uma vez que, é uma informação importante e essencial para consolidação de serviços de banco de dados em nuvem. Como esses serviços são compostos por uma pilha de camadas de recursos, a ideia é apresentar a capacidade de monitoramento da aplicação sobre uma mesma informação entre as camadas.

A Figura 4.18 apresenta o monitoramento sobre a carga de trabalho, onde foi possível coletar a seletividade, tempo de resposta e *throughput* das três consultas executadas. Durante esse espaço de tempo os agentes de monitoramento coletarem em ciclos de 5 segundos na camada física e virtual a métrica *Disk* e na camada de dados a métrica *DiskUtilization*.

Workload Trace

Query	Selectivity (rows)	Response Time (s)	Throughput (rows/s)	Record
select s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment from part, supplier, partsupp, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and p_size = 15 and p_type like '%BRASS' and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE' and ps_supplycost = (select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and s_suppkey = ps_suppkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'EUROPE') order by s_acctbal desc, n_name, s_name, p_partkey;	947	55.5	17.06	01:03:11 04-09-2013
select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'BUILDING' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15' group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate;	23170	719.5	32.20	01:04:07 04-09-2013
select supp_nation, cust_nation, l_year, sum(volume) as revenue from (select n1.n_name as supp_nation, n2.n_name as cust_nation, extract(year from l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as volume from supplier, lineitem, orders, customer, nation n1, nation n2 where s_suppkey = l_suppkey and o_orderkey = l_orderkey and c_custkey = o_custkey and s_nationkey = n1.n_nationkey and c_nationkey = n2.n_nationkey and ((n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY') or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE'))) as shipping group by supp_nation, cust_nation, l_year order by supp_nation, cust_nation, l_year;	14	16.2	0.86	01:16:07 04-09-2013

Figura 4.18 Métrica coletada sobre a camada de carga de trabalho.

As Figuras 4.19 e 4.20 apresentam os gráficos de I/O em disco para quantidade de requisições realizadas e *bytes* consumidos nas camadas física e virtual respectivamente.

Figura 4.19 Métrica coletada sobre a camada física.

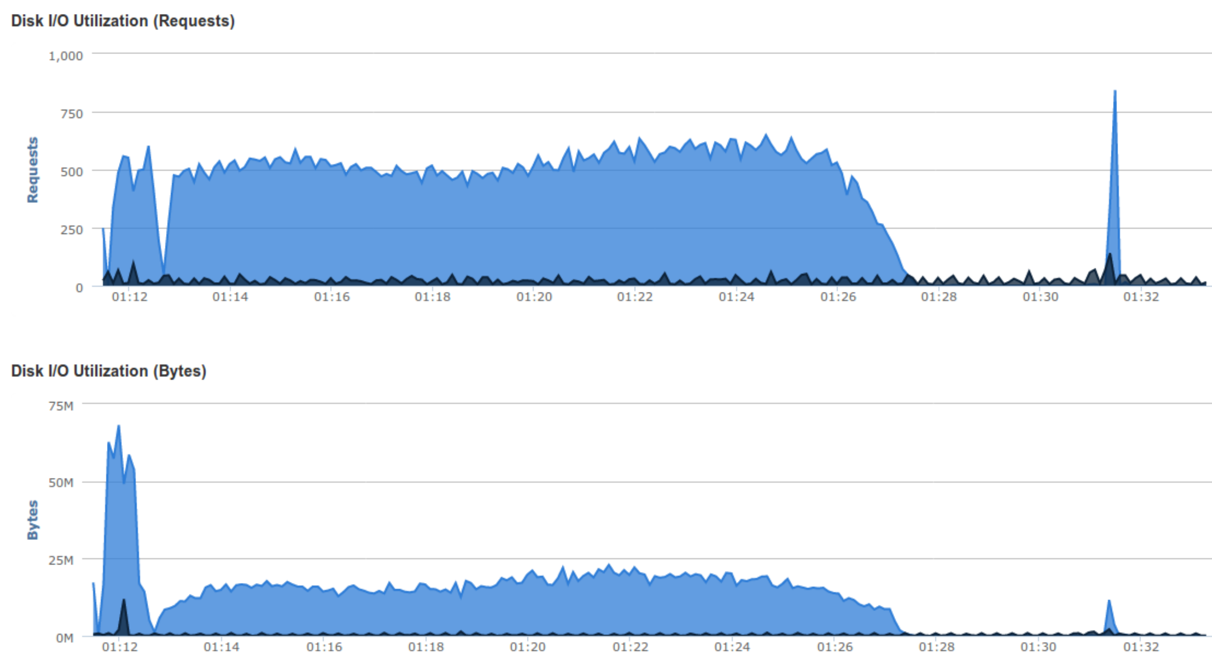


Figura 4.20 Métrica coletada sobre a camada virtual.

A Figura 4.21 apresenta os gráficos referentes as medidas coletadas sobre disco na camada de dados, onde é possível visualizar a quantidade de requisições de leitura (física/lógica) realizadas, quantidade de I/O em *bytes* e a quantidade de I/O em páginas realizadas pelo SGBD durante a execução das três consultas.

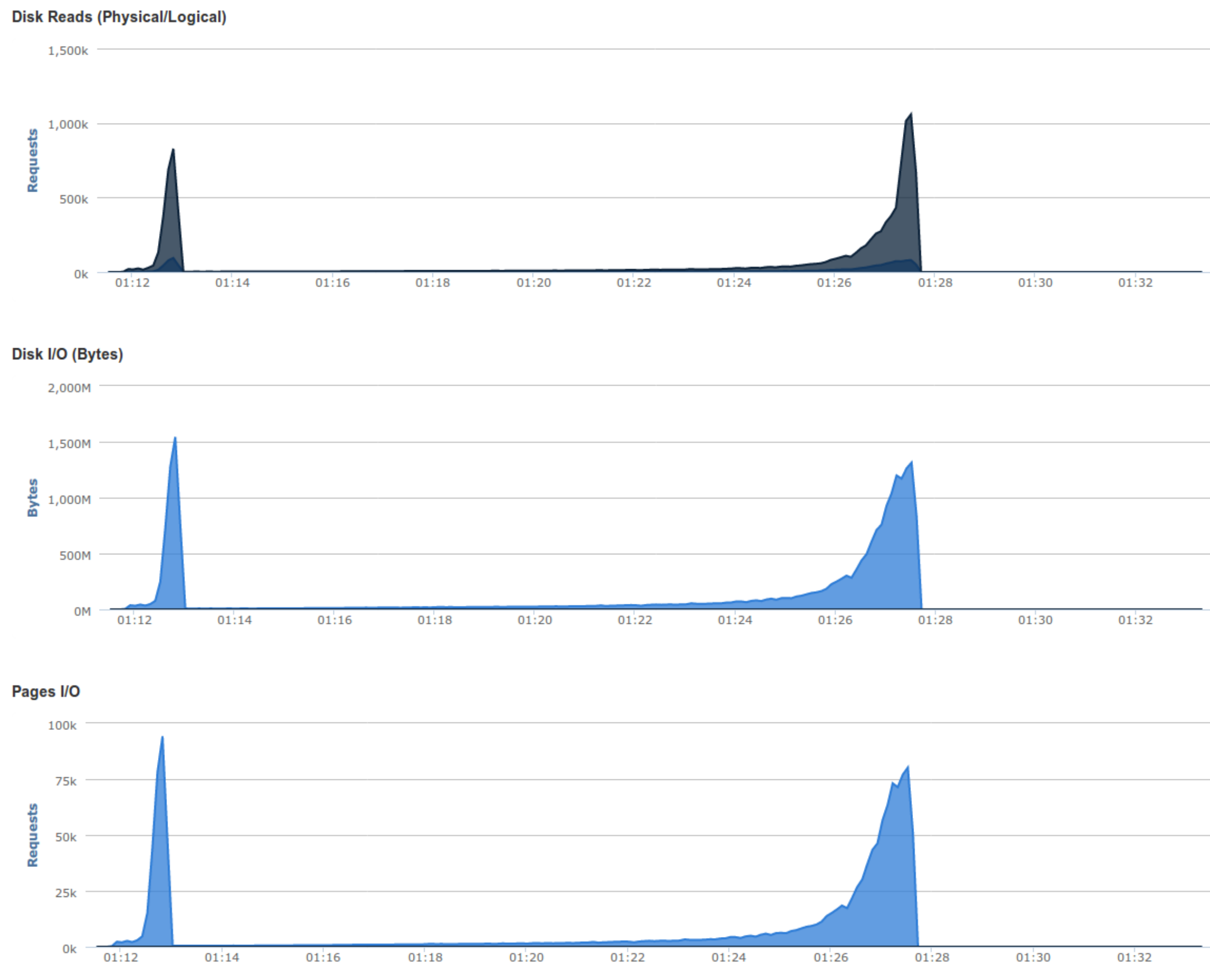


Figura 4.21 Métrica coletada sobre a camada de dados.

Com isso, destacamos que apesar dos gráficos apresentarem as mesmas métricas para as camadas, apresentam resultados bastantes diferentes, possibilitando a análise mais detalhada na execução das cargas de trabalho sobre cada camada. Como é possível visualizar na Figura 4.19 a quantidade de requisições de leitura na camada física alcançou mais de 15 mil e a quantidade de *bytes* lidos pouco mais de 600MB, por outro lado, na Figura 4.20 a quantidade de requisições de leitura na camada virtual no mesmo instante de tempo não passou de 750 e a quantidade de *bytes* lidos chegou a pouco mais de 60MB.

Na camada de dados, a Figura 4.21 demonstra que o SGBD no mesmo instante de tempo realizou mais requisições de leitura lógica do física e com relação aos *bytes* lidos registrou em torno de 1.500MB. A Figura 4.22 apresenta um *snapshot* de como o componente *Repository* armazenou a métrica *DiskUtilization*, utilizada nesse estudo de caso, com base na sua definição na base conhecimento histórica.

record_date	physicalreads	logicalreads	pendingreads	pendingwrites	dataread	datawritten	pagesread	pageswritten	keyread	keywrites	dbms	database
2013-09-04 23:03:01	0	0	0	0	0	0	0	0	0	0	1	NULL
2013-09-04 23:03:05	0	26	0	0	0	1536	0	0	0	0	1	NULL
2013-09-04 23:03:10	0	27	0	0	0	526336	0	16	0	0	1	NULL
2013-09-04 23:03:15	257	1461	1	0	4227072	1536	257	0	0	0	1	NULL
2013-09-04 23:03:20	2256	19603	0	0	36962304	395776	2256	12	0	0	1	NULL
2013-09-04 23:03:25	1996	17660	0	0	32702464	1536	1996	0	0	0	1	NULL
2013-09-04 23:03:30	2635	24150	0	0	43171840	197120	2635	0	0	0	1	NULL
2013-09-04 23:03:35	2100	15896	0	0	34406400	527872	2100	22	0	0	1	NULL
2013-09-04 23:03:40	2896	28327	0	0	47448064	1536	2896	0	0	0	1	NULL
2013-09-04 23:03:45	4679	43576	0	0	76660736	1536	4679	0	0	0	1	NULL
2013-09-04 23:03:50	15111	131679	0	0	247578624	461312	15111	14	0	0	1	NULL
2013-09-04 23:03:55	44225	377032	0	0	724582400	1536	44225	0	0	0	1	NULL

Figura 4.22 Exemplo do armazenamento da métrica *DiskUtilization* na base serial histórica.

4.2 CONCLUSÃO

Este capítulo apresentou o MyDBaaS Monitor, uma aplicação web open-source para o gerenciamento do monitoramento de serviços DBaaS, possibilitando o monitoramento e acompanhamento de forma transparente e automática dos diversos recursos que compõem esses serviços. Uma especificação do conjunto de métricas definidas nas diversas camadas, assim como, os coletores e *receivers* implementados para o funcionamento da aplicação foi apresentada. Foram destacados as funcionalidades que o MyDBaaS Monitor disponibiliza para o gerenciamento dos recursos, para a configuração do monitoramento e para o acompanhamento em tempo real do monitoramento realizado pelos agentes. Por fim, um estudo de caso demonstrando a abrangência da estrutura de monitoramento fornecida pelo MyDBaaS Monitor com base no *framework* MyDBaaS foi apresentada.

CAPÍTULO 5

CONCLUSÃO

Esta dissertação apresentou o MyDBaaS, um *framework* para o monitoramento de ambientes de banco de dados em nuvem cuja finalidade é possibilitar a criação de soluções de monitoramento personalizáveis e eficientes. O MyDBaaS oferece uma programação abrangente e um modelo extensível para o desenvolvimento de estratégias para o monitoramento de serviços DBaaS, disponibilizando um conjunto de *frozen spots* que tornam a utilização fácil e transparente, e também um conjunto de *hot spots* que tornam sua estrutura extensível e flexível conforme necessário.

