UNIVERSIDADE FEDERAL DO CEARÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

AURELIANO NETO DE OLIVEIRA DA SILVA

UM FRAMEWORK DE ADAPTAÇÃO BASEADA NO PADRÃO MAPEK

RUSSAS

2021

AURELIANO NETO DE OLIVEIRA DA SILVA

UM FRAMEWORK DE ADAPTAÇÃO BASEADA NO PADRÃO MAPEK

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientador: Prof. Ms. Filipe Marciel Roberto

RUSSAS

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S578f Silva, Aureliano Neto de Oliveira.
Um framework de adaptação baseada no padrão mapek / Aureliano Neto de Oliveira Silva. – 2021.
54 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Engenharia de Software, Russas, 2021.
Orientação: Prof. Me. Filipe Marciel Roberto.

1. Sistemas autônomos. 2. Auto adaptação. 3. Framework extensível. I. Título.

CDD 005.1

AURELIANO NETO DE OLIVEIRA DA SILVA

UM FRAMEWORK DE ADAPTAÇÃO BASEADA NO PADRÃO MAPEK

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: 26 de Março de 2021

BANCA EXAMINADORA

Prof. Ms. Filipe Marciel Roberto (Orientador)
Universidade Federal do Ceará (UFC)

Prof, Ms. José Osvaldo Mesquita Chaves
Universidade Federal do Ceará - UFC

Prof. Dr. Alexandre Matos Arruda
Universidade Federal do Ceará - UFC

RESUMO

Atualmente, e mais fortemente no futuro, vários sistemas basear-se-ão em dados para controlar algo, que inclusive pode ser o autocontrole. No primeiro caso estamos falando sobre sistema de controle adaptáveis, como feedbacks de controle. No último, estamos falando de sistemas autoadaptativos, e sobre eles há uma grande demanda de pesquisa e ferramentas para viabilizá-los ao grande público. Para que se possa chegar a um nível de autoadaptação é preciso ter autonomia de tomar decisões que modifiquem o comportamento durante a execução. A base dessa autonomia é realizar 4 atividades: monitoramento, análise, planejamento e execuação. O padrão MAPE-K proposto pela IBM, quando da proposição dos sistemas autônomos, é o que melhor permite representar esse ciclo de atividades. Neste trabalho é proposta uma implementação desse padrão que pode ser usada por sistemas programados em Python, para a inclusão de um comportamento autônomo em sua execuação. Uma simulação foi realizada com a manipulação de um datacenter que se adapta autonomamente à variação de requisições de usuários. Foi atestado, que o datacenter simulado teve seu comportamento alterado pelas decisões de mudança de execução definidas pela instanciação da proposta.

Palavras-chave: Sistemas autônomos. Auto adaptação. Framework extensível.

ABSTRACT

Currently, and more strongly in the future, several systems will rely on data to control something, which may even be self-control. In the first case, we are talking about adaptive control systems, such as control feedbacks. In the latter, we are talking about self-adaptive systems, and there is a great demand for research and tools to make them available to the general public. In order to reach a level of self-adaptation, it is necessary to have autonomy to make decisions that modify behavior during execution. The basis of this autonomy is to carry out 4 activities: monitoring, analysis, planning and execution. The MAPE-K standard proposed by IBM, when proposing autonomous systems, is the one that best allows representing this cycle of activities. This work proposes an implementation of this standard that can be used by systems programmed in Python, for the inclusion of an autonomous behavior in its execution. A simulation was performed with the manipulation of a datacenter that adapts autonomously to the variation of user requests. It was attested that the simulated data center had its behavior altered by the execution change decisions defined by the proposal instantiation.

Keywords: Autonomous systems. Self adaptation. Extensible framework.

LISTA DE FIGURAS

Figura 1 – Camadas da computação em nuvem e os serviços que podem ser ofertados. . 15
Figura 2 – Exemplo arquitetura microsserviços. 17
Figura 3 – Diferença entre virtualização do sistema e utilização de contêiner. 18
Figura 4 – Arquitetura do Kubernetes. 19
Figura 5 – Arquitetura do sistema de monitoramento Kubow. 22
Figura 6 – Ciclo de funcionamento do MAPE-K. 23
Figura 7 – Arquitetura do MORPH. 24
Figura 8 – Diagrama de classes do componente monitor. 26
Figura 9 – Diagrama de sequência ao disparar erro na leitura do sensor. 27
Figura 10 – Diagrama de fluxo normal de fase de monitoramento. 28
Figura 11 – Diagrama de classe do analisador 28
Figura 12 – Diagrama de sequência do analisador 29
Figura 13 – Diagrama de classe do Planejador. 30
Figura 14 – Diagrama de sequência do Planejador. 31
Figura 15 – Diagrama de fluxo normal de fase de monitoramento. 32
Figura 16 – Diagrama de sequência do componente Executor. 32
Figura 17 – Diagrama de classes do Knowledge. 33
Figura 18 – Diagrama de fluxo normal de fase de monitoramento. 35
Figura 19 – Diagrama MAPE-K e SWIM. 36
Figura 20 – Executando o simulador SWIM. 43
Figura 21 – Acessando o simulador SWIM via navegador. 44
Figura 22 – Executando a simulação. 44
Figura 23 – SWIM após o início da simulação. 45
Figura 24 – Gráfico de comportamento do SWIM executado de forma enxuta. 46
Figura 25 – Gráfico de comportamento do SWIM executando acoplado ao framework. . 47
Figura 26 – Gráfico comparação de requisições por segundo. 48
Figura 27 – Gráfico comparação de servers ativos. 49
Figura 28 – Log do simulador SWIM demonstrando o comando removeServer(). 49
Figura 29 – Gráfico comparação da potência dos servidores. 50
Figura 30 – Gráfico comparação média de tempo de resposta. 50

Figura 31 – Gráfico demonstrando disparo de estratégias em relação ao tempo médio de resposta. 51
Figura 32 – Gráfico demonstração unidade cumulativa. 52

LISTA DE TABELAS

Tabela 1 – Tabela de nomes e comandos de sensores. 37

SUMÁRIO

1	**INTRODUÇÃO**	11
2	**FUNDAMENTAÇÃO TEÓRICA**	14
2.1	**Sistemas Auto Adaptativos**	14
2.2	***Cloud Computing***	15
2.3	**Evento**	16
2.3.1	***Microsserviços***	16
2.3.2	***Contêiner***	17
2.3.3	***Docker***	18
2.3.4	***Kubernetes***	19
2.4	**Padrões de projeto**	19
3	**TRABALHOS RELACIONADOS**	21
3.1	**Kubow**	21
3.2	**MAPE-K**	22
3.3	**MORPH**	23
3.4	**SWIM**	24
4	**PROPOSTA**	26
4.1	**Monitor**	26
4.2	**Analisador**	28
4.3	**Planejador**	29
4.4	**Executor**	31
4.5	**knowledge**	33
5	**TESTES E RESULTADOS**	36
5.1	**Sensores**	37
5.2	**Knowledge**	37
5.3	**Sintomas**	38
5.3.1	***Classes Symptoms***	38
5.3.1.1	*LimitSymptom*	*38*
5.3.1.2	*CompareParamsSymptom*	*39*
5.3.1.3	*AverageResponseTimeSymptom*	*39*
5.3.2	***Instâncias de Symptoms***	39

5.3.2.1 *average_response_time_limit_upper* *39*
5.3.2.2 *average_response_time_limit_bottom* *39*
5.3.2.3 *can_add_servers* *39*
5.3.2.4 *has_many_active_servers* *39*
5.3.2.5 *arrival_rate_limit_upper* *40*
5.3.2.6 *arrival_rate_limit_bottom* *40*
5.3.2.7 *not_dimmer_limit_upper* *40*
5.3.2.8 *not_dimmer_limit_bottom* *40*
5.4 **Actions** 40
5.4.1 ***SocketAction*** 40
5.5 **Estratégias adicionadas ao planejador** 40
5.5.1 ***AddServerStrategy*** 41
5.5.2 ***RemoveServerStrategy*** 41
5.5.3 ***DimmerUpStepStrategy*** 41
5.5.4 ***DimmerDownStepStrategy*** 42
5.6 **Effectors** 42
5.6.1 ***SocketEffector*** 42
5.7 **SWIMServiceProvider** 42
5.8 **Execução do sistema alvo SWIM** 43
5.9 **Resultados** 45
5.9.1 ***Número de requisições por segundo*** 47
5.9.2 ***Número de servidores ativos*** 48
5.9.3 ***Potência máxima dos servidores*** 49
5.9.4 ***Tempo médio de resposta.*** 50
5.9.5 ***Utilidade cumulativa.*** 51
6 **CONCLUSÕES E TRABALHOS FUTUROS** 53
6.1 **Considerações gerais** 53
6.2 **Relevância do estudo** 53
6.3 **Trabalhos futuros** 54
REFERÊNCIAS 55

1 INTRODUÇÃO

A computação em nuvem surgiu como uma ferramenta para que empresas pudessem disponibilizar recursos. Com essa nova abordagem , sendo inicialmente chamada de computação utilitária, era possível que um cliente fizesse *upload* de tarefas para um *data center* e fosse cobrado depois pela alocação de recursos. (TANENBAUM, 2017).

A arquitetura de microsserviços é uma abordagem sobre a computação em nuvem que vem ganhando muito espaço. Ela é, acima de tudo, uma solução para o problema de criar e gerenciar aplicações de software complexos, podendo oferecer muitos benefícios para sistemas de médio e grande porte, dentre os que podem ser citados: redução de custos, melhoria de qualidade, agilidade e menor tempo de colocação no mercado. Como cada um dos microsserviços faz um pequeno papel, em aplicações grandes, pode-se diminuir bastante a complexidade de implementação. (SINGLETON, 2016).

Com a mudança de paradigma de comunicação entre sistemas na web, surge um crescente número de aplicações baseadas em microsserviços, além também de várias ferramentas de orquestração de serviços, como, por exemplo, o Kubernetes. A virtualização de infraestrutura é crucial nesse ponto para manter os serviços que são disponibilizados pelos contêineres de forma isolada e consistente, permitindo também a alta portabilidade de microsserviços em ambientes na nuvem. (ADERALDO, 2019).

À medida que as aplicações crescem em tamanho, complexidade e heterogeneidade, a diversidade de recursos de infraestrutura como máquinas virtuais e contêineres que precisam ser constantemente monitorados também cresce, ainda deve-se considerar que alguns serviços de um sistema são mais requisitados que outros, fazendo com que esses tenham de ser replicados várias vezes e em muitos casos em regiões espalhadas geograficamente, nesse cenário o desafio de monitoramento e a coleta de informações aumentam ainda mais, podendo-se exigir o gerenciamento de um número grandioso de eventos dessa natureza ocorrendo no sistema. (JAMSHIDI *et al.*, 2018).

Para continuar cumprindo seus objetivos, a autogestão do sistema é vista como a única saída praticável. Ela relaciona-se com a capacidade do sistema realizar automonitoramento (o sistema faz verificação contínua de seu estado) e autorreconfiguração (com base nas métricas observadas no seu monitoramento, escolhe uma nova configuração e a aplica). Os sistemas que possuem capacidade de autogestão são chamados também de sistemas autoadaptativos (ANGELOPOULOS *et al.*, 2018). Espera-se que sistemas operem em ambientes altamente

dinâmicos e que cumpram vários objetivos, se uma falha é detectada ou uma meta não alcançada, uma nova configuração é adotada. (ANGELOPOULOS *et al.*, 2018).

Ambientes de aplicação de serviços em nuvem podem variar muito seu estado de acordo com os aplicativos que rodem sobre eles, isso faz com que esses ambientes tenham requisitos não-funcionais muito dinâmicos, muitas vezes precisando de mais recursos para que os aplicativos possam operar de forma satisfatória e em outras ocasiões os recursos alocados para esses ambientes são superdimensionados, acabando por ficarem ociosos.

O objetivo deste trabalho é implementar um framework genérico e configurável que permite se acoplar e modificar todos os tipos de cenário. Nesse trabalho, para efeitos de teste foi escolhido utilizar como sistema alvo um ambiente em nuvem, pois ele disponibiliza um contexto que demanda monitoramento e configurações constantes em seu ambiente de execução.

Como objetivos específicos tem-se:

- Modelar uma proposta de um framework de adaptação;
- Implementá-lo utilizando melhores práticas de programação;
- Testá-la em um ambiente que simula um datacenter disponibilizando serviços em nuvem.

Para a relização de pesquisa e conseguir cumprir os objetivos, inicialmente foi feito um levantamento sobre o assunto de sistemas autoadaptativos, onde o objetivo era buscar artigos relacionados com o assunto, após esse levantamento o livro Padrões de projeto, Soluções reutilizáveis de software orientado a objetos (HELM RALPH JOHNSON, 2008) foi lido e os seus principais padrões foram estudados, assim foi possível implementar implementar com as melhores práticas e seguindo os padrões de projeto a camada proposta no documento.

Em uma segunda parte de leitura dos artigos, os artigos que demonstravam e falavam diretamente sobre sistemas adaptativos foram explanados e os principais fatores presentes em sistemas adaptativos foram levantados nessa leitura.

Após todos os estudos e levantamentos, a implementação do framework foi iniciada, foi utilizada a seguinte sequência pra a implementação dos módulos: Monitorador, Knowledge, Analisador, Planejador e por último foi implementado o executor, em cada um dos módulos implementados eram feitos testes manuais simples, com sistemas que simulavam entrada e saída de dados simples. Somente após a implementação e testes de todos os componentes é que foi feita a configuração necessária para o acoplamento ao simulador.

O restante desse documento está estruturado da seguinte forma: o capítulo 2 apresenta fundamentação teórica da proposta, apresentando os conceitos relacionados aos sistemas

autoadaptativos, sistemas de sistemas, microsserviços e discute mais sobre as principais arquiteturas e frameworks relacionados a proposta apresentada. No 3 os trabalhos de pesquisa, artigos e livros relacionados com o assunto, e utilizados como base no estudo são mostrados; O capítulo 4 descreve a proposta de pesquisa, onde detalhes sobre a implementação, dados de comunicação são apresentados e como eles se relacionam, no capítulo 5 os resultados são apresentados e é comparado a simulação do SWIM executando normalmente e também com a camada desse documento aplicada, o capítulo 6 mostra a conclusão de estudo falando também sobre sua relevância e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais termos e conceitos utilizados no decorrer da proposta.

2.1 Sistemas Auto Adaptativos

Segundo WEYNS (2018) Sistemas autoadaptativos são sistemas que conseguem verificar seu próprio estado e procurar uma configuração para adaptar-se ao contexto do ambiente, sendo ainda capazes de identificar falhas de operação e com isso disparar estratégias de configuração para continuar em operação sem deixar de oferecer um serviço. (WEYNS, 2018).

Várias técnicas podem ser consideradas para se construir aplicações com essa abordagem, uma das principais, está na construção de modelos arquitetônicos e de comportamentos, sendo essa técnica conhecida como modelo *Runtime* (Modelo em tempo de execução). Esse modelo se baseia nos requisitos pré-definidos em conjunto com propriedades de estado disponibilizados pelo ambiente em que o sistema opera. (BARBOSA *et al.*, 2017).

Um grupo de sistemas muito importantes para a adaptação, é o conceito de Sistemas de sistemas, que são aplicações formados pela composição de outros diversos sistemas menores e independentes entre si, criando uma nova aplicação. Os componentes indenpendentes foram projetados unicamente para atender seus objetivos, sendo assim, esses não podem atender aos requisitos gerais do sistema. (MAIA *et al.*, 2019).

