Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Tese de Doutorado

Escalonamento Orientado por Qualidade de Serviço em Infraestruturas de Computação na Nuvem

Giovanni Farias da Silva

Campina Grande, Paraíba, Brasil
Fevereiro/2020

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Escalonamento Orientado por Qualidade de Serviço em Infraestruturas de Computação na Nuvem

Giovanni Farias da Silva

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Metodologia e Técnicas de Computação

Francisco Vilar Brasileiro e Raquel Vigolvino Lopes
(Orientadores)

Campina Grande, Paraíba, Brasil

S586e Silva, Giovanni Farias da.

Escalonamento orientado por qualidade de serviço em infraestruturas de computação na nuvem / Giovanni Farias da Silva. – Campina Grande, 2020.

145 f. : il. color.

Tese (Doutorado em Ciências da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Computação, 2020.

"Orientação: Prof. Dr. Francisco Vilar Brasileiro, Profa. Dra. Raquel Vigolvino Lopes".

Referências.

1. Computação na Nuvem. 2. Escalonamento. 3. Qualidade de Serviço. 4. Provisionamento Justo. I. Brasileiro, Francisco Vilar. II. Lopes, Raquel Vigolvino. III. Título.

CDU 004.738.5(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

ESCALONAMENTO ORIENTADO POR QUALIDADE DE SERVIÇO EM INFRAESTRUTURAS DE COMPUTAÇÃO NA NUVEM

GIOVANNI FARIAS DA SILVA

TESE APROVADA EM 18/02/2020

FRANCISCO VILAR BRASILEIRO, Ph.D, UFCG
Orientador(a)

RAQUEL VIGOLVINO LOPES, Dra., UFCG
Orientador(a)

ANDREY ELÍSIO MONTEIRO BRITO, Dr., UFCG
Examinador(a)

REINALDO CÉZAR DE MORAIS GOMES, Dr., UFCG
Examinador(a)

AVELINO FRANCISCO ZORZO, Dr., PUC-RS
Examinador(a)

HERMES SENGER, Dr., UFSCar
Examinador(a)

CAMPINA GRANDE - PB

Resumo

O modelo de Computação na Nuvem de infraestrutura como serviço (IaaS, do inglês *Infrastructure as a Service*) tem se tornado cada vez mais a principal escolha de provisionamento de infraestrutura de computação. Com isso, cresce também a diversidade das cargas de trabalho submetidas pelos usuários. Essa característica, juntamente com a heterogeneidade das complexas infraestruturas usadas para executar essas cargas de trabalho, tornam o gerenciamento de recursos um dos principais desafios para provedores de computação na nuvem de larga escala. Para aumentar a utilização de recursos (consequentemente a lucratividade) e satisfazer as necessidades distintas dos usuários, os provedores podem oferecer múltiplas classes de serviço. Essas classes são diferenciadas pela qualidade de serviço (QoS, do inglês *Quality of Service*) prometida, que comumente é definida em termos de objetivos de nível de serviço (SLO, do inglês *Service Level Objectives*) estabelecidos em um contrato de nível de serviço (SLA, do inglês *Service Level Agreement*). Por outro lado, os provedores estão sujeitos ao pagamento de penalidades em caso de violações dos SLAs. O gerenciamento de recursos é dividido em etapas com responsabilidades bem definidas. Essas etapas operam em conjunto com o intuito de evitar cenários de super provisionamento (mantendo os custos o mais baixo possível) e sub provisionamento (mantendo a QoS em níveis aceitáveis). A atividade de escalonamento é responsável por definir quais requisições devem ter recursos alocados em um dado instante e quais os servidores devem prover esses recursos. No contexto de provedores de larga escala, políticas de escalonamento baseadas em prioridades geralmente são utilizadas para garantir que as requisições de diferentes classes de serviço recebam a QoS prometida. Prioridades maiores são associadas com classes cujas QoSs prometidas são maiores. Caso seja necessário, recursos alocados para requisições com prioridades menores podem ser preemptados para permitir que requisições com prioridades maiores sejam executadas. Porém, nesse contexto, a QoS entregue às requisições durante períodos com contenção de recursos pode ser injusta para certas requisições. Em particular, requisições com prioridades mais baixas podem ter seus recursos preemptados para acomodar outras com prioridades mais altas, mesmo se a QoS entregue para as últimas esteja acima da desejada, e a QoS das primeiras esteja abaixo da esperada. Além disso, requisições com mesma prioridade

que competem pelo mesmo recurso podem experimentar QoSs bem diferentes, visto que algumas delas podem sempre executar enquanto outras permanecem sempre pendentes. Este trabalho apresenta uma política de escalonamento que é orientada pela QoS prometida para as requisições. Esta segue uma nova abordagem de preempção na qual qualquer requisição cuja QoS atual esteja excedendo sua respectiva meta pode ter seus recursos preemptados em benefício de outras com QoS abaixo da meta (ou que estejam mais próximas de ter seus SLOs não satisfeitos). Os benefícios de usar uma política orientada por QoS são: (i) ela mantém a QoS de cada requisição a mais alta possível, considerando suas respectivas metas e recursos disponíveis; e (ii) ela minimiza a variância da QoS entregue para requisições de uma mesma classe, promovendo o provisionamento justo. Essas características permitem a definição de SLAs que são mais apropriados e que estão de acordo com os principais provedores de nuvem pública. A política proposta foi avaliada através de comparações entre seus resultados e os obtidos com um escalonador que representa o estado-da-prática, baseado em prioridade. Esta comparação se deu por experimentos de simulação — validados por experimentos de medição — alimentados por amostras de rastros de execução de um sistema em produção. No geral, o escalonamento orientado por QoS entrega um serviço melhor que o baseado em prioridade, especialmente quando a contenção de recursos não é tão alta. A similaridade da QoS entregue para requisições de uma mesma classe também foi muito mais alta quando o escalonamento orientado por QoS foi utilizado, particularmente quando nem todas as requisições receberam a QoS prometida. Além disso, com base na prática atual de grandes provedores públicos, os resultados mostraram que as penalidades incorridas pelo uso do escalonador baseado em prioridade podem ser, em média, até aproximadamente 2 vezes mais altas que aquelas incorridas pelo escalonador orientado por QoS. Por fim, o custo operacional do escalonador orientado por QoS foi aproximadamente 15 vezes maior. Porém, ainda há espaço para a implementação de otimizações, tal como um mecanismo de *cache* — já implementado no escalonamento baseado em prioridade. A nova política ainda se mostrou viável de ser implementada em um sistema real. As métricas utilizadas por seu escalonador foram definidas de forma que a política opere no sistema sem interferir na forma como os usuários interagem com o mesmo.

Abstract

The Infrastructure-as-a-Service (IaaS) model has increasingly become the primary choice for computing infrastructure provisioning. As a result, the diversity of workloads submitted by the users also increases. This feature, coupled with the heterogeneity of the complex infrastructures used to run these workloads, make resource management a major challenge for large-scale IaaS providers. In order to increase the resource utilization (consequently the profitability) and to satisfy the distinct needs of users, the provider may offer multiple service classes. Classes are distinguished by their expected Quality of Service (QoS), which is defined in terms of Service Level Objectives (SLO) established in a Service Level Agreement (SLA). On the other hand, the providers generally have to pay some kind of penalties in case of SLA violations. The resource management activities are divided into well-defined steps, which operate together in order to avoid overprovisioned and underprovisioned scenarios. The former scenarios must be avoided to keep the costs as low as possible, while the last ones must be avoided to deliver QoS at acceptable levels. The scheduling step is responsible for defining which requests should have resources allocated at any point in time and which servers should provide those resources. In the context of large scale providers, a priority-based scheduling policy is commonly used to guarantee that service requests submitted to the different service classes achieve the desired QoS. Higher priorities are associated with classes whose expected QoS is higher. If needed, resources servicing lower priority requests can be preempted to allow the servicing of higher priority ones. In this context, however, the QoS delivered during resource contention periods may be unfair on certain requests. In particular, lower priority requests may have their resources preempted to accommodate resources associated with higher priority ones, even if the actual QoS delivered to the latter is above the desired level, while the former is underserved. Also, competing requests with the same priority may experience quite different QoS, since some of them may have their resources preempted, while others do not. This document presents a scheduling policy that is driven by the QoS promised to individual requests. It follows a novel preemption approach in which any request whose actual QoS is above the promised target can have its associated resources preempted for the benefit of other requests whose actual QoS are below their targets or closer to miss them. Benefits of using the QoS-driven policy are twofold:

(i) it maintains the QoS of each request as high as possible, considering their QoS targets and available resources; and (ii) it minimizes the variance of the QoS delivered to requests of the same class, promoting fairness. These features allow the definition of SLAs that are more appropriate, and in line with those adopted by the major public cloud providers. The QoS-driven scheduling policy was assessed by comparing its results with the results obtained from a priority-based scheduling policy, which represents the state-of-practice. The simulation experiments fed with traces from a production system were used to compare the QoS-driven policy with a state-of-the-practice priority-based one. The simulation models used were validated by measurement experiments. In general, the QoS-driven policy delivers a better service than the priority-based one, especially when resource contention is not severe. The equity of the QoS delivered to requests of the same class was much higher when the QoS-driven policy was used, particularly when not all requests got the promised QoS, which is the most important scenario. Moreover, based on the current practice of large public cloud providers, the results show that penalties incurred by the priority-based scheduler can be, on average, as much as 193% higher than those incurred by the QoS-driven one. Finally, the operational cost of QoS-driven scheduling was arround 15 times higher. However, there still is space to implement optimizations for QoS-driven such as caching mechanism, which is already implemented in priority-based scheduling. The QoS-driven scheduler also proved viable to be implemented in a real system. The metrics used by this scheduler were defined in a way that the scheduling policy can operate in the system with no interference in the way that users interact with the system.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus por estar sempre ao meu lado, pela força nos momentos difíceis e por guiar minhas escolhas ao longo da vida.

Aos meus pais, Lúcia e Gilmar (*in memorian*), que apesar de todas as dificuldades, nunca mediram esforços para investir em minha educação. Agradeço os ensinamentos, conselhos, confiança e o amor sempre presentes. Vocês são minhas referências em amor, dedicação, entrega e generosidade.

A minha esposa Thaís, minha riqueza, por todo amor, carinho e paciência ao longo deste período. Obrigado pelo incentivo, companheirismo e também pela compreensão nos momentos em que meu corpo estava presente mas minha cabeça estava pensado em como resolver questionamentos deste trabalho.

Aos meus orientadores Fubica e Raquel por todos os ensinamentos a mim transmitidos através das várias discussões ao longo deste período. Eu destaco que esses ensinamentos vão além de conceitos em Ciência da Computação. Vocês são uma inspiração profissional para mim. Eu espero poder fazer por muitas pessoas o que vocês fizeram por mim.

A toda minha família (irmãos, avós, tias e tios, primas e primos) pelos momentos de descontração e por sempre acreditarem em mim. Agradeço especialmente aos meus irmãos Gilmara e Giordanni (onde também estendo os agradecimentos aos cunhados Sidney e Yêda) pela torcida e suporte de sempre, e, ao meu sobrinho Guilherme que sempre me proporciona momentos de alegria. Mesmo sem saber, Gui funcionou como uma válvula de escape para os momentos de aperreios. Também gostaria de agradecer ao meu sogro Cal, minha sogra Lêda e aos cunhados Juliana e Eduardo pelo incentivo constante.

Aos meu amigos do Laboratório de Sistemas Distribuídos, especialmente a João, Fábio, Marcus, Eduardo, Vinícius e Alessandro, pelas contribuições diretas para minha tese. Além disso, agradeço também aos amigos que tenho em Campina Grande pelas conversas, cervejadas, churrascos, e feijoadas que fizemos ao longo deste período. Esses momentos também foram muito importantes para a conclusão deste doutorado.