Com isso, essa dissertação alcançou as contribuições apresentadas no Capítulo 1, onde o *framework* de aplicação proposto para instanciação de ferramentas para monitoramento de ambientes de DBaaS foi totalmente baseado nos requisitos levantados no Capítulo 2, visando contemplar as necessidades de monitoramento existentes em serviços de nuvem, mais especificamente para serviços de DBaaS. Assim, o MyDBaaS contribuiu com padrões para definição de métricas para os diversos tipos de recursos e também para definição dos coletores das métricas, não limitando a forma como o usuário deseja monitorar seus serviços de DBaaS. Com base nesses padrões, foram definidos e implementados um conjunto de 21 métricas e coletores para o monitoramento de recursos nas camadas física, virtual, de dados e de carga de trabalho, como apresentado nas seções 4.1.1 e 4.1.2.

A definição e implementação de uma API para o consumo das métricas coletadas pelo ambiente de monitoramento criado pelo MyDBaaS, foi apresentada na seção 3.2.6, permitindo que aplicações possam consultar as métricas que estão sendo coletadas sobre os recursos cadastrados no ambiente de monitoramento e utilizá-las para diversos objetivos, tais como: controle de SLAs, gerenciamento da qualidade de serviço, estratégias de replicação e elasticidade, ou outras decisões que sejam necessárias. Também permitindo o usuário realizar o consumo das métricas de diversas maneiras pelas aplicações, possibilitando um consumo flexível, como por exemplo em função do tempo e/ou quantidade de coletas. A especificação e implementação de uma base de dados histórica para o armazenamento das métricas coletadas nas diversas camadas de recursos foram apresentadas na

seção 3.2.5, onde o componente *Repository* do MyDBaaS, é responsável por tornar totalmente transparente ao usuário o acesso a essa base de histórica, provendo métodos para salvar e consultar os dados independentemente de qual seja a métrica definida. A estrutura dessa base é expandida de acordo com as métricas definidas e coletadas, onde para cada métrica uma tabela é criada conforme a sua definição e relacionada a tabela de recurso em que a mesma é coletada.

A implementação de uma ferramenta para o monitoramento de serviços de banco de dados em nuvem baseada no *framework* MyDBaaS, foi apresentada no Capítulo 4. O MyDBaaS Monitor é uma aplicação web open-source para o gerenciamento do monitoramento de serviços DBaaS, possibilitando o monitoramento e acompanhamento de forma transparente e automática dos diversos recursos que compõem esses serviços. O MyDBaaS Monitor permite o usuário cadastrar os recursos existentes em cada camada do serviço DBaaS através de interfaces web, assim como, realizar a configuração do monitoramento de forma fácil e também acompanhar em tempo real o monitoramento executado sobre cada recurso através de gráficos dinâmicos. É importante destacar que o MyDBaaS Monitor é apenas um modelo de instanciação do *framework* MyDBaaS. Também no Capítulo 4, é apresentado um estudo de caso a partir do MyDBaaS Monitor, que refletiu a abrangência da estrutura de monitoramento fornecida pelo MyDBaaS, que possibilita a realização de análises mais detalhadas sobre as execuções das diversas cargas de trabalho que um serviço de DBaaS recebe e os reflexos das mesmas sobre cada camada de recurso existente.

O trabalho proposto mostrou bons resultados para o desenvolvimento de soluções de monitoramento para ambientes de DBaaS. Porém, melhorias podem ser realizadas através de trabalhos futuros, como destacado na seção a seguir.

5.1 OPORTUNIDADES PARA TRABALHOS FUTUROS

Como trabalhos futuros pretende-se, primeiramente, adicionar um módulo para exportar as métricas coletadas de forma personalizada. Com isso, pode-se fornecer dados para diferentes ferramentas de análise, R[1] e Weka[2], por exemplo.

Na nuvem, o custo é uma característica fundamental e deve ser considerada no

[1]http://www.r-project.org
[2]http://www.cs.waikato.ac.nz/ml/weka

monitoramento. Dessa forma, o MyDBaaS pode ser estendido para contabilizar todos os custos associados ao SGBD e, consequentemente, fornecer uma solução com contabilização total dos custos para os usuários. Além disso, poder-se utilizar as informações de SLAs para definir intervalo de valores, permitindo a notificação dos usuários de acordo com estes valores.

O MyDBaaS não fornece uma forma de escolha da estratégia de monitoramento a ser utilizada. Sendo assim, outro direcionamento a trabalhos futuros seria permitir ao usuário especificar informações sobre o monitoramento, tal como se alguma agregação deve ser utilizada antes do envio das métricas coletadas a base serial histórica. O *framework* MyDBaaS foi validado por meio de um estudo de caso, que demonstrou conformidade e extensibilidade da solução proposta. Apesar dos resultados apresentados serem interessantes, é fundamental verificar outros aspectos. Assim sendo, outro direcionamento para trabalhos futuros é avaliar o MyDBaaS considerando aspectos de escalabilidade, *overhead*, quantidade de mensagens e tráfego de dados na rede. Para tanto, será utilizada a infraestrutura de nuvem da UFC com uma grande quantidade de máquinas e diferentes cargas de trabalho. Tais experimentos permitirão avaliar o comportamento do MyDBaaS e, se for o caso, aprimorar seus algoritmos. Em relação aos experimentos referentes a disponibilidade, pretende-se desenvolver experimentos para validar a dependabilidade total do MyDBaaS, englobando também a confiabilidade. Para tanto, é necessário desenvolver experimentos detalhados com injeção de falhas.

Com relação à avaliação de desempenho, também é proposto a avaliação do MyDBaaS em ambientes de nuvens híbridas ou em diferentes zonas de disponibilidade com o intuito de identificar a variação no desempenho adicionado em decorrência da latência da rede. Para tanto, são necessários identificar parâmetros e desenvolver cenários que contemplem um ambiente mais geral.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: A software platform for .net-based cloud computing. *CoRR*, abs/0907.4622:267–295, 2009.

[2] Daniel J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009.

[3] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[4] Flávio R. Carvalho Sousa, Leonardo Moreira, and Javam Machado. Gerenciamento de dados em nuvem: Conceitos, sistemas e desafios. In *PEREIRA, A. C. M.; PAPPA, G. L.; WINCKLER, M.; GOMES, R. L. (Org.). Tópicos em Sistemas Colaborativos, Interativos, Multimídia, Web e Bancos de Dados, SIWB*, SBBD '10, pages 101–130, Belo Horizonte, MG, 2010.

[5] Peter Mell and Tim Grance. NIST W*orking* D*efinition of* C*loud* C*omputing* (Draft), 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing.* Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[7] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[8] Divyakant Agrawal, Amr El Abbadi, Fatih Emekci, and Ahmed Metwally. Database management as a service: Challenges and opportunities. volume 0, pages 1709–1716. IEEE Computer Society, 2009.

[9] Sherif Sakr. Cloud-hosted databases: technologies, challenges and opportunities. pages 1–16. Cluster Computing Journal, Springer US, 2013.

[10] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database service for the cloud. In *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2011.

[11] Cezar Taurion. *Cloud Computing: Computação em nuvem transformando o mundo da Tecnologia da Informação.* Addison-Wesley Professional, Boston, USA, 2009.

[12] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[13] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Survey cloud monitoring: A survey. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 57(9):2093–2115, June 2013.

[14] Amazon. Amazon CloudWatch, 2013. `http://aws.amazon.com/cloudwatch`.

[15] Paraleap Technologies. AzureWatch, 2013. `http://www.paraleap.com/AzureWatch`.

[16] Inc New Relic. New Relic: Application Performance Management and Monitoring, 2013. `http://newrelic.com`.

[17] OpenNebula Project. *OpenNebula: Open Source Data Center Virtualization*, 2013. `http://opennebula.org`.

[18] G Katsaros, R. Kubert, and G Gallizo. Building a service-oriented monitoring framework with rest and nagios. SCC '11, pages 426 – 431. IEEE International Conference on Services Computing, 2011.

[19] Markus Klems, David Bermbach, and Rene Weinert. A Runtime Quality Measurement Framework for Cloud Database Service Systems. QUATIC '11, pages 38–46. IEEE Computer Society, 2012.

[20] Vincent C. Emeakaroha, Ivona Brandic, Michael Maurer, and Schahram Dustdar. Low level metrics to high level slas - lom2his framework: Bridging the gap between

monitored metrics and sla parameters in cloud environments. In Waleed W. Smari and John P. McIntire, editors, *HPCS*, pages 48–54. IEEE, 2010.

[21] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A runtime model based monitoring approach for cloud. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 313–320, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Manoel Veras and Robert Tozer. *Cloud Computing: nova arquitetura da* TI. Editora Brasport, Rio de Janeiro, RJ, Brasil, 2012.

[23] Flávio R. Carvalho Sousa, Leonardo Moreira, and Javam Machado. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. In *In: III Escola Regional de Computação Ceará – Maranhão – Piauí*, ERCEMAPI '09, page Capítulo 7, Paraíba, PI, 2009.

[24] Fabio Perez Marzullo. *SOA na Prática: inovando seu negócio por meio de soluções orientadas a serviços.* Novatec Editora, São Paulo, SP, Brasil, 2009.

[25] Carlos Eduardo Marins. Desafios da informática forense no cenário de cloud computing. In *The Fourth International Conference on Forensic Computer Science*, ICoFCS '09, pages 78–85, Natal, RN, BRA, 2009.

[26] John W. Rittinghouse and James F. Ransome. *Cloud Computing: Implementation, Management and Security.* CRC Press, Boca Raton, FL, USA, 2010.

[27] Ronald L. Krutz and Russell Dean Vines. *Cloud Security: A Comprehensive Guide to Secure Cloud Computing.* Willey Publishing, Indianapolis, USA, 2010.

[28] Judith Hurwitz, Robin Bloor, Marcia Kaufman, and Fern Halper. *Cloud Computing For Dummies.* Wiley, For Dummies, 2009.

[29] George Reese. *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud.* O'Reilly Media, Sebastopol, USA, 2010.

[30] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.* O'Reilly Media, Sebastopol, USA, 2009.

[31] Vadim Matveev. Platform as a service: new opportunities for software development companies. Master's thesis, (Department of Information Technology) - Faculty of Technology Management. Lappeenranta University of Technology, 2010.

[32] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.* O'Reilly Media, Inc., 2009.

[33] Michael Miller. *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online.* Que Publishing, Indianapolis, USA, 2008.

[34] Ronald L. Krutz and Russell Dean Vines. *Cloud Security: A Comprehensive Guide to Secure Cloud Computing.* JJohn Wiley & Sons, 2010.

[35] Cloud Security Alliance. *Security Guidance for Critical Areas of Focus in Cloud Computing v3.0*, 2011. `https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf`.

[36] Amazon. *Elastic Compute Cloud*, 2013. `http://http://aws.amazon.com/ec2`.

[37] OpenNebula Project. *OpenNebula Detailed Features and Functionality*, 2013. `http://www.opennebula.org/documentation:features`.

[38] OpenNebula Project. *OpenNebula: Monitoring Subsystem*, 2013. `http://opennebula.org/documentation:rel4.2:img`.

[39] OpenNebula Project. *OpenNebula: Ganglia Monitoring*, 2013. `http://opennebula.org/documentation:rel4.2:ganglia`.

[40] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[41] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of the Third Conference on Pattern Languages and Programming.* Addison-Wesley, 1996.

[42] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[43] Ian Sommerville. *Software Engineering.* Pearson Education, Inc, Boston, Massachusetts, USA, 9th edition, 2010.

[44] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Companies, Inc, New York, NY, USA, 7th edition, 2010.

[45] Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM*, 40(10):70–77, October 1997.

[46] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, pages 150–162, London, UK, UK, 1994. Springer-Verlag.

[47] Garry Froehlich, H James Hoover, Ling Liu, and Paul G Sorenson. Reusing application frameworks through hooks. *John Wiley*, 1998.

[48] Leonard Richardson and Sam Ruby. *RESTful Web Services: Web services for the real world*. O'Reilly Media, Sebastopol, CA, USA, 2007.

[49] TPC. Tpc-h *Benchmark*, 2013. `http://www.tpc.org/tpch/spec/tpch2.16.0.pdf`.