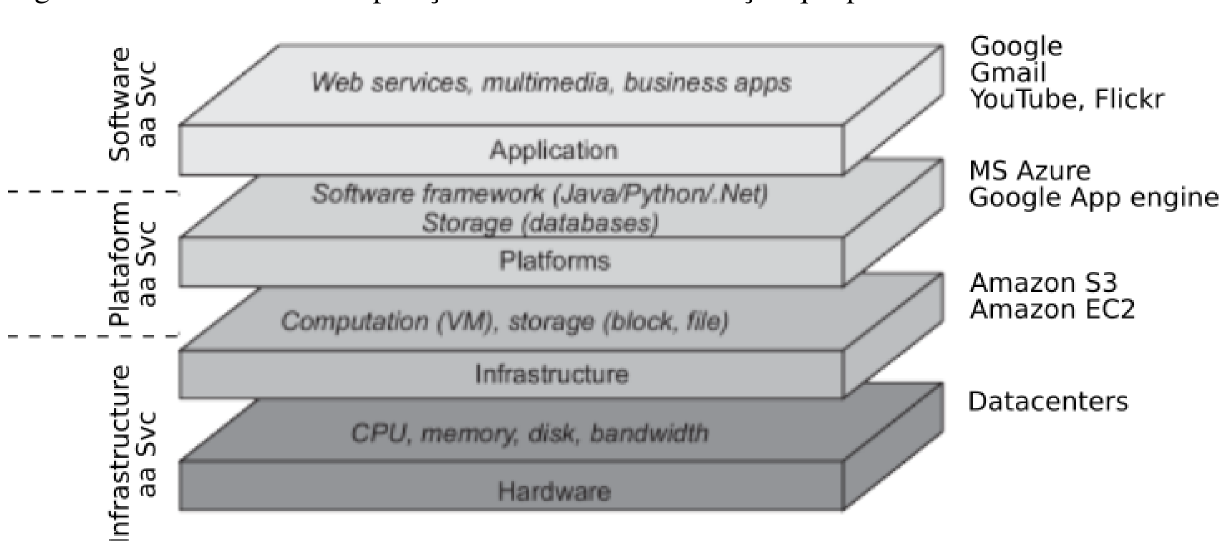
Sistemas de sistemas buscam alcançar funcionalidades que não são capazes de se conseguir com aplicações independentes por si só. Com essa combinação de componentes, surgem requisitos e comportamentos novos que devem ser considerados. Os componentes independentes, que participam da arquitetura, não foram implementados inicialmente para lidar com os novos comportamentos emergentes da integração entre os sistemas, necessitando utilizar técnicas de adaptação para continuar sendo um serviço autônomo e agregar um novo comportamento necessário ao sistema de sistemas. (MAIA *et al.*, 2019).

Outros termos são também largamente utilizados para referenciar sistemas autoadaptativos, entre eles estão: Sistemas Auto-Adaptáveis, Sistemas Autônomos, Sistemas Adaptativos, entre vários outros, nesse trabalho, sempre estarei os referenciando como Sistemas Autoadaptativos.

2.2 *Cloud Computing*

A computação em nuvem, surge como uma alternativa para que empresas de desenvolvimento pudessem disponibilizar seus sistemas de maneira fácil. Era inicialmente chamada de computação utilitária. Essa forma de disponibilização de recursos depois se torna a base do que hoje é conhecida como Computação em nuvem. Essa classe de computação é caracterizada por um *pool* de recursos virtualizados que provê vários benefícios, principalmente facilidade de utilização e de acessibilidade. Recursos em computação em nuvem podem ser configurados dinamicamente, fornecendo base para escalabilidade, sendo assim, se mais tarefas precisam ser realizadas, um cliente pode comprar mais recursos.(TANENBAUM, 2017).

Figura 1 – Camadas da computação em nuvem e os serviços que podem ser ofertados.



Fonte: (TANENBAUM, 2017)

Ainda é importante se falar que existem várias formas de disponibilização de serviços na nuvem, entre elas as principais são: Infrastructure-as-a-Service(IaaS), Platform-as-a-Service(PaaS), Software-as-a-Service(SaaS). O sistema IaaS ou em português infraestrutura como serviço, disponibiliza um serviço de infraestrutura de máquinas onde o ambiente e aplicação podem ser configuradas de acordo com a necessidade do cliente, o PaaS disponibiliza uma plataforma de ambiente já pré-instalada onde a aplicação do contratante será instalada e poderá ser configurada como desejada, e por último o Saas ou aplicação como serviço, é a plataforma de nível mais alta, onde uma aplicação é disponibilizada com objetivo já específico e pode realizar tarefas definidas. (TANENBAUM, 2017).

Um bom exemplo de uso de computação em nuvem, é uma empresa que precisa de um serviço de disponibilização de páginas web, ao invés dessa empresa montar uma estrutura

de servidores e de rede, que custaria muito caro e depende de serviços especializados, é muito comum serem alugados serviços de hospedagem para essa finalidade, e é cobrado da empresa somente sobre o serviço de armazenamento de dados e a disponibilização do serviço de entregas das páginas web.

2.3 Evento

Um termo bastante importante para o assunto é o Evento, que segundo IBM (2005), trata-se de uma alteração no estado do sistema de forma significativa, rede, aplicação ou resultado de um problema.

MAIA *et al.* (2019) utiliza-se de eventos ao referenciar sobre notificações de esgotamento de bateria e confirmação de chegada em um determinado local.

No contexto de uso desse artigo, sempre que se referenciar ao termo, será um estado no sistema alvo que é de interesse da camada adaptativa. Por exemplo, observado que o tempo de espera de uma requisição ultrapassar o limite de *timeout* de comunicação.

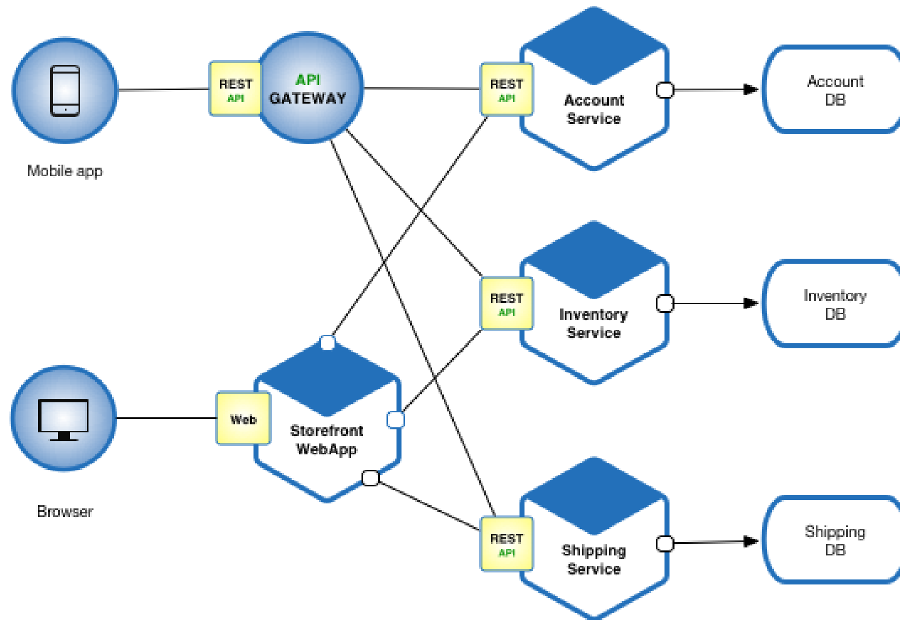
2.3.1 Microsserviços

A arquitetura de microserviçoes, é uma abordagem arquitetural que é montada e pensada para se trabalhar sobre a arquitetura em nuvem. Foi inicialmente pensada a partir da evolução de sistemas monolíticos que anteriormente eram utilizados nos *cluster's* de computação em nuvem. (JAMSHIDI *et al.*, 2018). Se baseia principalmente na ideia de dividir um problema complexo em problemas menores, cada um desses problemas são resolvidos de forma independente. Cada um dos microserviços executa de forma independente em um processo, se comunicando constatemente uns com os outros por meio de protocolos de comunicação simples. São implantados totalmente independentes, e o processo responsável pelo gerenciamento desses serviços, é também totalmente independente, e pode executar de forma separada. (NAMIOT; SNEPPE, 2014).

Se assemelham muito ao SOA (Arquitetura Orientada a serviços), alguns autores até o consideram como um subgrupo da arquitetura, no entanto, o SOA busca a utilização de transporte de dados pesados como barramentos de serviços de corporações, e os microsserviços buscam a utilização de tecnologias de comunicação leve. (JAMSHIDI *et al.*, 2018).

Na figura 2 é mostrado uma simples arquitetura de microsserviços.

Figura 2 – Exemplo arquitetura microsserviços.



Fonte: (FRANZINI, 2017)

A implementação de sistemas com a utilização de microsserviços trás muitos benefícios como:

- Entrega rápida do sistema, já que é possível modularizar totalmente cada um dos componentes que compõem o sistema como um todo e implementar cada um desses módulos de forma separada.
- Escalabilidade do sistema, caso seja necessário escalar alguma parte do sistema, somente aquele serviço que está sendo mais requisitado é escalado novamente.
- Maior autonomia do sistema como um todo, tais como decisões de ambiente, escolha de tecnologias, ect.

2.3.2 Contêiner

Um contêiner de transporte, empacota produtos de variados tipos e tamanhos, independente de quem os transporta, trens, navios caminhões. De maneira análoga, um contêiner de aplicação, empacota aplicações e suas dependências, mantendo-as independente do ambiente em que executam, seja do ambiente de desenvolvimento, testes e até mesmo ambiente de produção. (SILVA, 2016).

Muitos dos microsserviços de um sistema distribuído, utilizam contêineres como ambiente para a execução, tornando a migração da aplicação da fase de desenvolvimento para a produção muito menos dificultosa. Com essa tomada de decisão, utilizando a chamada cultura

DevOps (prática que surge para unir o desenvolvimento de software com a operação de sistema). (JAMSHIDI *et al.*, 2018).

Muitas empresas que disponibilizam recursos para computação em nuvem já oferecem serviços também de execução de contêineres em seus ambientes, principalmente baseados em contêineres docker, entre elas estão principalmente, *Amazon, Digital Ocean, Heroku* etc.

Na área de TI(Tecnologia da Informação) ao se referenciar aos contêineres de aplicação, é muito comum utilizar a palavra em inglês *container*, nesse trabalho em específico, utilizarei sempre a palavra em português contêiner.

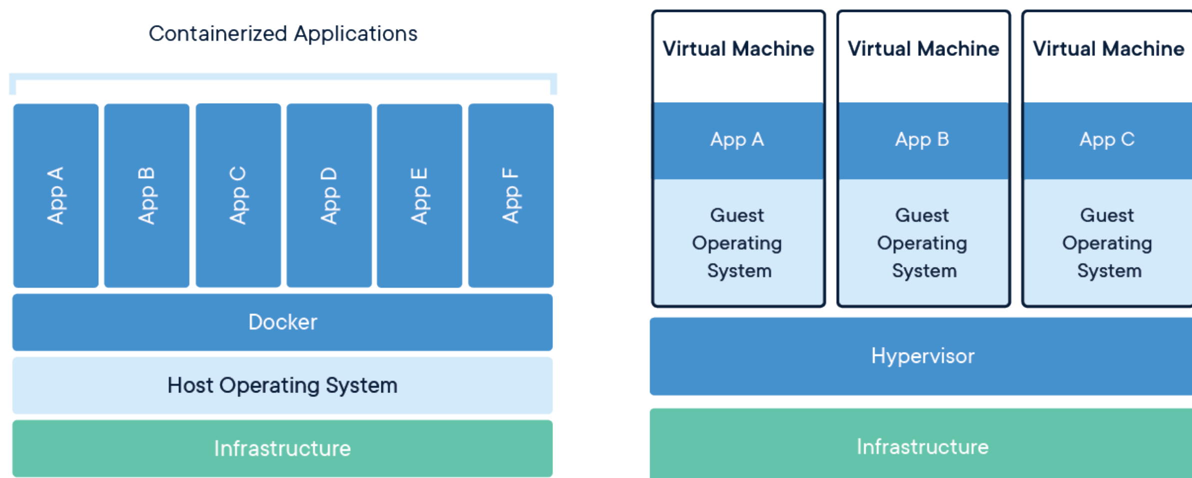
2.3.3 Docker

Docker, é uma ferramenta de gerenciamento de contêineres desde a criação de imagens, até exportação para ambientes em nuvem. (SILVA, 2016). A ferramenta hoje lidera a conteinização de contêineres em nuvem. (DOCKER, 2019).

Os LXC (Linux contêineres) existem desde 2008, mas ganharam popularidade e espaços em computação em nuvem somente no ano de 2014 com a crescente utilização do docker. Uma das principais vantagens de sua utilização ao invés da utilização de uma máquina virtual, é porque o kermel do Linux é compartilhado com os contêineres, ao mesmo momento que é utilizado na máquina hospedeira da aplicação, diminuindo uma camada de um sistema operacional totalmente virtualizado.

A Figura 3 mostra a diferença entre virtualização de um sistema e a utilização do docker.

Figura 3 – Diferença entre virtualização do sistema e utilização de contêiner.



Fonte: (DOCKER, 2019)

2.3.4 Kubernetes

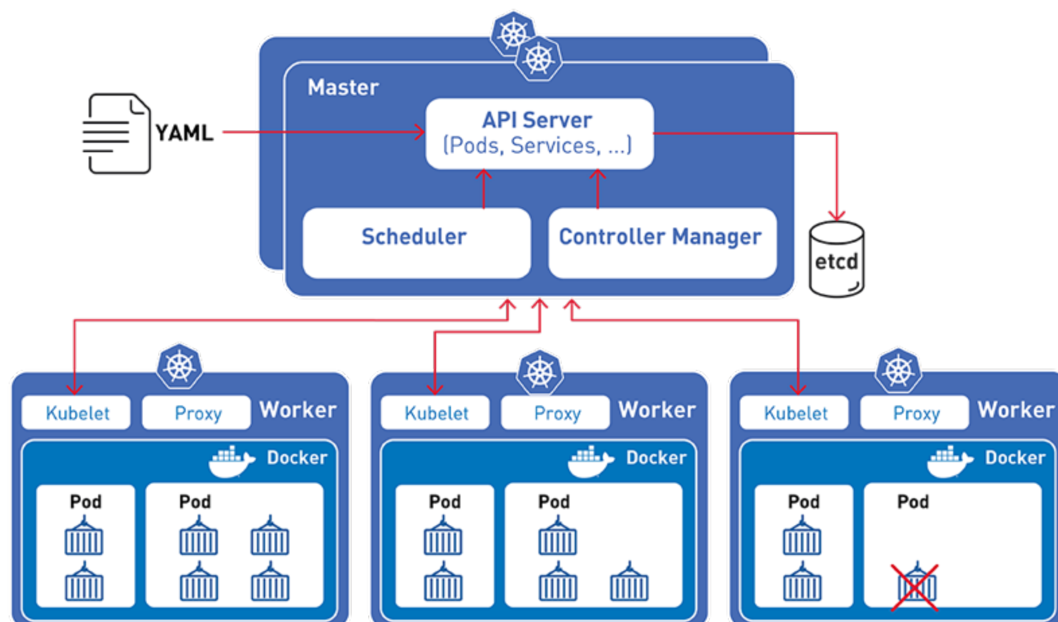
A ferramenta Kubernetes é uma tecnologia que auxilia no gerenciamento de contêineres em etapas como, dimensionamento, implantação e gerenciamento de aplicativos. O lema o principal da comunidade kubernetes é "Escalonando contêineres e não equipes."(KUBERNETES, 2019).

A ferramenta foi inicialmente desenvolvida pelo Google, e hoje é mantida pela comunidade própria e parceiros como, Github, Stack Overflow, entre outros.

Nos trabalhos relacionados, na sessão sobre o Kubow, é citada a ferramenta Kubernetes, e como ela auxilia o kubow na adaptação de contêineres de aplicação.

Na Figura 4 é mostrado a arquitetura interna do kubernetes.

Figura 4 – Arquitetura do Kubernetes.



Fonte: (MAPR, 2019)

2.4 Padrões de projeto

As soluções de software nem sempre são totalmente diferentes umas das outras, muitas vezes problemas parecidos surgem e deixam em dúvida a melhor forma de conseguir resolvê-los. RAMIREZ e CHENG (2018) falam que "O padrão de projeto, é uma solução geral já conhecida para um problema constantemente recorrente na literatura".

HELM, JOHNSON e VLISSIDES (2008) definem padrões de projeto como "Uma

solução para um problema já conhecido que se repete em vários projetos.", ainda segundo os mesmos, o padrão possui 4 características que o definem: um nome, um problema, uma solução, e a consequência na escolha e utilização daquele padrão.