Por fim, gostaria de agradecer ao povo brasileiro e a Ericsson Telecomunicações pelo subsídio financeiro que possibilitou a realização deste trabalho.

Conteúdo

Acrônimos

BoT - *Bag-of-Tasks*

EDF - *Earliest Deadline First*

FCFS - *First Come First Served*

IaaS - *Infrastructure as a Service*

PoC - *Proof of Concept*

QoS - *Quality of Service*

SLA - *Service Level Agreements*

SLI - *Service Level Indicator*

SLO - *Service Level Objectives*

TI - *Tecnologia da Informação*

TTV - *Time-To-Violate*

VM - *Virtual Machine*

Lista de Símbolos

Escalonamento Orientado por QoS

$A_j(t)$ - disponibilidade da requisição j no instante de tempo t

$e_j(t)$ - tempo de execução acumulado que as instâncias da requisição j contribuíram para a QoS de j até o instante de tempo t

$d_j(t)$ - tempo de execução acumulado que as instâncias da requisição j executaram mas não contribuíram para a melhoria da QoS de j até o instante de tempo t

$p_j(t)$ - tempo acumulado que as instâncias da requisição j estiveram pendentes até o instante de tempo t

O_i - SLO de disponibilidade estabelecido para a classe de serviço i

α_j - tempo esperado para a alocação de uma instância da requisição j em um servidor

$v_j^r(t)$ - o número de instâncias em execução da requisição j no instante de tempo t

$v_j^w(t)$ - o número de instâncias pendentes da requisição j no instante de tempo t

$\Delta t_j(t)$ - intervalo de tempo que a requisição j pode permanecer recebendo determinada contribuição para incrementar sua disponibilidade, a partir do instante de tempo t, sem que seu SLO seja violado

c_j - a contribuição que a requisição j recebe para melhorar sua disponibilidade ao longo do intervalo $\Delta t_j(t)$

$\Delta t_j^*(t)$ - TTV da requisição requisição j no instante de tempo t

$\Delta r_j(t)$ - quantidade de tempo que as instâncias da requisição j estiveram pendentes desde que seu SLO passou a não ser satisfeito até o instante de tempo t

$\Delta r_j^*(t)$ - *recoverability* da requisição j no instante de tempo t

$Q_j(t)$ - métrica de QoS usada para decisão do escalonador a ser associada com uma instância da requisição j no instante de tempo t

σ_i - margem de segurança para a métrica de QoS de requisições da classe de serviço i

Modelos de Simulação

$c_h^c(t)$ - capacidade total de CPU do servidor h no instante t

$a_h^c(t)$ - capacidade de CPU do servidor h alocada para as instâncias em execução no instante t

$c_h^r(t)$ - capacidade total de memória do servidor h no instante t

$a_h^r(t)$ - capacidade de memória do servidor h alocada para as instâncias em execução no instante t

$\mathcal{L}_h(t)$ - função de prioridade *Least Requested Priority* para alocar uma instância no servidor h no instante t

$\mathcal{B}_h(t)$ - função de prioridade *Balanced Resource Allocation* para alocar uma instância no servidor h no instante t

$P_h(t)$ - conjunto de todas as instâncias em execução no servidor h que precisam ser preemptadas para a alocação de uma instância pendente no instante t

$P_{h+}(t)$ - conjunto de instâncias que precisam ser preemptadas no servidor h no instante t, e que suas requisições não estão tão próximas de violar seus SLOs.

$P_{h-}^i(t)$ - conjunto de instâncias da classe i que precisam ser preemptadas no servidor h no instante t, e que já tem suas requisições violando ou próximas de violar seus respectivos SLOs

$C_k(t)$ - sobrecarga de preempções de uma instância k no tempo t

Modelo de Penalidades

D_j - déficit de QoS experimentado pela requisição j ao final de sua execução

u_j - quantidade de CPU solicitada pela requisição j

d_j - duração da execução da requisição j

D_j^* - déficit da requisição j em termos de CPU $\cdot$ tempo

b_j - bônus a ser oferecido pelo provedor por causa da violação do SLA da requisição j

P_j - penalidade do provedor por violar o SLA da requisição j

Análise de Viabilidade

C_s - conjunto finito de instâncias gerenciadas por um controlador de serviço

C_b - conjunto finito de instâncias gerenciadas por um controlador *Batch*

C_b^r - número máximo de instâncias que um controlador C_b pode executar em paralelo

p_i^e - *e-ésima* encarnação da instância p_i

$s(p_i^e, t)$ - estado da encarnação p_i^e no instante de tempo t

$p_{i\,w}^e$ - a encarnação p_i^e no estado *Esperando*

$p_{i\,r}^e$ - a encarnação p_i^e no estado *Executando*

$p_{i\,f}^e$ - a encarnação p_i^e no estado *Falha*

$p_{i\,c}^e$ - a encarnação p_i^e no estado *Completa*

$t_{p_{i\,w}^e}$ - instante de tempo que a encarnação p_i^e vai para o estado *Esperando*

$t_{p_{i\,r}^e}$ - instante de tempo que a encarnação p_i^e vai para o estado *Executando*

$t_{p_{i\,f}^e}$ - instante de tempo que a encarnação p_i^e vai para o estado *Falha*

$t_{p_{i\,c}^e}$ - instante de tempo que a encarnação p_i^e vai para o estado *Completa*

$w(p_i^e, t)$ - quantidade de tempo que a encarnação p_i^e esteve pendente desde sua criação até o instante de tempo t

$r(p_i^e, t)$ - quantidade de tempo que a encarnação p_i^e esteve em execução desde sua criação até o instante de tempo t

$c(p_i^e, t)$ - quantidade de tempo que a encarnação p_i^e esteve em execução simultaneamente com as outras instâncias do mesmo controlador, desde sua criação até o instante de tempo t

$W_C(t)$ - conjunto de encarnações das instâncias do controlador C que, no instante de tempo t, estão no estado *Esperando*

$R_C(t)$ - conjunto de encarnações das instâncias do controlador C que, no instante de tempo t, estão no estado *Executando*

$F_C(t)$ - conjunto de encarnações das instâncias do controlador C que, no instante de tempo t, estão no estado *Falha*

$C_C(t)$ - conjunto de encarnações das instâncias do controlador C que, no instante de tempo t, estão no estado *Completa*

$I_C(t)$ - conjunto de todas as encarnações das instâncias do controlador C no instante de tempo t

$W_{p_i}(t)$ - conjunto de encarnações da instância p_i que, no instante de tempo t, estão no estado *Esperando*

$R_{p_i}(t)$ - conjunto de encarnações da instância p_i que, no instante de tempo t, estão no

estado *Executando*

$F_{p_i}(t)$ - conjunto de encarnações da instância p_i que, no instante de tempo t, estão no estado *Falha*

$C_{p_i}(t)$ - conjunto de encarnações da instância p_i que, no instante de tempo t, estão no estado *Completa*

$I_{p_i}(t)$ - conjunto de todas as encarnações da instância p_i no instante de tempo t

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo introdutório apresenta o contexto de nuvens computacionais, no qual este trabalho está inserido, bem como delimita o escopo de seu objeto de estudo. Em seguida, caracteriza-se o problema existente com a técnica de escalonamento tradicionalmente utilizada por grandes provedores no estado-da-prática, bem como a solução proposta por este trabalho. Logo após, são apresentados os objetivos geral e específicos, as principais contribuições deste trabalho e como este documento está organizado.

1.1 Contextualização e Escopo

O paradigma de Computação na Nuvem tem se consolidado cada vez mais no cenário global de Tecnologia da Informação. Este paradigma tem como princípio a terceirização de serviços computacionais para provedores de serviços, assim como os usuários já fazem para outros tipos de serviços como eletricidade e água [52]. Por causa das diferentes necessidades dos usuários, os provedores de Computação na Nuvem comumente oferecem diferentes modelos para aquisição de seus serviços computacionais.

Um dos modelos oferecidos é o de infraestrutura como serviço (IaaS, do inglês *Infrastructure as a Service*). Neste modelo, os provedores oferecem recursos computacionais como ciclos de CPU, espaço de memória, armazenamento em disco e banda da rede. Geralmente, esses recursos são empacotados e oferecidos no formato de Máquina Virtual (VM, do inglês *Virtual Machine*) [48] ou contêiner [44]. Essas tecnologias isolam os recursos físicos que estão sendo compartilhados entre os vários usuários. Dessa forma, evita-se que as execuções

relacionadas a uma requisição afetem o desempenho de outra que esteja alocada na mesma máquina física (servidor). Neste trabalho, o conjunto de recursos agrupados para atender uma requisição do usuário é referenciado como *instância*, independente de ser empacotado pelo provedor no formato de VM ou contêiner.

O modelo de IaaS tem se tornado cada vez mais a principal escolha para provisionamento de infraestruturas de computação [8]. Do ponto de vista do consumidor, as instâncias podem ser rapidamente provisionadas, e, em seguida, liberadas. Esse comportamento possibilita elasticidade teoricamente ilimitada na medida em que a demanda aumente ou diminua. Além disso, o modelo "pague conforme utilização" (*pay-as-you-go* em inglês) é a forma de tarifação geralmente praticada por provedores públicos. Nesse cenário, os usuários pagam apenas pelos recursos de fato utilizados [28]. Assim sendo, os consumidores têm custos apenas durante os períodos que os recursos de fato são utilizados. Após a liberação dos recursos, o provedor é o responsável por assumir os custos relacionados ao suporte do serviço, tais como manutenção da infraestrutura, atualizações, equipe de manutenção, etc.

O gerenciamento de infraestruturas de computação bastante complexas e de larga escala é tido como um dos principais desafios para os grandes provedores de IaaS [24]. Aliás, essas infraestruturas também recebem cargas de trabalho normalmente bastante heterogêneas e que variam ao longo do tempo. Com o objetivo de aumentar a utilização dos recursos (consequentemente, a lucratividade), os provedores podem oferecer múltiplas classes de serviço. Tipicamente, grandes provedores oferecem sua capacidade temporariamente não utilizada de forma oportunista, essencialmente sem garantia de Qualidade do Serviço (QoS, do inglês *Quality of Service*) [38]. A ausência de uma expectativa mínima de QoS restringe as aplicações que podem se beneficiar desses recursos, e, por isso, o preço cobrado por eles é menor (por exemplo, instâncias Amazon EC2 spot [1] e Google Cloud preemptible [3]). No entanto, Carvalho et al. [13; 15] mostraram que um provedor pode agregar mais valor aos seus recursos ociosos, oferecendo-os por meio de uma ou mais novas classes de serviço. Cada nova classe define uma expectativa de QoS mínima a longo prazo para as requisições por essas novas classes. Por causa da QoS prometida, o provedor consegue cobrar mais pelas instâncias dessas classes em comparação àquelas oferecidas oportunistamente. Isso permite que o provedor aumente a utilização dos recursos de forma mais rentável [54]. As novas classes são úteis para aplicações que podem aceitar recursos levemente degradados,

mas ainda precisam de garantias moderadas de QoS (por exemplo, aplicações não interativas como indexação da web e transcodificação de vídeo).