APÊNDICE A

CLASSES DE DEFINIÇÃO DAS MÉTRICAS

A.1 CAMADA FÍSICA

```
public class HostDomains extends AbstractMetric {

    private int hostDomainsInactive;
    private int hostDomainsActive;

    public int getHostDomainsInactive() {
        return hostDomainsInactive;
    }

    public void setHostDomainsInactive(int hostDomainsInactive) {
        this.hostDomainsInactive = hostDomainsInactive;
    }

    public int getHostDomainsActive() {
        return hostDomainsActive;
    }

    public void setHostDomainsActive(int hostDomainsActive) {
        this.hostDomainsActive = hostDomainsActive;
    }

    @Override
    public String toString() {
        return "host";
    }

    @Override
    public List<HostDomains> jsonToList(String json) {
        Gson gson = new Gson();
        List<HostDomains> hostDomainsList = gson.fromJson(json, new TypeToken<List<HostDomains>>(){}.getType());
        return hostDomainsList;
    }
}
```

Figura A.1 Implementação da métrica *HostDomains*.

```java
public class DomainStatus extends AbstractMetric {

    private String domainStatusHostIdentifier;
    private double domainStatusCpuPercent;
    private double domainStatusMemoryPercent;

    public String getDomainStatusHostIdentifier() {
        return domainStatusHostIdentifier;
    }

    public void setDomainStatusHostIdentifier(String domainStatusHostIdentifier) {
        this.domainStatusHostIdentifier = domainStatusHostIdentifier;
    }

    public double getDomainStatusCpuPercent() {
        return domainStatusCpuPercent;
    }

    public void setDomainStatusCpuPercent(double domainStatusCpuPercent) {
        this.domainStatusCpuPercent = domainStatusCpuPercent;
    }

    public double getDomainStatusMemoryPercent() {
        return domainStatusMemoryPercent;
    }

    public void setDomainStatusMemoryPercent(double domainStatusMemoryPercent) {
        this.domainStatusMemoryPercent = domainStatusMemoryPercent;
    }

    @Override
    public String toString() {
        return "host";
    }

    @Override
    public List<DomainStatus> jsonToList(String json) {
        Gson gson = new Gson();
        List<DomainStatus> domainStatusList = gson.fromJson(json, new TypeToken<List<DomainStatus>>(){}.getType());
        return domainStatusList;
    }
}
```

Figura A.2 Implementação da métrica *DomainStatus*.

```java
public class HostInfo extends AbstractMetric {

    private String hostInfoName;
    private String hostInfoHypervisor;
    private int hostInfoCores;
    private int hostInfoCpus;
    private double hostInfoMemory;
    private double hostInfoMhz;
    private String hostInfoModel;

    public String getHostInfoName() {
        return hostInfoName;
    }

    public void setHostInfoName(String hostInfoName) {
        this.hostInfoName = hostInfoName;
    }

    public String getHostInfoHypervisor() {
        return hostInfoHypervisor;
    }

    public void setHostInfoHypervisor(String hostInfoHypervisor) {
        this.hostInfoHypervisor = hostInfoHypervisor;
    }

    public int getHostInfoCores() {
        return hostInfoCores;
    }

    public void setHostInfoCores(int hostInfoCores) {
        this.hostInfoCores = hostInfoCores;
    }

    public int getHostInfoCpus() {
        return hostInfoCpus;
    }

    public void setHostInfoCpus(int hostInfoCpus) {
        this.hostInfoCpus = hostInfoCpus;
    }

    public double getHostInfoMemory() {
        return hostInfoMemory;
    }

    public void setHostInfoMemory(double hostInfoMemory) {
        this.hostInfoMemory = hostInfoMemory;
    }

    public double getHostInfoMhz() {
        return hostInfoMhz;
    }

    public void setHostInfoMhz(double hostInfoMhz) {
        this.hostInfoMhz = hostInfoMhz;
    }

    public String getHostInfoModel() {
        return hostInfoModel;
    }

    public void setHostInfoModel(String hostInfoModel) {
        this.hostInfoModel = hostInfoModel;
    }

    @Override
    public String toString() {
        return "host";
    }

    @Override
    public List<HostInfo> jsonToList(String json) {
        Gson gson = new Gson();
        List<HostInfo> hostInfoList = gson.fromJson(json, new TypeToken<List<HostInfo>>(){}.getType());
        return hostInfoList;
    }
}
```

Figura A.3 Implementação da métrica *HostInfo*.

A.2 CAMADA VIRTUAL

```
public class Cpu extends AbstractMetric {

    private double cpuUser;
    private double cpuSystem;
    private double cpuNice;
    private double cpuWait;
    private double cpuIdle;
    private double cpuCombined;

    public double getCpuUser() {
        return cpuUser;
    }

    public void setCpuUser(double cpuUser) {
        this.cpuUser = cpuUser;
    }

    public double getCpuSystem() {
        return cpuSystem;
    }

    public void setCpuSystem(double cpuSystem) {
        this.cpuSystem = cpuSystem;
    }

    public double getCpuNice() {
        return cpuNice;
    }

    public void setCpuNice(double cpuNice) {
        this.cpuNice = cpuNice;
    }

    public double getCpuWait() {
        return cpuWait;
    }

    public void setCpuWait(double cpuWait) {
        this.cpuWait = cpuWait;
    }

    public double getCpuIdle() {
        return cpuIdle;
    }

    public void setCpuIdle(double cpuIdle) {
        this.cpuIdle = cpuIdle;
    }

    public double getCpuCombined() {
        return cpuCombined;
    }

    public void setCpuCombined(double cpuCombined) {
        this.cpuCombined = cpuCombined;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Cpu> jsonToList(String json) {
        Gson gson = new Gson();
        List<Cpu> cpuList = gson.fromJson(json, new TypeToken<List<Cpu>>(){}.getType());
        return cpuList;
    }
}
```

Figura A.4 Implementação da métrica *Cpu*.

```java
public class Memory extends AbstractMetric {

    private long memorySwapUsed;
    private long memorySwapFree;
    private double memorySwapUsedPercent;
    private double memorySwapFreePercent;
    private long memoryUsed;
    private long memoryFree;
    private double memoryUsedPercent;
    private double memoryFreePercent;
    private long memoryBuffersCacheUsed;
    private long memoryBuffersCacheFree;

    public long getMemorySwapUsed() {
        return memorySwapUsed;
    }

    public void setMemorySwapUsed(long memorySwapUsed) {
        this.memorySwapUsed = memorySwapUsed;
    }

    public long getMemorySwapFree() {
        return memorySwapFree;
    }

    public void setMemorySwapFree(long memorySwapFree) {
        this.memorySwapFree = memorySwapFree;
    }

    public long getMemoryUsed() {
        return memoryUsed;
    }

    public void setMemoryUsed(long memoryUsed) {
        this.memoryUsed = memoryUsed;
    }

    public long getMemoryFree() {
        return memoryFree;
    }

    public void setMemoryFree(long memoryFree) {
        this.memoryFree = memoryFree;
    }

    public double getMemoryUsedPercent() {
        return memoryUsedPercent;
    }

    public void setMemoryUsedPercent(double memoryUsedPercent) {
        this.memoryUsedPercent = memoryUsedPercent;
    }

    public double getMemoryFreePercent() {
        return memoryFreePercent;
    }

    public void setMemoryFreePercent(double memoryFreePercent) {
        this.memoryFreePercent = memoryFreePercent;
    }

    public long getMemoryBuffersCacheUsed() {
        return memoryBuffersCacheUsed;
    }

    public void setMemoryBuffersCacheUsed(long memoryBuffersCacheUsed) {
        this.memoryBuffersCacheUsed = memoryBuffersCacheUsed;
    }

    public long getMemoryBuffersCacheFree() {
        return memoryBuffersCacheFree;
    }

    public void setMemoryBuffersCacheFree(long memoryBuffersCacheFree) {
        this.memoryBuffersCacheFree = memoryBuffersCacheFree;
    }

    public double getMemorySwapUsedPercent() {
        return memorySwapUsedPercent;
    }

    public void setMemorySwapUsedPercent(double memorySwapUsedPercent) {
        this.memorySwapUsedPercent = memorySwapUsedPercent;
    }

    public double getMemorySwapFreePercent() {
        return memorySwapFreePercent;
    }

    public void setMemorySwapFreePercent(double memorySwapFreePercent) {
        this.memorySwapFreePercent = memorySwapFreePercent;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Memory> jsonToList(String json) {
        Gson gson = new Gson();
        List<Memory> memoryList = gson.fromJson(json, new TypeToken<List<Memory>>(){}.getType());
        return memoryList;
    }
}
```

Figura A.5 Implementação da métrica *Memory*.

```java
public class Network extends AbstractMetric {

    private long networkBytesSent;
    private long networkBytesReceived;
    private long networkPacketsSent;
    private long networkPacketsReceived;

    public long getNetworkBytesSent() {
        return networkBytesSent;
    }

    public void setNetworkBytesSent(long networkBytesSent) {
        this.networkBytesSent = networkBytesSent;
    }

    public long getNetworkBytesReceived() {
        return networkBytesReceived;
    }

    public void setNetworkBytesReceived(long networkBytesReceived) {
        this.networkBytesReceived = networkBytesReceived;
    }

    public long getNetworkPacketsSent() {
        return networkPacketsSent;
    }

    public void setNetworkPacketsSent(long networkPacketsSent) {
        this.networkPacketsSent = networkPacketsSent;
    }

    public long getNetworkPacketsReceived() {
        return networkPacketsReceived;
    }

    public void setNetworkPacketsReceived(long networkPacketsReceived) {
        this.networkPacketsReceived = networkPacketsReceived;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Network> jsonToList(String json) {
        Gson gson = new Gson();
        List<Network> networkList = gson.fromJson(json, new TypeToken<List<Network>>(){}.getType());
        return networkList;
    }
}
```

Figura A.6 Implementação da métrica *Network*.

```java
public class Disk extends AbstractMetric {

    private long diskBytesRead;
    private long diskBytesWritten;
    private long diskReads;
    private long diskWrites;
    private double diskFree;
    private double diskUsed;
    private double diskTotal;
    private double diskPercent;

    public long getDiskBytesRead() {
        return diskBytesRead;
    }

    public void setDiskBytesRead(long diskBytesRead) {
        this.diskBytesRead = diskBytesRead;
    }

    public long getDiskBytesWritten() {
        return diskBytesWritten;
    }

    public void setDiskBytesWritten(long diskBytesWritten) {
        this.diskBytesWritten = diskBytesWritten;
    }

    public long getDiskReads() {
        return diskReads;
    }

    public void setDiskReads(long diskReads) {
        this.diskReads = diskReads;
    }

    public long getDiskWrites() {
        return diskWrites;
    }

    public void setDiskWrites(long diskWrites) {
        this.diskWrites = diskWrites;
    }

    public double getDiskFree() {
        return diskFree;
    }

    public void setDiskFree(double diskFree) {
        this.diskFree = diskFree;
    }

    public double getDiskUsed() {
        return diskUsed;
    }

    public void setDiskUsed(double diskUsed) {
        this.diskUsed = diskUsed;
    }

    public double getDiskTotal() {
        return diskTotal;
    }

    public void setDiskTotal(double diskTotal) {
        this.diskTotal = diskTotal;
    }

    public double getDiskPercent() {
        return diskPercent;
    }

    public void setDiskPercent(double diskPercent) {
        this.diskPercent = diskPercent;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Disk> jsonToList(String json) {
        Gson gson = new Gson();
        List<Disk> diskList = gson.fromJson(json, new TypeToken<List<Disk>>(){}.getType());
        return diskList;
    }
}
```

Figura A.7 Implementação da métrica *Disk*.

```java
public class Partition extends AbstractMetric {

    private String partitionDirectoryName;
    private String partitionDeviceName;
    private long partitionBytesRead;
    private long partitionBytesWritten;
    private long partitionReads;
    private long partitionWrites;
    private long partitionFreeBytes;
    private long partitionUsedBytes;
    private long partitionTotalBytes;
    private double partitionPercent;

    public String getPartitionDirectoryName() {
        return partitionDirectoryName;
    }

    public void setPartitionDirectoryName(String partitionDirectoryName) {
        this.partitionDirectoryName = partitionDirectoryName;
    }

    public String getPartitionDeviceName() {
        return partitionDeviceName;
    }

    public void setPartitionDeviceName(String partitionDeviceName) {
        this.partitionDeviceName = partitionDeviceName;
    }

    public long getPartitionBytesRead() {
        return partitionBytesRead;
    }

    public void setPartitionBytesRead(long partitionBytesRead) {
        this.partitionBytesRead = partitionBytesRead;
    }

    public long getPartitionBytesWritten() {
        return partitionBytesWritten;
    }

    public void setPartitionBytesWritten(long partitionBytesWritten) {
        this.partitionBytesWritten = partitionBytesWritten;
    }

    public long getPartitionReads() {
        return partitionReads;
    }

    public void setPartitionReads(long partitionReads) {
        this.partitionReads = partitionReads;
    }

    public long getPartitionWrites() {
        return partitionWrites;
    }

    public void setPartitionWrites(long partitionWrites) {
        this.partitionWrites = partitionWrites;
    }

    public long getPartitionFreeBytes() {
        return partitionFreeBytes;
    }

    public void setPartitionFreeBytes(long partitionFreeBytes) {
        this.partitionFreeBytes = partitionFreeBytes;
    }

    public long getPartitionUsedBytes() {
        return partitionUsedBytes;
    }

    public void setPartitionUsedBytes(long partitionUsedBytes) {
        this.partitionUsedBytes = partitionUsedBytes;
    }

    public long getPartitionTotalBytes() {
        return partitionTotalBytes;
    }

    public void setPartitionTotalBytes(long partitionTotalBytes) {
        this.partitionTotalBytes = partitionTotalBytes;
    }

    public double getPartitionPercent() {
        return partitionPercent;
    }

    public void setPartitionPercent(double partitionPercent) {
        this.partitionPercent = partitionPercent;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Partition> jsonToList(String json) {
        Gson gson = new Gson();
        List<Partition> partitionList = gson.fromJson(json, new TypeToken<List<Partition>>(){}.getType());
        return partitionList;
    }
}
```