3 TRABALHOS RELACIONADOS

Nesse capítulo, são destacados os trabalhos que foram encontrados na literatura na fase de levantamento bibliográfico.

3.1 Kubow

O trabalho realizado por Aderaldo, Nabor, Medonça (2018) *kubow* é uma tecnologia que auxilia na adaptação de contêineres Docker. Para atingir tal objetivo, personaliza e amplia o *loop* MAPE-K, e o framework Rainbow. Os passos seguidos pela aplicação são demonstrados a seguir.

- Um conjunto de medidores monitoram o estado de um ambiente de contêineres Docker, mantendo estado da aplicação alvo atualizado.
- Um avaliador analisa o estado da aplicação com base nos dados colhidos no passo anterior, e, com base em um modelo de arquitetura, avalia se alguma adaptação é necessária.
- Uma estratégia é selecionada por um gerente de adaptação, com base em várias estratégias pré-carregadas no sistema.
- A estratégia é executada, por meio de um gerenciador de execução.
- Os mecanismos necessários para que a estratégia selecionada seja aplicada, são efetuados por um componente de promulgação do sistema, alterando o comportamento do aplicativo alvo.

Duas unidades separadas, recebem componentes do Rainbow, uma chamada de Mestre, composta principalmente pelo gerenciador de modelo e o analisador de arquitetura, e o Delegate, composto por medidores, sondas e efetuadores.

O kubow ainda utiliza um conjunto de serviços do kubernetes como base para suas sondas e efetuadores, sendo esses serviços chamados via *API*. Duas *API's* estão disponíveis para serem acessadas, a API kubernetes, disponibilizada pela própria ferramenta, e a API Metrics, que é uma extensão da API Kubernetes.

A figura 5 logo a seguir mostra a arquitetura do sistema Kubow e o detalhes de seus componentes.

Figura 5 – Arquitetura do sistema de monitoramento Kubow.

Fonte: (ADERALDO, 2019)

3.2 MAPE-K

Os sistemas de computação vêm se tornando cada vez mais interconectados e diversificados, os arquitetos são cada vez menos capazes de conhecer à priori os componentes que farão interação.

Sistemas autonômicos são vistos como a única saída para essa de situação, são esses sistemas que podem gerenciar seu próprio comportamento com base em metas gerais do sistema.

Foi apresentado pela primeira vez um conceito sobre computação autonômica em um artigo da IBM na universidade de Havard. (KEPHART; CHESS, 2003).

O loop apresentado na universidade no ano de 2003 é denominado MAPE-K, nesse modelo, o fluxo de execução funciona em loop constante onde cada um das fases é executada, então verifica-se se alguma ação deve ser tomada no sistema.

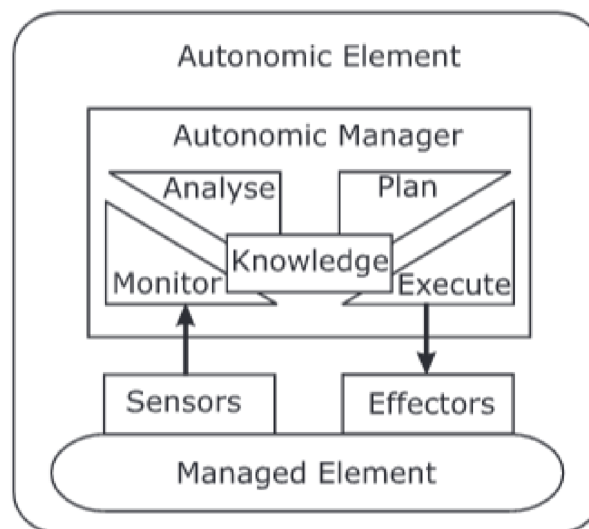
- Monitoring: Fase de monitoramento onde o loop monitora o estado do componente. Podem

também serem feitas ações de filtragem de dados e engatilhar eventos para a camada de analyze.

- Analyze: Fase de análise, onde os dados que estão sendo monitorados na fase anterior são verificados. Com base nas decisões tomadas por essa fase é decidido se alguma ação deve ser tomada no sistema. Existema vários mecanismos que são capazes de modelar situações e prever comportamentos do sistema.
- Planning: Após a fase de análise, se alguma ação precisar ser executada no sistema (por exemplo, enviar um comando para diminuir a quantidade de CPU's ativas), nessa fase será o processamento necessário para criar o comando da ação. Planejamentos são modelados para que o sistema alcance metas e objetivos.
- Execute: Fase de execução do loop, fornece mecanismos de execução de ações e controles que o sistema necessita para que a adaptação seja feita e metas sejam alcançadas.

A figura 6 logo a seguir mostra os componentes do MAPE-K sugeridos pela IBM.

Figura 6 – Ciclo de funcionamento do MAPE-K.



Fonte: (HUESBCHER, 2008)

3.3 MORPH

O MORPH é uma abordagem arquitetural que voltada para sistemas adaptativos, que trabalha com eventos em tempo de execução. Essa arquitetura é composta por 3 camadas principais que executam sobre o sistema alvo, cada uma com objetivos únicos e bem definidos. As camadas superiores se comunicam com as camadas logo abaixo, pedindo informações e

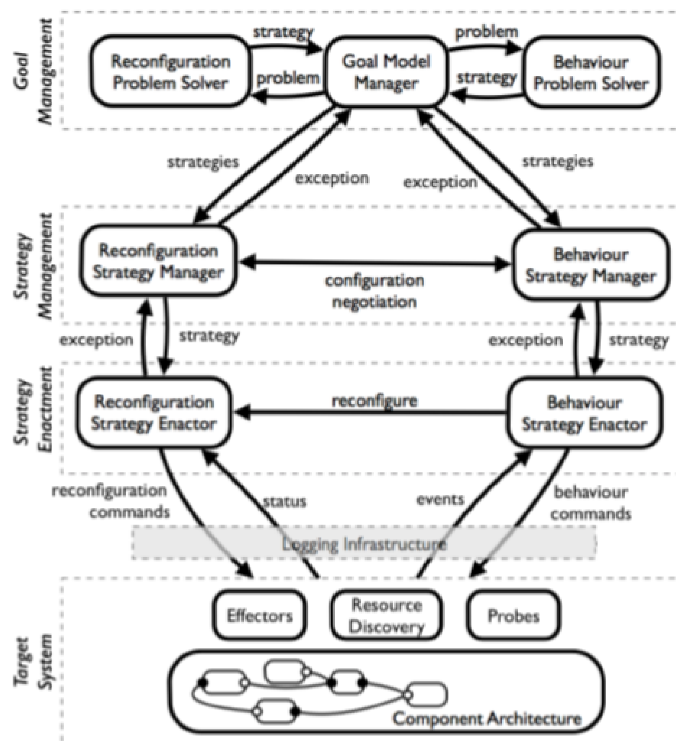
definindo comandos a serem executados exatamente como em uma arquitetura em camadas. (BRABERMAN *et al.*, 2015).

As três camadas da arquitetura são:

- Gerenciamento de objetivos: Busca que objetivo geral do sistema deve ser atingido.
- Gerenciamento de estratégias: Com base nos objetivos selecionados, as estratégias que podem ser utilizadas para que se chegue aquele objetivo.
- Promulgação de estratégias: camada responsável por executar as estratégias selecionadas no sistema alvo, e, ao mesmo tempo monitorar o efeito delas ao longo do tempo, para verificar se o objetivo geral foi atingido.

Uma imagem da arquitetura do MORPH com suas camadas e fluxos de mensagens entre os componentes está disposta logo abaixo na Figura 7.

Figura 7 – Arquitetura do MORPH.



Fonte: (BRABERMAN *et al.*, 2015)

3.4 SWIM

Vários problemas podem surgir na etapa de testes de sistemas autoadaptativos, principalmente se o ambiente de teste for um ambiente real. O problema que surge com o teste

nesses ambientes, é a alta variedade de estados diferentes que os servidores podem assumir, dificilmente podendo replicar o mesmo cenário para comparação com outros sistemas.

Nesse contexto, MORENO, SCHMERL e GARLAN (2018), apresentam o *SWIM*, um simulador de infraestrutura na WEB. Esse simulador apresentado no artigo, foi construído utilizando *omnetpp*, um simulador de eventos para sistemas em rede.

O SWIM simula um servidor web, semelhante ao ZNN e RUBiS (aplicações como o mesmo propósito de simulação), e consiste basicamente em uma camada de servidor web e uma camada de banco de dados. Além desses recursos básicos, o SWIM ainda faz o uso de um balanceador de carga é utilizado para fazer a simulação de vários servidores.

O diferencial do SWIM com os outros simuladores já existentes com o mesmo propósito, é que a ferramenta simula o processo de solicitação de vários servidores, mas roda em um único processo na mesma máquina. Existem vários elementos aleatórios na execução do simulador, como, por exemplo, tempo de resposta da requisição, número de pedidos que chegam, vazão de dados suportados, apesar disso, é possível replicar o mesmo ambiente várias vezes, para simular com várias estratégias e poder compara-las.

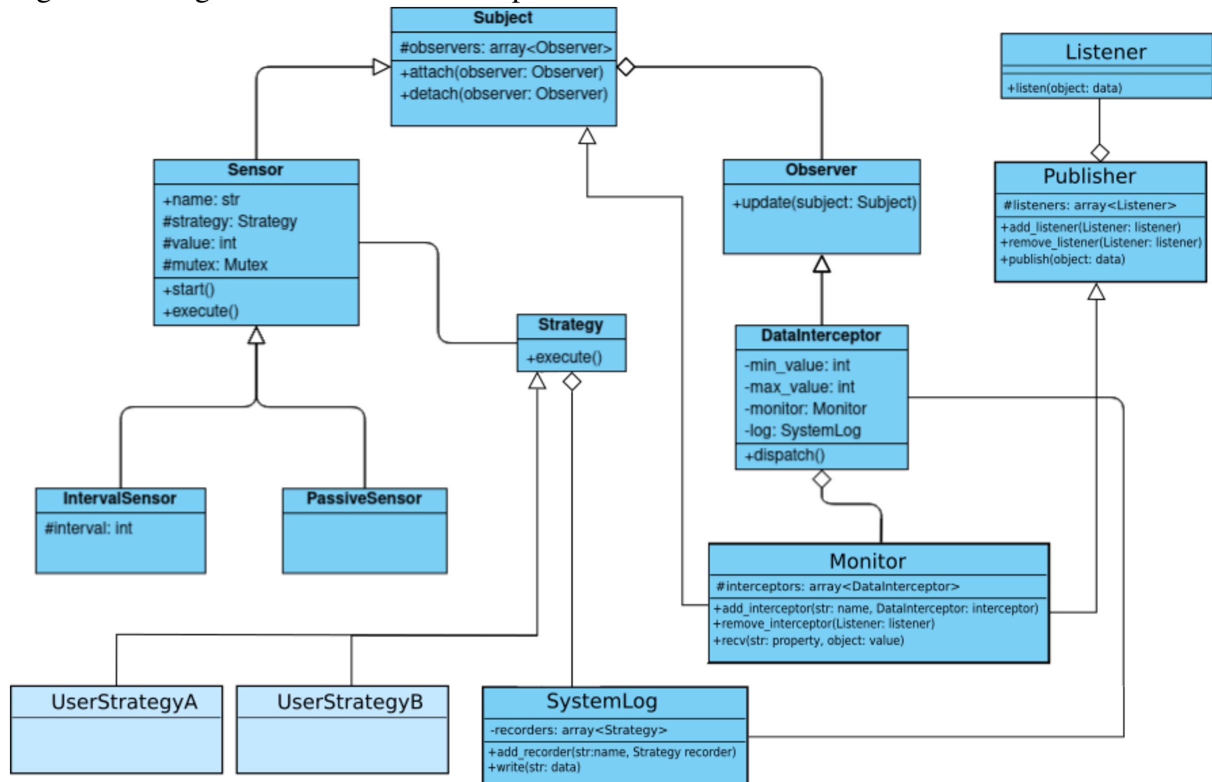
4 PROPOSTA

4.1 Monitor

Esse componente tem como responsabilidade monitorar o sistema alvo através da coleta de dados via sensores e disponibilizá-los para os componentes seguintes do MAPE-K. A disponibilização é realizada através da interface Publisher onde qualquer componente que quiser se ouvir as alterações deve herdar da classe Listener. Há ainda um publicador especial, que disponibiliza os dados via socket ZMQ (ZEROMQ, 2020) e pode ser escultado por sistemas externos.

Na figura 8 vemos o diagrama de classes do componente.

Figura 8 – Diagrama de classes do componente monitor.



Fonte: O Autor.

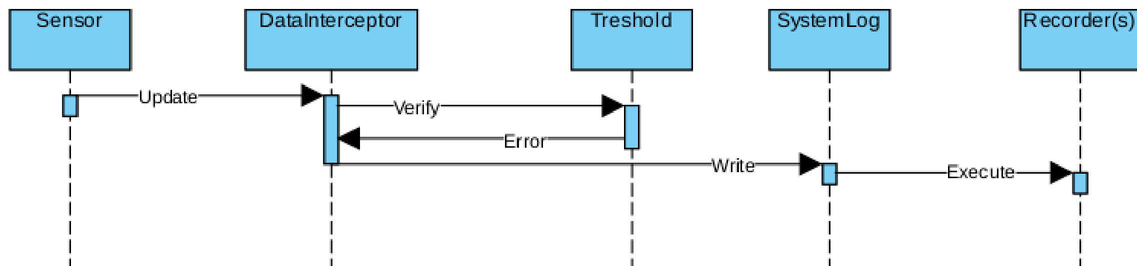
O passo inicial de monitoramento são os sensores, esses são responsáveis por se comunicar com o sistema alvo e verificar o estado das propriedades. Há dois tipos de sensores que podem ser instanciados, o IntervalSensor e o PassiveSensor, onde o primeiro busca a informação em um determinado intervalo de tempo e o segundo fica esperando algum evento para que possa atualizar e disponibilizar um valor.

As classes em destaque na figura 8 UserStrategyA e UserStrategyB, são classes que devem ser criadas herdando de Strategy e a implementação do método execute deve seguir a especificação do sistema em que se deseja monitorar, posteriormente esse método será chamado por um objeto Sensor permitindo sua comunicação com o sistema alvo.

O Sensor repassa os dados para o objeto de DataInterceptor que tem por finalidade receber o dado e verificar se o dado está corrompido ou fora de uma faixa de valor, caso isso aconteça uma exceção é lançada e um registro é criado no sistema de log. Se o dado é lido corretamente seu valor é disparado para o Monitor.

O diagrama de sequência na figura 9 mostra o fluxo de dados quando o valor é interceptado pelo DataInterceptor e seu valor é inválido.

Figura 9 – Diagrama de sequência ao disparar erro na leitura do sensor.

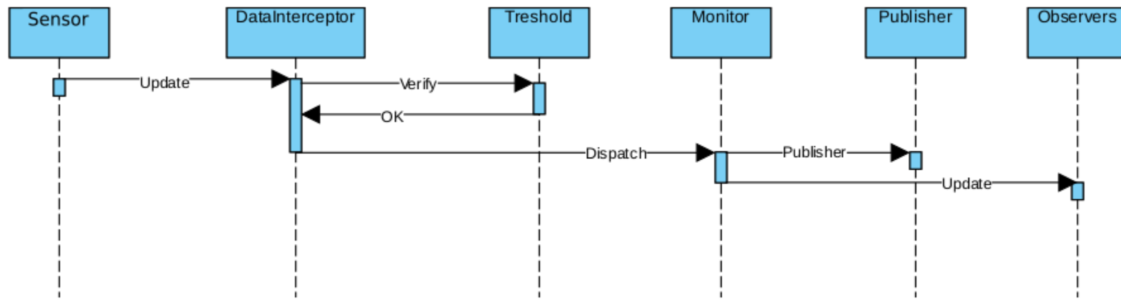


Fonte: O Autor.

O Monitor é o principal componente dessa fase, ele extende a classe Subject e Publisher. Ao ouvir um dado do DataInterceptor, o Monitor atualiza o valor do dado no SystemState, dispara os dados através do método publish proveniente da classe Publisher, e notifica os interessados através do método notify, proveniente da classe Subject.