Na medida em que o mercado de provedores de computação na nuvem se torna mais competitivo, os provedores se diferenciam oferecendo uma gama mais ampla de classes de serviço [7]. Cada classe está associada a um esquema diferente de precificação e uma QoS mínima esperada. Nesse cenário, alguns usuários podem querer pagar para ter recursos dedicados sempre disponíveis, mas a maioria pode aceitar uma expectativa menor de QoS em troca de preços mais baixos [13]. Por isso, um número maior de classes possibilita que os usuários escolham as classes adequadamente com base na QoS necessária para sua aplicação. Nesse contexto, a Amazon EC2, um dos principais provedores de IaaS disponíveis no mercado, oferece instâncias computacionais através de quatro classes de serviço: *On-demand*, *Reserved*, *Spot* e *Dedicated Hosts* [1].

A QoS prometida para cada classe é estipulada através de metas específicas. Essas metas são estabelecidas pelos "Objetivos de Nível de Serviço" (SLO, do inglês *Service Level Objectives*) definidos em um "Contrato de Nível de Serviço" (SLA, do inglês *Service Level Agreement*) correspondente a cada classe. Os SLAs são acordados entre o provedor e seus clientes no momento da contratação do serviço. Nesse sentido, os provedores definem SLOs para diversas métricas de QoS e se esforçam para cumpri-los. Em um instante qualquer no tempo, a QoS que está sendo entregue para uma requisição é denominado por um ou mais "Indicadores de Nível de Serviço" (SLI, do inglês *Service Level Indicator*). Cada SLI está associado com um SLO, *i.e.* o SLI representa a medição atual para uma métrica de QoS e o SLO representa a meta estabelecida para a mesma métrica de QoS. As diferentes classes de serviço podem ser determinadas de acordo com variadas combinações de SLOs. Alguns exemplos de SLOs que podem ser fixados pelo provedor são: os recursos estarão disponíveis por mais que *99%* do tempo desde o momento que o usuário submeteu sua requisição; no mínimo 99% das requisições serão admitidas para execução; uma requisição levará no máximo *30* segundos para que sua instância esteja acessível ao usuário; dentre outros.

É importante destacar que há a possibilidade do provedor não conseguir entregar a QoS mínima prometida. Um SLO tem uma meta a ser atendida, enquanto o SLA define essas metas e em quais circunstâncias essas metas podem não ser atendidas sem que isso gere uma violação do SLA (por exemplo, o SLA pode prever que o SLO possa não ser atendido

um certo número de vezes). O não atendimento de SLOs pode levar a violações de SLAs. Por exemplo, em períodos com alta contenção de recursos, o provedor pode não atender os SLOs para todas as requisições no sistema. Esses períodos podem ser causados pelo aumento inesperado da demanda ou pela falha de servidores. Nesse caso, tipicamente, o provedor está sujeito a pagar penalidades por requisições que tiveram seus SLAs violados. Essas penalidades também estão estabelecidas no SLA de cada classe de serviço.

A penalidade a ser paga pelo provedor não necessariamente é diretamente proporcional ao número de requisições violadas. Por exemplo, dois dos principais provedores públicos de computação na nuvem (AWS[1] e Azure[2]) têm SLAs cujas penalidades variam de maneira não linear com o déficit de QoS experimentado pela requisição. Nesse sentido, déficits de QoS maiores levam a penalidades cada vez maiores.

O gerenciamento eficiente de seus recursos permite que o provedor possa cumprir os SLAs das diferentes classes de serviço, ao mesmo tempo que reduz os custos envolvidos com o provisionamento do serviço. As atividades relacionadas a este gerenciamento podem ser divididas em três etapas principais: *Planejamento de Capacidade*, *Controle de Admissão* e *Escalonamento* [15]. O *Planejamento de Capacidade* define a quantidade de recursos adequada para que o provedor consiga atender a demanda esperada. Esta etapa estabelece a capacidade da infraestrutura do provedor para um período relativamente longo (horizonte de meses). A meta dessa atividade é evitar tanto o super quanto o sub provisionamento dos recursos, considerando a demanda futura esperada. Na etapa de *Controle de Admissão* são estabelecidos critérios de admissão/rejeição de novas requisições. A possível rejeição de parte das requisições tem o intuito de não permitir que novas requisições gerem uma carga que possa afetar negativamente — a ponto de causar violações de SLAs — o desempenho daquelas que já foram admitidas anteriormente. Por último, na fase de *Escalonamento*, decide-se quais requisições terão recursos alocados em um dado instante e quais os servidores proverão os recursos para as mesmas. Em momentos em que há contenção de recursos (*i.e.* quando não é possível alocar recursos para todas as requisições ativas no sistema), geralmente, um subconjunto das instâncias solicitadas permanecem pendentes esperando que recursos ocupados sejam liberados. Assim sendo, sempre que é executado, o escalonador é

[1] https://aws.amazon.com/compute/sla/

[2] https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_8/

responsável por escolher quais das instâncias devem estar em execução e quais devem ficar pendentes (se necessário). De forma genérica, uma requisição pode solicitar recursos para uma ou mais instâncias. Neste caso, considera-se que cada requisição estará associada com a quantidade de instâncias por ela solicitada. Uma vez que uma requisição é admitida, o escalonador trabalha para alocar as instâncias solicitadas por esta requisição, considerando os SLOs definidos no SLA da classe de serviço da requisição. No entanto, em períodos com contenção de recursos, é possível que nem todas as instâncias solicitadas por uma requisição tenham recursos alocados para sua execução. Destaca-se que *este trabalho tem como objeto de estudo a etapa de escalonamento*.

O escalonador é responsável por executar uma política de escalonamento, que busca otimizar um ou mais objetivos. Alguns dos critérios utilizados por políticas de escalonamento são [12]: evitar a sobrecarga de servidores para que as instâncias não afetem o desempenho umas das outras; consolidar a carga de trabalho de forma a diminuir a quantidade de servidores ativos sem causar violações de SLA, buscando reduzir custos; deixar próximas as instâncias que executam aplicações dependentes e que possuem fortes requisitos de comunicação; dentre outros.

Diferentes escalonadores executam políticas de escalonamento distintas, que buscam atender objetivos diversos. Por exemplo, a política "Primeiro a Chegar Primeiro a ser Servido" (FCFS, do inglês *First Come First Served*) busca garantir que as requisições que foram admitidas primeiro sejam as primeiras a serem atendidas. Dessa forma, as requisições são atendidas com base em seus respectivos tempos de admissão. Por esta razão, quando a política FCFS é implementada, não há interrupções (preempções) de instâncias que já estejam em execução para permitir que outras sejam executadas. Por não levar em conta a QoS prometida para cada classe de serviço, esta política se torna ineficiente para garantir a QoS para as diferentes classes.

No contexto de múltiplas classes de serviço, diversos escalonadores de provedores de larga escala usam políticas *baseadas em prioridade* [10; 11; 18; 22; 23; 34; 45; 50; 51]. Esses escalonadores diferenciam as classes de serviço através da associação de uma prioridade para cada uma delas. Quanto maior é a QoS prometida para a classe, maior será a prioridade atribuída a ela. Nesse cenário, cada instância associada com uma requisição é relacionada com a mesma prioridade da classe de serviço da requisição. Com uma prioridade

atribuída para cada instância, um escalonador deste tipo toma decisões que levam em conta a QoS prometida para as diferentes classes. A prioridade de uma instância funciona como um *proxy* para sua QoS prometida. Nesse sentido, um escalonador baseado em prioridade pode preemptar uma instância em benefício de uma outra, desde que a última tenha uma prioridade maior que a primeira. Isso ocorre independente do momento de admissão de suas respectivas requisições. Entre requisições de mesma prioridade, usualmente a mais antiga tem preferência no momento da alocação. Portanto, nesse contexto, as instâncias com prioridades mais altas têm menos chances de serem preemptadas, e, consequentemente, elas têm mais chances de alcançar QoS melhor que as instâncias com prioridades mais baixas. No entanto, apesar desta política ser amplamente utilizada por escalonadores que conhecem múltiplas classes de serviço, essa abordagem pode ser ineficiente em algumas situações descritas a seguir.

1.2 Caracterização do Problema

Apesar de considerar a QoS prometida para a classe de serviço, a política de escalonamento baseada em prioridade pode tomar decisões ineficientes em pelo menos duas situações, quando há contenção por recursos:

- As instâncias com prioridades mais baixas têm maiores chances de não receber a QoS esperada. Isso se deve à possibilidade delas sempre serem preemptadas em benefício de instâncias com prioridades mais altas. Isso ocorre inclusive quando as últimas já estiverem recebendo QoS acima da esperada. Por exemplo, suponha que a instância a, de prioridade 2 (com QoS esperada de 90% de disponibilidade[3]), esteja alocada no servidor x e recebendo 100% de disponibilidade em um dado instante. Neste mesmo instante, considere que a instância b, de prioridade 0 (com QoS esperada de 50% de disponibilidade), esteja alocada no servidor y e recebendo 30% de disponibilidade como QoS. Além disso, suponha também que a e b demandem a mesma quantidade de recursos. Nesse instante, considere que x precisa ser removido da infraestrutura para manutenção. Se não houver recursos disponíveis para alocar a em qualquer outro

[3]Considere a disponibilidade de uma instância como o percentual de tempo que a mesma esteve em execução em um servidor desde a admissão de sua respectiva requisição. Esta métrica será descrita e discutida no Capítulo 3.

servidor, b (por ser de prioridade mais baixa) pode ser preemptada em benefício de a. Isso acontece mesmo com a recebendo uma QoS acima da esperada e b, ao contrário, recebendo QoS abaixo da esperada.

- Dado que uma instância não é preemptada em benefício de uma outra de mesma classe (*i.e.*, mesma prioridade), a QoS entregue para instâncias de uma mesma classe pode apresentar alta variabilidade. Isso ocorre quando, em períodos com contenção de recursos, duas instâncias de mesma classe estão competindo pelo mesmo recurso. Por exemplo, suponha que existam duas instâncias a e b de prioridade 2 (com QoS esperada de 90% de disponibilidade) e que o provedor tenha capacidade para alocar apenas uma delas. De acordo com o escalonamento baseado em prioridade, uma instância ficará sempre em execução e a outra sempre pendente. Considerando que a estará em execução, uma vez que a não será preemptada em benefício de b, a diferença entre as QoSs recebidas por elas será cada vez maior. A QoS de a ficará cada vez melhor ao longo do tempo (*i.e.* disponibilidade de a será incrementada para um valor cada vez maior), enquanto que a QoS de b ficará cada vez pior (*i.e.* disponibilidade diminuirá para um valor cada vez menor).

Em resumo, o escalonador baseado em prioridade não foi concebido para respeitar a QoS prometida para todas as classes de serviço simultaneamente. Enquanto houver requisições com prioridades mais altas necessitando de recursos, o escalonador alocará qualquer recurso livre para elas. Além disso, caso seja necessário, ele sempre preemptará recursos de requisições com prioridades mais baixas para liberar recursos para outras de maior prioridade, independente das QoSs atuais das requisições envolvidas. Finalmente, uma vez que os recursos são alocados para uma requisição, eles nunca serão preemptados em benefício de uma outra requisição de mesma classe. Por esta razão, em períodos com contenção de recursos, é possível que aconteça alta variabilidade na QoS entregue para requisições de uma mesma classe (*i.e.* o provisionamento de recursos não é justo nesse cenário).

Uma possível solução para essas situações seria fazer com que o provedor sempre trabalhasse com uma infraestrutura super provisionada. No entanto, os custos envolvidos em manter essa infraestrutura proíbem esta alternativa. É importante destacar que as atividades de planejamento de capacidade e controle de admissão, que operam em conjunto com o es-

calonamento, sempre buscam evitar ambos os cenários: o super provisionamento, mantendo os custos o mais baixo possível; e, o sub provisionamento, mantendo a QoS entregue em níveis aceitáveis. Nesse contexto, pode-se inferir que o provedor tem como objetivo operar em cenários com contenção moderada, onde a infraestrutura e carga de trabalho são dimensionadas de forma a manter alta utilização de recursos e ainda entregar a QoS prometida para as requisições. No entanto, imprecisões tanto no controle de admissão como no planejamento de capacidade podem causar períodos com alta contenção, por isso, o escalonador precisa lidar com essas situações.