Figura A.8 Implementação da métrica *Partition*.

```java
public class Machine extends AbstractMetric {

    private String machineOperatingSystem;
    private String machineKernelName;
    private String machineKernelVersion;
    private String machineArchitecture;
    private double machineTotalMemory;
    private double machineTotalSwap;
    private int machineTotalCPUCores;
    private int machineTotalCPUSockets;
    private int machineTotalCoresPerSocket;

    public String getMachineOperatingSystem() {
        return machineOperatingSystem;
    }

    public void setMachineOperatingSystem(String machineOperatingSystem) {
        this.machineOperatingSystem = machineOperatingSystem;
    }

    public String getMachineKernelName() {
        return machineKernelName;
    }

    public void setMachineKernelName(String machineKernelName) {
        this.machineKernelName = machineKernelName;
    }

    public String getMachineKernelVersion() {
        return machineKernelVersion;
    }

    public void setMachineKernelVersion(String machineKernelVersion) {
        this.machineKernelVersion = machineKernelVersion;
    }

    public String getMachineArchitecture() {
        return machineArchitecture;
    }

    public void setMachineArchitecture(String machineArchitecture) {
        this.machineArchitecture = machineArchitecture;
    }

    public double getMachineTotalMemory() {
        return machineTotalMemory;
    }

    public void setMachineTotalMemory(double machineTotalMemory) {
        this.machineTotalMemory = machineTotalMemory;
    }

    public double getMachineTotalSwap() {
        return machineTotalSwap;
    }

    public void setMachineTotalSwap(double machineTotalSwap) {
        this.machineTotalSwap = machineTotalSwap;
    }

    public int getMachineTotalCPUCores() {
        return machineTotalCPUCores;
    }

    public void setMachineTotalCPUCores(int machineTotalCPUCores) {
        this.machineTotalCPUCores = machineTotalCPUCores;
    }

    public int getMachineTotalCPUSockets() {
        return machineTotalCPUSockets;
    }

    public void setMachineTotalCPUSockets(int machineTotalCPUSockets) {
        this.machineTotalCPUSockets = machineTotalCPUSockets;
    }

    public int getMachineTotalCoresPerSocket() {
        return machineTotalCoresPerSocket;
    }

    public void setMachineTotalCoresPerSocket(int machineTotalCoresPerSocket) {
        this.machineTotalCoresPerSocket = machineTotalCoresPerSocket;
    }

    @Override
    public String toString() {
        return "machine";
    }

    @Override
    public List<Machine> jsonToList(String json) {
        Gson gson = new Gson();
        List<Machine> machineList = gson.fromJson(json, new TypeToken<List<Machine>>(){}.getType());
        return machineList;
    }
}
```

Figura A.9 Implementação da métrica *Machine*.

A.3 CAMADA DE DADOS

```
public class ActiveConnection extends AbstractDatabaseMetric {

    private int activeConnectionAmount;

    public int getActiveConnectionAmount() {
        return activeConnectionAmount;
    }

    public void setActiveConnectionAmount(int activeConnectionAmount) {
        this.activeConnectionAmount = activeConnectionAmount;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<ActiveConnection> jsonToList(String json) {
        Gson gson = new Gson();
        List<ActiveConnection> processStatusList = gson.fromJson(json, new TypeToken<List<ActiveConnection>>(){}.getType());
        return processStatusList;
    }
}
```

Figura A.10 Implementação da métrica *ActiveConnection*.

```
public class Size extends AbstractDatabaseMetric {

    private double sizeUsed;

    public double getSizeUsed() {
        return sizeUsed;
    }

    public void setSizeUsed(double sizeUsed) {
        this.sizeUsed = sizeUsed;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<Size> jsonToList(String json) {
        Gson gson = new Gson();
        List<Size> sizeList = gson.fromJson(json, new TypeToken<List<Size>>(){}.getType());
        return sizeList;
    }
}
```

Figura A.11 Implementação da métrica *Size*.

```java
public class NetworkTraffic extends AbstractDatabaseMetric {

    private int networkTrafficBytesReceived;
    private int networkTrafficBytesSent;

    public int getNetworkTrafficBytesReceived() {
        return networkTrafficBytesReceived;
    }

    public void setNetworkTrafficBytesReceived(int networkTrafficBytesReceived) {
        this.networkTrafficBytesReceived = networkTrafficBytesReceived;
    }

    public int getNetworkTrafficBytesSent() {
        return networkTrafficBytesSent;
    }

    public void setNetworkTrafficBytesSent(int networkTrafficBytesSent) {
        this.networkTrafficBytesSent = networkTrafficBytesSent;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<NetworkTraffic> jsonToList(String json) {
        Gson gson = new Gson();
        List<NetworkTraffic> networkTrafficList = gson.fromJson(json, new TypeToken<List<NetworkTraffic>>(){}.getType());
        return networkTrafficList;
    }
}
```

Figura A.12 Implementação da métrica *NetworkTraffic*.

```java
public class ProcessStatus extends AbstractDatabaseMetric {

    private double processStatusCpu;
    private double processStatusMemory;

    public double getProcessStatusCpu() {
        return processStatusCpu;
    }

    public void setProcessStatusCpu(double processStatusCpu) {
        this.processStatusCpu = processStatusCpu;
    }

    public double getProcessStatusMemory() {
        return processStatusMemory;
    }

    public void setProcessStatusMemory(double processStatusMemory) {
        this.processStatusMemory = processStatusMemory;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<ProcessStatus> jsonToList(String json) {
        Gson gson = new Gson();
        List<ProcessStatus> processStatusList = gson.fromJson(json, new TypeToken<List<ProcessStatus>>(){}.getType());
        return processStatusList;
    }
}
```

Figura A.13 Implementação da métrica *ProcessStatus*.

```java
public class InformationData extends AbstractDatabaseMetric {

    private int informationDataDatabases;
    private int informationDataTables;
    private int informationDataIndexs;
    private int informationDataTriggers;
    private int informationDataViews;
    private int informationDataRoutines;

    public int getInformationDataDatabases() {
        return informationDataDatabases;
    }

    public void setInformationDataDatabases(int informationDataDatabases) {
        this.informationDataDatabases = informationDataDatabases;
    }

    public int getInformationDataTables() {
        return informationDataTables;
    }

    public void setInformationDataTables(int informationDataTables) {
        this.informationDataTables = informationDataTables;
    }

    public int getInformationDataIndexs() {
        return informationDataIndexs;
    }

    public void setInformationDataIndexs(int informationDataIndexs) {
        this.informationDataIndexs = informationDataIndexs;
    }

    public int getInformationDataTriggers() {
        return informationDataTriggers;
    }

    public void setInformationDataTriggers(int informationDataTriggers) {
        this.informationDataTriggers = informationDataTriggers;
    }

    public int getInformationDataViews() {
        return informationDataViews;
    }

    public void setInformationDataViews(int informationDataViews) {
        this.informationDataViews = informationDataViews;
    }

    public int getInformationDataRoutines() {
        return informationDataRoutines;
    }

    public void setInformationDataRoutines(int informationDataRoutines) {
        this.informationDataRoutines = informationDataRoutines;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<InformationData> jsonToList(String json) {
        Gson gson = new Gson();
        List<InformationData> informationDataList = gson.fromJson(json, new TypeToken<List<InformationData>>(){}.getType());
        return informationDataList;
    }
}
```

Figura A.14 Implementação da métrica *InformationData*.

```java
public class InformationTable extends AbstractDatabaseMetric {

    private String informationTableName;
    private double informationTableDataLength;
    private double informationTableIndexLength;
    private long informationTableAmountRows;
    private double informationTableRowAverage;
    private double informationTableTotalSize;

    public String getInformationTableName() {
        return informationTableName;
    }

    public void setInformationTableName(String informationTableName) {
        this.informationTableName = informationTableName;
    }

    public double getInformationTableDataLength() {
        return informationTableDataLength;
    }

    public void setInformationTableDataLength(double informationTableDataLength) {
        this.informationTableDataLength = informationTableDataLength;
    }

    public double getInformationTableIndexLength() {
        return informationTableIndexLength;
    }

    public void setInformationTableIndexLength(double informationTableIndexLength) {
        this.informationTableIndexLength = informationTableIndexLength;
    }

    public long getInformationTableAmountRows() {
        return informationTableAmountRows;
    }

    public void setInformationTableAmountRows(long informationTableAmountRows) {
        this.informationTableAmountRows = informationTableAmountRows;
    }

    public double getInformationTableRowAverage() {
        return informationTableRowAverage;
    }

    public void setInformationTableRowAverage(double informationTableRowAverage) {
        this.informationTableRowAverage = informationTableRowAverage;
    }

    public double getInformationTableTotalSize() {
        return informationTableTotalSize;
    }

    public void setInformationTableTotalSize(double informationTableTotalSize) {
        this.informationTableTotalSize = informationTableTotalSize;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<InformationTable> jsonToList(String json) {
        Gson gson = new Gson();
        List<InformationTable> informationTableList = gson.fromJson(json, new TypeToken<List<InformationTable>>(){}.getType());
        return informationTableList;
    }
}
```

Figura A.15 Implementação da métrica *InformationTable*.

```java
public class StatementDML extends AbstractDatabaseMetric {

    private int statementDMLInserts;
    private int statementDMLSelects;
    private int statementDMLUpdates;
    private int statementDMLDeletes;

    public int getStatementDMLInserts() {
        return statementDMLInserts;
    }

    public void setStatementDMLInserts(int statementDMLInserts) {
        this.statementDMLInserts = statementDMLInserts;
    }

    public int getStatementDMLSelects() {
        return statementDMLSelects;
    }

    public void setStatementDMLSelects(int statementDMLSelects) {
        this.statementDMLSelects = statementDMLSelects;
    }

    public int getStatementDMLUpdates() {
        return statementDMLUpdates;
    }

    public void setStatementDMLUpdates(int statementDMLUpdates) {
        this.statementDMLUpdates = statementDMLUpdates;
    }

    public int getStatementDMLDeletes() {
        return statementDMLDeletes;
    }

    public void setStatementDMLDeletes(int statementDMLDeletes) {
        this.statementDMLDeletes = statementDMLDeletes;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<StatementDML> jsonToList(String json) {
        Gson gson = new Gson();
        List<StatementDML> statementDMLList = gson.fromJson(json, new TypeToken<List<StatementDML>>(){}.getType());
        return statementDMLList;
    }
}
```

Figura A.16 Implementação da métrica *StatementDML*.

```java
public class StatementTCL extends AbstractDatabaseMetric {

    private int statementTCLCommits;
    private int statementTCLRollback;
    private int statementTCLSavepoint;

    public int getStatementTCLCommits() {
        return statementTCLCommits;
    }

    public void setStatementTCLCommits(int statementTCLCommits) {
        this.statementTCLCommits = statementTCLCommits;
    }

    public int getStatementTCLRollback() {
        return statementTCLRollback;
    }

    public void setStatementTCLRollback(int statementTCLRollback) {
        this.statementTCLRollback = statementTCLRollback;
    }

    public int getStatementTCLSavepoint() {
        return statementTCLSavepoint;
    }

    public void setStatementTCLSavepoint(int statementTCLSavepoint) {
        this.statementTCLSavepoint = statementTCLSavepoint;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<StatementTCL> jsonToList(String json) {
        Gson gson = new Gson();
        List<StatementTCL> statementTCLList = gson.fromJson(json, new TypeToken<List<StatementTCL>>(){}.getType());
        return statementTCLList;
    }
}
```

Figura A.17 Implementação da métrica *StatementTCL*.

```java
public class StatementDDL extends AbstractDatabaseMetric {

    private int statementDDLCreate;
    private int statementDDLAlter;
    private int statementDDLDrop;
    private int statementDDLTruncate;
    private int statementDDLRename;

    public int getStatementDDLCreate() {
        return statementDDLCreate;
    }

    public void setStatementDDLCreate(int statementDDLCreate) {
        this.statementDDLCreate = statementDDLCreate;
    }

    public int getStatementDDLAlter() {
        return statementDDLAlter;
    }

    public void setStatementDDLAlter(int statementDDLAlter) {
        this.statementDDLAlter = statementDDLAlter;
    }

    public int getStatementDDLDrop() {
        return statementDDLDrop;
    }

    public void setStatementDDLDrop(int statementDDLDrop) {
        this.statementDDLDrop = statementDDLDrop;
    }

    public int getStatementDDLTruncate() {
        return statementDDLTruncate;
    }

    public void setStatementDDLTruncate(int statementDDLTruncate) {
        this.statementDDLTruncate = statementDDLTruncate;
    }

    public int getStatementDDLRename() {
        return statementDDLRename;
    }

    public void setStatementDDLRename(int statementDDLRename) {
        this.statementDDLRename = statementDDLRename;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<StatementDDL> jsonToList(String json) {
        Gson gson = new Gson();
        List<StatementDDL> statementDDLList = gson.fromJson(json, new TypeToken<List<StatementDDL>>(){}.getType());
        return statementDDLList;
    }
}
```

