UNIVERSIDADE DO ESTADO DE SANTA CATARINA
CURSO DE SISTEMAS DE INFORMACAO

LEONARDO AUGUSTO METZGER

OPTVM: UM SERVIÇO DE SUPORTE PARA MIGRAÇÃO DE VMS

TRABALHO DE CONCLUSÃO DE CURSO

SÃO BENTO DO SUL
2019

LEONARDO AUGUSTO METZGER

OPTVM: UM SERVIÇO DE SUPORTE PARA MIGRAÇÃO DE VMS

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informacao da Universidade do Estado de Santa Catarina, como requisito parcial para a obtenção do título de Bacharel.

Orientador: Mário Ezequiel Augusto
Universidade do Estado de Santa Catarina

SÃO BENTO DO SUL
2019

Aos meus professores, pais, irmão e amigos que me ajudaram e me motivaram durante o desenvolvimeto.

AGRADECIMENTOS

"A ciência é o que nós compreendemos suficientemente bem para explicar a um computador. A arte é tudo mais.- *Donald Knuth*

RESUMO

METZGER, Leonardo. OptVM: Um serviço de suporte para migração de VMs. 2019. 22 f. Trabalho de Conclusão de Curso – Curso de Sistemas de Informacao, Universidade do Estado de Santa Catarina. São Bento do Sul, 2019.

@TODO
Palavras-chave: VM. Optimization. Rest.

ABSTRACT

METZGER, Leonardo. Title in English. 2019. 22 f. Trabalho de Conclusão de Curso – Curso de Sistemas de Informacao, Universidade do Estado de Santa Catarina. São Bento do Sul, 2019.

@TODO
Keywords: VM. Optimization. Rest.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AE	Algoritmos Evolucionários
GA	Algoritmos Genéticos
MOO	Otimização multiobjetivo

LISTA DE SÍMBOLOS

Γ	Letra grega Gama
λ	Comprimento de onda
$\in$	Pertence

LISTA DE ALGORITMOS

1 INTRODUÇÃO

Com a evolução da computação em nuvem, surgiram necessidades cada vez maiores de utilizar ao máximo o poder dos computadores sem sobrecarregar os mesmos, e os componentes que o fazem funcionar, como o uso de energia. Estas necessidades surgem para atender requisitos de diminuição de custos, aumento de desempenho, diminuição no consumo de energia, entre outros objetivos que fazem com que usuários de computação em nuvem e empresas que usam este tipo de serviço obtenham vantagem no uso dela.

Para isso, é muito comum que para otimizar o uso dos computadores de um ambiente em nuvem, os provedores utilizem o mecanismo de virtualização. Hoje, os *datacenters* são compostos por máquinas físicas(PMs) e máquinas virtuais(VMs), sendo que, cada PM normalmente possui pelo menos uma ou mais VMs. Essa utilização das VMs permite que seja construído um ambiente flexível em relação a organização e quantidade de VM por Hosts em cada *datacenter*.

Em um cenário que o ambiente em que temos a possibilidade de utilizar as PMs como host de múltiplas VMs, é possível que as VMs da nuvem sejam organizadas de diferentes maneiras em relação as PMs. A decisão de organizar a núvem de uma maneira ou de outra, envolvem os objetivos de quem está gerenciando a núvem.

Os objetivos podem ser os mais variados. Por exemplo, uma empresa que use o serviço da nuvem pode querer ter um alto desempenho, assim como pode querer ter o menor custo possível. Por esse motivo, existem pesquisas que buscam maneiras de melhorar a alocação e realocação de VMs, e uma das formas de resolver este tipo problema, é utilizar a otimização multiobjetivo (MOO).

A migração de uma VM envolve algumas etapas, como, a descoberta da necessidade de migração, a escolha de uma VM a ser migrada e a escolha de um host de destino para essa VM. A etapa em que este trabalho está interessado é a escolha de um host de destino para a VM. Considerando que uma migração seja considerada cara do ponto de vista computacional, o momento da migração deve ser bem escolhido para evitar que a migração da VM feita não gere prejuízos ou problemas maiores do que a própria sobrecarga do host. Assim como o momento da migração é importante, a escolha de um destino também é, pois o host selecionado tem que melhorar a maneira em que os hosts estão organizados naquele determinado momento, para que não haja novas migrações por conta da migração inicial.

Este trabalho tem papel de servir como apoio para a migração de VMs em ambientes de computação em nuvem. Isto é feito através de uma abordagem em que um gerenciador de núvem, que precise migrar uma VM, possa utilizar um serviço que selecionará as melhores opções de host para fazer a migração de uma VM. O serviço possui uma abordagem que utiliza algoritmos que fazem uma seleção dos melhores hosts baseando-se nos objetivos do consumidor do serviço. Contudo, o serviço é uma solução caixa preta, esta característica traz uma grande vantagem, o usuário não precisa conhecer nada sobre os algoritmos utilizados, precisa apenas utilizar a interface que é definida pelo serviço. A interface do serviço é construida em cima de *webservices*, utilizando padrões bem conhecidos para facilitar a integração dos gerenciadores das núvens computacionais.

Este trabalho apresenta uma aplicação prática da construção de um *webservice* utilizando padrões bem conhecidos na indústria. Além do *webservice*, o trabalho também apresenta uma aplicação de algoritmos de otimização multiobjetivo para um problema que existe hoje. Também são feitos testes e gerados métricas para a avaliação dos resultados.

1.1 ORGANIZAÇÃO DO TRABALHO

O trabalho está organizado em seis capítulos. O primeiro, destina-se a dar uma introdução do problema e forma de resolvê-lo Já o segundo capítulo, irá dar o embasamento teórico necessário para o entendimento dos demais capítulos. O terceiro destina-se a metodologia utilizada para o desenvolvimento do trabalho. No quarto serão apresentados aspectos da implementação da solução para o problema encontrado. Os resultados obtidos com a implementação apresentada no capítulo quarto serão feitas no capítulo cinco. E por último as conclusões que foram obtidas na realização do trabalho.