1.3 Objetivos

O objetivo geral deste trabalho é propor e avaliar uma nova política de escalonamento que possa *lidar com a QoS prometida para todas as classes de serviço simultaneamente e tratar de forma mais justa requisições de uma mesma classe*, particularmente em períodos com contenção de recursos. Por tratamento justo, entende-se que as requisições de uma mesma classe devem receber QoSs semelhantes quando estiverem competindo pelos mesmos recursos, inclusive quando não for possível entregar a QoS prometida. Tendo em vista este objetivo principal, os seguintes objetivos específicos foram definidos:

1. Propor uma política de escalonamento que tome decisões com base na QoS que estiver sendo entregue (SLI) para cada requisição no sistema no momento da tomada de decisão, bem como em suas respectivas metas de QoS. Esta política tem como objetivo cumprir os SLAs das requisições admitidas, independentemente da classe de serviço solicitada. Além disso, a política também visa provisionar recursos de forma mais justa para requisições de uma mesma classe que estejam competindo pelos mesmos recursos;

2. Identificar como os rastros de execução de provedores públicos disponíveis podem ser utilizados para a avaliação da nova política de escalonamento;

3. Avaliar a nova política de escalonamento através da comparação dos resultados obtidos por ela com os obtidos por escalonadores baseados em prioridades. Para esta avaliação é fundamental utilizar cargas de trabalho e infraestruturas realistas, *i.e.* representativas

e relevantes em ambientes de provedores operando em larga escala, mas também é importante a utilização de cargas sintéticas para avaliação sob condições específicas;

4. Analisar a viabilidade da nova política. Esta análise deve ser baseada em aspectos práticos relacionados, por exemplo, ao desempenho do escalonador e a forma como tipicamente os usuários do provedor submetem suas cargas de trabalho.

1.4 Solução Proposta

Como mencionado na seção anterior, este trabalho propõe uma nova política de escalonamento para provedores de computação na nuvem, denominada **Escalonamento Orientado por QoS**. Neste cenário, o escalonador toma decisões levando em conta a QoS entregue (SLI) para cada uma das requisições no sistema no momento da tomada de decisão e suas correspondentes metas de QoS. Esta política tem basicamente dois objetivos: (i) atender as metas de QoS das requisições admitidas, quaisquer que sejam suas classes de serviço; e, (ii) oferecer tratamento justo para requisições de uma mesma classe, entregando QoSs semelhantes para as requisições que competem pelo mesmo recurso.

Os objetivos da política proposta são alcançados através de um novo mecanismo de preempção. Este novo mecanismo preempta instâncias cuja QoS esteja excedendo a QoS esperada, e, usa os recursos liberados para alocar instâncias cuja QoS esteja abaixo de sua respectiva meta (ou mais próxima de violá-la). Isso ocorre mesmo que as instâncias beneficiadas sejam de classes com prioridades mais baixas que as classes daquelas preemptadas. Na prática, o Escalonador Orientado por QoS está sempre trabalhando para diminuir a diferença entre a QoS entregue para as requisições e suas respectivas metas de QoS. Nesse contexto, o escalonador pode preemptar qualquer instância em benefício de uma outra que esteja com um risco maior ter seu SLA violado.

A relevância deste trabalho pode ser destacada tanto do ponto de vista do provedor quanto de seus consumidores. Na perspectiva do provedor, este pode usar seus recursos de forma mais efetiva em períodos em que não há recursos não utilizados. A política proposta visa alocar os recursos de forma que uma maior quantidade de requisições tenham seus SLAs cumpridos, independentemente de suas respectivas classes. Já na perspectiva do usuário, este sempre vai receber a melhor QoS que o provedor pode entregar para a classe solicitada, com

base na demanda atual do provedor e no provisionamento justo (igualitário) entre requisições de mesma classe.

1.5 Contribuições

A **política de Escalonamento Orientada por QoS** (Capítulo 3) é a principal contribuição deste trabalho. Além disso, são contribuições importantes: (i) a **avaliação experimental** (Capítulos 4, 5 e 6) e (ii) a **análise de viabilidade** (Capítulos 5 e 7) da política proposta. A avaliação se deu através de experimentos de simulação, comparando a nova política com uma política baseada em prioridades utilizada na prática [11]. Já a análise de viabilidade ocorreu através de implementação de um protótipo em um sistema real, o Kubernetes[4]. Este protótipo foi usado para validar os modelos de simulação utilizados mas também como prova de conceito da nova política de escalonamento.

O resultados provenientes deste trabalho são apresentados a seguir: (i) A **publicação do artigo** "*Escalonamento justo em infraestruturas de nuvem com múltiplas classes de serviço*" [47] no Simpósio Brasileiro de Redes de Computadores (SBRC) 2019. Este artigo foi *premiado como o melhor artigo* da trilha principal do evento; (ii) A **patente** "*Method and resource manager for scheduling of instances in a data centre*" foi registrada com número de registro PCT/SE2017/050375 [14]; (iii) a **submissão do artigo** "*QoS-driven scheduling in the cloud*" [26] para a revista científica *Journal of Information Security and Applications* está sob revisão; e, (iv) o **artigo** "*Availability-driven scheduling in Kubernetes*" [25] **a ser submetido** para a revista científica *IEEE Transactions on Cloud Computing*.

Em termos de software, basicamente dois artefatos foram gerados ao longo deste trabalho: o **simulador** (Capítulo 4) e o **protótipo** do escalonador orientado por QoS para o Kubernetes (Capítulos 5 e 7). Para este último, dois componentes adicionais foram utilizados para permitir o cálculo das métricas necessárias ao novo escalonador: *kubewatch* (para monitorar eventos relacionados com as instâncias, tais como criação, alocação, preempção e término) e *prometheus* (para armazenar as métricas necessárias ao novo escalonador).

Por último, a **publicação do artigo** *"On the Efficiency Gains of Using Disaggregated Hardware to Build Warehouse-Scale Clusters"* [24] na *IEEE International Conference on*

[4]Informações sobre o sistema Kubernetes estão disponíveis em https://kubernetes.io/.

Cloud Computing Technology and Science (CloudCom) 2017 também é um resultado deste trabalho de tese. Este artigo descreve um estudo preliminar a partir da investigação inicial sobre o tema de escalonamento em provedores de IaaS, analisando o impacto de tecnologias de Hardware Desagregado no escalonamento em infraestruturas de nuvem.

1.6 Organização do documento

O restante deste documento é organizado da seguinte forma. No Capítulo 2 são apresentados os trabalhos relacionados ao objeto de estudo desta tese. A política de escalonamento orientada por QoS proposta é apresentada no Capítulo 3. Em seguida, a metodologia de avaliação e o processo de validação dos modelos de simulação são detalhados nos Capítulos 4 e 5 respectivamente. A avaliação dos resultados obtidos dos experimentos de simulação são discutidos no Capítulo 6. Em seguida, analisa-se o escalonamento orientado por QoS sob um ponto de vista mais prático no Capítulo 7, tais como o desempenho do escalonador e sua viabilidade de implementação no contexto de um sistema real. Por fim, no Capítulo 8 são resumidas as principais conclusões deste trabalho e possíveis direcionamentos para trabalhos a serem desenvolvidos no futuro. Além disso, o Apêndice A contém instruções para reprodução dos experimentos executados ao longo deste trabalho.

Capítulo 2

Trabalhos Relacionados

Este capítulo descreve os trabalhos relacionados ao objeto de estudo deste trabalho de tese. Esse levantamento do estado da arte tem como principal objetivo posicionar este trabalho em relação à literatura atual com relação a alguns aspectos, tais como arquitetura da solução, o suporte a múltiplas classes de serviço, a outras abordagens de escalonamento cientes de QoS e uma taxonomia proposta para escalonadores.

2.1 Arquitetura do Escalonador

Os provedores de Computação na Nuvem comumente lidam com cargas de trabalho heterogêneas, que são executadas tipicamente em *clusters* de larga escala. Esses *clusters* normalmente são formados por recursos também heterogêneos. A heterogeneidade da infraestrutura combinada com a heterogeneidade da carga de trabalho torna a tarefa de escalonamento nesses ambientes bastante desafiadora. Os algoritmos de escalonamento propostos para *clusters* de larga escala podem ser classificados de acordo com suas arquiteturas. Dessa forma, é possível destacar as seguintes categorias de arquiteturas para esse tipo de escalonador:

(i) *Escalonadores monolíticos* (em inglês, *monolithic schedulers*) tomam suas decisões de forma centralizada, considerando o *cluster* como um todo. Por possuir uma visão geral do estado do *cluster* (infraestrutura e carga de trabalho), muitas verificações normalmente são realizadas enquanto toma decisões. Por esta razão, o tempo necessário para a formulação de uma decisão geralmente é maior para escalonadores deste tipo, e, por isso, são menos escaláveis. No entanto, os escalonadores com essa arquitetura

geralmente tomam as melhores decisões, visto que o estado completo do sistema é conhecido pelo escalonador. Firmament [29], Quincy [33], Quasar [21] e YARN [18; 50] são exemplos de sistemas com essa arquitetura. Em um primeiro momento, o YARN [18; 50] parece seguir o escalonamento em dois níveis. Isso ocorre pelo fato de que cada aplicação tem um componente que negocia por recursos com um gerenciador de recursos global do sistema. No entanto, as solicitações por recursos das diferentes aplicação são enviadas para um único escalonador global no gerenciador de recursos do sistema. Este último escalona recursos em vários servidores de acordo com as restrições das aplicações. Nesse contexto, o componente de cada aplicação promove serviço de gerenciamento de aplicações, não de escalonamento. Assim sendo, o YARN é classificado como um escalonador monolítico;

(ii) *Escalonadores em dois níveis* (em inglês, *two-level schedulers*) possuem um único gerenciador com a visão dos recursos do *cluster* e múltiplas e paralelas instâncias do escalonador. O Mesos [32] é um exemplo de sistema com essa arquitetura. Nesse sistema, o gerenciador particiona dinamicamente os recursos do *cluster* e oferece os recursos disponíveis (não utilizados) para múltiplos escalonadores, que podem aceitar ou não as ofertas com base em suas demandas. Uma vez que uma oferta é aceita pelo escalonador, este é responsável por alocar os recursos para suas demandas.

(iii) *Escalonadores distribuídos* (em inglês, *distributed schedulers*) geralmente são implementados sob um das duas abordagens a seguir. Na primeira delas, diferentes escalonadores executam de forma independente sem informações sobre o estado geral do *cluster*. O Sparrow [40] é um exemplo de implementação dessa abordagem. Este sistema realiza escalonamento sob um conjunto de servidores de forma autônoma e sem um estado centralizado ou logicamente centralizado do *cluster*. Assim, os múltiplos escalonadores que executam em paralelo tomam decisões com base em informações instantâneas obtidas dos servidores. A outra alternativa para escalonadores distribuídos é através do compartilhamento do estado do *cluster*. Nesse caso, os diversos escalonadores operam com acesso ao estado do sistema enquanto possíveis conflitos de concorrência nas decisões são tratados quando identificados. Os escalonadores Apollo [10], Borg [51], Omega [45] e Tarcil [22] são exemplos de escalonadores im-

plementados de acordo com esta abordagem;

(iv) *Escalonadores híbridos* (em inglês, *hybrid schedulers*) possuem diferentes escalonadores no sistema. Uma determinada parte da carga de trabalho é alocada por um escalonador centralizado e outros escalonadores distribuídos tomam decisões para o restante da carga. Os escalonadores Hawk [19] e Mercury [34] são exemplos de escalonadores híbridos. Tipicamente, as tarefas classificadas como as mais importantes são escalonadas pelo escalonador centralizado e o restante da carga é processado pelos escalonadores distribuídos.