Figura A.18 Implementação da métrica *StatementDDL*.

```java
public class StatementDCL extends AbstractDatabaseMetric {

    private int statementDCLGrant;
    private int statementDCLRevoke;

    public int getStatementDCLGrant() {
        return statementDCLGrant;
    }

    public void setStatementDCLGrant(int statementDCLGrant) {
        this.statementDCLGrant = statementDCLGrant;
    }

    public int getStatementDCLRevoke() {
        return statementDCLRevoke;
    }

    public void setStatementDCLRevoke(int statementDCLRevoke) {
        this.statementDCLRevoke = statementDCLRevoke;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<StatementDCL> jsonToList(String json) {
        Gson gson = new Gson();
        List<StatementDCL> statementDCLList = gson.fromJson(json, new TypeToken<List<StatementDCL>>(){}.getType());
        return statementDCLList;
    }
}
```

Figura A.19 Implementação da métrica *StatementDCL*.

```java
public class DiskUtilization extends AbstractDatabaseMetric {

    private int diskUtilizationPhysicalReads;
    private int diskUtilizationLogicalReads;
    private int diskUtilizationPendingReads;
    private int diskUtilizationPendingWrites;
    private int diskUtilizationDataRead;
    private int diskUtilizationDataWritten;
    private int diskUtilizationPagesRead;
    private int diskUtilizationPagesWritten;
    private int diskUtilizationKeyRead;
    private int diskUtilizationkeyWrites;

    public int getDiskUtilizationPhysicalReads() {
        return diskUtilizationPhysicalReads;
    }

    public void setDiskUtilizationPhysicalReads(int diskUtilizationPhysicalReads) {
        this.diskUtilizationPhysicalReads = diskUtilizationPhysicalReads;
    }

    public int getDiskUtilizationLogicalReads() {
        return diskUtilizationLogicalReads;
    }

    public void setDiskUtilizationLogicalReads(int diskUtilizationLogicalReads) {
        this.diskUtilizationLogicalReads = diskUtilizationLogicalReads;
    }

    public int getDiskUtilizationPendingReads() {
        return diskUtilizationPendingReads;
    }

    public void setDiskUtilizationPendingReads(int diskUtilizationPendingReads) {
        this.diskUtilizationPendingReads = diskUtilizationPendingReads;
    }

    public int getDiskUtilizationPendingWrites() {
        return diskUtilizationPendingWrites;
    }

    public void setDiskUtilizationPendingWrites(int diskUtilizationPendingWrites) {
        this.diskUtilizationPendingWrites = diskUtilizationPendingWrites;
    }

    public int getDiskUtilizationDataRead() {
        return diskUtilizationDataRead;
    }

    public void setDiskUtilizationDataRead(int diskUtilizationDataRead) {
        this.diskUtilizationDataRead = diskUtilizationDataRead;
    }

    public int getDiskUtilizationDataWritten() {
        return diskUtilizationDataWritten;
    }

    public void setDiskUtilizationDataWritten(int diskUtilizationDataWritten) {
        this.diskUtilizationDataWritten = diskUtilizationDataWritten;
    }

    public int getDiskUtilizationPagesRead() {
        return diskUtilizationPagesRead;
    }

    public void setDiskUtilizationPagesRead(int diskUtilizationPagesRead) {
        this.diskUtilizationPagesRead = diskUtilizationPagesRead;
    }

    public int getDiskUtilizationPagesWritten() {
        return diskUtilizationPagesWritten;
    }

    public void setDiskUtilizationPagesWritten(int diskUtilizationPagesWritten) {
        this.diskUtilizationPagesWritten = diskUtilizationPagesWritten;
    }

    public int getDiskUtilizationKeyRead() {
        return diskUtilizationKeyRead;
    }

    public void setDiskUtilizationKeyRead(int diskUtilizationKeyRead) {
        this.diskUtilizationKeyRead = diskUtilizationKeyRead;
    }

    public int getDiskUtilizationkeyWrites() {
        return diskUtilizationkeyWrites;
    }

    public void setDiskUtilizationkeyWrites(int diskUtilizationkeyWrites) {
        this.diskUtilizationkeyWrites = diskUtilizationkeyWrites;
    }

    @Override
    public String toString() {
        return "database";
    }

    @Override
    public List<DiskUtilization> jsonToList(String json) {
        Gson gson = new Gson();
        List<DiskUtilization> diskUtilizationList = gson.fromJson(json, new TypeToken<List<DiskUtilization>>(){}.getType());
        return diskUtilizationList;
    }
}
```

Figura A.20 Implementação da métrica *DiskUtilization*.

APÊNDICE B

CLASSES PARA DEFINIÇÃO DOS COLETORES

B.1 CAMADA FÍSICA

```java
public class HostDomainsCollector extends AbstractCollector<HostDomainsMetric> {

    public HostDomainsCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws LibvirtException {
        this.metric = HostDomainsMetric.getInstance();
        Connect connect = (Connect) args[0];
        this.metric.setHostDomainsActive(connect.numOfDefinedDomains());
        this.metric.setHostDomainsInactive(connect.numOfDomains());
    }

    @Override
    public void run() {
        //Load the metric
        Connect connect = null;
        try {
            connect = new Connect(null);
            this.loadMetric(new Object[] {connect});
        } catch (LibvirtException e) {
            System.out.println("Problem loading the HostDomains metric values (Libvirt)");
            e.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("HostDomains request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        try {
            connect.close();
        } catch (LibvirtException e) {
            System.out.println("Problem to close the Libvirt connection.");
            e.printStackTrace();
        }
    }
}
```

Figura B.1 Implementação do coletor *HostDomainsCollector*.

```java
public class DomainStatusCollector extends AbstractCollector<DomainStatusMetric> {

    private List<DomainStatus> domainStatusMetrics;

    public DomainStatusCollector(int identifier, String type) {
        super(identifier, type);
        this.domainStatusMetrics = new ArrayList<DomainStatus>();
    }

    @Override
    public void loadMetric(Object[] args) throws LibvirtException {
        this.metric = DomainStatusMetric.getInstance();
        Connect connect = (Connect) args[0];
        int[] domains = connect.listDomains();
        for (int domainId : domains) {
            DomainStatus domainStatus = new DomainStatus();
            Domain domain = connect.domainLookupByID(domainId);
            domainStatus.setDomainStatusHostIdentifier(domain.getName());
            long domainPid = ShellCommand.getDomainPid(domain.getName());
            String[] results = ShellCommand.getProcessStatus(domainPid);
            domainStatus.setDomainStatusCpuPercent(Double.valueOf(results[0]));
            domainStatus.setDomainStatusMemoryPercent(Double.valueOf(results[1]));
            this.domainStatusMetrics.add(domainStatus);
        }
    }

    @Override
    public void run() {
        //Load the metric
        Connect connect = null;
        try {
            connect = new Connect(null);
            this.loadMetric(new Object[] {connect});
        } catch (LibvirtException e) {
            System.out.println("Problem loading the DomainStatus metric values (Libvirt)");
            e.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), domainStatusMetrics, 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("Domain Status request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        this.domainStatusMetrics.clear();
        try {
            connect.close();
        } catch (LibvirtException e) {
            System.out.println("Problem to close the Libvirt connection.");
            e.printStackTrace();
        }
    }
}
```

Figura B.2 Implementação do coletor *DomainStatusCollector*.

```java
public class HostInfoCollector extends AbstractCollector<HostInfoMetric> {

    public HostInfoCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws LibvirtException {
        this.metric = HostInfoMetric.getInstance();
        Connect connect = (Connect) args[0];
        NodeInfo nodeInfo = connect.nodeInfo();
        this.metric.setHostInfoName(connect.getHostName());
        this.metric.setHostInfoHypervisor(connect.getType());
        this.metric.setHostInfoCores(nodeInfo.cores);
        this.metric.setHostInfoCpus(nodeInfo.cpus);
        this.metric.setHostInfoMemory(Math.round(((nodeInfo.memory/1024)/1024)*100.0)/100.0);
        this.metric.setHostInfoMhz(nodeInfo.mhz);
        this.metric.setHostInfoModel(nodeInfo.model);
    }

    @Override
    public void run() {
        //Load the metric
        Connect connect = null;
        try {
            connect = new Connect(null);
            this.loadMetric(new Object[] {connect});
        } catch (LibvirtException e) {
            System.out.println("Problem loading the HostInfo metric values (Libvirt)");
            e.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("HostInfo request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        try {
            connect.close();
        } catch (LibvirtException e) {
            System.out.println("Problem to close the Libvirt connection.");
            e.printStackTrace();
        }
    }
}
```

Figura B.3 Implementação do coletor *HostInfoCollector*.

B.2 CAMADA VIRTUAL

```
public class CpuCollector extends AbstractCollector<CpuMetric> {

    private List<Cpu> cpuMetrics;

    public CpuCollector(int identifier, String type) {
        super(identifier, type);
        this.cpuMetrics = new ArrayList<Cpu>();
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = CpuMetric.getInstance();
        for (CpuPerc cpuPerc : sigar.getCpuPercList()) {
            Cpu cpu = new Cpu();
            cpu.setCpuUser(Double.valueOf(CpuPerc.format(cpuPerc.getUser()).replace("%", "")));
            cpu.setCpuSystem(Double.valueOf(CpuPerc.format(cpuPerc.getSys()).replace("%", "")));
            cpu.setCpuIdle(Double.valueOf(CpuPerc.format(cpuPerc.getIdle()).replace("%", "")));
            cpu.setCpuNice(Double.valueOf(CpuPerc.format(cpuPerc.getNice()).replace("%", "")));
            cpu.setCpuWait(Double.valueOf(CpuPerc.format(cpuPerc.getWait()).replace("%", "")));
            cpu.setCpuCombined(Double.valueOf(CpuPerc.format(cpuPerc.getCombined()).replace("%", "")));
            this.cpuMetrics.add(cpu);
        }
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the CPU metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), cpuMetrics, 0 , 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("CPU request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        this.cpuMetrics.clear();
        sigar.close();
    }
}
```

Figura B.4 Implementação do coletor *CpuCollector*.

```java
public class MemoryCollector extends AbstractCollector<MemoryMetric>  {

    public MemoryCollector(int identifier, String type) {
        super(identifier, type);
    }

    private static Long format(long value) {
        return new Long(value / 1024);
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = MemoryMetric.getInstance();
        Mem mem = sigar.getMem();
        Swap swap = sigar.getSwap();
        this.metric.setMemorySwapUsed(format(swap.getUsed()));
        this.metric.setMemorySwapFree(format(swap.getFree()));
        if (swap.getTotal() > 0) {
            this.metric.setMemorySwapUsedPercent(Math.round((swap.getUsed()*100)/swap.getTotal()*100.0)/100.0);
            this.metric.setMemorySwapFreePercent(Math.round((swap.getFree()*100)/swap.getTotal()*100.0)/100.0);
        }
        this.metric.setMemoryUsed(format(mem.getUsed()));
        this.metric.setMemoryFree(format(mem.getFree()));
        this.metric.setMemoryUsedPercent(Math.round(mem.getUsedPercent()*100.0)/100.0);
        this.metric.setMemoryFreePercent(Math.round(mem.getFreePercent()*100.0)/100.0);
        this.metric.setMemoryBuffersCacheUsed(format(mem.getActualUsed()));
        this.metric.setMemoryBuffersCacheFree(format(mem.getActualFree()));
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the Memory metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("Memory request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        sigar.close();
    }
}
```

Figura B.5 Implementação do coletor *MemoryCollector*.

```java
public class NetworkCollector extends AbstractCollector<NetworkMetric> {

    public NetworkCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = NetworkMetric.getInstance();
        long bytesReceived = 0;
        long bytesSent = 0;
        long packetsSent = 0;
        long packetsReceived = 0;
        String[] netInterfacesList = sigar.getNetInterfaceList();

        for (String netInterface : netInterfacesList) {
            NetInterfaceStat netInterfaceStat = sigar.getNetInterfaceStat(netInterface);
            bytesReceived = bytesReceived + netInterfaceStat.getRxBytes();
            packetsReceived = packetsReceived + netInterfaceStat.getRxPackets();
            bytesSent = bytesSent + netInterfaceStat.getTxBytes();
            packetsSent = packetsSent + netInterfaceStat.getTxPackets();
        }
        this.metric.setNetworkBytesSent(bytesSent);
        this.metric.setNetworkBytesReceived(bytesReceived);
        this.metric.setNetworkPacketsSent(packetsSent);
        this.metric.setNetworkPacketsReceived(packetsReceived);
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the Network metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;

        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("Net request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        sigar.close();
    }
}
```