2 REVISÃO DE LITERATURA

Neste capítulo, são apresentados alguns conceitos e termos utilizados no decorrer do trabalho. Os conceitos apresentados tem relação com a construção do sistema. São apresentadas as técnicas, arquiteturas e algoritmos disponíveis para solução do problema, assim como as utilizadas no desenvolvimento da solução.

As primeiras seções do capítulo abordam os padrões e protocolos disponíveis na literatura que são utilizados na construção de *webservices*. Esses padrões e protocolos são conhecidos por facilitar e flexibilizar a integração de sistemas. Entre eles destacam-se o *Service Oriented Architecture* (SOA), que é uma arquitetura que trata os serviços como componentes e visa utilizar esses componentes para resolver problemas de negócios complexos através de composição de serviços. *Representational State Transfer* (REST), que é um estilo arquitetural que pode ser utilizado, ou não, para a criação de serviços SOA. O *Simple Object Access Protocol* (SOAP), que é um protocolo que muitas vezes é comparado ao REST por também servir para a criação de serviços SOA.

Além disso, nas seções posteriores, o capítulo também apresenta algoritmos que ajudam a resolver o problema da seleção de host para migração de VM através de otimização multiobjetivo (MOO). Existem vários algoritmos que podem ser utilizados para a resolução desse tipo de problema. Neste trabalho, são abordados os algoritmos genéticos (GAs), que fazem parte de um grupo maior de algoritmos, chamado algoritmos evolucionários (EAs). O conceito de EA também será abordado neste capítulo.

Nas primeiras quatro seções, as ferramentas apresentadas estão relacionadas em como é feita a comunicação e disponibilização do serviço. E as outras seções estão relacionados ao funcionamento interno, em como solucionar a seleção dos melhores hosts.

2.1 SERVICE ORIENTED ARCHITECTURE (SOA)

O desenvolvimento de software para um ambiente corporativo é uma tarefa complexa. Conforme Brown [Brown Simon Johnston 2002], no decorrer dos anos, a comunidade de desenvolvimento de software se dedicou em desenvolver novas abordagens, processos e ferramentas para a construção de softwares de grande escala.

Brown considera que uma maneira de descrever um sistema de software é como sendo um composto de uma coleção de serviços. Cada serviço, provém um conjunto de funcionalidades bem definidas. As funcionalidades do serviço sendo bem definidas tornam possível a construção de serviços compostos, ou seja, uma funcionalidade que faça a utilização de outras funcionalidades ou serviços. Esta modularização e coordenação de serviços e funcionalidades caracteriza uma Service Oriented Architecture (SOA).

Segundo [Valipour Bavar Amirzafari e Daneshpour 2009], SOA pode ser definido como um design de software utilizado para conectar negócios e recursos computacionais sob demanda, e isso possibilita os usuários do serviço (podendo ser

outros serviços ou usuários finais) a alcançarem seus objetivos.

Existem diversas maneiras de implementar uma aplicação baseada em SOA, o importante é que sua interface seja bem definida com as operações que podem ser realizadas. Uma das grandes vantagens do SOA, é a facilidade que ele provém na integração de sistemas. Segundo [Valipour Bavar Amirzafari e Daneshpour 2009], com as operações bem definidas e disponíveis, o consumidor do SOA, pode se preocupar somente com o que determinado serviço faz e não como é implementado.

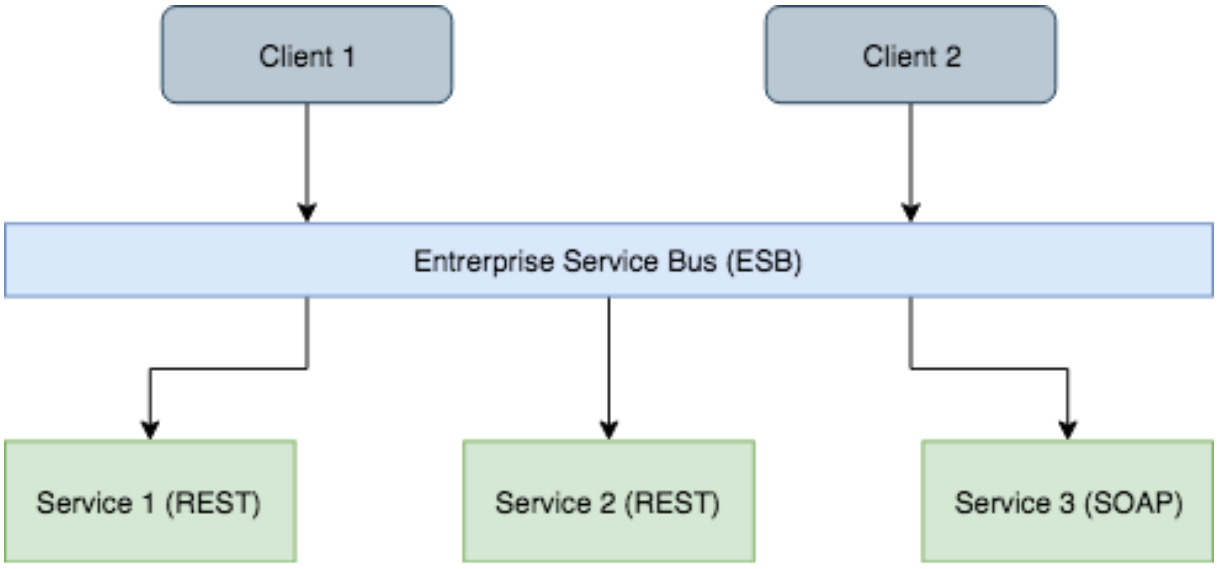
As principais características de um software feito utilizando SOA são que ele é auto contido e modular, interoperável, fracamente acoplado, passível de composição e possui transparência de localização. Como SOA não limita a estratégia utilizada para o desenvolvimento do mesmo, pode-se utilizar qualquer técnica para implementá-lo. No ambiente corporativo, os serviços comumente são implementados utilizando web services SOAP, REST ou chamadas RPCs.