O diagrama de sequência na figura 10 mostra o fluxo adotado normalmente pela fase de monitoramento.

Figura 10 – Diagrama de fluxo normal de fase de monitoramento.



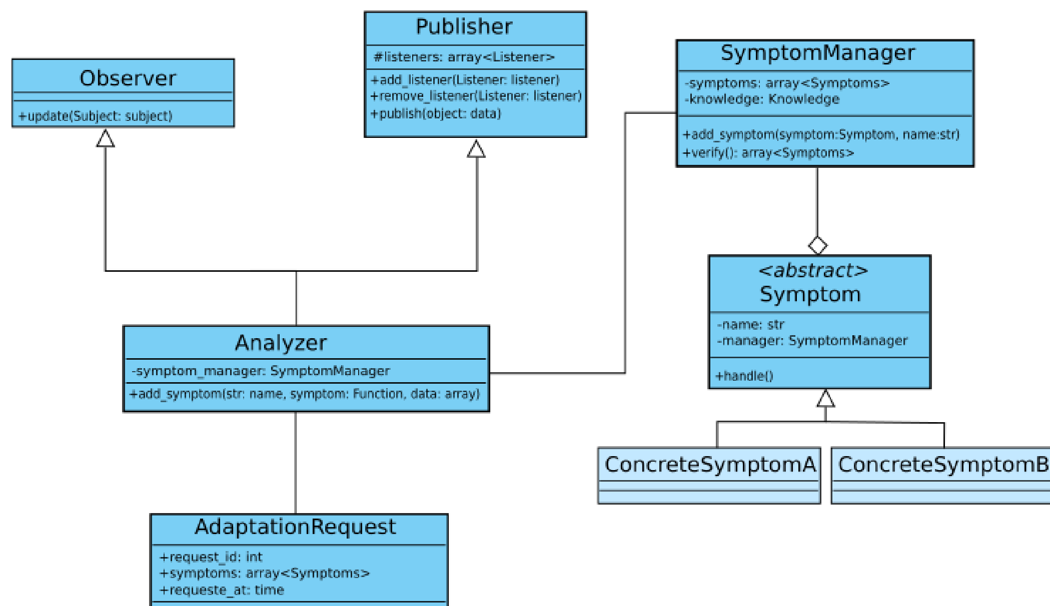
Fonte: O Autor.

4.2 Analisador

O componente Analisador é o responsável por verificar o sistema e procurar por possíveis sintomas, representados pela classe Symptom, que possam levar a um estado indesejável do sistema. Ao verificar que um ou mais sintomas estejam ocorrendo no sistema alvo, um pedido de adaptação, representado pela classe AdaptationRequest, é feito publicando dados para componentes subsequentes da estrutura MAPE.

A figura 11 mostra o diagrama de casse da estrutura do analisador.

Figura 11 – Diagrama de classe do analisador



Fonte: O Autor.

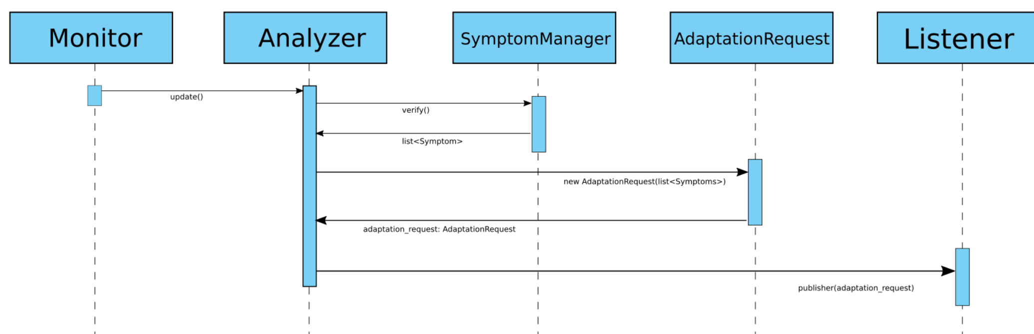
Para representar e verificar dados de um sintoma, o usuário deve estender a classe

Symptom e implementar o seu método abstrato handle de acordo com a especificação do sistema alvo. Um objeto da classe SymptomManager que contenha esse sintoma vai retorná-lo através da chamada do método verify.

A classe principal dessa estutura analyzer, a classe Analyzer, é um Observer e atualiza sempre que recebe notificações do Monitor, disparando a chamada de verify do objeto SymptomManager que ele possui. Após a verificação, caso tenham sintomas ocorrendo, o objeto Analyzer cria um AdaptationRequest com esses sintomas e o dispara.

O diagrama da figura 12 a seguir mostra com detalhes o fluxo para um pedido de adaptação feito pelo analisador.

Figura 12 – Diagrama de sequência do analisador



Fonte: O Autor.

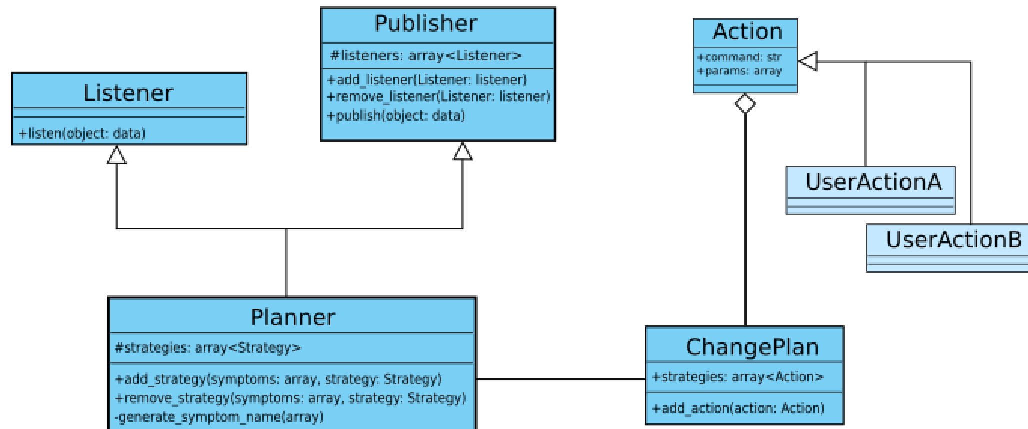
4.3 Planejador

O componente planejador tem como responsabilidade ouvir os sintomas vindos do componente analisador, visto na última seção, e com isso criar um Plano de Mudança (representado pela classe ChangePlan), composto principalmente por ações que devem ser executadas no sistema alvo.

Na figura 13 é possível ver as classes que compõem esse componente.

Na classe Planner, principal objeto desse componente, possui um array de strategies, onde uma delas será selecionada através dos sintomas ouvidos pelo componente, esse array pode ser configurado através dos métodos públicos add_strategy e remove_strategy, onde em add strategy, o primeiro parâmetro do método é um array de strings, que representa um conjunto de symptoms indesejáveis do sistema, e o segundo é a strategy que será responsável por gerar o Plano de mudanças para aquele conjunto de sintomas. Remove strategy recebe como parâmetro

Figura 13 – Diagrama de classe do Planejador.



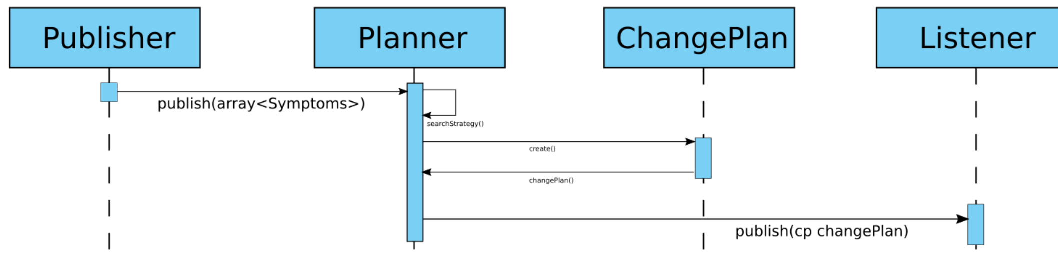
Fonte: O Autor.

somente o array do conjunto de sintomas, e faz somente a remoção daquela estratégia referente aquele conjunto.

A classe action possui somente os atributos command e params, e deve ser extendida pelo usuário do framework. Posteriormente o conteúdo dessa classe será utilizado por efetuadores do componente Executador. Ao adicionar uma action ao plano de mudança deve-se passar uma string que indica que efetuador deve receber aquela Action, assim é possível designar actions específicas para cada um dos efetuadores.

O fluxo de execução do planejador é mostrado na figura 14.

Figura 14 – Diagrama de sequência do Planejador.



Fonte: O Autor.

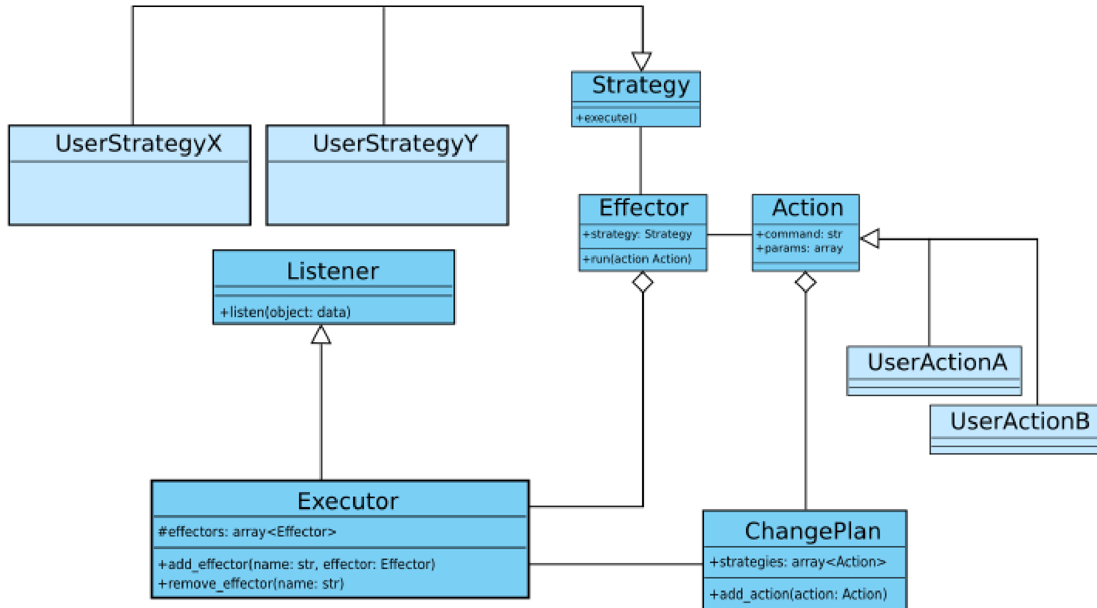
Ao ouvir os sintomas publicados pelo publicador do componente analisador, a classe Planner busca em suas estratégias representadas por um conjunto chave-valor uma que foi pensada para aquele conjunto, ao encontrar essa será executada e o resultado será utilizado para criar um plano de mudança. Ao obter um plano de mudaça a classe Planner executa seu método publish, assim avisando aos interessados sobre um plano de mudanças no sistema alvo.

Esta implementação não disponibiliza para o usuário um conjunto de interfaces que guia a implementação de um gerador de estratégias em reação às modificações no contexto de execução. Cabe ao usuário implementar sob seu projeto esse mecanismo. Ele deve usar os sintomas como entrada para definir que ações devem ser executadas quando uma determinada condição é satisfeita.

4.4 Executor

O componente executor é reponsável pela execução das ações contidas no plano de mudança no sistema alvo, a principal classe desse componente a classe Executor utiliza objetos efetuadores, representado pela classe Effector para chamar uma estratégia e conseguir assim esse acesso. O objeto Strategy contido dentro do objeto efector deve possuir o comportamento que faz a conexão final entre o MAPE e o sistema alvo, assim fechando o ciclo inteiro. O diagrama da figura 15 mostra as classes participantes e suas relações.

Figura 15 – Diagrama de fluxo normal de fase de monitoramento.

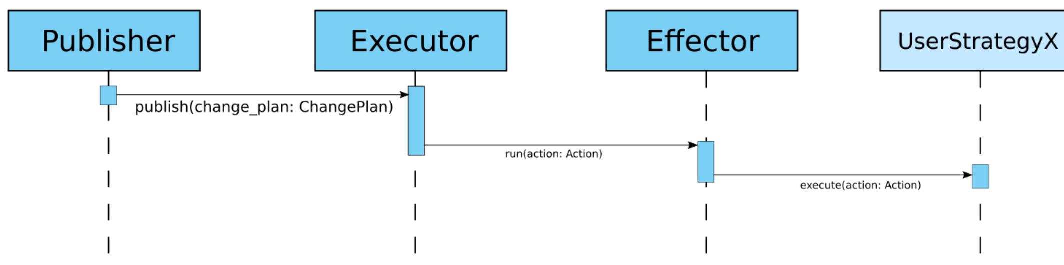


Fonte: O Autor.

As classes UserStrategyX e UserStrategyY são classes ilustrativas que devem ser implementadas pelo usuário, elas recebem um objeto que estende a classe Action e assim saber com que valores deve atuar no sistema alvo. O objeto Executor chamará o método execute de cada uma dessas strategies passando o plano de mudança para cada uma delas.

Na figura 16 é mostrado o diagrama de sequência que mostra o fluxo de dados para a execução de uma action para o UserStrategyA, para os demais efetuadores seguem a mesma estrutura, um efetuador que possui a estratégia associada irá chamar seu método execute().

Figura 16 – Diagrama de sequência do componente Executor.



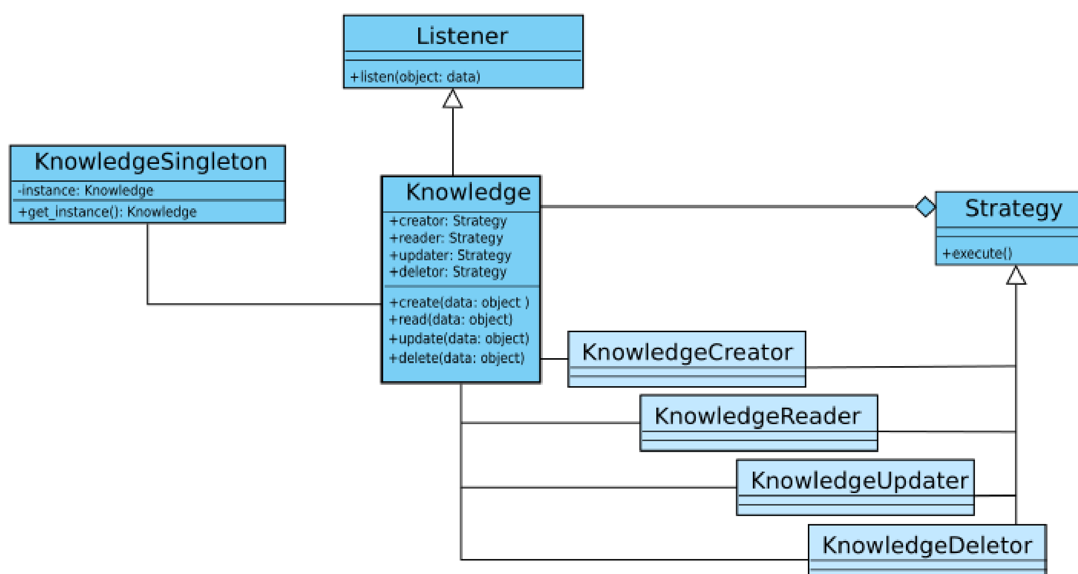
Fonte: O Autor.

4.5 knowledge

O componente Knowledge é responsável por armazenar dados que podem ser acessados por todos os componentes MAPE e ser capaz de manter o seu estado consistente durante toda sua execução.

Veja na figura 17 o diagrama de classes que compõe o knowledge.

Figura 17 – Diagrama de classes do Knowledge.



Fonte: O Autor.