Figura B.6 Implementação do coletor *NetworkCollector*.

```java
public class DiskCollector extends AbstractCollector<DiskMetric> {

    public DiskCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = DiskMetric.getInstance();
        long diskBytesRead = 0;
        long diskBytesWritten = 0;
        long diskReads = 0;
        long diskWrites = 0;
        long diskFreeBytes = 0;
        long diskUsedBytes = 0;
        long diskTotalBytes = 0;
        FileSystem[] fileSystemList = sigar.getFileSystemList();
        FileSystemUsage fileSystemUsage;

        for (FileSystem fileSystem : fileSystemList) {
            fileSystemUsage = sigar.getFileSystemUsage(fileSystem.getDirName());
            diskBytesRead = diskBytesRead + fileSystemUsage.getDiskReadBytes();
            diskBytesWritten = diskBytesWritten + fileSystemUsage.getDiskWriteBytes();
            diskReads = diskReads + fileSystemUsage.getDiskReads();
            diskWrites = diskWrites + fileSystemUsage.getDiskWrites();
            diskFreeBytes = diskFreeBytes + fileSystemUsage.getAvail();
            diskUsedBytes = diskUsedBytes + fileSystemUsage.getUsed();
            diskTotalBytes = diskTotalBytes + fileSystemUsage.getTotal();
        }
        this.metric.setDiskReads(diskReads);
        this.metric.setDiskWrites(diskWrites);
        this.metric.setDiskBytesRead(diskBytesRead);
        this.metric.setDiskBytesWritten(diskBytesWritten);
        this.metric.setDiskFree(Math.round((diskFreeBytes/1024/1024)*100.0)/100.0);
        this.metric.setDiskUsed(Math.round((diskUsedBytes/1024/1024)*100.0)/100.0);
        this.metric.setDiskTotal(Math.round((diskTotalBytes/1024/1024)*100.0)/100.0);
        this.metric.setDiskPercent(Math.round(((diskUsedBytes*100)/diskTotalBytes)*100.0)/100.0);
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the Disk metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("Disk request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        sigar.close();
    }
}
```

Figura B.7 Implementação do coletor *DiskCollector*.

```java
public class PartitionCollector extends AbstractCollector<PartitionMetric> {

    List<Partition> partitionMetrics;

    public PartitionCollector(int identifier, String type) {
        super(identifier, type);
        this.partitionMetrics = new ArrayList<Partition>();
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = PartitionMetric.getInstance();
        FileSystem[] fileSystemList = sigar.getFileSystemList();
        FileSystemUsage fileSystemUsage;
        for (FileSystem fileSystem : fileSystemList) {
            if ((fileSystem.getDirName().trim().equals("/")) || (fileSystem.getDirName().trim().equals("/home"))) {
                Partition partition = new Partition();
                partition.setPartitionDirectoryName(fileSystem.getDirName());
                partition.setPartitionDeviceName(fileSystem.getDevName());
                fileSystemUsage = sigar.getFileSystemUsage(fileSystem.getDirName());
                partition.setPartitionReads(fileSystemUsage.getDiskReads());
                partition.setPartitionWrites(fileSystemUsage.getDiskWrites());
                partition.setPartitionBytesRead(fileSystemUsage.getDiskReadBytes());
                partition.setPartitionBytesWritten(fileSystemUsage.getDiskWriteBytes());
                partition.setPartitionFreeBytes(fileSystemUsage.getAvail());
                partition.setPartitionUsedBytes(fileSystemUsage.getUsed());
                partition.setPartitionTotalBytes(fileSystemUsage.getTotal());
                partition.setPartitionPercent(fileSystemUsage.getUsePercent()*100.0);
                this.partitionMetrics.add(partition);
            }
        }
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the Disk Partitions metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), partitionMetrics, 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;
        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("Partitions request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        this.partitionMetrics.clear();
        sigar.close();
    }
}
```

Figura B.8 Implementação do coletor *PartitionCollector*.

```
public class MachineCollector extends AbstractCollector<MachineMetric>  {

    public MachineCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws SigarException {
        Sigar sigar = (Sigar) args[0];
        this.metric = MachineMetric.getInstance();
        OperatingSystem sys = OperatingSystem.getInstance();
        Mem mem = sigar.getMem();
        Swap swap = sigar.getSwap();
        CpuInfo cpuInfo = sigar.getCpuInfoList()[0];
        this.metric.setMachineOperatingSystem(sys.getDescription());
        this.metric.setMachineKernelName(sys.getName());
        this.metric.setMachineKernelVersion(sys.getVersion());
        this.metric.setMachineArchitecture(sys.getArch());
        this.metric.setMachineTotalMemory(Math.round((((mem.getTotal()/1024)/1024)/1024)*100.0)/100.0);
        this.metric.setMachineTotalSwap(Math.round((((swap.getTotal()/1024)/1024)/1024)*100.0)/100.0);
        this.metric.setMachineTotalCPUCores(cpuInfo.getTotalCores());
        this.metric.setMachineTotalCPUSockets(cpuInfo.getTotalSockets());
        this.metric.setMachineTotalCoresPerSocket(cpuInfo.getCoresPerSocket());
    }

    @Override
    public void run() {
        Sigar sigar = new Sigar();
        //Collecting metrics
        try {
            this.loadMetric(new Object[] {sigar});
        } catch (SigarException e2) {
            System.out.println("Problem loading the System metric values (Sigar)");
            e2.printStackTrace();
        }

        //Setting the parameters of the POST request
        List<NameValuePair> params = null;
        try {
            params = this.loadRequestParams(new Date(), 0, 0);
        } catch (IllegalAccessException e1) {
            e1.printStackTrace();
        } catch (IllegalArgumentException e1) {
            e1.printStackTrace();
        } catch (InvocationTargetException e1) {
            e1.printStackTrace();
        } catch (NoSuchMethodException e1) {
            e1.printStackTrace();
        } catch (SecurityException e1) {
            e1.printStackTrace();
        }

        HttpResponse response;

        try {
            response = this.sendMetric(params);
            System.out.println(response.getStatusLine());
            if (response.getStatusLine().getStatusCode() != 202) {
                System.out.println("System request error!");
                EntityUtils.consume(response.getEntity());
            }
            EntityUtils.consume(response.getEntity());
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        //Release any native resources associated with this sigar instance
        sigar.close();
    }
}
```

Figura B.9 Implementação do coletor *MachineCollector*.

B.3 CAMADA DE DADOS

```java
public class ActiveConnectionCollector extends AbstractCollector<ActiveConnectionMetric> {

    public ActiveConnectionCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws ClassNotFoundException, SQLException {
        this.metric = ActiveConnectionMetric.getInstance();
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Connection connection = null;
        ResultSet resultSet = null;
        Object[] params;
        switch (String.valueOf(args[0])) {
        case "dbms":
            params = databaseConnection.getConnection(Integer.valueOf((String) args[1]), null);
            connection = (Connection) params[1];
            //Checking database type to make the SQL
            if (params[0].equals("MySQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select count(*) as connections from information_schema.processlist;", null);
            } else if (params[0].equals("PostgreSQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select count(*) as connections from pg_stat_activity;", null);
            }
            while (resultSet != null && resultSet.next()) {
                this.metric.setActiveConnectionAmount(resultSet.getInt("connections"));
            }
            resultSet.close();
            connection.close();
            break;
        case "database":
            params = databaseConnection.getConnection(null, Integer.valueOf((String) args[1]));
            connection = (Connection) params[1];
            //Checking database type to make the SQL
            if (params[0].equals("MySQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select count(*) as connections from information_schema.processlist where db = database();", null);
            } else if (params[0].equals("PostgreSQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select count(*) as connections from pg_stat_activity where datname = '"+params[2]+"';", null);
            }
            while (resultSet != null && resultSet.next()) {
                this.metric.setActiveConnectionAmount(resultSet.getInt("connections"));
            }
            resultSet.close();
            connection.close();
            break;
        }
    }

    @Override
    public void run() {
        this.metric = ActiveConnectionMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;
        Object[] args = new Object[2];
        //Checking the DBMSs enabled for collection
        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                args[0] = "dbms";
                args[1] = dbms;
                try {
                    this.loadMetric(args);
                } catch (ClassNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the ActiveConnections metric value (DBMS)");
                    e.printStackTrace();
                }

                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("Active connections request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //Checking the Databases enabled for collection
        if (this.metric.getDatabases().length > 0) {
            for (String database : this.metric.getDatabases()) {
                args[0] = "database";
                args[1] = database;
                try {
                    this.loadMetric(args);
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the ActiveConnections metric value (Database)");
                    e.printStackTrace();
                }

                try {
                    params = this.loadRequestParams(new Date(), 0, Integer.parseInt(database));
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("Active connections request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
```

Figura B.10 Implementação do coletor *ActiveConnectionCollector*.

```java
public class SizeCollector extends AbstractCollector<SizeMetric> {

    public SizeCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws ClassNotFoundException, SQLException {
        this.metric = SizeMetric.getInstance();
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Connection connection = null;
        ResultSet resultSet = null;
        Object[] params;

        switch (String.valueOf(args[0])) {
        case "dbms":
            params = databaseConnection.getConnection(Integer.valueOf((String) args[1]), null);
            connection = (Connection) params[1];
            //Checking database type to make the SQL
            if (params[0].equals("MySQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select sum(data_length + index_length)/1024/1024 as size from information_schema.tables;", null);
            } else if (params[0].equals("PostgreSQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select (sum(pg_database_size(pg_database.datname))/1024/1024) as size from pg_database;", null);
            }
            while (resultSet != null && resultSet.next()) {
                this.metric.setSizeUsed(Math.round(resultSet.getDouble("size")*100.0)/100.0);
            }
            resultSet.close();
            connection.close();
            break;
        case "database":
            params = databaseConnection.getConnection(null, Integer.valueOf((String) args[1]));
            connection = (Connection) params[1];
            //Checking database type to make the SQL
            if (params[0].equals("MySQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select sum(data_length + index_length)/1024/1024 as size from information_schema.tables where table_schema = database();", null);
            } else if (params[0].equals("PostgreSQL")) {
                resultSet = databaseConnection.executeSQL(connection, "select (pg_database_size('"+params[2]+"')/1024/1024) as size;", null);
            }
            while (resultSet != null && resultSet.next()) {
                this.metric.setSizeUsed(Math.round(resultSet.getDouble("size")*100.0)/100.0);
            }
            resultSet.close();
            connection.close();
            break;
        }
    }

    @Override
    public void run() {
        this.metric = SizeMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;
        Object[] args = new Object[2];
        //Checking the DBMSs enabled for collection
        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                args[0] = "dbms";
                args[1] = dbms;
                try {
                    this.loadMetric(args);
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the ActiveConnections metric value (DBMS)");
                    e.printStackTrace();
                }

                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("Active connections request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //Checking the Databases enabled for collection
        if (this.metric.getDatabases().length > 0) {
            for (String database : this.metric.getDatabases()) {
                args[0] = "database";
                args[1] = database;
                try {
                    this.loadMetric(args);
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the ActiveConnections metric value (DBMS)");
                    e.printStackTrace();
                }

                try {
                    params = this.loadRequestParams(new Date(), 0, Integer.parseInt(database));
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("Active connections request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Figura B.11 Implementação do coletor *SizeCollector*.

```java
public class NetworkTrafficCollector extends AbstractCollector<NetworkTrafficMetric> {

    private boolean firstCycle;
    private int bytesReceived;
    private int bytesSent;

    public NetworkTrafficCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
        this.bytesReceived = 0;
        this.bytesSent = 0;
    }

    @Override
    public void loadMetric(Object[] args) throws NumberFormatException, ClassNotFoundException, SQLException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            resultSet = databaseConnection.executeSQL(connection, "show global status where variable_name in ('Bytes_received', 'Bytes_sent');", null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Bytes_received":
                    bytesReceived = resultSet.getInt("Value");
                    break;
                case "Bytes_sent":
                    bytesSent = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setNetworkTrafficBytesReceived(0);
            this.metric.setNetworkTrafficBytesSent(0);
        //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
        } else {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Bytes_received":
                    this.metric.setNetworkTrafficBytesReceived(resultSet.getInt("Value") - bytesReceived);
                    bytesReceived = resultSet.getInt("Value");
                    break;
                case "Bytes_sent":
                    this.metric.setNetworkTrafficBytesSent(resultSet.getInt("Value") - bytesSent);
                    bytesSent = resultSet.getInt("Value");
                    break;
                }
            }
        }
        //Close the connection and resultset
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = NetworkTrafficMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the NetworkTraffic metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("NetworkTraffic request error!");
                        EntityUtils.consume(response.getEntity());
                    } catch (ClientProtocolException e) {
                        e.printStackTrace();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
            //After the first cycle the flag is changed to false
            this.firstCycle = false;
        }
    }
```

Figura B.12 Implementação do coletor *NetworkTrafficCollector*.