2.1.1 Enterprise Service Bus (ESB)

Para obter essa flexibilidade na construção dos serviços, o é comum o SOA utilizar uma camada que recebe a chamada dos serviços e encaminha para o serviço correto. Essa camada, além da responsabilidade enfileirar as mensagens enviadas aos serviços, também faz a tradução da comunicação, caso um serviço trabalhe somente com SOAP e outro somente com REST.

A arquitetura utilizando SOA fica semelhante a da imagem.

Figura 1 – SOA com ESB



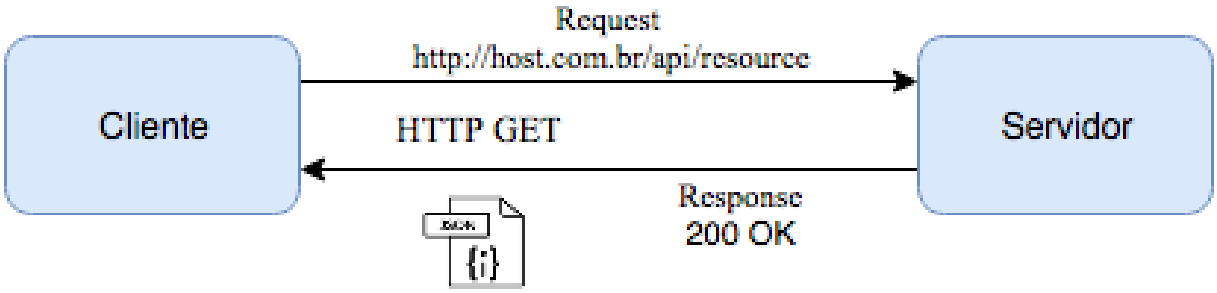
2.2 REPRESENTATIONAL STATE TRANSFER (REST)

REST foi formalizado por Fielding [Fielding 2000] em sua tese de doutorado, onde ele tem por objetivo apresentar uma arquitetura para criação de sistemas network-based. Na tese, REST é definido como um estilo arquitetural. Fielding também

define seis restrições que devem ser respeitadas na criação de um software que é implementado utilizando este estilo arquitetural.

A comunicação no REST, é feita via *Hypertext Trasfer Protocol* (HTTP). Desta meneira a comunicação cliente-servidor é feita utilizando as funcionalidades que o protocolo suporta, como: a utilização de cabeçalhos, *query-strings*, *Universal Resource Identifiers*(URIs), entre outros recursos do protocolo. Como foco deste trabalho é o estilo arquitetural REST, o mesmo não contempla detalhes sobre o funcionamento deste protocolo, apenas a utilização dele.

Figura 2 – REST sobre HTTP simples



Neste capítulo será apresentado as seis restrições que são definidas por Fielding que um sistema deve atender quando REST é utilizado e a maneira de utilização dos verbos HTTP no padrão arquitetural.

As restrições, podem ser atendidas em sua totalidade ou não, isso depede do nível de maturidade da API. Uma API que não atende todas as restrições não está deixando de utilizar REST, apenas o utiliza com um nível de maturidade menor. As seis restrições são:

1. Interface uniforme
2. *Stateless*
3. Cacheável
4. Cliente-servidor
5. Sistema em camadas
6. Código por demanda (Opcional)

2.2.1 Interface Uniforme

Fielding [Fielding 2000] destaca que uma das características centrais do REST, e o que difere ele de outros estilos arquiteturais, é a utilização de uma interface uniforme entre os componentes. Esta interface uniforme define a forma com que o cliente e o servidor se comunicam. Segundo [**?**] isto desacopla e simplifica a arquitetura.

Segundo [**?**] além disso, essa interface, é uniforme quando segue as seguintes características:

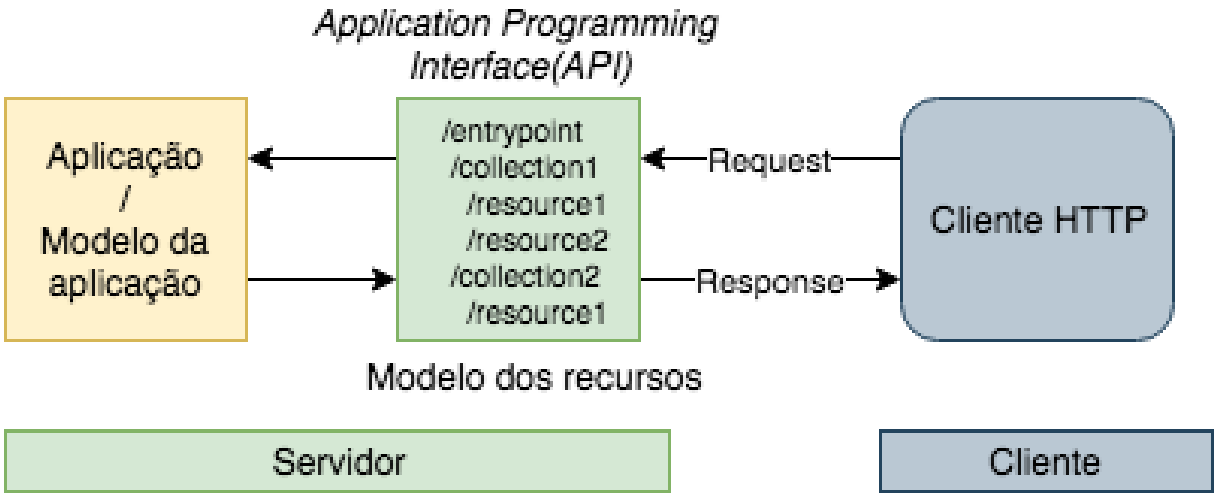
1. Baseada em recurso
2. Manipula os recursos através de representações
3. Possui mensagens autodescritivas
4. *Hypermedia as the Engine of Application State* (HATEOAS)

Ser **baseada em recursos**, significa que a interface deve separar seus recursos através de URIs. Cada recurso possui alguma URI que serve como *endpoint* para interação com o usuário. Essa URI é utilizada para fazer a **manipulação dos recursos utilizando representações**. As representações de um recursos podem ter diversos formatos, como por exemplo *JavaScript Object Notation* (JSON) ou *eXtensible Markup Language*(XML).