A classe Knowledge é a responsável pela instanciação da base de conhecimento. Ela disponibiliza métodos para que um CRUD sobre dados seja realizado. Cada operação é realizada por um objeto instanciado a partir das classes: KnowledgeCreator, KnowledgeReader, KnowledgeUpdater e KnowledgeDeletor. Como essas classes implementam a interface Strategy, o comportamento é adaptável, pelo usuário, para cada base. O objetivo de cada classe que implementa as estratégias de armazenamento de dados são detalhados abaixo:

- O KonowledgeCreator, responsável por criar estado de variáveis no sistema.
- O KnowledgeReader, responsável por fazer leituras e retornar o valor da variável pesquisada.
- O KnowledgeUpdate, que pode ser utilizado para fazer atualização em variáveis de estados já existentes.

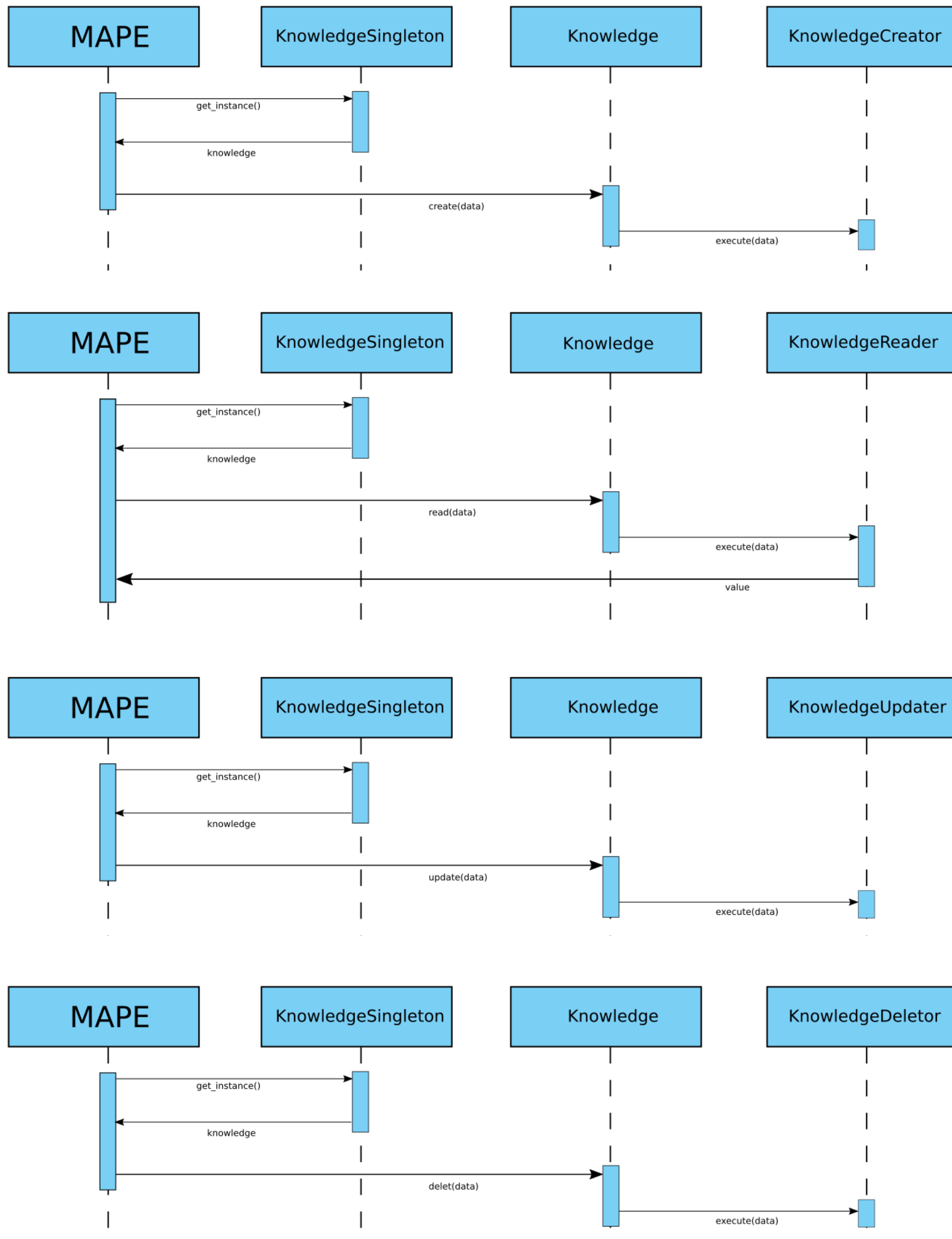
- O KnowledgeDeletor que exclui variáveis de estado.

O framework disponibiliza a classe KnowledgeSingleton que dá acesso, via método get_instance a uma instância constante da base de conhecimento representada pela classe knowledge. Isso é importante para que se mantenha um único ponto de acesso ao estado do sistema monitorado representado na base de conhecimento.

Para que seja possível o Knowledge receber dados do Monitor ou qualquer de qualquer outro componente, a classe Knowledge estende Listener, permitindo se inscrever e ouvir publicações. Ao ouvir dados de um componente é chamado o método create e criará o dado recebido na base de conhecimento.

A figura 18 mostra o diagrama de sequência das 4 operações que podem serem feitas pelo componente.

Figura 18 – Diagrama de fluxo normal de fase de monitoramento.

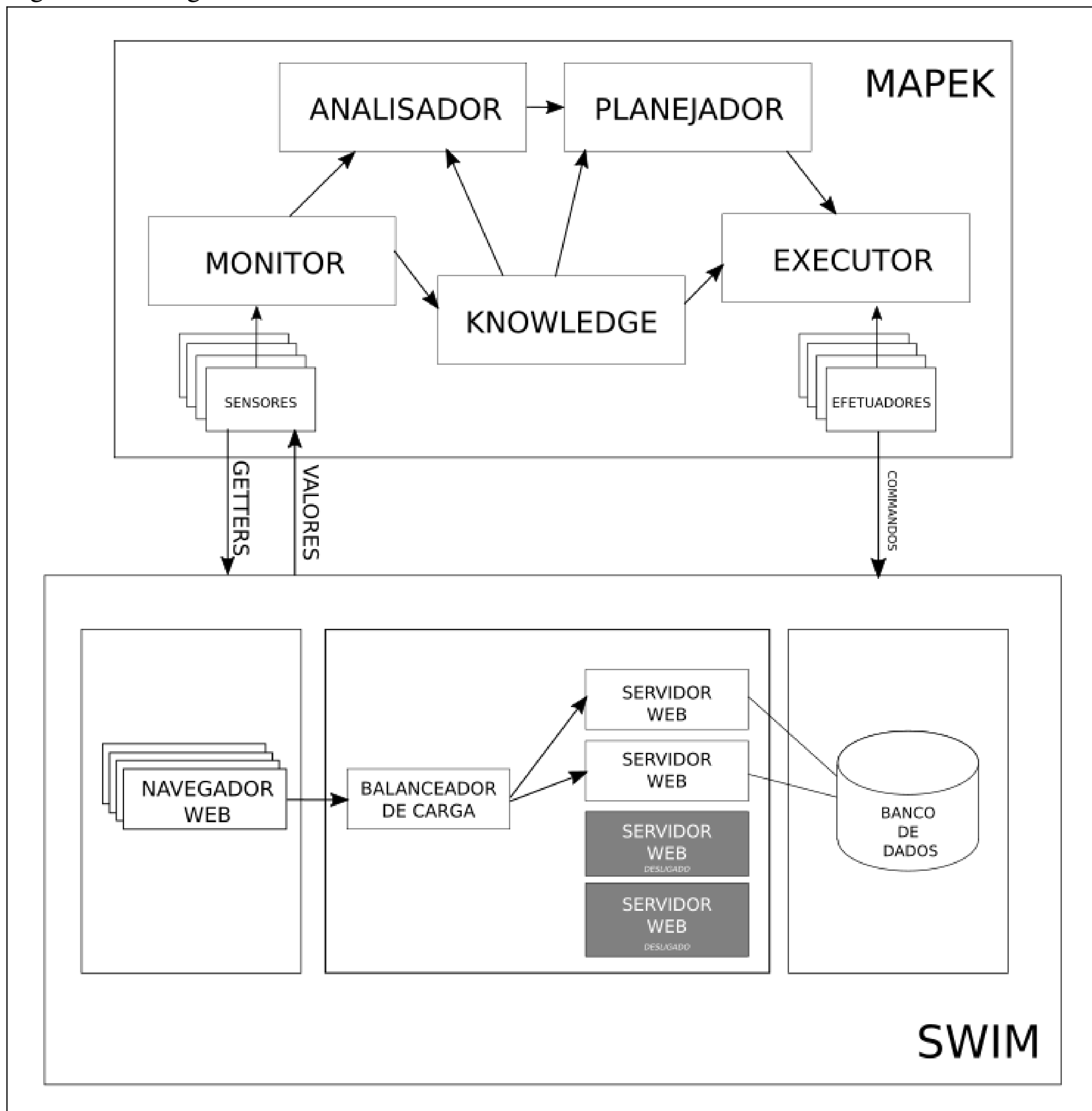


Fonte: O Autor.

5 TESTES E RESULTADOS

Como ambiente de testes para validar a implementação do MAPE-K, foi utilizado o simulador SWIM (MORENO *et al.*, 2018), um esquema da comunicação entre os componentes e sua estrutura interna é mostrada na figura 19.

Figura 19 – Diagrama MAPE-K e SWIM.



Fonte: O Autor.

O Simulador disponibiliza uma porta de conexão socket, por padrão na porta 4242, ainda é disponibilizado um documento, com os comandos de getters e setters que podem ser enviados ao simulador, esses comandos foram utilizados para a fase de monitoramento e planejamento.

5.1 Sensores

A implementação do MAPE disponibiliza basicamente 2 tipos de sensores que podem ser utilizados, o IntervalSensor e PassiveSensor, todos os sensores utilizados na integração com o SWIM foram do tipo IntervalSensor. A implementação do comportamento do sensor basicamente envia um comando ao SWIM e recebe o valor correspondente.

A tabela 1 mostra a lista de comandos, que é o comando enviado a api do SWIM, a lista de variáveis, que é nome identificador que será guardado no knowledge, e uma pequena descrição dessa variável.

Tabela 1 – Tabela de nomes e comandos de sensores.

Commandos	Variável	Descrição
get_active_servers	active_servers	Número de servidores ativos.
get_arrival_rate	arrival_rate	Média de requisições nos últimos 60 segundos.
get_basic_rt	basic_rt	Média de tempo de resposta básica nos últimos 60 segundos.
get_basic_throughput	basic_throughput	Média de saída de respostas dos últimos 60 segundos.
get_dimmer	dimmer	Nível de potência máxima dos servidores.
get_max_servers	max_servers	Número máximo de servidores.
get_opt_rt	opt_rt	Média de tempo de resposta com conteúdo opcional nos últimos 60 segundos.
get_opt_throughput	opt_throughput	Média de saída de respostas com conteúdo opcional dos últimos 60 segundos.
get_servers	servers	Número de servidores.

5.2 Knowledge

Como base de conhecimento dessa implementação optou-se por não utilizar um banco de dados convencional, em vez disso optou-se a utilização de armazenar as informações em um arquivo de texto com configuração chave valor, em um arquivo formato .json, onde as chaves representam as variáveis mostradas na tabela 1 da seção anterior, e o valor representam o estado atual das variáveis.

Cada uma das estratégias, KnowledgeCreator, KnowledgeReader, KnowledgeUpdater e KnowledgeDeletor ao executar faz a leitura do arquivo, manipula sua estrutura e o salva

novamente.

É mostrado logo a seguir um exemplo do arquivo .json com os dados do simulador SWIM.

```json
{
    "basic_rt": 0.00320049,
    "arrival_rate": 9.15,
    "active_servers": 3.0,
    "dimmer": 0.4,
    "opt_rt": 0.0856854,
    "max_servers": 3.0,
    "opt_throughput": 3.81667,
    "servers": 3.0,
    "basic_throughput": 5.38333
}
```

5.3 Sintomas

Foram criadas 4 classes que estendem a classe Symptom, a seguir é exibida a lista com o nome e descrição de seu comportamento.

5.3.1 Classes Symptoms

5.3.1.1 LimitSymptom

O LimitSymptom tem como finalidade verificar limites de valores, sejam eles superiores ou inferiores no knowledge. Para isso são recebidos 3 parâmetros para que possa ser possível instanciá-los, property, que indica que propriedade será lida, signal, um dos sinais de comparação maior(>) que ou menor que(<), e limit, que tem como finalidade indicar o valor de limite da propriedade.

5.3.1.2 CompareParamsSymptom

Tem por finalidade comparar propriedades do knowledge, recebe como parâmetros param_1 e param_2, e retorna verdadeiro se o valor da propriedade com o nome de param_1 é maior que o valor da propriedade com o nome de param_2.

5.3.1.3 AverageResponseTimeSymptom

Verifica a média de tempo de resposta e faz comparação com um limite passado por parâmetro, a média de tempo de resposta definido na classe como average_response_time é definido também no exemplo de execução do SWIM no arquivo /examples/simple_am/SwimClient.cpp, no método double SwimClient::getAverageResponseTime().

5.3.2 Instâncias de Symptoms

5.3.2.1 average_response_time_limit_upper

Faz a verificação da média de tempo de resposta ultrapassa o limite de 0.75, instanciada a partir da classe AverageResponseTimeSymptom.

5.3.2.2 average_response_time_limit_bottom

Verifica se a média de tempo de resposta do servidor está abaixo do limite de 0,75, instanciada a partir da classe AverageResponseTimeSymptom.

5.3.2.3 can_add_servers

Verifica se o número de servidores ativos é menor que o número máximo de servidores. Instanciado a partir da classe CompareParamsSymptom, passando como parâmetros max_servers e active_servers.

5.3.2.4 has_many_active_servers

Verifica se há pelo menos 1 servidor ativo no sistema alvo, para isso é instanciado a partir da classe LimitSymptom.

5.3.2.5 arrival_rate_limit_upper

Verifica se a média de requisições ultrapassa o limite de 15 requisições/segundo, utiliza a classe LimitSympsom para fazer a verificação.

5.3.2.6 arrival_rate_limit_bottom

Verifica se a média de requisições está abaixo de 10 requisições/segundo, utila a classe LimitSympsom para fazer a verificação.

5.3.2.7 not_dimmer_limit_upper

Verifica se o limite do dimmer está abaixo de seu limite superior, ou seja, 1, a classe LimitSymptom é utilizada para fazer essa verificação.

5.3.2.8 not_dimmer_limit_bottom

Verifica se o limite do dimmer está acima de seu limite inferior, 0,1, a classe LimitSymptom é utilizada para fazer essa verificação.

5.4 Actions

5.4.1 SocketAction

Essa action foi criada pensando nos dados que serão necessários para chamadas de execução no sistema alvo via socket, para isso recebe como parâmetros um comando (command) e uma lista de parâmetros (params). Posteriormente, na fase de execução essa action será utilizada no efetuador SocketEffector (será abordado na seção efetuadores) para que seu comando, com os seus parâmetros sejam executados utilizando o SWIMServiceProvider.

5.5 Estratégias adicionadas ao planejador

Na simulação com o SWIM foram adicionados 4 grupos de estratégias. E cada uma delas foi interligada a um grupo de sintomas, a descrição de cada uma das estratégias e seu grupo de sintomas está especificado a seguir.

5.5.1 *AddServerStrategy*

Essa estratégia é disparada quando se deseja adicionar um novo servidor no simulador, para isso seu método execute retorna um plano de mudanças contendo uma action do tipo SocketAction, onde possui o comando add_server. A sua execução ainda faz a atualização de active_sservers (representa o número de servidores ativos) na base de conhecimento adicionando incrementando em 1 o seu valor. É Interligada ao grupo de sintomas.

- average_response_time_limit_upper
- can_add_servers

5.5.2 *RemoveServerStrategy*

Essa estratégia visa fazer a remoção de 1 servidor no sistema alvo, no plano de mudanças retornado pelo método execute, contém uma action do tipo SocketAction com o comando remove_server, além disso faz a atualização na base de conhecimento da variável active_servers, decrementando o ser valor em 1. É interligada ao grupo de sintomas.