Nesse cenário, observa-se que a arquitetura das soluções tem evoluído de uma solução centralizada em direção a uma distribuída (ou híbrida). No geral, as soluções centralizadas conseguem tomar melhores decisões quando o escalonador é executado. Isso é possível porque o escalonador tem acesso às informações sobre as instâncias a serem escalonadas e ao estado geral do *cluster*. No entanto, o escalonador centralizado pode se tornar complexo para lidar com cargas de trabalho heterogêneas, como também pode ser um gargalo para o sistema quando grandes cargas de trabalho são submetidas. No caso de soluções distribuídas, os escalonadores podem tomar decisões que não são tão boas (soluções completamente distribuídas sem estado compartilhado) ou precisam lidar com identificação e controle de conflitos nas decisões (soluções com estado compartilhado). Entretanto, quando uma mesma carga de trabalho é considerada, essas soluções conseguem tomar decisões de forma mais rápida em comparação com uma solução centralizada, elevando a escalabilidade do escalonador. Os principais escalonadores para *clusters* de larga escala [51; 45; 10] utilizam algoritmos de escalonamento proprietários, ou seja, suas implementações reais não são detalhadas nem estão disponíveis publicamente para serem estendidas e implantadas em ambientes diferentes. O Kubernetes [5] é um exemplo de sistema de gerenciamento de contêineres cujo código é aberto. Esse sistema é desenvolvido e mantido primordialmente pela equipe da Google e sofreu forte influência dos sistemas desenvolvidos anteriormente por esta empresa [11] (Borg [51] e Omega [45]).

Sob o ponto de vista arquitetural, um escalonador que implementa a política orientada por QoS (detalhada no Capítulo 3) pode ser desenvolvido seguindo qualquer uma das arquiteturas mencionadas acima. Neste trabalho, tanto os modelos de simulação quanto o protótipo

desenvolvidos (apresentados nos Capítulos 4 e 5, respectivamente) são desenvolvidos com base em uma arquitetura monolítica. Em outras palavras, nesses artefatos há um único escalonador no sistema que toma decisões com base na visão geral do *cluster*. Este tipo de arquitetura foi adotado visto que geralmente tomam as melhores decisões e a escalabilidade do escalonador não foi uma métrica avaliada neste trabalho.

No entanto, caso a escalabilidade se torne um problema, a política de escalonamento proposta pode ser implementada seguindo uma outra arquitetura. Por exemplo, ao invés de existir um único escalonador, a arquitetura dos modelos pode ser evoluída sem muito esforço de forma a permitir a existência de escalonadores distribuídos compartilhando o estado do *cluster* (arquitetura distribuída com compartilhamento do estado). Basicamente, a fila de instâncias pendentes seria processada por mais de um escalonador, e um mecanismo para identificar e controlar conflitos de concorrência nas decisões deve ser implementado. Nesse sentido, o mesmo controle de concorrência otimista utilizado em outras soluções [51; 45] pode ser implementado. Neste cenário, cada réplica do escalonador opera com uma cópia local do estado do *cluster*, e, repetidamente: atualiza as informações do estado; executa a atribuição de recursos de um servidor a uma instância; e informa ao servidor escolhido sobre esta atribuição. Caso a alocação seja apropriada, o servidor alocará recursos para a instância. Caso contrário, por exemplo, porque a decisão foi tomada com base em informações desatualizadas, a alocação dos recursos não acontece e a instância pendente será reprocessada pelo mesmo ou outro escalonador. Schwarzkop et al. [45] mostraram que o uso desta abordagem para o controle de concorrência é uma abordagem viável e atrativa para o escalonamento em *clusters* com arquitetura distribuída com compartilhamento de estado.

Por último, também é importante mencionar que algumas das soluções com arquiteturas distribuídas surgiram a partir de versões anteriores com arquiteturas monolíticas. Por exemplo, o Borg [51] inicialmente foi desenvolvido como um único escalonador centralizado. A partir do momento que este escalonador se tornou um gargalo para o sistema, sua arquitetura foi evoluída para permitir múltiplos escalonadores que compartilham o estado geral do *cluster*. Segundo Burns et al. [11], as lições aprendidas ao longo do desenvolvimento e utilização do Borg tiveram forte influencia nos sistemas desenvolvidos posteriormente pela Google, por isso, desde o início o Omega [45] já foi projetado com uma arquitetura distribuída.

2.2 Escalonamento com Múltiplas Classes de Serviço

Sob a perspectiva do provedor, o cenário ideal é aquele onde a utilização dos recursos é mantida o mais alto possível, elevando sua lucratividade, enquanto o índice de violações de SLAs é mantido o mais baixo possível, reduzindo os gastos com as penalidades.

Uma maneira comum de elevar a utilização dos recursos de um provedor é oferecer a capacidade temporariamente ociosa de maneira oportunista, essencialmente sem garantias de QoS [38] (*i.e.* sem nenhum SLO associado a esses recursos). A ausência de SLOs restringe as aplicações que podem se beneficiar dessas ofertas. Por isso, o preço cobrado por esses recursos é menor em relação aos recursos que são alocados com SLOs associados (*i.e.* com uma expectativa de QoS mínima a ser provida). As instâncias Amazon EC2 Spot [1] e Google Cloud Preemptible [3] são exemplos de recursos oferecidos de forma oportunista. Nesse contexto, Carvalho et al. [13; 15] mostraram que um provedor pode oferecer esses recursos na forma de uma ou mais novas classes de serviço ao invés de oferecê-los apenas de forma oportunista Cada nova classe tem SLOs de longo prazo associados a suas instâncias, permitindo cobrar mais por essas instâncias em comparação a uma outra com recursos oportunistas. Essas classes são úteis especialmente para aplicações que podem aceitar recursos com menos QoS, mas que ainda precisam de garantias de QoS moderadas [13] — por exemplo, aplicações não voltadas à interação com usuários, como indexação web e transcodificação de vídeo. Isso permite que o provedor aumente a utilização dos recursos de forma mais rentável [54].

Com o crescimento da competitividade no mercado de provedores de IaaS, o oferecimento de um número maior de classes de serviço torna-se um diferencial entre os diversos provedores [16]. Cada classe de serviço é associada a um modelo de precificação diferente e a uma QoS mínima esperada. Nesse cenário, os usuários podem escolher as classes de serviço a serem solicitadas de acordo com a QoS realmente necessária para suas aplicações. Alguns usuários podem estar dispostos a pagar por recursos dedicados sempre disponíveis, mas outros podem aceitar uma expectativa menor de QoS em troca de preços mais baixos [13].

Com o objetivo de oferecer múltiplas classes de serviço, diversas políticas de escalonamento associam prioridades distintas para cada classe de serviço. Geralmente, quanto maior é a QoS que precisa ser provida para as instâncias de uma classe, maior é a prioridade

atribuída a esta classe. Com esta abordagem, preempções de instâncias já em execução associadas com prioridades mais baixas podem ocorrer em benefício de instâncias associadas com prioridades mais altas. Por esta razão, um escalonador que implementa esta política é classificado como *preemptivo*. Ao longo do tempo, políticas de escalonamento preemptivas têm sido propostas como uma opção para lidar com cargas de trabalho heterogêneas em áreas como Sistemas Operacionais [49, Capítulo 3], sistemas de tempo real [36] e *clusters* de execução de *jobs* paralelos [27]. No contexto dos principais sistemas de gerenciamento de *clusters* de larga escala, diversos escalonadores adotam uma política baseada em prioridades [10; 11; 18; 22; 23; 34; 45; 50; 51]. Nesta política de escalonamento, a prioridade funciona como um *proxy* para a QoS prometida para as instâncias de determinada classe. Isso facilita as decisões de escalonamento e o gerenciamento de variáveis de decisão, no entanto, como discutido na Introdução deste documento, pode levar a alta variabilidade na QoS entregue pelo sistema, como também a um uso de recursos menos eficiente quando o sistema estiver com contenção de recursos.

O escalonamento orientado por QoS proposto também é uma política de escalonamento *preemptiva*. Porém, ao invés de usar um *proxy* para a QoS prometida para as instâncias, a política proposta toma decisões com base na meta de QoS prometida e na QoS atual de todas as requisições no sistema no momento da tomada de decisão. Este mecanismo de preempção visa respeitar a QoS prometida para todas as classes de serviço simultaneamente, além de provisionar recursos de forma mais justa para requisições de uma mesma classe que estejam competindo pelos mesmos recursos.

Destaca-se que também existem políticas de escalonamento baseada em prioridades que consideram uma prioridade não estática. Nesse caso, a prioridade de uma requisição é computada com base no tempo em que ela esteve na fila de espera [53]. Nesse sentido, a prioridade de uma requisição passa a ser uma métrica dinâmica e pode aumentar continuamente caso a requisição não tenha recursos alocados para a mesma. No entanto, neste caso, não há uma forma direta de mapear prioridades mais altas com classes de serviço que devem oferecer melhor QoS, portanto, esses escalonadores não são utilizados por grandes provedores de nuvem que oferecem múltiplas classes de serviço.

Por fim, quase todos os escalonadores seguem o mesmo algoritmo básico para alocar instâncias que estão na fila à espera por recursos. Sempre que o escalonador é executado,

primeiramente a fila de espera é ordenada seguindo alguma política (por exemplo, a prioridade associada com a instância). Em seguida, a fila de espera é processada, uma instância por vez, e, um servidor adequado é procurado para cada instância. Nesse contexto, a busca pelo servidor ocorre em dois passos principais: *verificação de viabilidade* e *classificação*. Na *verificação de viabilidade*, o escalonador filtra os servidores que estão habilitados para prover recursos para a instância, levando em consideração a quantidade de recursos solicitada e as possíveis restrições de alocação e/ou de afinidade especificadas pelo usuário. As restrições de alocação restringem os servidores onde a instância pode ser alocada, por exemplo, devido a requisitos relacionados a uma versão específica do Sistema Operacional ou tipo de processador. Já as restrições de afinidade restringem os servidores onde a instância pode ser alocada de forma que duas requisições pertencentes a um mesmo grupo (por exemplo, um *job*) não podem ser alocadas em um mesmo servidor. Neta etapa, alguns servidores podem ter recursos suficientes para alocar a instância sem a necessidade de preempções. Outros servidores tornam-se elegíveis condicionados à preempção de uma ou mais instâncias já em execução.

Já na etapa de *classificação*, o escalonador usa uma função de pontuação para escolher o servidor mais adequado do conjunto de servidores elegíveis, selecionado na etapa anterior. Esta função atribui uma pontuação a cada servidor elegível e deve ser implementada de acordo com os objetivos do provedor (por exemplo, consolidação da carga de trabalho, balanceamento na alocação de recursos dos servidores ou redução na quantidade de recursos fragmentados). No contexto dos sistemas de gerenciamento de larga escala, Borg [51], Omega [45], Kubernetes [11], Apollo [10] e Sparrow [40] são exemplos de sistemas cujos escalonadores operam em duas etapas. O escalonamento orientado por QoS proposto neste trabalho também é executado em duas etapas, igualmente denominadas de verificação de viabilidade e classificação.