Mensagens autodescritivas significa que na requisição feita pelo cliente para o servidor, devem ser incluidas informações sobre o que deve ser feito com o recurso que se está utilizando. Por exemplo, deve ser incluido o formato que está sendo feita a comunicação (XML, JSON), o que está querendo modificar naquele recurso (atualizar, deletar, etc.). Além dessas informações, que são enviadas pelo cliente, as mensagens de resposta também devem ser autodescritivas, contendo informações sobre a resposta, por exemplo, se o recurso pode ser cacheado e por quanto tempo.

Segundo [**?**] **HATEOAS** significa clientes entregar estado via conteúdo no corpo da mensagem, *query-strings*, cabeçalhos e URIs. E o servidor retornar estados através de *status-codes*, conteúdo no corpo da mensagem e meta-informações através de cabeçalhos. Seguindo essas regras, a comunicação é considerada comunicação via **hyperteto**. Além disso, também significa enviar a relação entre recursos quando necessário. Isso pode ser feito através de links contidos nos corpos da mensagem ou cabeçalhos da requisição.

Figura 3 – Recursos REST



2.2.2 Cliente-servidor

REST é implementado utilizando o modelo de comunicação cliente-servidor. Isso ajuda com a separação de responsabilidades, e permite que uma portabilidade de clientes do serviço implementado. O REST permite uma reutilização do serviço. Do ponto de vista de arquitetura de software, isto é muito importante, pois permite que componentes fiquem bem modularizados.

2.2.3 Stateless

No REST, é necessário que a aplicação seja *Stateless*, ou seja, as requisições devem ser autocontidas. Isso significa que as requisições de usuários devem conter todas as informações necesárias para o servidor entender e processar a requisição. Não deve-se assumir que o cliente da API utilize-a de uma forma específica para a requisição funcionar.

2.2.4 Cacheável

Essa é uma característica que faz com que aumente o desempenho da aplicação. Deve ser possível fazer cache de informações. Isso faz com que gere menos comunicação entre cliente e servidor. O *cache* pode ser feito de maneira implicita (o cliente define quando cachear) e de maneira explícita (o servidor indica que a requisição pode ser cacheada).

2.2.5 Sistemas em camadas

Deve ser possível adicionar camadas(*middlewares*) na utilização do REST. Essas camadas podem ter diversas finalidades, como, por exemplo, aumento de performance através de *load-balance*, aumento de segurança (autenticação), entre outros.

2.2.6 Código por demanda (Opcional)

Esta restrição é opcional e pouco utilizada. Segundo [**?**] é uma maneira de extender a funcionalidade através de envio de código para ser executado pelo cliente. Esta restrição deve ser aplicada apenas quando não viole outras restrições.

2.2.7 Verbos HTTP

Além dos itens citados nas seções anteriores, outra parte importante no REST, para a uniformidade da interface, são os **verbos http**. Os verbos HTTP representam a ação que o usuário da API está querendo tomar através da requisição enviada. Isso é muito importante, pois da um contexto para cada tipo de requisição da API.

Tabela 1 – Tabela de verbos HTTP

Verbo	Descrição
GET	busca/le informações na API
POST	cria recursos na API
PUT	atualiza algum recurso
DELETE	deleta algum recurso ou alguns recursos

2.3 SIMPLE OBJECT ACCESS PROTOCOL (SOAP)

SOAP é um protocolo de comunicação baseado em XML(eXtension Markup Language) que foi criado no final dos anos 90. Seu objetivo é fazer a comunicação en-

tre o cliente e o servidor através de informações passadas através de um documento XML. O protocolo utiliza um *schema* XML, que é uma maneira de descrever e validar o formato os dados das requisições e respostas. Esse *schema* é utilizado pelo cliente e pelo servidor para saber como interpretar a resposta, no caso de recebimento de mensagem, e formatar a requisição, no caso de envio. Esse *schema* é chamado de WSDL (Web Services Description Language).

O objetivo do SOAP, é expor regras de negócio de aplicação através de serviços. Por esse motivo, o SOAP é uma opção comumente utilizada na construção de aplicações SOA. Outra característica do SOAP, é que ele não precisa ser implementado sobre um protocolo de transporte específico, e na maioria das vezes é utilizado HTTP, porém é possível implementá-lo utilizando outros protocolos. Além do SOAP permitir utilizar outro protocolo, ele não restringe a implementação em alguma linguagem de programação específica, ou seja, é possível implementá-lo em qualquer linguagem de programação.

O SOAP é comparado com ao REST, pois os dois podem ser utilizados para um objetivo semelhante, porém, os dois tem um foco diferente, onde o SOAP tem como objetivo expor regras de negócio como serviço, e o REST visa representar um determinado estado e manipulá-lo através de operações bem definidas.

Segundo [**?**] como o SOAP é um grande XML, no contexto dos webservices, ele começou a parder espaço para o REST, o qual é mais simples e permite enviar informações em formatos mais leves, como o JSON por exemplo. Isso é uma grande vantagem, principalmente por ser um problema que tem o potencial de ter quantidade de dados trafegando em larga escala.

2.4 OTIMIZAÇÃO MULTI-OBJETIVO (MOO)

Problemas de otimização buscam, de alguma maneira, um bom resultado para a execução de alguma tarefa. Segundo Veloso [Matos 2017] para os problemas de otimização, existem de maneira geral, dois tipos de problemas, os mono-objetivos e os multi-objetivos. Os mono-objetivos buscam otimizar uma solução baseando-se em um único objetivo, por consequência, problemas mono-objetivo resultam em um único resultado, que pode ser considerada a solução ótima para o problema. Já os multi-objetivo, buscam atender vários fatores, e isso torna a solução ótima mais difícil de ser encontrada.

Conforme Ticona [Ticona 2003] um problema de otimização multi-objetivo, é representado por um conjunto de funções objetivo que devem ser otimizadas.