- average_response_time_limit_bottom
- has_many_active_servers
- arrival_rate_limit_bottom

5.5.3 *DimmerUpStepStrategy*

Aumentará a potência dos servidores ativos (dimmer) em 1 passo. A estratégia de dividir o dimmer em passsos é presente nos exemplos do sistema alvo SWIM no arquivo /examples/simple_am/simple_am.cpp, igualmente a presente nesse arquivo, foi escolhido utilizar 5 passos de potência, um valor de dimmer selecionado por essa estratégia é maior que o nível atual fazendo um aumento total no nível de potência do simulador. O método execute dessa estratégia retorna um plano de mudança contendo um SocketAction com o comando set_dimmer e como parâmetro, o valor de dimmer selecionado. É ligada ao grupo de sintomas listado a seguir.

- average_response_time_limit_upper
- arrival_rate_limit_upper
- not_dimmer_limit_upper

5.5.4 *DimmerDownStepStrategy*

Igualmente à estratégia anterior, divide o dimmer em passos e seleciona nível para ser aplicado ao sistema alvo com a única diferença de, ao invés de selecionar um nível de potência maior é selecionado um menor, diminuindo o nível de potência do sistema alvo. No plano de mudanças contido em seu retorno irá conter o comando set_dimmer e o valor de potência selecionado como parâmetro. Os seguintes sintomas são responsáveis por disparar essa estratégia.

- average_response_time_limit_bottom
- arrival_rate_limit_bottom
- not_dimmer_limit_bottom

5.6 Effectors

5.6.1 *SocketEffector*

Tem por função acessar o sistema alvo e executar actions as quais lhe foram atribuídos, utiliza a classe SWIMServiceProvider (será visto na próxima seção) para fazer esse acesso e poder executar as actions. O seu método execute recebe como parâmetro uma action do tipo SocketAction e faz a transformação de sua estrutura para formato string, seguindo a estrutura:

```
1 <command> <param_1> <param_2> ...
```

Após isso a string é utilizada como comando e executada no sistema alvo através do serviço que é disponibilizado pela classe SWIMServiceProvider.

5.7 SWIMServiceProvider

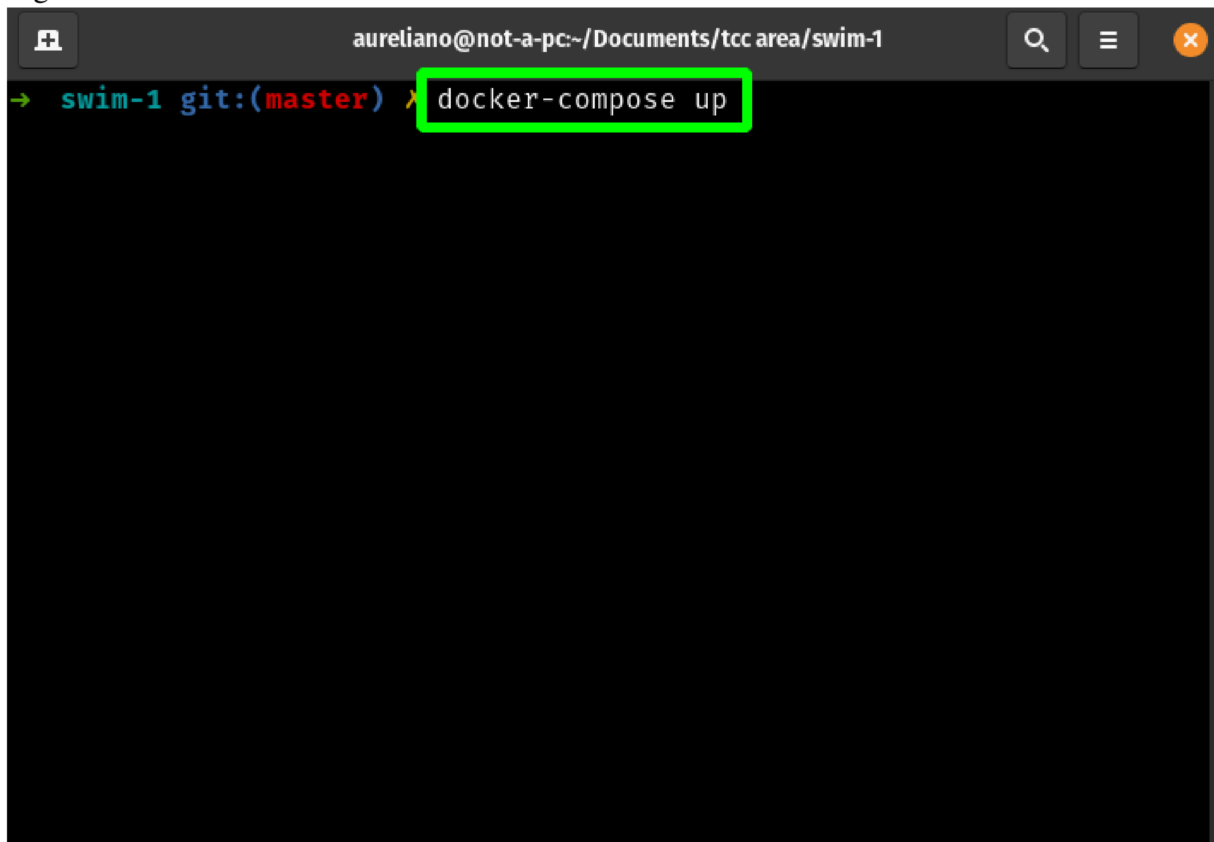
A classe SWIMServiceProvider é uma classe que disponibiliza a conexão para acessar o sistema alvo, o uso desta é necessário, pois o SWIM disponibiliza somente uma conexão para ser dividida entre todos os componentes. Para a disponibilização do objeto de conexão é disponibilizada por uma estrutura que implementa o padrão Singleton, disponibilizando uma única conexão para todos os sensores e os efetuadores. Além disso, o acesso à conexão do SWIM é controlado por um Mutex, liberando o acesso à cada um dos componentes de forma exclusiva naquele dado momento de execução.

5.8 Execução do sistema alvo SWIM

O simulador SWIM disponibiliza no total 10 cenários de execução, sendo numerados do número 0 até o número 9, para a execução e teste da camada MAPE-K implementada foi escolhido o cenário 2. Esse cenário foi selecionado, sendo o mesmo da demonstração de execução mostrada no artigo do simulador, sendo executado da seguinte forma.

1. É Executado o framework SWIM com o comando docker-compose up em sua pasta raiz, como mostrado na figura 20.

Figura 20 – Executando o simulador SWIM.

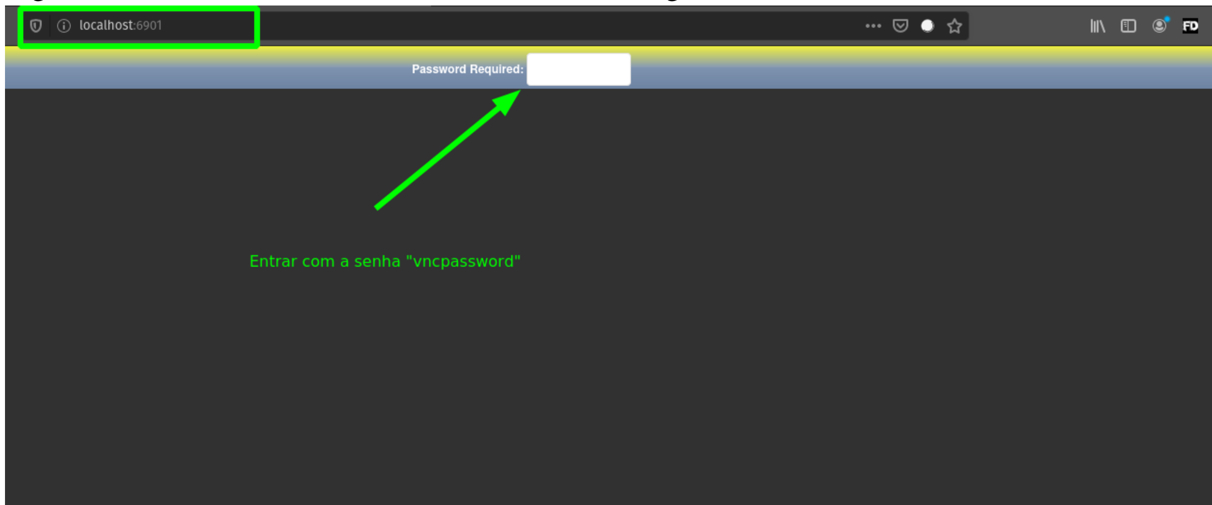


Fonte: O Autor.

2. É utilizado um navegador qualquer para acessar o sistema via interface gráfica, visitando o link http://localhost:6901.
3. Acesse com a senha vncpassword.
4. É necessário ir até o path /headerless/seams-swim/swim/simulations/swim.
5. O comando ./run.sh sim 2 é executado.

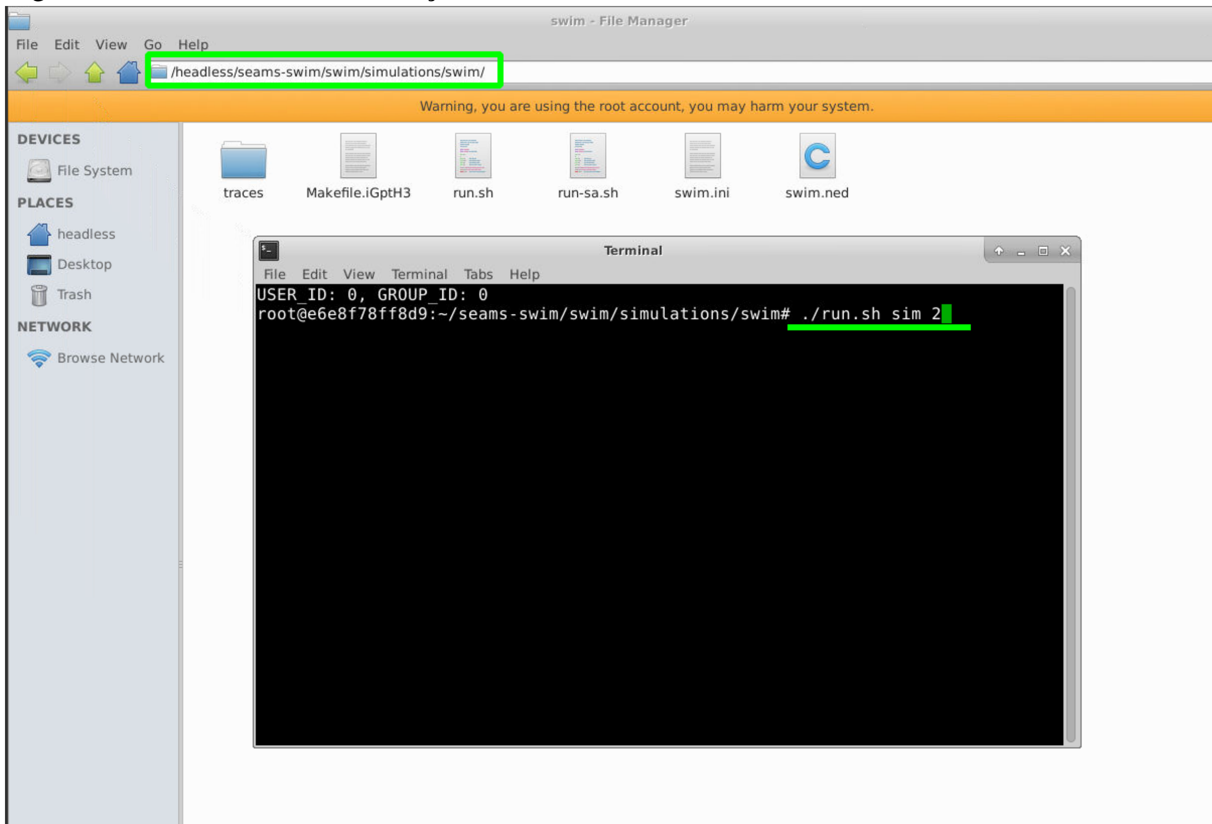
Após esses passos o simulador estará sendo executado e disponibilizará um socket na porta 4242 do seu localhost, onde o objeto SWIMServiceProvider se conecta. A Figura 23

Figura 21 – Acessando o simulador SWIM via navegador.



Fonte: O Autor.

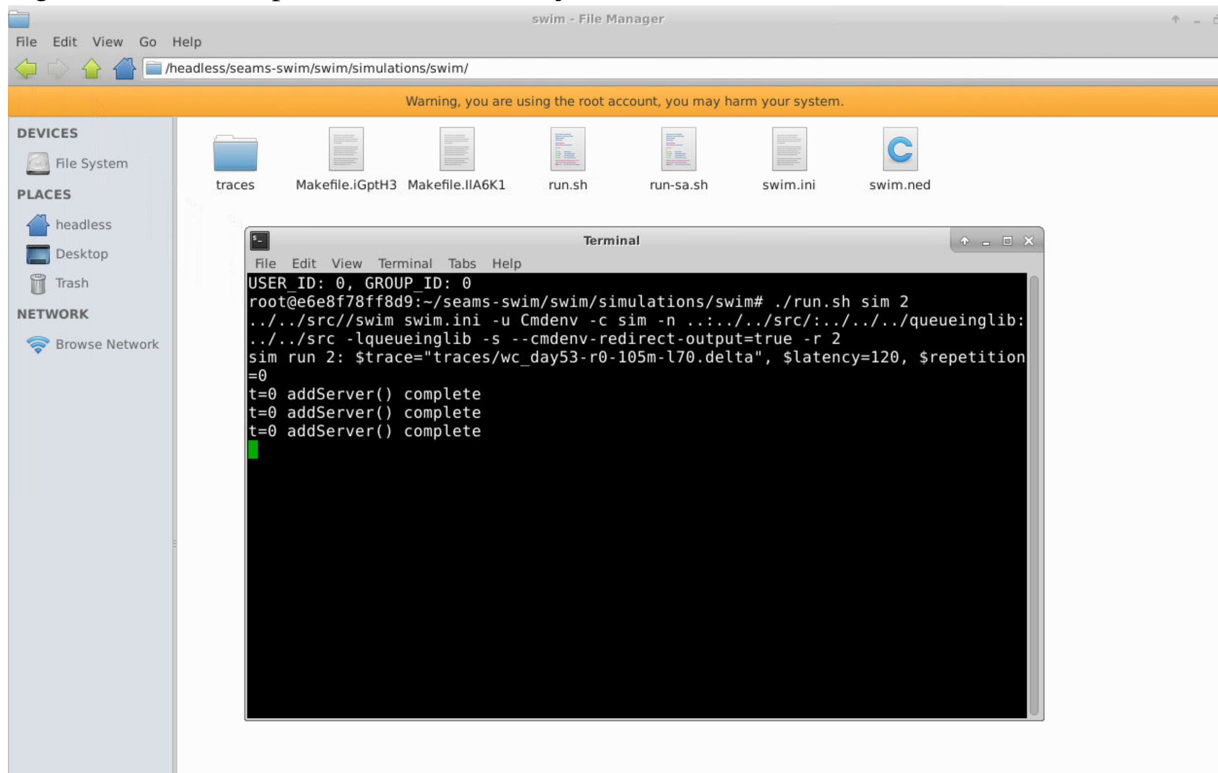
Figura 22 – Executando a simulação.



Fonte: O Autor.

mostra o terminal de comandos após o início da simulação.

Figura 23 – SWIM após o início da simulação.