```
public class ProcessStatusCollector extends AbstractCollector<ProcessStatusMetric> {

    public ProcessStatusCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws Exception {
        this.metric = ProcessStatusMetric.getInstance();
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        for (DBMS dbms : databaseConnection.getDBMSs()) {
            if (dbms.getId() == args[0]) {
                switch (dbms.getType()) {
                case "MySQL":
                    String[] mysql = ShellCommand.getProcessStatus(Long.parseLong(ShellCommand.getPidsFromName("mysqld")[0]));
                    this.metric.setProcessStatusCpu(Double.valueOf(mysql[0]));
                    this.metric.setProcessStatusMemory(Double.valueOf(mysql[1]));
                    break;
                case "PostgreSQL":
                    this.metric.setProcessStatusCpu(ShellCommand.getPostgreSQLCpuPercentage());
                    this.metric.setProcessStatusMemory(ShellCommand.getPostgreSQLMemPercentage());
                    break;
                }
            }
        }
    }

    @Override
    public void run() {
        this.metric = ProcessStatusMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        //Checking the DBMSs enabled for collection
        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                try {
                    this.loadMetric(new Object[] {Integer.parseInt(dbms)});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (Exception e) {
                    System.out.println("Problem loading the ProcessStatus metric value (DBMS)");
                    e.printStackTrace();
                }

                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("Process status request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Figura B.13 Implementação do coletor *ProcessStatusCollector*.

```java
public class InformationDataCollector extends AbstractCollector<InformationDataMetric> {

    public InformationDataCollector(int identifier, String type) {
        super(identifier, type);
    }

    @Override
    public void loadMetric(Object[] args) throws NumberFormatException, ClassNotFoundException, SQLException  {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            String sql = "select (select count(*) from information_schema.schemata where schema_name not in ('mysql','information_schema','performance_schema')) as amount_db, " +
                         "(select count(*) from information_schema.tables where table_schema not in ('mysql','information_schema','performance_schema')) as amount_tables, " +
                         "(select count(*) from information_schema.statistics where table_schema not in ('mysql','information_schema','performance_schema')) as amount_index, " +
                         "(select count(*) from information_schema.triggers where trigger_schema not in ('mysql','information_schema','performance_schema')) as amount_trigger, " +
                         "(select count(*) from information_schema.views where table_schema not in ('mysql','information_schema','performance_schema')) as amount_views, " +
                         "(select count(*) from information_schema.routines where routine_schema not in ('mysql','information_schema','performance_schema')) as amount_routines " +
                         "from dual;";
            resultSet = databaseConnection.executeSQL(connection, sql, null);
        }
        //Dealing with the return of the query and inserting the data in the object of the metric
        while (resultSet != null && resultSet.next()) {
            this.metric.setInformationDataTables(resultSet.getInt("amount_tables"));
            this.metric.setInformationDataDatabases(resultSet.getInt("amount_db"));
            this.metric.setInformationDataIndexs(resultSet.getInt("amount_index"));
            this.metric.setInformationDataTriggers(resultSet.getInt("amount_trigger"));
            this.metric.setInformationDataViews(resultSet.getInt("amount_views"));
            this.metric.setInformationDataRoutines(resultSet.getInt("amount_routines"));
        }
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = InformationDataMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the InformationData metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("InformationData request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Figura B.14 Implementação do coletor *InformationDataCollector*.

```java
public class StatementDMLCollector extends AbstractCollector<StatementDMLMetric> {

    private boolean firstCycle;
    private int inserts;
    private int updates;
    private int selects;
    private int deletes;

    public StatementDMLCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
        this.inserts = 0;
        this.updates = 0;
        this.selects = 0;
        this.deletes = 0;
    }

    @Override
    public void loadMetric(Object[] args) throws NumberFormatException, ClassNotFoundException, SQLException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            resultSet = databaseConnection.executeSQL(connection, "show global status where variable_name in ('Com_insert', 'Com_select', 'Com_update', 'Com_delete');", null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_insert":
                    inserts = resultSet.getInt("Value");
                    break;
                case "Com_select":
                    selects = resultSet.getInt("Value");
                    break;
                case "Com_update":
                    updates = resultSet.getInt("Value");
                    break;
                case "Com_delete":
                    deletes = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setStatementDMLInserts(0);
            this.metric.setStatementDMLUpdates(0);
            this.metric.setStatementDMLSelects(0);
            this.metric.setStatementDMLDeletes(0);
            //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
            } else {
                while (resultSet != null && resultSet.next()) {
                    switch (resultSet.getString("Variable_name")) {
                    case "Com_insert":
                        this.metric.setStatementDMLInserts(resultSet.getInt("Value") - inserts);
                        inserts = resultSet.getInt("Value");
                        break;
                    case "Com_select":
                        this.metric.setStatementDMLSelects(resultSet.getInt("Value") - selects);
                        selects = resultSet.getInt("Value");
                        break;
                    case "Com_update":
                        this.metric.setStatementDMLUpdates(resultSet.getInt("Value") - updates);
                        updates = resultSet.getInt("Value");
                        break;
                    case "Com_delete":
                        this.metric.setStatementDMLDeletes(resultSet.getInt("Value") - deletes);
                        deletes = resultSet.getInt("Value");
                        break;
                    }
                }
            }
            //Close the connection and resultset
            resultSet.close();
            connection.close();
        }

        @Override
        public void run() {
            this.metric = StatementDMLMetric.getInstance();
            HttpResponse response;
            List<NameValuePair> params = null;

            if (this.metric.getDBMSs().length > 0) {
                for (String dbms : this.metric.getDBMSs()) {
                    //Load the metric
                    try {
                        this.loadMetric(new Object[] {dbms});
                    } catch (NumberFormatException e) {
                        e.printStackTrace();
                    } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                    } catch (SQLException e) {
                        System.out.println("Problem loading the StatementDML metric value (DBMS)");
                        e.printStackTrace();
                    }

                    //Creates request parameters
                    try {
                        params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                    } catch (NumberFormatException e) {
                        e.printStackTrace();
                    } catch (IllegalAccessException e) {
                        e.printStackTrace();
                    } catch (IllegalArgumentException e) {
                        e.printStackTrace();
                    } catch (InvocationTargetException e) {
                        e.printStackTrace();
                    } catch (NoSuchMethodException e) {
                        e.printStackTrace();
                    } catch (SecurityException e) {
                        e.printStackTrace();
                    }

                    //Sends the collected metric
                    try {
                        response = this.sendMetric(params);
                        System.out.println(response.getStatusLine());
                        if (response.getStatusLine().getStatusCode() != 202) {
                            System.out.println("StatementDML request error!");
                            EntityUtils.consume(response.getEntity());
                        }
                        EntityUtils.consume(response.getEntity());
                    } catch (ClientProtocolException e) {
                        e.printStackTrace();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
            //After the first cycle the flag is changed to false
            this.firstCycle = false;
        }
    }
```

Figura B.15 Implementação do coletor *StatementDMLCollector*.

```java
public class StatementTCLCollector extends AbstractCollector<StatementTCLMetric> {

    private boolean firstCycle;
    private int commits;
    private int rollbacks;
    private int savepoints;

    public StatementTCLCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
        this.commits = 0;
        this.rollbacks = 0;
        this.savepoints = 0;
    }

    @Override
    public void loadMetric(Object[] args) throws SQLException, NumberFormatException, ClassNotFoundException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            resultSet = databaseConnection.executeSQL(connection, "show global status where variable_name in ('Com_commit', 'Com_rollback', 'Com_savepoint');", null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_commit":
                    commits = resultSet.getInt("Value");
                    break;
                case "Com_rollback":
                    rollbacks = resultSet.getInt("Value");
                    break;
                case "Com_savepoint":
                    savepoints = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setStatementTCLCommits(0);
            this.metric.setStatementTCLRollback(0);
            this.metric.setStatementTCLSavepoint(0);
        //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
        } else {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_commit":
                    this.metric.setStatementTCLCommits(resultSet.getInt("Value") - commits);
                    commits = resultSet.getInt("Value");
                    break;
                case "Com_rollback":
                    this.metric.setStatementTCLRollback(resultSet.getInt("Value") - rollbacks);
                    rollbacks = resultSet.getInt("Value");
                    break;
                case "Com_savepoint":
                    this.metric.setStatementTCLRollback(resultSet.getInt("Value") - savepoints);
                    rollbacks = resultSet.getInt("Value");
                    break;
                }
            }
        }
        //Close the connection and resultset
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = StatementTCLMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the StatementTCL metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("StatementTCL request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //After the first cycle the flag is changed to false
        this.firstCycle = false;
    }
}
```

Figura B.16 Implementação do coletor *StatementTCLCollector*.

```java
public class StatementDDLCollector extends AbstractCollector<StatementDDLMetric> {

    private boolean firstCycle;
    private int creates;
    private int alters;
    private int drops;
    private int truncates;
    private int renames;

    public StatementDDLCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
        this.creates = 0;
        this.alters = 0;
        this.drops = 0;
        this.truncates = 0;
        this.renames = 0;
    }

    @Override
    public void loadMetric(Object[] args) throws NumberFormatException, ClassNotFoundException, SQLException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            resultSet = databaseConnection.executeSQL(connection, "show global status where variable_name in ('Com_create_table', 'Com_alter_table', 'Com_drop_table', 'Com_truncate', 'Com_rename_table');", null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_create_table":
                    creates = resultSet.getInt("Value");
                    break;
                case "Com_alter_table":
                    alters = resultSet.getInt("Value");
                    break;
                case "Com_drop_table":
                    drops = resultSet.getInt("Value");
                    break;
                case "Com_truncate":
                    truncates = resultSet.getInt("Value");
                    break;
                case "Com_rename_table":
                    renames = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setStatementDDLCreate(0);
            this.metric.setStatementDDLAlter(0);
            this.metric.setStatementDDLDrop(0);
            this.metric.setStatementDDLTruncate(0);
            this.metric.setStatementDDLRename(0);
        //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
        } else {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_create_table":
                    this.metric.setStatementDDLCreate(resultSet.getInt("Value") - creates);
                    creates = resultSet.getInt("Value");
                    break;
                case "Com_alter_table":
                    this.metric.setStatementDDLAlter(resultSet.getInt("Value") - alters);
                    alters = resultSet.getInt("Value");
                    break;
                case "Com_drop_table":
                    this.metric.setStatementDDLDrop(resultSet.getInt("Value") - drops);
                    drops = resultSet.getInt("Value");
                    break;
                case "Com_truncate":
                    this.metric.setStatementDDLTruncate(resultSet.getInt("Value") - truncates);
                    truncates = resultSet.getInt("Value");
                    break;
                case "Com_rename_table":
                    this.metric.setStatementDDLRename(resultSet.getInt("Value") - renames);
                    renames = resultSet.getInt("Value");
                    break;
                }
            }
        }
        //Close the connection and resultset
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = StatementDDLMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the StatementDDL metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("StatementDDL request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //After the first cycle the flag is changed to false
        this.firstCycle = false;
    }
}
```

Figura B.17 Implementação do coletor *StatementDDLCollector*.

```java
public class StatementDCLCollector extends AbstractCollector<StatementDCLMetric> {

    private boolean firstCycle;
    private int grants;
    private int revokes;

    public StatementDCLCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
        this.grants = 0;
        this.revokes = 0;
    }

    @Override
    public void loadMetric(Object[] args) throws SQLException, NumberFormatException, ClassNotFoundException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            resultSet = databaseConnection.executeSQL(connection, "show global status where variable_name in ('Com_grant', 'Com_revoke');", null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_grant":
                    grants = resultSet.getInt("Value");
                    break;
                case "Com_revoke":
                    revokes = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setStatementDCLGrant(0);
            this.metric.setStatementDCLRevoke(0);
        //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
        } else {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Com_commit":
                    this.metric.setStatementDCLGrant(resultSet.getInt("Value") - grants);
                    grants = resultSet.getInt("Value");
                    break;
                case "Com_rollback":
                    this.metric.setStatementDCLRevoke(resultSet.getInt("Value") - revokes);
                    revokes = resultSet.getInt("Value");
                    break;
                }
            }
        }
        //Close the connection and resultset
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = StatementDCLMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the StatementDCL metric value (DBMS)");
                    e.printStackTrace();
                }

                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("StatementDCL request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //After the first cycle the flag is changed to false
        this.firstCycle = false;
    }
}
```

Figura B.18 Implementação do coletor *StatementDCLCollector*.