Segundo [**?**] num problema multi-objetivo, considera-se um conjunto de variáveis, que são chamadas de solução, que busca otimizar mais de uma função objetivo satisfazendo algumas restrições.

No universo dos algoritmos de otimização multi-objetivo, existem diversos tipos, e nesse trabalho será utilizado algoritmos genéticos, os quais fazem parte da classe de algoritmos evolucionários. Os mesmos serão descritos a seguir nesse capítulo.

2.5 ALGORITMOS EVOLUCIONÁRIOS (EA)

Segundo [Ticona 2003], algoritmos evolucionários (AE) tem sido muito utilizados para problemas de otimização. Um dos motivos do uso deles, é por causa da possibilidade de resolver problemas que envolvam múltiplos objetivos. A abordagem utilizada neste tipo de algoritmo é baseada na evolução humana. O processo é baseado seleção natural de Darwin, da mesma maneira que acontece com a seleção das espécies. O algoritmo reproduz artificialmente o processo de seleção natural para encontrar os mais aptos a resolver determinado problema. O objetivo desses algoritmos é encontrar aproximações da solução perfeita para problemas difíceis.

Dentro da categoria dos AEs para otimização baseada em multiplos objetivos, existem diferentes modelos.

O modelo de utilizado neste trabalho é a de algoritmos genéticos (AG). Esta é uma classe de algoritmos muito utilizada em otimizações multi-objetivo.

2.6 ALGORITMOS GENÉTICOS (GA)

@TODO

2.7 TRABALHOS RELACIONADOS

Neste capítulo será apresentado o problema em que o OptVM se propõem resolver. Isso será feito através da apresentação de alguns fatos relacionados a nuvens computacionais e como elas costumam ser utilizadas nos dias de hoje.

2.7.1 Migração de máquinas virtuais

As nuvens computacionais são utilizadas pela maioria das empresas de software da atualidade. Por esse motivo, as maiores empresas do setor investem muito neste segmento, oferecendo vários tipos de serviços diferenciados para seus consumidores. Estas empresas concorrem em alguns aspectos, como: velocidade, preço, disponibilidade e etc. Para aumentar sua competitividade nesses aspectos, muitas vezes é utilizada a virtualização.

Com a virtualização é possível alocar partes de um recurso físico para diferentes consumidores, fazendo com que um recurso físico se torne melhor utilizado. Isso deixa a alocação de recursos muito mais flexível, e torna possível obter uma elasticidade nos serviços oferecidos.

Uma dos objetivos de utilizar a virtualização é obter uma elasticidade dos recursos oferecidos. Ou seja, é possível aumentar sua capacidade de processamento, armazenamento mesmo depois que o já foi alocada uma VM para o usuário. Isso permite que um consumidor do serviço possa escolher o quanto precisa para executar as tarefas que deseja, assim como o provedor também consegue otimizar o uso de seus recursos

Essa realocação de uma VM pode ser feita a nível de nuvem, datacenter(DC) ou host. Quando a realocação é feita em nível de DC ou nuvem, é muito provável que seja necessário migrar uma VM do local em que ela se encontra. Caso seja necessário uma migração de uma máquina, existem alguns pontos que devem ser avaliados.

Três momentos podem ser considerados os pontos principais a serem avaliados para uma migração, são eles:

1. A descoberta de uma necessidade de migração
2. Qual máquina virtual deve ser mgirada
3. Para onde deve ocorrer a migração

Estas otimizações e migrações das VMs são necessárias em ambientes que envolvem uma infraestruturas grande, onde existem multiplos hosts, datacenters e nuvens. Por esse motivo, não é recomendado que um sistema ou serviço gerencie a infraestrutura inteira sozinho, pois sua escalabilidade poderia se tornar um gargalo. Isso faz com que sejam construidos diferentes serviços e aplicações que se integram e gerenciam a infraesturua.

2.7.2 Otimização na escolha de um host

Como citado anteriormente uma das partes essenciais na migração de uma VM é a escolha de um destino para ela. Para isso, é importante escolher um destino que aloque muito bem a VM e não seja necessário fazer uma outra migração logo em seguida.

3 METODOLOGIA

@TODO

4 OPTVM

O OptVM é um sistema que tem o propósito de dar suporte para a migração de VMs em um ambientes de núvens federadas, isso é feito através *webservices* utilizando o modelo cliente-servidor, mais específicamente o padrão arquitetural Representational State Transfer(REST).

O sistema possui dois principais componentes para atingir seu objetivo: um deles, faz uma filtragem de hosts aplicando restrições. E o outro faz uma otimização na escolha de um subconjunto de possíveis hosts para uma VM migrar.

As restrições que são aplicadas aos possíveis destinos (hosts), elas servem para desconsiderar os destinos que não são passiveis de migração por causa de regras de negócio. As restrições no OptVM são pré-definidas, ou seja, elas devem ser apenas escolhidas e parametrizadas pelo usuário da API, não é possível criar um tipo de restrição. Por exemplo, podem ser definidas regras do tipo: uma VM que está localizada em um país, não pode ser migrada para outro país, onde cada país da regra pode ser parametrizado. Esta regra, pode ser parametrizada com EUA e Israel, por exemplo. Além disso, as restrições são aplicadas antes da otimização, para o sistema desconsiderar hosts que não são realmente alvos de migração e limitar o espaço de busca da otimização.

O outro componente, o qual define um melhor subconjunto de hosts faz isso utilizando MOO. Essa otimização, possui um conjunto de funções objetivo pré-definidas. Esses objetivos também podem ser escolhidos e parametrizados pelo usuário da aplicação conforme sua necessidade. Os objetivos, assim como as restrições, são pré-definidos pelo OptVM e o usuário tem a opção de utilizá-los ou não. Os **objetivos da otimização**, são interesses do usuário, por exemplo, minimização do consumo de energia. No OptVM, os objetivos são traduzidos para funções objetivo, do problema de otimização. Ou seja, internamente são tratados como funções matemáticas que representam o objetivo da otimização.

Como a escolha e parametrização das restrições e funções objetivo da otimização são escolhidos pelo usuário da API, Foi criado um recurso que representa a configuração e parametrização das **restriçoes** e **objetivos de otimização**, e para este recurso foi dado o nome de **política**.