2.3 Escalonamento ciente de QoS

Vários escalonadores para sistemas de computação na nuvem usam QoS para direcionar as decisões de escalonamento, no entanto, quase todos eles utilizam QoS em um contexto diferente daquele usado neste trabalho. Por exemplo, um dos possíveis cenários

é utilizar a QoS associada às aplicações dos usuários, as quais executam nos recursos oferecidos pelo provedor. Existem políticas de escalonamento que direcionam suas decisões com base em parâmetros de QoS das aplicações dos usuários [30; 35; 20; 21; 22]. Neste caso, o escalonador utiliza técnicas de classificação para identificar o impacto que as diferentes decisões de alocação teriam no desempenho das aplicações. Por isso, o escalonador monitora métricas de QoS das aplicações e as utiliza na tomada de decisões. Por exemplo, quando ocorre a degradação da QoS de aplicações que executam em um servidor específico (*e.g.* o tempo de resposta das aplicações passam a não atender os níveis aceitáveis), o escalonador decide não mais alocar outras instâncias neste mesmo servidor e/ou até mesmo migrar instâncias para outros servidores. Por um lado, esses escalonadores permitem um controle mais preciso da QoS fornecida pelas próprias aplicações. Por outro lado, esses escalonadores precisam conhecer completamente as especificidades das aplicações que executam no sistema, o que os torna bastante específico. Por esta razão, esses escalonadores não são utilizados por grandes provedores de nuvem que executam cargas de trabalho arbitrárias e muito heterogêneas. A política de escalonamento proposta neste trabalho toma decisões com base na QoS prometida pelo provedor para cada requisição admitida; *i.e.*, para as instâncias no sistema. Isso ocorre independente do tipo de aplicação que é executada pelo usuário nos recursos provisionados para essas requisições.

O termo QoS também pode estar relacionado com parâmetros do provedor, tais como utilização e/ou alocação balanceada dos recursos, tempo de tomada de decisão, quantidade de requisições processadas por unidade de tempo, etc. Alguns desses parâmetros tipicamente são utilizados por políticas de escalonamento [51; 45; 11; 40; 10; 21; 20; 22] que buscam maximizar ou minimizar um ou mais parâmetros de QoS do provedor enquanto seleciona o servidor mais adequado para a alocação de uma instância (etapa de classificação discutida na seção anterior). Por esta razão, esses escalonadores também são classificados como escalonadores cientes de QoS por alguns autores. Diferentemente desses escalonadores, o escalonamento proposto neste trabalho adota a orientação por QoS não apenas na etapa de seleção do servidor, mas também quando a fila de espera é ordenada e na etapa de verificação de viabilidade. Além disso, em alguns casos, as métricas de QoS utilizadas para a seleção do servidor são novas. Particularmente, nos modelos de simulação e no protótipo implementados, duas funções de prioridade são utilizadas para escolher o

servidor mais adequado para prover recursos para uma dada requisição: "Prioridade Menos Solicitada" e "Alocação Balanceada de Recurso". Ambas as funções são utilizadas com o mesmo peso e detalhadas adiante na Seção 4.1.2. Essas funções foram implementadas por serem funções padrões no sistema Kubernetes [5]. No entanto, destaca-se que a otimização deste tipo de parâmetro de QoS não é o objetivo principal do escalonador proposto. Note que a configuração de alocação que resulta no cenário com melhores parâmetros de QoS para o provedor (por exemplo, menor fragmentação de recursos[1]), pode não ser a configuração que resultará na menor quantidade de violações de SLAs. Por esta razão, em períodos com contenção de recursos, a otimização desse tipo de parâmetros de QoS do provedor fica em segundo plano no escalonamento orientado por QoS.

A QoS dos recursos do provedor também podem ser levados em conta nas decisões do escalonador. Alguns algoritmos de escalonamento [46; 31] consideram a QoS dos recursos enquanto os aloca para as instâncias das requisições. Nesse cenário, servidores com menor valor para o parâmetro de QoS considerado (por exemplo, servidor com maior probabilidade de falhar) tem mais chances de serem escolhidos para prover recursos para atender requisições com SLOs menos exigentes. Neste trabalho, esse tipo de parâmetro de QoS não é levado em conta pelo escalonador orientado por QoS. No entanto, uma terceira função de prioridade que analisasse esse aspecto do servidor poderia ser implementada e utilizada na etapa de classificação do escalonamento.

Além disso, Shahrad e Wentzlaff [46] propõem um modelo de nuvem no qual os consumidores podem solicitar SLOs de disponibilidade customizados para os provedores. Nesse cenário, o provedor busca atender os SLOs definidos pelos usuários dentro de uma janela de tempo específica (por exemplo, um período mensal de cobrança). Os autores argumentam que essa abordagem possibilita mercados mais eficientes. Similar ao escalonamento proposto neste trabalho, o escalonador utilizado por Shahrad e Wentzlaff [46] também toma decisões com base na QoS atualmente entregue e nos SLOs prometidos (no caso deles, SLOs solicitados). No entanto, o trabalho é focado em aspectos econômicos do modelo proposto e usa um escalonador muito simples. Em particular, o escalonamento no modelo proposto por eles é baseado na probabilidade de falhas dos servidores. Dessa forma, o escalonador pode

[1] A fragmentação de recursos ocorre através de pequenas quantidades de recursos que sobram nos servidores por não serem grandes o suficiente para satisfazer nenhuma das requisições admitidas.

migrar VMs para servidores com maior probabilidade de falhas (mais baratos) se a falha deste servidor não comprometer o tempo máximo que a VM pode ficar inativa (para cumprir o SLO solicitado). O escalonador também pode deliberadamente preemptar instâncias de forma proposital, pausando as VMs que não terão seus SLOs não atendidos se ficarem inativas até o final da janela de tempo específica. Isso é feito para evitar que as instâncias recebam QoS maior do que a contratada. Contudo, diferentemente da política orientada por QoS, o escalonador proposto por eles não consegue preemptar recursos de uma instância em execução em benefício de outra instância pendente, independente dos SLOs das instâncias envolvidas. Isso diminui os ganhos que podem ser alcançados com o uso de escalonamento orientado por QoS. Além disso, o escalonador proposto neste trabalho também não reduz a QoS das instâncias quando houver recursos disponíveis para que executem, *i.e.* a QoS entregue para as instâncias será a maior possível.

Finalmente, o escalonador orientado por QoS também pode ser associado com o clássico algoritmo de escalonamento *Earliest Deadline First* (EDF). Este é um algoritmo bem estudado na área de sistemas em tempo real [36; 39; 17]. Neste algoritmo, a prioridade de uma tarefa é inversamente proporcional à diferença entre seu tempo máximo para conclusão e o tempo atual. Assim sendo, a prioridade dinâmica de uma tarefa aumenta monotonicamente. Neste trabalho, o escalonador orientado por QoS utiliza a disponibilidade da requisição como métrica de QoS de interesse. Nesse sentido, ele toma decisão levando em conta métricas de QoS (detalhadas adiante na Seção 3.3) que indicam por quanto tempo uma requisição pode não ter recursos alocados antes de ter seu SLO violado, ou o tempo excedente que uma requisição esteve pendente desde de que seu SLO começou a ser violado. Caso os valores dessas métricas sejam adequadamente consideradas como *deadlines* e as requisições como tarefas a serem executadas, o escalonador orientado por QoS também pode ser classificado como um tipo de escalonador EDF. No entanto, os valores das métricas utilizadas pelo escalonador proposto podem tanto aumentar quanto reduzir, a depender do estado das instâncias associadas com a requisição em questão.

2.4 Classificação Taxonômica

Há centenas de trabalhos direcionados a problemas de escalonamento em sistemas distribuídos, o que dificulta na classificação dos mesmos. Lopes e Menascé [37] propuseram uma taxonomia para classificar escalonadores, com base nas funcionalidades, eles classificam mais de 100 artigos em 10 grupos. De acordo com esta taxonomia, o escalonador orientado por QoS proposto neste trabalho pode ser classificado como apresentado na Tabela 2.1.

A política de escalonamento orientada por QoS é capaz de escalonar diferentes e diversos *jobs* que podem chegar a qualquer tempo de múltiplos usuários. Da forma que está definido atualmente, o escalonador lida com tarefas individualmente, ou seja, *jobs* com uma única tarefa ou com tarefas independentes (heterogêneas ou homogêneas). Neste último caso, o escalonador atribui tarefas para os servidores uma por uma, observando a carga da infraestrutura no momento. Os *jobs* executam em instâncias (VMs ou contêineres) dedicadas com uma quantidade fixa de recursos solicitados e não podem reservar menos ou mais recursos. Este escalonador é ciente de SLO, mas não é preparado para lidar com *jobs* de tempo real. A infraestrutura considerada é local, com um único domínio e pode ser homogênea ou heterogênea. Além disso, o escalonador trabalha a nível de tarefa, decidindo qual tarefa de um dado *job* deve executar e em qual servidor a mesma deve ser alocada. O escalonador também não considera localidade de dado e é capaz de reiniciar as instâncias alocadas em recursos que tenham falhado. Por último, este escalonador é estático no sentido de que sua política de escalonamento não é alterada ao longo do tempo. Ainda de acordo com a mesma taxonomia, esta política é sub-ótima, especialmente porque o escalonador opera online sem conhecimento do futuro e as decisões de escalonamento são realizadas em resposta aos eventos. A topologia do escalonador é distribuída: a decisão é centralizada e as requisições são enviadas aos servidores (*push-based*) pelo escalonador. Finalmente, como um dos pontos principais desta solução, o escalonador é flexível: preempções e migrações ocorrem como parte da política de escalonamento.

Tabela 2.1: Caracterização da política de escalonamento orientada por QoS

Propriedade	**Valor**
Origem da carga de trabalho	Multi-usuário e multi-*job*
Estrutura do *Job*	Única tarefa
Flexibilidade do *Job*	Rígido
Processo de chegada	Aberto
Composição da carga de trabalho	Heterogênea
Qualidade do Serviço	Ciente de SLO
Tempo Real	Não
Heterogeneidade dos Recursos	Qualquer
Escala	Estática
Compartilhamento de Recursos	VMs/contêineres dedicados
Cobertura Geográfica	Local
Federação	Único domínio
Meta do Escalonamento	Cumprimento dos SLOs e Justiça
Nível	Nível de tarefa
Localidade do dado	Sem afinidade
Modelo de Falha	Recuperação à falha
Adaptabilidade	Estática
Operação	Online
Topologia	Distribuída, centralizada e *push-based*
Flexibilidade	Flexível: ciente de migração, preemptivo

Capítulo 3

Escalonamento Orientado por QoS

Neste capítulo é apresentada a política de *Escalonamento Orientado por QoS*. Esta política toma decisões levando em conta a QoS que está sendo entregue para cada requisição no momento da tomada de decisão e suas respectivas metas de QoS. Este escalonador tem como objetivo cumprir as expectativas de QoS, independente da classe de serviço solicitada, além de promover um provisionamento mais justo entre requisições de uma mesma classe que competem pelo uso dos mesmos recursos.

3.1 Introdução

Essencialmente, a missão do escalonador de um provedor de computação na nuvem é decidir, no momento de sua execução, quais das requisições devem ter suas respectivas instâncias alocadas e quais servidores devem prover recursos para essas instâncias. De forma genérica, uma requisição pode solicitar recursos para uma ou mais instâncias, consequentemente, cada requisição estará associada com a quantidade de instâncias por ela solicitada.