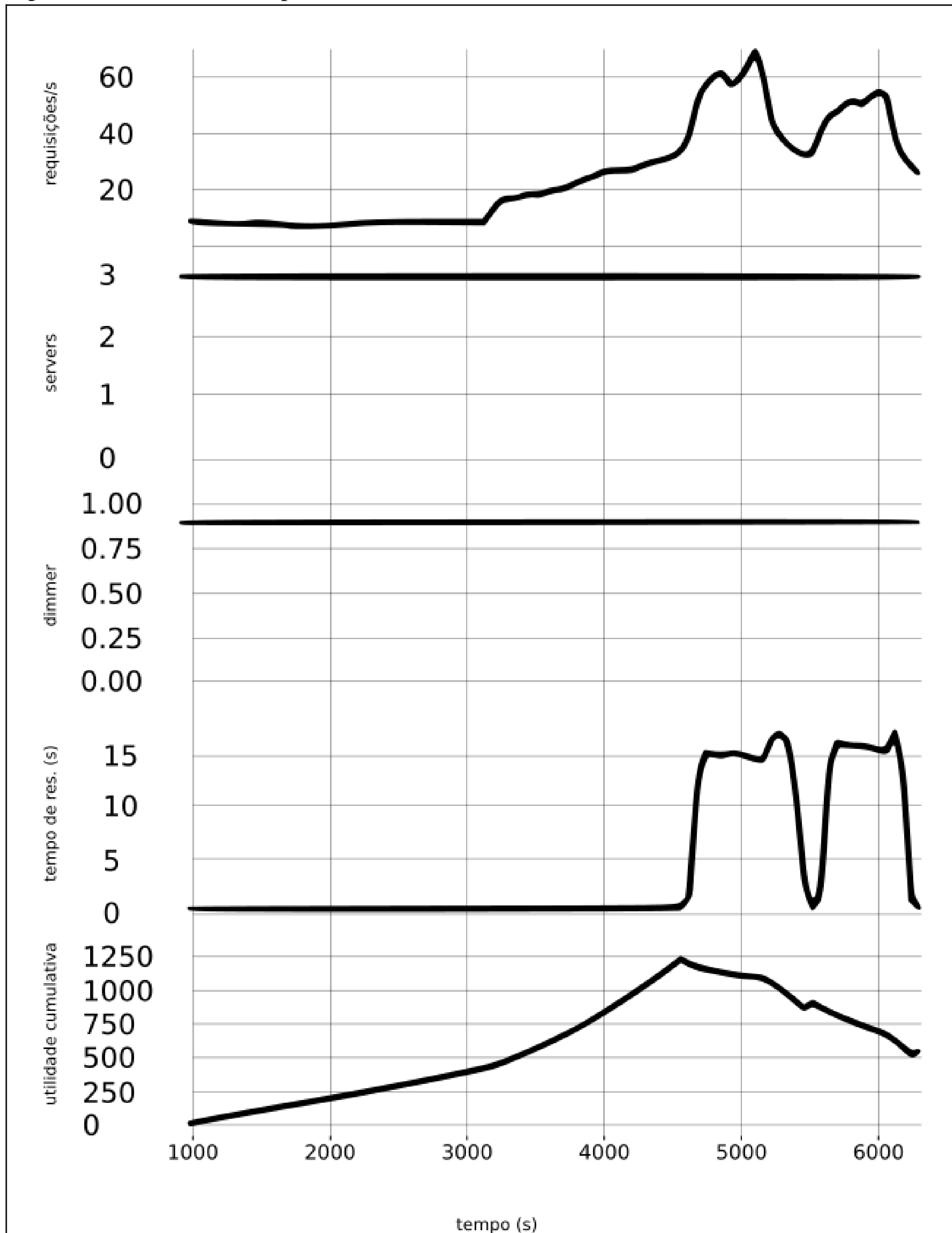


Fonte: O Autor.

5.9 Resultados

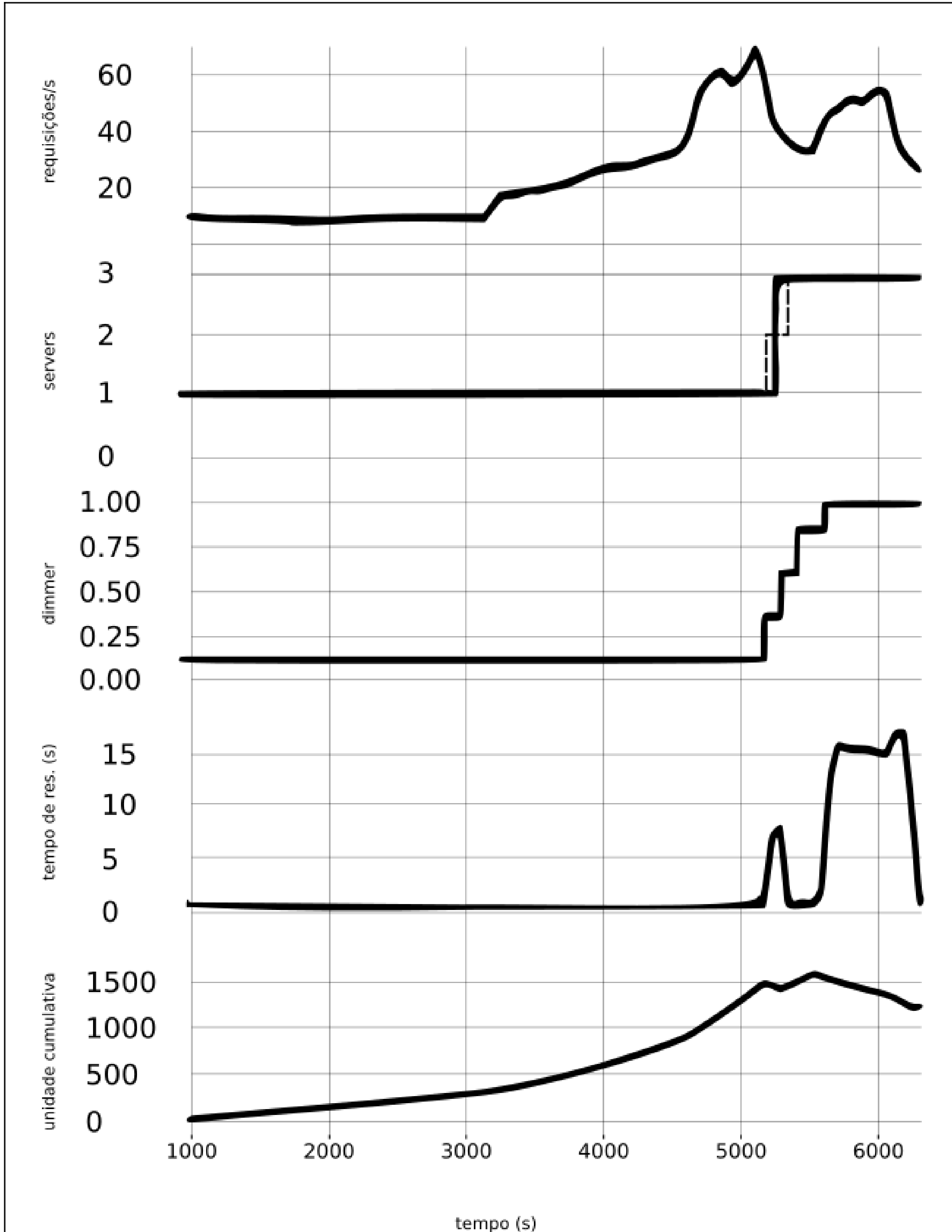
As figuras a seguir mostram os resultados obtidos a partir das execuções do simulador SWIM, a Figura 24 mostra sua execução se forma enxuta, e a figura 25 mostra a execução acoplada ao framework de adaptação. Ambas foram obtidas a partir de uma ferramenta plotResults.R, escrita na linguagem R, disponibilizada pelo próprio simulador, e disponível na pasta /headless/seams-swim/swim/tools. Para melhorar a qualidade das imagens, essas foram editadas utilizando um software de vetorização.

Figura 24 – Gráfico de comportamento do SWIM executado de forma enxuta.



Fonte: O Autor.

Figura 25 – Gráfico de comportamento do SWIM executando acoplado ao framework.



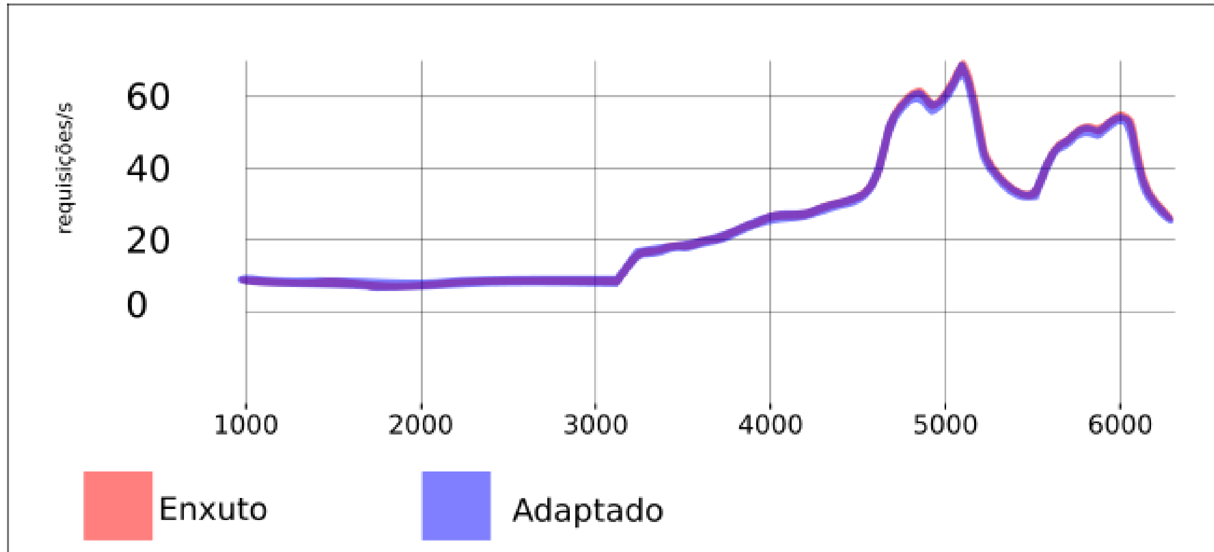
Fonte: O Autor.

5.9.1 *Número de requisições por segundo*

O gráfico da figura 26 mostra em detalhes o gráfico de requisições por segundo das duas execuções sobrepostas.

O simulador SWIM executa de forma igual nas duas execuções, já que o cenário de execução escolhido é o mesmo, é posssível ver os dois gráficos totalmente sobrepostos o que sustenta que os dois cenários de execução são exatamente os mesmos.

Figura 26 – Gráfico comparação de requisições por segundo.

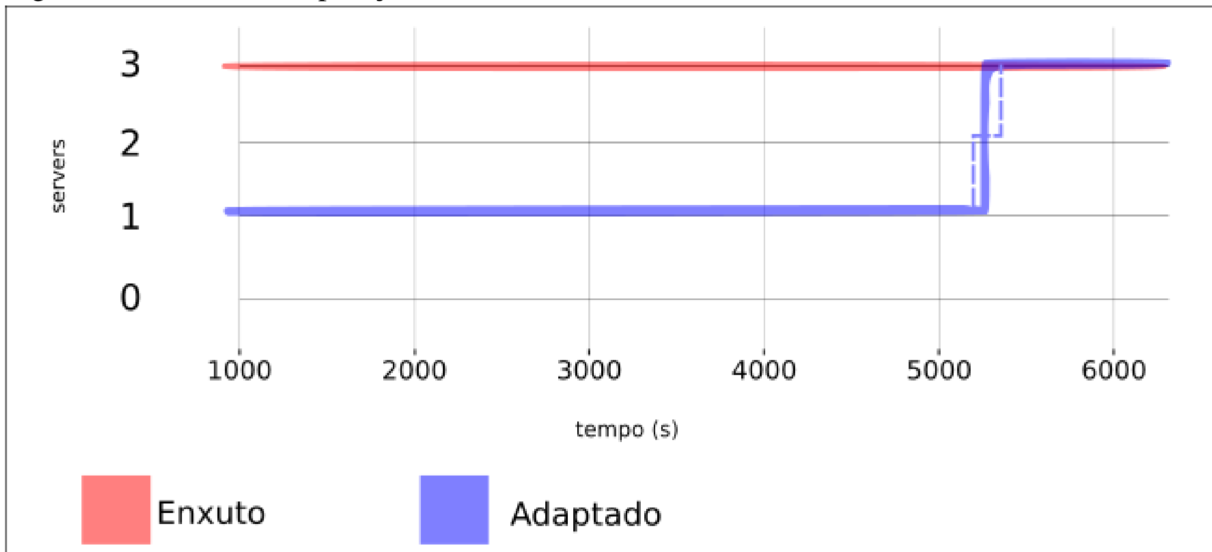


Fonte: O Autor.

5.9.2 *Número de servidores ativos*

A figura 27 contém o gráfico de servidores ativos ao longo do tempo de execução, é visível que ao executar o simulador com o framework MAPE-K acoplado o seu comportamento se altera. Um detalhe importante de se comentar, é que nessa simulação o número de servidores no início são 3. Na simulação com a adaptação apesar de iniciar o gráfico com o valor de 1, é possível ver em sua execução que logo nos primeiros momentos de execução 2 servidores são desligados devido a ocorrência de sintomas no sistema, isso é mostrado na figura 28 onde o log do sistema mostra que os servers são desligados.

Figura 27 – Gráfico comparação de servers ativos.



Fonte: O Autor.

Figura 28 – Log do simulador SWIM demonstrando o comando removeServer().

```
Terminal
File  Edit  View  Terminal  Tabs  Help
USER_ID: 0, GROUP_ID: 0
root@b34e58a6c3d0:~/seams-swim/swim/simulations/swim# ./run.sh sim 2
../../src//swim swim.ini -u Cmdenv -c sim -n ..:../../src/:../../../queueinglib:
../../src -lqueueinglib -s --cmdenv-redirect-output=true -r 2
sim run 2: $trace="traces/wc_day53-r0-105m-l70.delta", $latency=120, $repetition
=0
t=0 addServer() complete
t=0 addServer() complete
t=0 addServer() complete
t=242.34715 executing removeServer()
scheduled complete remove at 242.34715
t=272.389361 executing removeServer()
scheduled complete remove at 272.41917176344
```

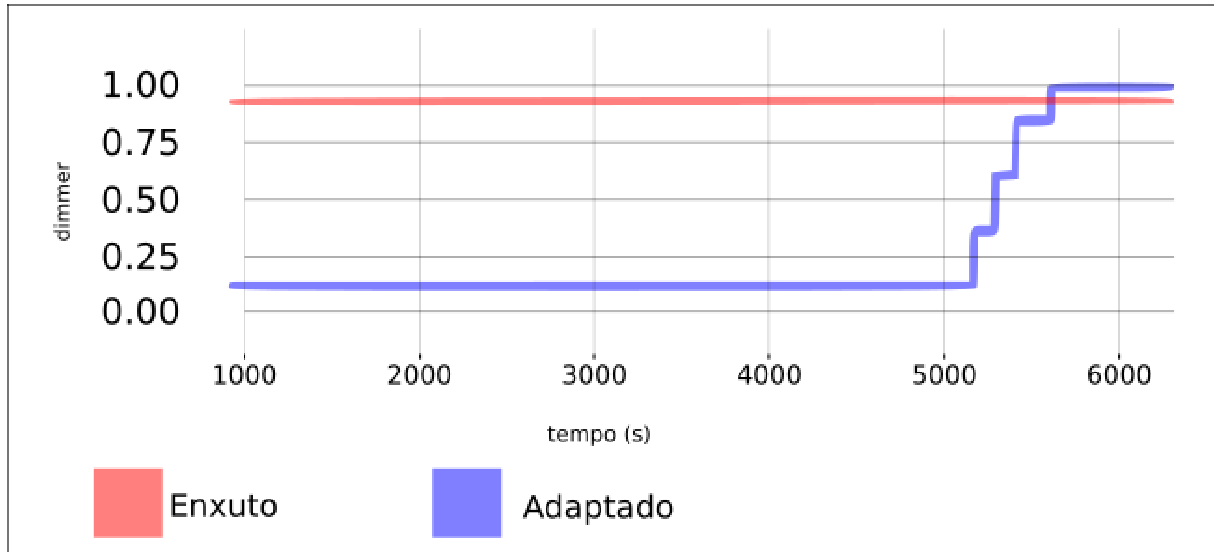
Fonte: O Autor.

5.9.3 *Potência máxima dos servidores*

O gráfico presente na figura 29 mostra a comparação do nível máximo de potência nas duas execuções, é possível ver também nesse gráfico que ao longo do tempo de execução a execução com a camada adaptativa altera esse nível de potência. a potência inicial do sistema é 0.9, porém igualmente ao que é mostrado no gráfico anterior, nos instantes iniciais da execução

essa potência é diminuída e é mostrado no gráfico somente a partir de um dado instante.