O OptVM busca ser uma solução caixa preta, onde, o usuário não necessita saber nada sobre o funcionamento interno, algoritmos utilizados, etc. Basta utilizar as *Application Programming Interfaces* (APIs) para fazer uso de suas funcionalidades. A intenção é que o usuário não precise entender sobre como as coisas são feitas internamente para obter as vantagens do uso do OptVM.

Neste capítulo, serão apresentados aspectos gerais em relação a implementação e uso do OptVM. No primeiro momento serão mostrados os componentes que integram o sistema. Depois disso, técnicas e ferramentas utilizadas, e no final a ideia do funcionamento do serviço e como ele é implementado.

4.1 COMPONENTES

O OptVM é dividido em dois principais componentes: O aplicador de constraints (*constraint applyier*) e o otimizador(*optmizer*). Os dois componentes estão relacionados, porém são representados e aplicados separadamente.

4.2 APLICADOR DE CONSTRAINTS (CONSTRAINT APPLYIER)

Como citado na introdução do capítulo, o OptVM utiliza restrições pré-definidddas. Para o entendimento de como funciona as o aplicador de restrições, é importante conhecer os tipos de restrições e como elas funcionam.

4.2.1 Tipos de restrições (constraints)

As restrições pré definidas, tem um *alias* para que o usuário do serviço faça a identificação no uso delas. Os tipos de restrições disponibilizadas pelo OptVM operam em 3 níveis, são eles: **cloud**, **datacenter** (DC) e **hosts**.

Cada restrição tem uma responsabilidade específica, uma regra de negócio que não deve ser infringida. As restrições disponibilizadas e seus respectivos níveis são as seguintes:

1. Cloud:
    a) @TODO
2. DC:
    a) *Conflito*: Pode conflitar uma localização, por exemplo, uma VM dos USA, não pode habitar em um host de ISRAEL;
    b) *Custo*: Os custos que irão gerar ultrapassam os parâmetros.
3. Host
    a) *Dependência*: Depende que o host tenha um sistema operacional(OS) ou hypervisor específico (baseado em seus parâmetros);
    b) *Coabitação*: Dependendo dos parâmetros da constraints, exige que o host seja vizinho (1 hop de distância).

4.2.2 Algoritmo

O *constraint applyier* tem a responsabilidade de remover os hosts que não atendem à uma ou mais restrições estabelecidas pelo usuário. Como as restrições operam nos 3 níveis, é importante que algoritmo tenha acesso a um contexto onde obtenha informações sobre as núvens, datacenters e hosts disponíveis.

O algoritmo que aplica as constraints, consiste em iterar sobre as constraints ordenadas da mais significativa (operam em nível de cloud), para a menos significativa (operam em nível de host), e remover os itens que não atendem as restrições, após cada iteração.

O pseudocódigo do algoritmo é representado da seguinte maneira:

A intenção de aplicar as restrições ordenadas das mais significativas, que são as entidades que eliminarão mais destinos de uma vez (*Clouds* e *Datacenters*), para as menos significativas (*Hosts*), para que diminua o número de iterações. Pois quando, por exemplo, um *datacenter* não atende uma restrição, nenhum host daquele

Algoritmo 1: Constraint Applyier

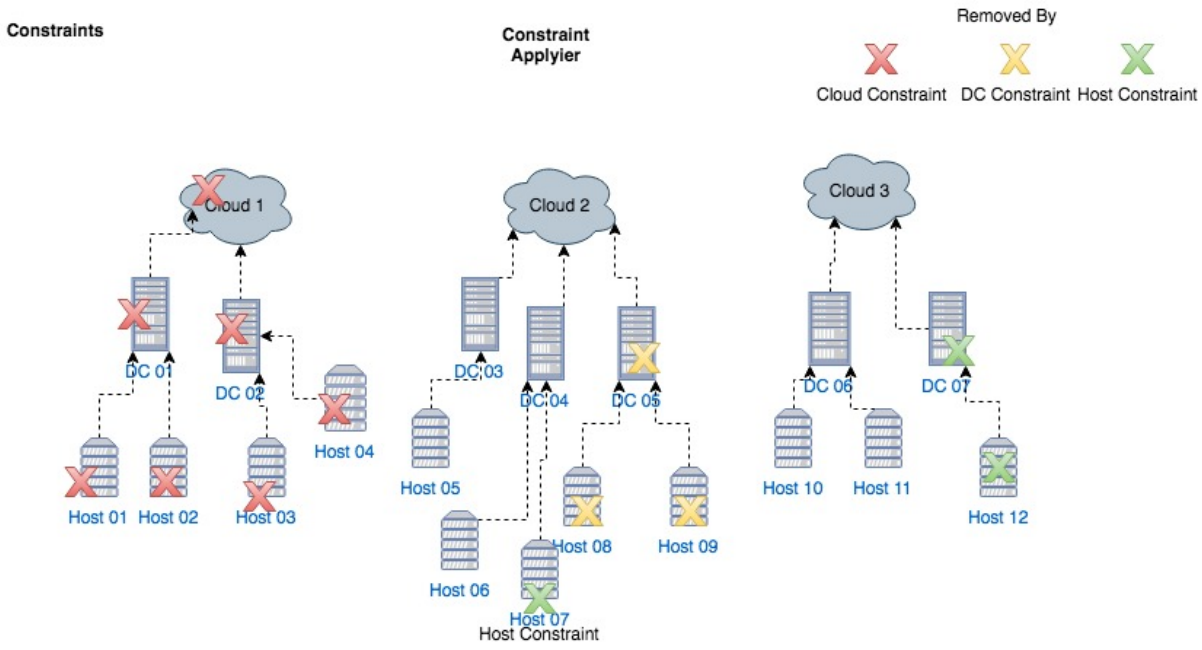
Input: as $constraints$ restrições, $context(Clouds, DCs, Hosts)$ contexto com todas as opções disponíveis
Output: $context'(Clouds, DCs, Hosts)$ somente com Clouds/DCs/Hosts que atendem as restrições

$context' \leftarrow context$
for $constraints \in constraints$ **do**
  if c *é do tipo CLOUD* **then**
    aplica a $constraints$ nas núvens do contexto
    $context' \leftarrow nuvensAtualizadas$
  end
  else if $constraints$ *é do tipo DC* **then**
    aplica a $constraints$ nas DCs do contexto
    $context' \leftarrow datacentersAtualizados$
  end
  else if $constraints$ *é do tipo HOST* **then**
    aplica a $constraints$ nas Hosts do contexto
    $context' \leftarrow hostsAtualizados$
  end
end

datacenter são elegíveis como um possível destino, então, as restrições em nível de host para os hosts desse determinado datacenter nem devem ser executadas.