Em períodos com contenção de recursos, uma requisição pode ser atendida parcialmente, *i.e.* ter recursos alocados para algumas de suas instâncias, mas não todas. Nesse cenário, quando não há recursos suficientes para alocar todas as instâncias de todas as requisições admitidas, tipicamente algumas instâncias são mantidas em uma *fila de espera* aguardando que recursos sejam liberados. Em um dado instante no tempo, as instâncias que têm recursos alocados são classificadas como instâncias *em execução*, enquanto que as mantidas na fila de espera são chamadas de instâncias *pendentes*. No caso de *escalonadores preemptivos*, o

escalonador pode interromper (preemptar) uma instância em execução. Uma vez que ocorre uma preempção, os recursos que estavam alocados para a instância preemptada são liberados e podem ser alocados para uma das instâncias pendentes.

Com o intuito de aumentar a utilização dos recursos e satisfazer diferentes perfis de usuários, os provedores geralmente oferecem múltiplas classes de serviço. Cada uma das classes é associada a um esquema diferente de precificação e uma QoS mínima esperada. A QoS prometida para uma classe em particular é estipulada através de metas específicas estabelecidas por SLOs. Os SLOs de cada classe são definidos em seu SLA correspondente, acordado entre o provedor e seus clientes na contratação do serviço. Visto que cada requisição admitida está relacionada com uma classe, os SLOs definidos no SLA da classe da requisição indicam a QoS mínima esperada para cada requisição.

Os provedores se esforçam para atender os SLOs de cada requisição admitida, consequentemente, cumprindo os SLAs. Quando a QoS prometida para as classes não é entregue pelo provedor, ocorrem violações dos SLAs. Por exemplo, em períodos com alta contenção de recursos, o provedor pode não ser capaz de satisfazer todos os SLOs de todas as requisições no sistema. Esses períodos podem ocorrer por causa de aumento inesperado da demanda, de falha de servidores e/ou comportamentos inadequados nas etapas de planejamento de capacidade e/ou controle de admissão. Nesse cenário, o provedor está sujeito a pagar penalidades por essas requisições que tiveram seus SLAs violados. A política de *Escalonamento Baseada em Prioridade* é utilizada por diversos escalonadores que suportam múltiplas classes de serviço em provedores de computação na nuvem [10; 11; 18; 22; 23; 34; 45; 50; 51]. Como já discutido na introdução desta tese, o escalonamento baseado em prioridades pode tomar decisões inadequadas em algumas situações.

Uma nova política de escalonamento é proposta neste trabalho, denominada de *Escalonamento Orientado por QoS*. Um escalonador que implementa esta política toma decisões levando em conta a QoS entregue para cada requisição no sistema no momento da tomada de decisão, bem como suas respectivas metas de QoS. Em determinado ponto no tempo, a QoS entregue para uma requisição é representada pelo SLI da métrica de QoS, *i.e.* o SLI representa a medição atual para uma métrica de QoS. A ideia central desta política é que, quando não for possível alocar recursos para todas as requisições admitidas, aquelas requisições cuja QoS esteja excedendo sua respectiva meta tenham recursos preemptados em benefício

de outras com QoS abaixo da meta (ou que estejam mais próximas de violar seus SLAs). O restante deste capítulo discute como o escalonador orientado por QoS opera para entregar QoS e provisionamento de recursos mais justo para as requisições admitidas ao longo do tempo.

3.2 Conceitos Básicos

Uma importante característica do escalonador orientado por QoS é ser um *escalonador preemptivo*. Assim, quando não é possível alocar recursos para as instâncias de todas as requisições admitidas, algumas delas são mantidas em uma *fila de pendentes*. As instâncias na fila ou foram preemptadas em algum momento ou nunca executaram. Essas instâncias são mantidas na fila até que o escalonador decida executá-las, alocando os recursos solicitados em algum servidor da infraestrutura.

O escalonador monitora a QoS atual entregue para cada requisição no sistema, bem como sua classe de serviço e meta de QoS. Este mecanismo é genérico o suficiente para ser implementados sob o ponto de vista de qualquer métrica de QoS, ou até mesmo considerar mais de uma métrica. Neste trabalho, a *disponibilidade da requisição* — o percentual de tempo que as instâncias de uma requisição estiveram alocadas em um servidor contribuindo para aumentar a QoS da requisição — é considerada como a única métrica de QoS de interesse. Esta escolha é justificada pelo fato de a disponibilidade ser uma das principais preocupações de consumidores de computação em nuvem [46], como também estar presente em 73% dos SLAs negociados entre consumidores e provedores [41]. Portanto, considera-se que cada classe de serviço oferecida pelo provedor é associada com um SLA que inclui um único SLO de disponibilidade. Uma vez que cada requisição admitida também está relacionada com uma classe, o SLO de disponibilidade definido para classe da requisição indica a QoS mínima esperada para cada requisição. Além disso, após o término da execução de uma requisição, se a disponibilidade entregue para a mesma foi menor que seu SLO, assume-se que seu SLA foi violado, levando o provedor a pagar uma penalidade. A partir deste ponto, o SLO de disponibilidade de uma requisição será referenciado apenas por SLO.

No caso de requisições associadas com mais de uma instância, é importante destacar que não necessariamente uma instância em execução estará contribuindo para melhorar a

disponibilidade de sua requisição. A semântica dada a disponibilidade de uma requisição associada com múltiplas instâncias impacta nos períodos que o provedor interpreta que o serviço está sendo entregue a uma requisição. Por esta razão, o tempo de execução de uma instância pode ser classificado em períodos de *contribuição* e *não contribuição* para a melhoria da disponibilidade de sua requisição. Por exemplo, suponha que o provedor considere que o serviço está sendo entregue para uma requisição apenas quando todas as suas instâncias estiverem executando simultaneamente. Neste caso, se ao menos uma das instâncias de uma requisição estiver pendente, as outras que estão em execução não estarão contribuindo para melhorar a disponibilidade desta requisição. Por outro lado, se o provedor considerar que a disponibilidade de uma requisição é dada pela disponibilidade média entregue para todas as suas instâncias, sempre que uma instância estiver em execução ela estará contribuindo para melhorar a disponibilidade de sua requisição.

Considere que $e_j(t)$ é o tempo acumulado em que todas as instâncias da requisição j estiveram em execução e efetivamente contribuíram para melhorar a QoS de j, desde sua admissão até um instante no tempo t; $d_j(t)$ é a quantidade de tempo que as instâncias da requisição j estiveram em execução mas não contribuíram para a melhoria da QoS de j até o tempo t (*i.e.* a quantidade acumulada de tempo de execução descartada das instâncias de j até t); e, $p_j(t)$ é o tempo acumulado em que as instâncias da requisição j estiveram pendentes até t. Portanto, no instante de tempo t, a disponibilidade da requisição j, $A_j(t)$, é dada pela seguinte equação:

$$A_j(t) = \frac{e_j(t)}{e_j(t) + d_j(t) + p_j(t)} \tag{3.1}$$

É possível usar a equação acima para calcular a disponibilidade de uma requisição em duas situações: (i) para calcular a disponibilidade atual entregue para uma requisição no tempo t; e, (ii) para calcular a disponibilidade final entregue para uma requisição que foi terminada (completou sua execução). Esta última é usada para avaliar se o SLA da requisição foi cumprido ou não. Neste caso, t é o instante de tempo que a requisição foi terminada — note que este tempo não é conhecido previamente pelo escalonador.

3.3 Métrica de QoS para decisão

O escalonamento orientado por QoS tem como objetivo manter a QoS entregue para todas as requisições admitidas igual ou acima de seus respectivos SLOs. Além disso, a nova política também pretende promover provisionamento mais justo para requisições de uma mesma classe, particularmente em períodos com contenção de recursos. Nesse sentido, o mecanismo de preempção da política proposta permite preempções de instâncias em benefício de outras instâncias de mesma classe, buscando reduzir a variância da QoS de suas requisições. No entanto, é inadequado utilizar diretamente a disponibilidade atual das requisições para decidir qual delas deve ter instância(s) alocada(s).

É possível ilustrar isso através de um exemplo simples. Suponha a existência de duas requisições j e k, cada uma associada com apenas uma instância e com o mesmo SLO de disponibilidade de 90%. Além disso, admita que o provedor considera que sempre que uma instância executa ela contribui para aumentar a disponibilidade de sua requisição. Em determinado instante no tempo, o provedor precisa escolher apenas uma dentre as requisições j e k para ter recursos alocados à sua instância. A requisição j chegou no sistema 1 hora atrás e sua instância executou por 58 minutos. Com base na Equação 3.1, a disponibilidade de j é de $96,6\%$. A requisição k está no sistema por 10 minutos e sua instância tem executado desde que k foi admitida (*i.e.* por 10 minutos). Portanto, a disponibilidade de k é 100%. Se apenas as disponibilidades das requisições forem consideradas para decidir qual delas deve ter sua instância pendente, a escolha seria a requisição k, visto que sua disponibilidade é maior. No entanto, após aproximadamente $1,1$ minuto na fila de espera, a requisição k teria disponibilidade de $10/11,1 = 90,09\%$ e sua instância teria que voltar a executar, caso contrário seu SLO seria violado. Por outro lado, a instância de j poderia ficar na fila de espera por até $4,4$ minutos, antes que seu SLO fosse violado.

O exemplo acima é importante para ilustrar que quando as instâncias ficam pendentes, as disponibilidades de suas requisições são decrementadas em velocidades diferentes. Quanto mais tempo as instâncias de uma requisição tiverem executado e contribuído para sua disponibilidade no passado, mais tempo a requisição pode receber uma determinada ou nenhuma contribuição para sua QoS antes de ter seu SLO não satisfeito.

Esse entendimento leva à definição de uma nova métrica chamada de *Time-to-Violate*

(TTV) de uma requisição. Esta métrica indica um intervalo de tempo em que uma requisição j pode permanecer recebendo uma contribuição c_j para sua QoS antes de ter seu SLO violado. Por exemplo, considerando que a requisição j está associada com apenas uma instância, o TTV de j indica por quanto tempo j pode ficar sem receber nenhuma contribuição para sua disponibilidade antes de ter seu SLO não atendido. No caso de j estar associada com múltiplas instâncias (n, onde $n > 1$), as instâncias que já contribuem para melhorar a disponibilidade de j em um instante de tempo podem continuar contribuindo ao longo do TTV. Portanto, supondo que m instâncias de j ($m < n$) continuarão contribuindo para a QoS de j ao longo do TTV, o TTV resultará em um intervalo de tempo onde j poderá continuar recebendo a contribuição das m instâncias para sua QoS (*i.e.* c_j) antes ter seu SLO não atendido.

Portanto, para uma instância que está em execução, o TTV indica por quanto tempo ela pode, a partir do instante atual, ficar na fila de espera antes que sua requisição tenha seu SLO violado (*i.e.*, caso a instância fosse preemptada neste instante). Para uma instância pendente, o TTV indica por quanto tempo a instância pode permanecer na fila sem que sua requisição tenha seu SLO violado. Assim sendo, o TTV associado às instâncias pendentes é monitorado e, idealmente, não deve atingir valores próximos de zero. Valores próximos de zero indicam que a requisição está próxima de não atender seu SLO.