Figura 29 – Gráfico comparação da potência dos servidores.

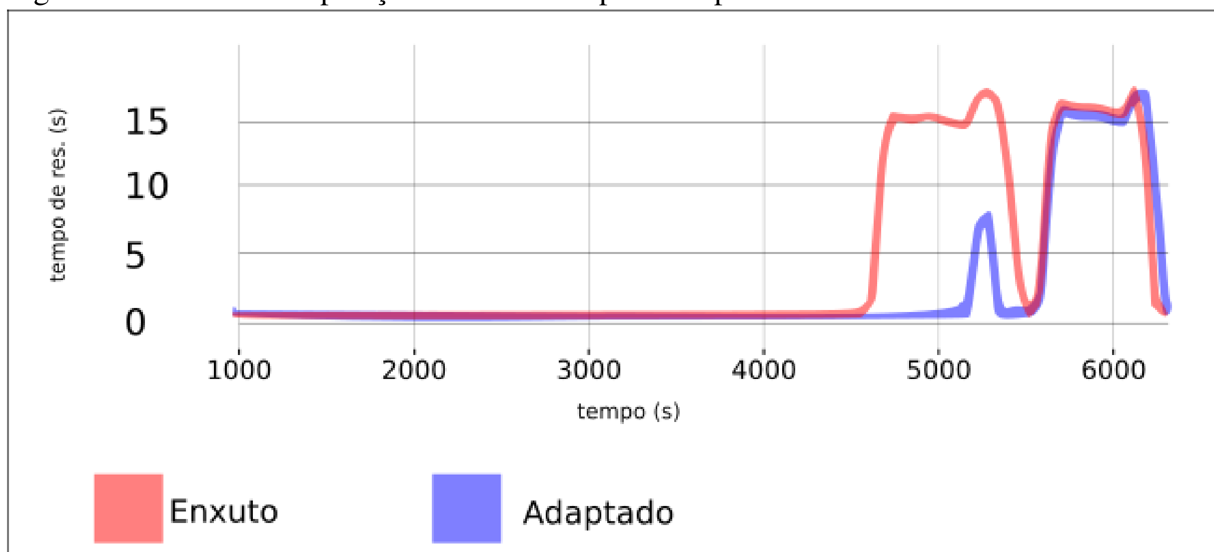


Fonte: O Autor.

5.9.4 *Tempo médio de resposta.*

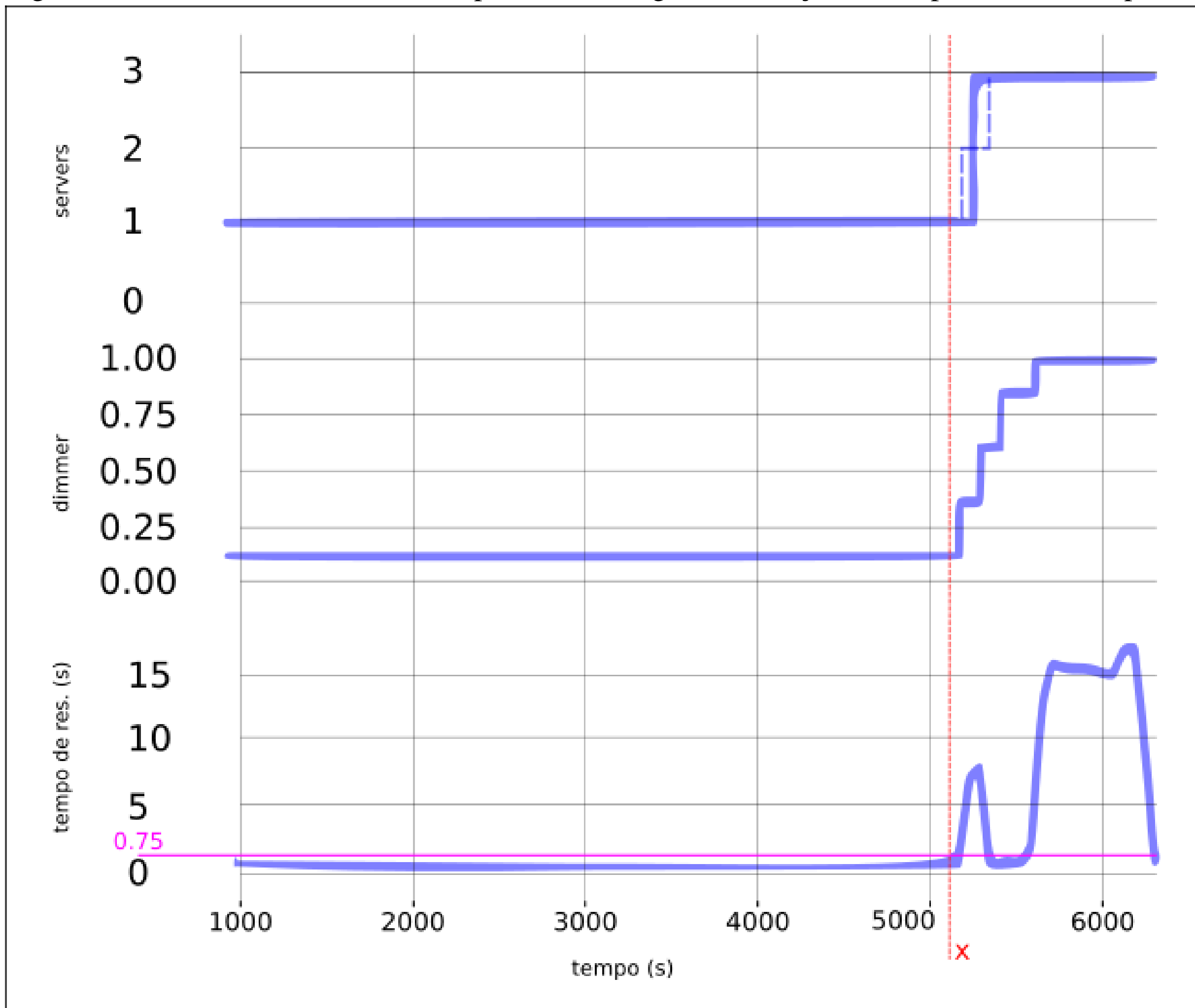
O gráfico de tempo médio de resposta, presente na figura 30 demonstra como a alteração de servidores ativos e seus níveis de potência (dimmer) podem alterar o comportamento geral do sistema. Logo a seguir na figura 31 é ainda possível ver que o gatilho do sintoma average_response_time_limit_upper em tempo x é o responsável por disparar as estratégias AddServerStrategy e DimmerUpStrategy.

Figura 30 – Gráfico comparação média de tempo de resposta.



Fonte: O Autor.

Figura 31 – Gráfico demonstrando disparo de estratégias em relação ao tempo médio de resposta.



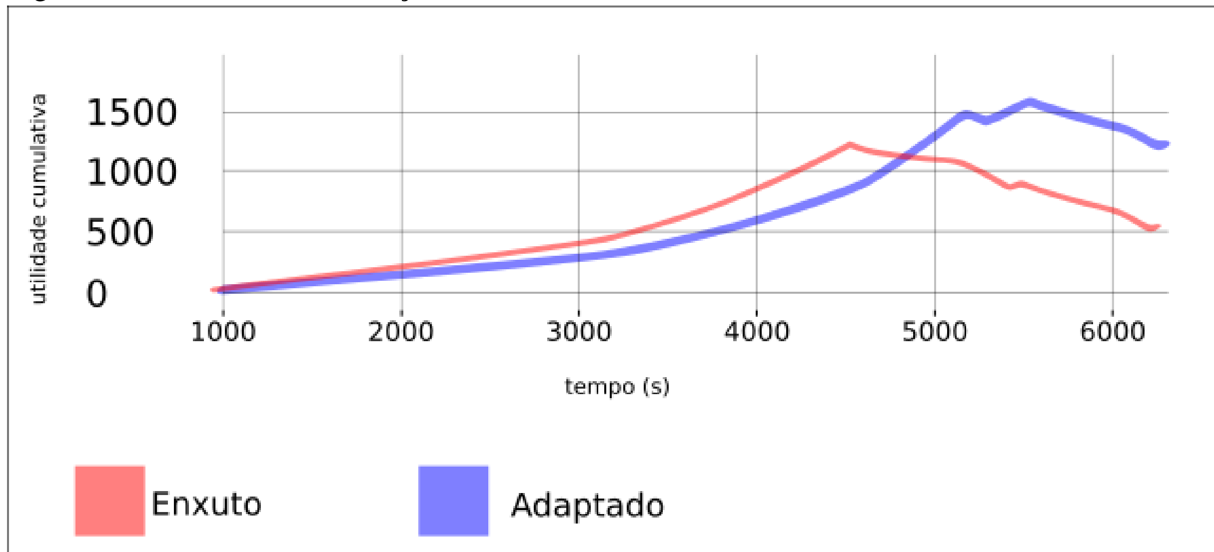
Fonte: O Autor.

5.9.5 Utilidade cumulativa.

O último gráfico, mostrado na figura 32, segundo (MORENO *et al.*, 2018) "é usando a função de utilidade usada em um estudo comparativo de auto-previsão". Nesse estudo essa função para comparar as abordagens CobRA e PLA, ambas baseadas em controle preditivo de modelo (MPC) (MORENO *et al.*, 2017).

É possível ver através da geração do SWIM que ambas as abordagens geram linhas diferentes ao longo do tempo de execução.

Figura 32 – Gráfico demonstração unidade cumulativa.



Fonte: O Autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Considerações gerais

Os sistemas de informação estão cada vez ficando maiores e complexos, precisando de monitoramento de seu estado constantemente, ferramentas com o propósito de alterá-lo para um nível desejável vem surgindo cada vez mais.

Durante o decorrer do documento, foram apresentadas várias ferramentas e tecnologias com o propósito de facilitar essa tarefa, cada uma em um nicho específico da área de tecnologia, principalmente quando se fala de sistemas distribuídos e contêiner's de aplicação.

O estudo surgiu após uma pesquisa sobre os sistemas auto-adaptativos e verificou-se que há uma falta de uma abordagem genérica, onde as camadas de comunicação com o sistema, de monitoramento e execução principalmente fossem extensíveis, podendo se acoplar a qualquer sistema que se deseja fazer o processo de adaptação.

Como mostrado no capítulo 4 o framework é totalmente configurável, sendo necessário somente implementar as classes de monitoramento e execução para fazer a comunicação e implementar as estratégias, sintomas e ações, fazendo toda sua parte lógica de adaptação funcionar.

Os componentes monitorador e executor trabalham com valores genéricos, fazendo sensores de qualquer tipo de comunicação, ex: HTTP, canais de comunicação, etc. independente do formato que essa comunicação forneça, basta fazer uma conversão para essa estrutura e executá-la no sistema alvo.

Observando as figuras do capítulo 5, exceto a figura 26 que é a entrada de dados de requisição, portanto se tornando igual, há diferença notável entre as linhas de tempo no gráfico, portanto é possível afirmar que o sistema alvo teve seu comportamento alterado devido seu acoplamento ao framework, validando os objetivos pretendidos.

6.2 Relevância do estudo

Como já dito na seção anterior, com a técnica aplicada no estudo é possível acessar e configurar um sistema através da implementação de classes estendendo métodos abstratos. Com a utilização desse framework já preparado, o esforço gasto com o desenvolvimento de um sistema poderia ser gasto com a implementação de outros módulos, diminuindo a complexidade

de seu desenvolvimento.

Com a utilização do SWIM como sistema alvo, foi possível usar a camada para alterar o seu comportamento, colhendo seus resultados com as ferramentas disponibilizadas pelo próprio simulador foi possível verificar a alteração.

6.3 Trabalhos futuros

Em seções anteriores foi descrito a implementação da camada, seus componentes, os testes realizados, bem como a relevância de tal estudo e seus resultados expressivos.

No entanto, há ainda espaços que podem ser estudados e pesquisados e que pretenderemos futuramente, como, por exemplo, a implementação de um módulo de inteligência artificial, aprendizado de máquina, ou qualquer outro tipo que possa ser utilizado para criar estratégias, relacionamentos entre sintomas e essas estratégias, assim, conseguindo fazer a melhoria de sistemas. Esse componente poderia ser utilizado no módulo analisador e planejador, fazendo com que o framework consiga pensar e planejar estratégias.

REFERÊNCIAS

ADERALDO, C. Kubow: An architecture-based self-adaptation service for cloud native applications. 03 2019.

ANGELOPOULOS, K.; PAPADOPOULOS, A.; SOUZA, V. S.; MYLOPOULOS, J. Engineering self-adaptive software systems: From requirements to model predictive control. **ACM Transactions on Autonomous and Adaptive Systems**, v. 13, p. 1–27, 04 2018.

BARBOSA, D. M.; Lima, R. G. D. M.; Maia, P. H. M.; Costa, E. Lotus@runtime: A tool for runtime monitoring and verification of self-adaptive systems. In: **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.: s.n.], 2017. p. 24–30.

BRABERMAN, V.; D'IPPOLITO, N.; KRAMER, J.; SYKES, D.; UCHITEL, S. Morph: A reference architecture for configuration and behaviour self-adaptation. In: **Proceedings of the 1st International Workshop on Control Theory for Software Engineering**. New York, NY, USA: ACM, 2015. (CTSE 2015), p. 9–16. ISBN 978-1-4503-3814-1. Disponível em: <http://doi.acm.org/10.1145/2804337.2804339>.

DOCKER. **Modernize your applications, accelerate innovation**. 2019. <https://www.docker.com/>. Accessed: 06/11/2019.

FRANZINI, F. **O que são microservices ?** 2017. <https://imasters.com.br/desenvolvimento/o-que-sao-microservices>. Accessed: 06/11/2019.

HELM RALPH JOHNSON, J. V. R. **Padrões de projeto, Soluções reutilizáveis de software orientado a objetos.** [S.l.: s.n.], 2008.

HUESBCHER, J. A. M. M. C. A survey of autonomic computing — degrees,models and applications. 2008.

JAMSHIDI, P.; Pahl, C.; Mendonça, N. C.; Lewis, J.; Tilkov, S. Microservices: The journey so far and challenges ahead. **IEEE Software**, v. 35, n. 3, p. 24–35, May 2018.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 1, p. 41–50, jan. 2003. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2003.1160055>.

KUBERNETES. **Orquestração de contêiner pronto para produção**. 2019. <https://kubernetes.io/pt>. Accessed: 29/09/2019.

MAIA, P.; VIEIRA, L.; CHAGAS, M.; YU, Y.; ZISMAN, A.; NUSEIBEH, B. Cautious adaptation of defiant components. 10 2019.

MAPR. **WHAT IS KUBERNETES?** 2019. <https://mapr.com/products/kubernetes/>. Accessed: 07/11/2019.

MORENO, G. A.; PAPADOPOULOS, A. V.; ANGELOPOULOS, K.; CÁMARA, J.; SCHMERL, B. Comparing model-based predictive approaches to self-adaptation: Cobra and pla. In: **2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**. [S.l.: s.n.], 2017. p. 42–53.

MORENO, G. A.; SCHMERL, B.; GARLAN, D. Swim: An exemplar for evaluation and comparison of self-adaptation approaches for web applications. In: **Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems**. New York, NY, USA: ACM, 2018. (SEAMS '18), p. 137–143. ISBN 978-1-4503-5715-9. Disponível em: <http://doi.acm.org/10.1145/3194133.3194163>.

NAMIOT, D.; SNEPPE, M. sneps. On micro-services architecture. **Interenational Journal of Open Information Technologies**, v. 2, p. 24–27, 09 2014.

SILVA, W. F. da. **Aprendendo Docker, do básica à orquestração de contêineres**. [S.l.: s.n.], 2016.

SINGLETON, A. The economics of microservices. **IEEE Cloud Computing**, v. 3, n. 5, p. 16–20, Sep. 2016.

TANENBAUM, M. V. S. A. S. **Distribuited Systems**. 3. ed. [S.l.: s.n.], 2017.

WEYNS, D. Engineering self-adaptive software systems – an organized tour. In: . [S.l.: s.n.], 2018. p. 1–2.

ZEROMQ. **An open-source universal messaging library**. 2020. <https://zeromq.org/>. Acessed: 17/12/2020.