Uma representação visual, de como o algoritmo funcionaria, pode ser vista na Figura 4. A ideia é que funcione semelhante a uma estrutura de árvore com três níveis, e que quando um nó pai não atende a uma restrição, consequentemente a sub-árvore também não atende. E a avaliação é feita top-down, começando pela raiz e indo em direção aos nós folhas.

Figura 4 – Exemplo de aplicação das constraints



A Figura 4 mostra como as constraints aplicadas em seus respectivos níveis, eliminariam a possíbilidade da VM migrar para determinado host. Na Figura 4, foram representados os seguintes casos:

1. A **Cloud 1** não atende à alguma restrição escolhida em nível de núvem, então nenhum de seus DCs e por consequência hosts dos DCs, serão alvos válidos;
2. O **Datacenter 05** não atende alguma restrição em nível de DC, então nenhum de seus hosts estarão disponíveis como alvo de migração;
3. E os **Host 07 e Host 12** não atendem alguma restrição em nível de host;
4. Um caso mais específico é quando o **Host 12** não atende a alguma restrição, logo o **DC 07** não terá nenhum host válido, então também deve ser desconsiderado.

4.3 OTIMIZADOR (OPTIMIZER)

Considerando que o cliente da API é uma federação de núvens computacionais. O *optimizer* é responsável por otimizar a seleção dos hosts disponíveis nas núvens da federação, para a VM que necessita migrar. O objetivo é alcançar um melhor subconjunto para a alocação da VM. Além disso, a seleção desse subconjunto é selecionado atendendo os objetivos da **política** selecionada para a otimização (conjunto de objetivos e restrições), escolhida pelo usuário.

A otimização feita no OptVM faz uso de GAs. Os GAs fazem iterações sobre uma população que é iniciada aleatoriamente no algoritmo. Para cada iteração, o algoritmo avalia os indivíduos ou soluções da população pela *fitness function*, que também é chamada de função objetivo. Após a avaliação dos indivíduos, é feita a seleção dos melhores avaliados e também é feito o crossover, gerando uma nova população, que é mais evoluida e mais adequada ao problema. O número de iterações feitas é escolhido pelo usuário do algoritmo. Quanto mais iterações houver, mais evoluida estará a população, porém, maior será o consumo computacional. Caso o número de iterações for muito alto, pode inviabilizar o uso do algoritmo por causa do tempo de resposta.

No OptVM, os objetivos são dinâmicos, ou seja, é possível que o usuário da API escolha por alguns objetivos e não por outros. Os objetivos disponíveis no OptVM são três, dos quais uma combinação de dois ou os três podem ser escolhidos, são eles:

1. Minimização do consumo de energia;
2. Minimização do tempo de instalação;
3. Minimização da sobrecarga da migração.

Como a otimização possui um conjunto de indivíduos, que formam uma população, cada indivíduo da população representa uma possível solução para o problema. Nos algoritmos de otimização o indivíduo possui uma representação (*encoding*), que pode ser feita de diversas maneiras, de maneira binária, por inteiros, entre outros tipos de representação.

Como uma representação representa uma solução para o problema, no OptVM, a representação escolhida foi uma representação binária, ou seja, uma solução para o problema é um array de booleanos onde cada booleano representa um host. O valor (*true* ou *false*), se o host está no subconjunto de soluções ou não estaria alocada.

$$Solution = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

No exemplo, o tamanho escolhido para o subconjunto é de 2 hosts, ou seja, a otimização busca achar o melhor subconjunto de 2 hosts dentre os N disponíveis. No exemplo, os 2 hosts escolhidos na solução, são os hosts 1 e 4 do conjunto de N hosts.

4.3.1 NSGAII

@TODO (Se for usado o NSGAII mesmo, explicar aqui como ele funciona)

4.4 COMUNICAÇÃO

Em termos gerais, uma API é uma interface de software que pode ser chamada e executada [Eizinger 2017].

Como o OptVM é um serviço que deve ser disponibilizado para uma arquitetura de cliente-servidor de maneira distribuída, haviam três possíveis maneiras de implementá-lo, que eram REST, SOAP e via chamadas RPC. Para o desenvolvimento do OptVM foi escolhida a implementação utilizando o modelo REST. A escolha desta opção se deu principalmente pelos seguintes motivos:

1. É um padrão arquitetural maduro;
2. É agnóstico em relação a liguangens de programação;
3. É flexivel em relação ao modelo de comunicação.

Como o padrão arquitetural REST é agnostico em relação ao formato utilizado para fazer a comunicação dos dados. O *econding* dos dados pode ser feito da maneira que for mais conveniente para o usuário. No caso do OptVM é possível fazer a comunicação tanto no formato XML como no formato JSON.

Isso é controlado pelo próprio cliente da aplicação. É controlado através do cabeçalho *Accept* da requisição HTTP.

4.5 REPRESENTAÇÃO DO SERVIÇO

O padrão REST, definido por Fielding [Fielding 2000], sugere que se deve criar uma interface para interação com o sistema. Essa interface é representada através de recursos, e a interação com esses recursos é feita através de requisições HTTP, as quais contém verbos, o corpo da mensagem, cabeçalhos, entre outras informações. Além disso, o padrão arquitetural também sugere que haja links para verificar outras informações e tomar ações sobre os recursos, chamado HATEOAS.