```java
public class DiskUtilizationCollector extends AbstractCollector<DiskUtilizationMetric> {

    private boolean firstCycle;
    private int physicalReads;
    private int logicalReads;
    private int pendingReads;
    private int pendingWrites;
    private int dataRead;
    private int dataWritten;
    private int pagesRead;
    private int pagesWritten;
    private int keyRead;
    private int keyWrites;

    public DiskUtilizationCollector(int identifier, String type) {
        super(identifier, type);
        this.firstCycle = true;
    }

    @Override
    public void loadMetric(Object[] args) throws SQLException, NumberFormatException, ClassNotFoundException {
        DatabaseConnection databaseConnection = DatabaseConnection.getInstance();
        Object[] params = databaseConnection.getConnection(Integer.valueOf((String) args[0]), null);
        Connection connection = (Connection) params[1];
        ResultSet resultSet = null;

        //Checking DBMS type to make and execute the SQL
        if (params[0].equals("MySQL")) {
            String sql = "show global status where variable_name in ('Innodb_buffer_pool_reads', 'Innodb_buffer_pool_read_requests', " +
                         "'Innodb_data_pending_reads', 'Innodb_data_pending_writes', 'Innodb_data_read', 'Innodb_data_written', 'Innodb_pages_read', " +
                         "'Innodb_pages_written', 'Key_reads', 'Key_writes');";
            resultSet = databaseConnection.executeSQL(connection, sql, null);
        }

        //Dealing with the return of the query and inserting the data in the object of the metric
        //If first cycle: the metric has zero values and the monitored value is stored
        if (firstCycle == true) {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Innodb_buffer_pool_reads":
                    physicalReads = resultSet.getInt("Value");
                    break;
                case "Innodb_buffer_pool_read_requests":
                    logicalReads = resultSet.getInt("Value");
                    break;
                case "Innodb_data_pending_reads":
                    pendingReads = resultSet.getInt("Value");
                    break;
                case "Innodb_data_pending_writes":
                    pendingWrites = resultSet.getInt("Value");
                    break;
                case "Innodb_data_read":
                    dataRead = resultSet.getInt("Value");
                    break;
                case "Innodb_data_written":
                    dataWritten = resultSet.getInt("Value");
                    break;
                case "Innodb_pages_read":
                    pagesRead = resultSet.getInt("Value");
                    break;
                case "Innodb_pages_written":
                    pagesWritten = resultSet.getInt("Value");
                    break;
                case "Key_reads":
                    keyRead = resultSet.getInt("Value");
                    break;
                case "Key_writes":
                    keyWrites = resultSet.getInt("Value");
                    break;
                }
            }
            this.metric.setDiskUtilizationPhysicalReads(0);
            this.metric.setDiskUtilizationLogicalReads(0);
            this.metric.setDiskUtilizationPendingReads(0);
            this.metric.setDiskUtilizationPendingWrites(0);
            this.metric.setDiskUtilizationDataRead(0);
            this.metric.setDiskUtilizationDataWritten(0);
            this.metric.setDiskUtilizationPagesRead(0);
            this.metric.setDiskUtilizationPagesWritten(0);
            this.metric.setDiskUtilizationKeyRead(0);
            this.metric.setDiskUtilizationKeyWrites(0);
        //Otherwise: the new value is subtracted by the value stored and the result is set in the metric
        } else {
            while (resultSet != null && resultSet.next()) {
                switch (resultSet.getString("Variable_name")) {
                case "Innodb_buffer_pool_reads":
                    this.metric.setDiskUtilizationPhysicalReads(resultSet.getInt("Value") - physicalReads);
                    physicalReads = resultSet.getInt("Value");
                    break;
                case "Innodb_buffer_pool_read_requests":
                    this.metric.setDiskUtilizationLogicalReads(resultSet.getInt("Value") - logicalReads);
                    logicalReads = resultSet.getInt("Value");
                    break;
                case "Innodb_data_pending_reads":
                    this.metric.setDiskUtilizationPendingReads(resultSet.getInt("Value") - pendingReads);
                    pendingReads = resultSet.getInt("Value");
                    break;
                case "Innodb_data_pending_writes":
                    this.metric.setDiskUtilizationPendingWrites(resultSet.getInt("Value") - pendingWrites);
                    pendingWrites = resultSet.getInt("Value");
                    break;
                case "Innodb_data_read":
                    this.metric.setDiskUtilizationDataRead(resultSet.getInt("Value") - dataRead);
                    dataRead = resultSet.getInt("Value");
                    break;
                case "Innodb_data_written":
                    this.metric.setDiskUtilizationDataWritten(resultSet.getInt("Value") - dataWritten);
                    dataWritten = resultSet.getInt("Value");
                    break;
                case "Innodb_pages_read":
                    this.metric.setDiskUtilizationPagesRead(resultSet.getInt("Value") - pagesRead);
                    pagesRead = resultSet.getInt("Value");
                    break;
                case "Innodb_pages_written":
                    this.metric.setDiskUtilizationPagesWritten(resultSet.getInt("Value") - pagesWritten);
                    pagesWritten = resultSet.getInt("Value");
                    break;
                case "Key_reads":
                    this.metric.setDiskUtilizationKeyRead(resultSet.getInt("Value") - keyRead);
                    keyRead = resultSet.getInt("Value");
                    break;
                case "Key_writes":
                    this.metric.setDiskUtilizationKeyWrites(resultSet.getInt("Value") - keyWrites);
                    keyWrites = resultSet.getInt("Value");
                    break;
                }
            }
        }
        //Close the connection and resultset
        resultSet.close();
        connection.close();
    }

    @Override
    public void run() {
        this.metric = DiskUtilizationMetric.getInstance();
        HttpResponse response;
        List<NameValuePair> params = null;

        if (this.metric.getDBMSs().length > 0) {
            for (String dbms : this.metric.getDBMSs()) {
                //Load the metric
                try {
                    this.loadMetric(new Object[] {dbms});
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                } catch (SQLException e) {
                    System.out.println("Problem loading the DiskUtilization metric value (DBMS)");
                    e.printStackTrace();
                }
                //Creates request parameters
                try {
                    params = this.loadRequestParams(new Date(), Integer.parseInt(dbms), 0);
                } catch (NumberFormatException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (InvocationTargetException e) {
                    e.printStackTrace();
                } catch (NoSuchMethodException e) {
                    e.printStackTrace();
                } catch (SecurityException e) {
                    e.printStackTrace();
                }

                //Sends the collected metric
                try {
                    response = this.sendMetric(params);
                    System.out.println(response.getStatusLine());
                    if (response.getStatusLine().getStatusCode() != 202) {
                        System.out.println("DiskUtilization request error!");
                        EntityUtils.consume(response.getEntity());
                    }
                    EntityUtils.consume(response.getEntity());
                } catch (ClientProtocolException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //After the first cycle the flag is changed to false
        this.firstCycle = false;
    }
}
```

Figura B.19 Implementação do coletor *DiskUtilizationCollector*.

B.4 CAMADA DE CARGA DE TRABALHO

```
public class WorkloadStatusCollector extends AbstractWorkloadCollector<WorkloadStatus> {

    public WorkloadStatusCollector(MyDriverClient client, Properties properties) {
        super(client, properties);
    }

    @Override
    public Object[] executeQuery(Database database, String workload) throws ClassNotFoundException, SQLException, Exception {
        WorkloadStatus workloadStatus = new WorkloadStatus();
        //Creates the connection to the instance database
        Connection connection = this.myDBaaSConnection.getConnection(database);
        Statement statement = connection.createStatement();
        ResultSet resultSet = null;
        long startTime = System.currentTimeMillis();
        resultSet = statement.executeQuery(workload);
        resultSet.last();
        //Collects the selectivity of the workload - ResultSet size
        long selectivity = resultSet.getRow();
        resultSet.first();
        //Calculate the total response time
        double estimatedTime = System.currentTimeMillis() - startTime;
        workloadStatus.setWorkloadStatusQuery(workload);
        workloadStatus.setWorkloadStatusSelectivity(selectivity);
        //Collects the throughput of workload
        workloadStatus.setWorkloadStatusThroughput(Math.round(selectivity/estimatedTime*100.0)/100.0);
        workloadStatus.setWorkloadStatusResponseTime(estimatedTime);

        List<NameValuePair> params = this.loadRequestParams(workloadStatus, new Date(), database);
        HttpResponse response = this.sendMetric(params);
        System.out.println(response.getStatusLine());
        if (response.getStatusLine().getStatusCode() != 202) {
            System.out.println("WorkloadStatus request error!");
            EntityUtils.consume(response.getEntity());
        }
        EntityUtils.consume(response.getEntity());
        return new Object[]{connection, statement, resultSet, workloadStatus};
    }

    @Override
    public Object[] executeUpdate(Database database, String workload) throws ClassNotFoundException, SQLException, Exception {
        WorkloadStatus workloadStatus = new WorkloadStatus();
        //Creates the connection to the instance database
        Connection connection = this.myDBaaSConnection.getConnection(database);
        Statement statement = connection.createStatement();
        long startTime = System.currentTimeMillis();
        //Collects the selectivity of the workload
        long selectivity = statement.executeUpdate(workload);
        //Calculate the total response time
        double estimatedTime = System.currentTimeMillis() - startTime;
        workloadStatus.setWorkloadStatusQuery(workload);
        workloadStatus.setWorkloadStatusSelectivity(selectivity);
        workloadStatus.setWorkloadStatusResponseTime(estimatedTime);
        if (selectivity > 0) {
            //Collects the throughput of workload
            workloadStatus.setWorkloadStatusThroughput(Math.round(selectivity/estimatedTime*100.0)/100.0);
        }

        List<NameValuePair> params = this.loadRequestParams(workloadStatus, new Date(), database);
        HttpResponse response = this.sendMetric(params);
        System.out.println(response.getStatusLine());
        if (response.getStatusLine().getStatusCode() != 202) {
            System.out.println("WorkloadStatus request error!");
            EntityUtils.consume(response.getEntity());
        }
        EntityUtils.consume(response.getEntity());
        return new Object[]{connection, statement, workloadStatus};
    }
}
```

Figura B.20 Implementação do coletor *WorkloadStatusCollector*.

APÊNDICE C

CLASSES DE DEFINIÇÃO DOS RECEIVERS

C.1 CAMADA VIRTUAL

```
@Resource
@Path("/machine")
public class MachineReceiverController extends AbstractReceiver<MetricRepository> {

    private VirtualMachineRepository machineRepository;

    public MachineReceiverController(DefaultStatus status, MetricRepository repository,
            VirtualMachineRepository machineRepository) {
        super(status, repository);
        this.machineRepository = machineRepository;
    }

    @Post("/info")
    public void information(Machine metric, int machine) {
        if (machineRepository.updateSystemInformation(metric, machine)) {
            status.accepted();
        }
    }

    @Post("/cpu")
    public void cpu(List<Cpu> metric, int machine, String recordDate) {
        for (Cpu cpu : metric) {
            try {
                if (repository.saveMetric(cpu, recordDate, machine, 0, 0, 0)) {
                    status.accepted();
                }
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }

    @Post("/memory")
    public void memory(Memory metric, int machine, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, machine, 0, 0, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
```

Figura C.1 Implementação do *receiver* para métricas de máquina virtual (Parte 1).

```
    @Post("/network")
    public void network(Network metric, int machine, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, machine, 0, 0, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    @Post("/disk")
    public void disk(Disk metric, int machine, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, machine, 0, 0, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    @Post("/partition")
    public void partition(List<Partition> metric, int machine, String recordDate) {
        for (Partition partition : metric) {
            try {
                if (repository.saveMetric(partition, recordDate, machine, 0, 0, 0)) {
                    status.accepted();
                }
            } catch (NoSuchMethodException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figura C.2 Implementação do *receiver* para métricas de máquina virtual (Parte 2).

C.2 CAMADA DE DADOS

```java
@Resource
@Path("/storage")
public class StorageReceiverController extends AbstractReceiver<MetricRepository> {

    public StorageReceiverController(DefaultStatus status, MetricRepository repository) {
        super(status, repository);
    }

    @Post("/activeconnections")
    public void activeConnections(ActiveConnection metric, int dbms, int database, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, 0, dbms, database)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    @Post("/size")
    public void size(Size metric, int dbms, int database, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, 0, dbms, database)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    @Post("/networktraffic")
    public void networkTraffic(NetworkTraffic metric, int dbms, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    @Post("/processstatus")
    public void processStatus(ProcessStatus metric, int dbms, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
```

Figura C.3 Implementação do *receiver* para métricas de SGBD/instância de banco de dados (Parte 1).

```java
@Post("/informationdata")
public void informationData(InformationData metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

@Post("/informationtable")
public void informationTable(List<InformationTable> metric, int database, String recordDate) {
    for (InformationTable informationTable : metric) {
        try {
            if (repository.saveMetric(informationTable, recordDate, 0, 0, 0, database)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}

@Post("/statementdml")
public void statementDML(StatementDML metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

@Post("/statementtcl")
public void statementTCL(StatementTCL metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

@Post("/statementddl")
public void statementDDL(StatementDDL metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

@Post("/statementdcl")
public void statementDCL(StatementDCL metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}

@Post("/diskutilization")
public void diskUtilization(DiskUtilization metric, int dbms, String recordDate) {
    try {
        if (repository.saveMetric(metric, recordDate, 0, 0, dbms, 0)) {
            status.accepted();
        }
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}
}
```

Figura C.4 Implementação do *receiver* para métricas de SGBD/instância de banco de dados (Parte 2).

C.3 CAMADA DE CARGA DE TRABALHO

```
@Resource
@Path("/workload")
public class WorkloadReceiverController extends AbstractReceiver<MetricRepository>{

    public WorkloadReceiverController(DefaultStatus status, MetricRepository repository) {
        super(status, repository);
    }

    @Post("/workloadstatus")
    public void workloadStatus(WorkloadStatus metric, int database, String recordDate) {
        try {
            if (repository.saveMetric(metric, recordDate, 0, 0, 0, database)) {
                status.accepted();
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
}
```

Figura C.5 Implementação do *receiver* para métricas de carga de trabalho.