4.5.1 Recursos

O OptVM trabalha em cima de 2 recursos, chamados políticas(*policies*) e otimizações(*optimizations*). Ambos os recursos, conseguem trabalhar de maneira independente.

O recurso de políticas é responsável por gerenciar (criar, atualizar, deletar), as políticas cadastradas. As políticas são compostas por objetivos e restrições.

Já o recurso das otimizações, é responsável por gerenciar as otimizações executadas pelo sistema. O recurso além de fazer a otimização, também guarda um

histórico, com informações mais detalhadas, que pode ser consultado após feitas as otimizações. Além do histórico, o recurso também guarda métricas e informações sobre a otimização em si, como tempo de execução por exemplo.

Conforme REST propõe, foram utilizados os verbos para realizar as operações correspondentes ao que eles se propõe a fazer. Então o verbo POST é utilizado para a criação de recursos, e o GET para a busca de recurso(s), DELETE para deleção e PUT para atualização.

No geral, os recursos do OptVM ficaram organizados da seguinte maneira:

Tabela 2 – Tabela recurso otimização

Verbo	URI	Operação
GET	*/optimizations*	Todas as otimizações
POST	*/optimizations*	Cria uma otimização
GET	*/optimizations/:id*	Otimização específica
GET	*/optimizations/:id/details*	Detalhes da otimização
GET	*/optimizations/:id/metrics*	Métricas da otimização

Tabela 3 – Tabela recurso policy

Verbo	URI	Operação
GET	*/policies*	Todas as políticas
POST	*/policies*	Cria uma política
PUT	*/policies*	Atualiza uma política
GET	*/policies/:id*	Política específica
DELETE	*/policies/:id*	Exclui específica

4.6 FUNCIONAMENTO DO SERVIÇO

A utilização do serviço, exige que seja seguido um fluxo. O próprio serviço ajuda o usuário seguir o fluxo, através do HATEOAS, indicando links e próximas ações e consultas que podem ser feitas pelo consumidor da aplicação.

A ideia é que no caso de uso mais simples de utilização do serviço, para fazer uma otimização, é seguido o seguinte fluxo:

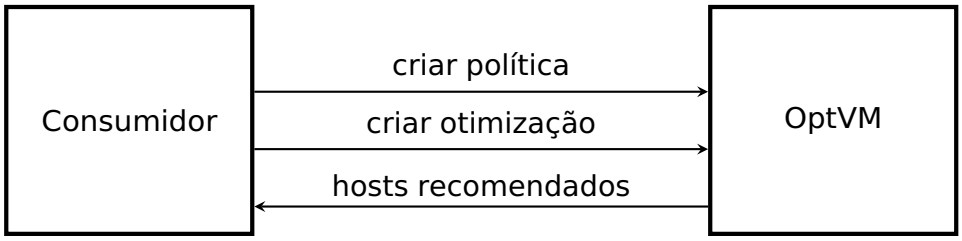
1. Criação de uma política, com seus objetivos e restrições de negócios;
2. Envio do conjunto de núvens/DCs/Hosts para ser feita a otimização;
3. Obtenção dos detalhes da otimização.

Apesar do caso de uso básico, é possível utilizar a API de maneiras diferentes, e fazer as combinações que forem mais úteis para o usuário.

A criação de uma política, é feita enviando os objetivos e as restrições. Após criada a política, a mesma pode ser utilizada em uma otimização.

Após a criação de um política, é possível usá-la para fazer otimizações.

A criação de uma otimização, retorna um objeto com um resumo dos melhores hosts e sua identificação, para melhorar o destino da VM. Além disso, é possível obter



detalhes de execução, através de */metrics*, assim como detalhes da otimização através da URI */details*.

No exemplo, a criação de um recurso de otimização deve-se ser utilizado o seguinte formato, para o corpo da requisição. Está sendo utilizado o formato JSON, porém, o mesmo se aplica também para o XML:

```
{
  "id": 1,
  "policy": 2,
  "clouds": [
    {
      "id": 1,
      "name": "Test",
      "datacenters": [
        {
          "id": 1,
          "hosts": [
            {
              "id": 1,
              "vms": [{"id": 10}]
            },
            ...
          ]
        },
        ...
      ]
      ...
    }
  ]
}
```

E a resposta terá o seguinte formato:

```
{
  "id": 1,
  "hosts": [
    {
      "cloud": 1,
      "datacenter": 2,
      "host": 1
    },
```

```
    ...
  ],
  "details": "/optimizations/1/details",
  "metrics": "/optimizations/1/metrics"
}
```

Apesar do exemplo estar mostrando arrays com apenas um item, tanto para a requisição como para a resposta, muito provavelmente haverá outros itens.

5 RESULTADOS

@TODO

6 CONCLUSÃO

@TODO

6.1 TRABALHOS FUTUROS

@TODO

6.2 CONSIDERAÇÕES FINAIS

@TODO

Referências

BROWN SIMON JOHNSTON, K. K. A. Using service-oriented architecture and component based development to build web service applications. Rational Software, 2002. Citado na página 3.

EIZINGER, T. Api design in distributed systems: A comparison between graphql and rest. Engineering at the University of Applied Sciences Technikum Wien, 2017. Citado na página 16.

FIELDING, R. T. Architectural styles and the design of network-based software architectures. University of California, 2000. Citado 3 vezes nas páginas 4, 5 e 16.

MATOS, A. V. de. A migração de máquinas virtuais no gerenciamento de recursos em ambientes de nuvens computacionais. Universidade Federal do Paraná (UFPR), 2017. Citado na página 8.

TICONA, A. C. B. D. W. G. C. Algoritmos evolutivos para otimização multi-objetivo. Universidade de São Paulo, 2003. Citado 2 vezes nas páginas 8 e 9.

VALIPOUR BAVAR AMIRZAFARI, K. N. M. M. H.; DANESHPOUR, N. A brief survey of software architecture concepts and service oriented architecture. Department of Electrical and Computer Engineering Shahid Rajaee University, 2009. Citado 2 vezes nas páginas 3 e 4.