Considere que j é uma requisição submetida para a classe de serviço i, cujo SLO de disponibilidade é O_i. Além disso, admita que $v_j^r(t)$ e $v_j^w(t)$ são, respectivamente, o número de instâncias da requisição j em execução e pendentes em um ponto no tempo t. O intervalo de tempo que j pode permanecer recebendo determinada contribuição para incrementar sua QoS sem que seu SLO seja violado é representado por $\Delta t_j(t)$. Nesse cenário, c_j indica a contribuição que a requisição j estará recebendo para melhorar sua disponibilidade ao longo do intervalo $\Delta t_j(t)$. Essa contribuição c_j é definida pelo número de instâncias de j que, ao longo de $\Delta t_j(t)$, continuarão efetivamente contribuindo para sua QoS. Assumindo que a disponibilidade atual de j no tempo t é maior ou igual à prometida ($A_j(t) \geq O_i$), a Equação 3.1 pode ser utilizada para descobrir quando a meta de disponibilidade O_i da requisição j será alcançada.

$$O_i = \frac{e_j(t) + c_j \Delta t_j(t)}{e_j(t) + d_j(t) + p_j(t) + (v_j^r(t) + v_j^w(t))\Delta t_j(t)}. \tag{3.2}$$

Como observado acima, o numerador da Equação 3.1 é incrementado com o tempo que as instâncias da requisição j que estarão em execução ao longo de $\Delta t_j(t)$ estarão contribuindo efetivamente para aumentar sua disponibilidade. Já o denominador da mesma equação é incrementado pelo tempo que todas as instâncias de j permanecerão no sistema, estejam elas pendentes ou em execução. Assim, $\Delta t_j(t)$ pode ser calculado como apresentado na Equação 3.3 a seguir.

$$\Delta t_j(t) = \frac{e_j(t) - O_i(e_j(t) + d_j(t) + p_j(t))}{(p_j^r(t) + p_j^w(t))O_i - c_j}, \forall j | A_j(t) \geq O_i. \tag{3.3}$$

Na prática, uma vez que uma instância de uma requisição j é escolhida para deixar a fila de espera, essa instância precisa ser alocada no servidor escolhido. Esta alocação requer, minimamente, o carregamento dos dados da instância para a memória. A instância de j estará pronta para ser executada apenas após esse *tempo de alocação* tiver decorrido. Assim, o cálculo do TTV deve levar em consideração esse tempo. O tempo de alocação de j pode ser mais longo ou mais curto a depender da instância já ter executado anteriormente no servidor escolhido ou não. Considere que α_j é o tempo máximo de alocação esperado para preparar o servidor para executar uma instância de j. Idealmente, o escalonador deve remover a instância de j da fila de espera antes que $\Delta t_j(t) - \alpha_j = 0$, caso contrário, a instância terá seu SLO momentaneamente não satisfeito. Assim o TTV de uma requisição j no tempo t, $\Delta t_j^*(t)$, é computado como definido na Equação 3.4.

$$\Delta t_j^*(t) = \Delta t_j(t) - \alpha_j. \tag{3.4}$$

No entanto, quando o sistema estiver enfrentando temporariamente um pico de demanda, é possível que o provedor não consiga entregar a QoS prometida para algumas ou até mesmo todas as requisições. Nesse cenário, não faz sentido calcular o TTV para requisições que já estejam recebendo QoS abaixo da prometida, visto que elas já têm seus respectivos SLOs não satisfeitos. Para essas requisições ($A_j(t) < O_i$), uma outra métrica é definida, denominada *recoverability* da requisição. Esta métrica indica por quanto tempo as instâncias de uma requisição estiveram pendentes desde que seu SLO passou a não ser atendido (*i.e.* o tempo excedente em fila de suas instâncias), dando uma ideia de quão recuperável é a requisição.

No tempo t, $\Delta r_j(t)$ indica a quantidade de tempo que as instâncias da requisição j estiveram pendentes desde que seu SLO passou a não ser satisfeito até t. De forma semelhante

ao intervalo $\Delta t_j(t)$, a Equação 3.1 pode ser utilizada para descobrir quando a meta de disponibilidade O_i da requisição j foi alcançada.

$$O_i = \frac{e_j(t)}{e_j(t) + d_j(t) + p_j(t) + \Delta r_j(t)}. \tag{3.5}$$

Assim, $\Delta r_j(t)$ pode ser calculado como apresentado na Equação 3.6 a seguir.

$$\Delta r_j(t) = \frac{e_j(t)}{O_i} - (e_j(t) + d_j(t) + p_j(t)), \forall j | A_j(t) < O_i. \tag{3.6}$$

Destaca-se que $\Delta t_j(t)$ nunca assumirá valores negativos, enquanto $\Delta r_j(t)$ sempre assumirá valores negativos. Quanto maior o valor absoluto desta métrica, mais distante a requisição está de recuperar sua disponibilidade a fim de ter seu SLO atendido.

Como já discutido nesta seção, na prática, após um servidor ser escolhido para prover recursos para uma instância, a instância estará pronta para a execução no servidor após um tempo de alocação tiver decorrido. Portanto, mesmo que uma requisição esteja recebendo menos disponibilidade que a prometida (*i.e.* suas instâncias estiveram pendentes por mais tempo do que poderiam), suas instâncias necessariamente estarão pendentes por mais um tempo extra (o tempo de alocação das mesmas). Assim, faz sentido que a *recoverability* seja calculada de forma conservadora e leve em conta o tempo extra que é necessário para alocar a instância de uma requisição no servidor. Portanto, a *recoverability* de uma requisição j no tempo t, $\Delta r_j^*(t)$, é computada como definido na Equação 3.7

$$\Delta r_j^*(t) = \Delta r_j(t) - \alpha_j. \tag{3.7}$$

Por fim, conhecendo a disponibilidade $A_j(t)$ de todas as requisições no sistema no tempo t, a métrica de QoS $Q_j(t)$ é calculada como segue:

$$Q_j(t) = \begin{cases} \Delta t_j^*(t), & \text{se } A_j(t) \geq O_i \\ \Delta r_j^*(t), & \text{caso contrário} \end{cases}$$

Embora as métricas apresentadas estejam associadas com requisições ativas sistema, quando o escalonador é executado, ele decide sobre alocação e/ou preempção de instâncias. Por esta razão, ao executar o escalonador no tempo t, uma métrica $Q_j(t)$ é associada com

cada instância (pendente ou executando) no sistema. Conhecendo essas métricas, o escalonador pode decidir quais das instâncias em execução devem ser preemptadas (se necessário), e quais das instâncias pendentes devem passar a executar.

Destaca-se que apesar deste trabalho considerar a disponibilidade como única métrica de QoS de interesse, é importante destacar que outras métricas relacionadas ao provisionamento de recursos também poderiam ser utilizadas. Além disso, diferentes métricas também poderiam ser utilizadas em conjunto, com pesos específicos para cada uma delas na tomada de decisão do escalonador.

3.4 Política de Escalonamento

Sempre que o escalonador é executado, inicialmente ele ordena a fila de espera em ordem crescente das métricas de QoS associadas com as instâncias pendentes. Em seguida, o escalonador processa uma instância por vez, tentando encontrar um servidor capaz de alocar cada instância pendente. Como outros escalonadores propostos na literatura [51; 45; 10; 11; 40], quando o escalonador precisa decidir onde alocar uma instância pendente da fila de espera, ele executa dois passos principais: *verificação de viabilidade* e *classificação*. Como discutido no Capítulo 2 (Seção 2.2), na etapa de *verificação de viabilidade*, o escalonador filtra os servidores elegíveis para prover recursos para a instância, enquanto que na etapa de *classificação* o escalonador ordena os servidores elegíveis de acordo com algum critério definido pelo provedor.

3.4.1 Verificação de Viabilidade

Nesta etapa, um servidor é considerado habilitado para alocar uma instância pendente da requisição j se: (i) não violar as possíveis restrições de alocação e/ou afinidade de j; e, (ii) se sua capacidade livre adicionada com os recursos que podem ser liberados por preempções são suficientes para alocar a instância de j. Quando a capacidade livre já é suficiente, não é necessário verificar por instâncias a serem preemptadas. Caso contrário, o escalonador precisa decidir se uma instância de uma requisição k em execução no servidor h pode ser preemptada em benefício da instância de j pendente. Nesse sentido, no tempo t, a instância de k pode ser preemptada apenas se a métrica de QoS de j for menor que a métrica de QoS

de k ($Q_j(t) < Q_k(t)$).

Na busca por instâncias que são elegíveis para ter seus recursos preemptados, o escalonador inicia avaliando a instância k em execução no servidor h com o maior valor para a métrica de QoS ($Q_k(t)$). Se houver múltiplas instâncias com o maior valor, então uma delas é escolhida aleatoriamente.

Além disso, o escalonador considera uma *margem de segurança* específica para cada classe de serviço σ_i para a métrica de QoS. Admitindo que a classe de serviço da requisição k é i, então σ_i é a margem de segurança para a métrica de QoS de qualquer requisição k da classe i. Para efeito de representação, denota-se i_k como a classe de serviço da requisição k e σ_{i_k} como a margem de segurança da classe da requisição k. Assim sendo, instâncias de requisições com valores da métrica de QoS menores que a margem de segurança correspondente (*i.e.* $Q_k(t) < \sigma_{i_k}$) não devem ser preemptadas.

O procedimento discutido acima não faz diferença entre as classes de serviço, tratando todas as instâncias da mesma forma. Esse comportamento é bom quando os SLAs de todas as requisições podem ser cumpridos, mas pode não ser o caso quando o sistema estiver enfrentando temporariamente um pico de demanda. Isso pode levar ao não atendimento de SLOs de requisições de algumas classes de serviço. Dependendo de como os SLAs são definidos, é possível que o provedor queira dar importâncias diferentes para as diferentes classes e evitar as chances de violar os SLAs de determinadas classes, consideradas as mais importantes (por exemplo, porque o não cumprimento do SLA para essas classes leva a penalidades mais pesadas).

Este problema é abordado permitindo que as classes de serviço sejam classificadas de acordo com sua importância. Por exemplo, um provedor pode definir que classes que prometem SLOs mais altos são mais importantes que aquelas que prometem SLOs menores. O objetivo é prestar um melhor serviço para as classes mais importantes, preservando a justiça temporal dentro de cada classe. Como mencionado antes, isso é especialmente crítico durante períodos de alta contenção de recursos. Dessa forma, relaxam-se as regras de verificação de viabilidade que proíbem a preempção de instâncias associadas com métrica de QoS abaixo de suas correspondentes margens de segurança. No tempo t, uma instância de uma requisição k da classe i_k pode ser preemptada em benefício da instância de uma requisição j da classe i_j em duas situações adicionais: (i) as métricas de QoS de ambas estão abaixo de

suas margens de segurança correspondentes ($Q_k(t) < \sigma_{i_k}$ e $Q_j(t) < \sigma_{i_j}$) e a requisição j é de uma classe mais importante que a classe da requisição k; ou, (ii) as métricas de QoS de ambas estão abaixo de suas margens de segurança correspondentes, as requisições k e j são igualmente importantes, e, a métrica de QoS da requisição j é menor que a da requisição k ($Q_j(t) < Q_k(t)$).

Se nenhum servidor for considerado elegível, a instância pendente não pode ser alocada e continuará na fila de espera. Caso contrário, a etapa de classificação é executada.

3.4.2 Classificação

A etapa de classificação inicia dividindo os servidores elegíveis em dois conjuntos distintos: servidores que *não necessitam de preempções* e servidores que *necessitam de preempções* para alocar uma instância pendente de uma requisição j. Se o primeiro conjunto não estiver vazio, o servidor selecionado será deste conjunto. Neste caso, uma *função de pontuação de alocação* configurável é utilizada para comparar os servidores deste conjunto, e o servidor com maior pontuação é selecionado. A função de pontuação implementada neste trabalho é descrita em detalhes mais adiante na Seção 4.1.3.

Se o conjunto de servidores que não necessitam de preempções estiver vazio, todos os servidores elegíveis exigem que preempções ocorram para alocar a instância pendente. Neste caso, a pontuação de um servidor é calculada usando uma *função de pontuação de custo de preempção* configurável. Esta função fornece uma avaliação dos custos das preempções necessárias para a alocação da instância em um servidor. Neste caso, o servidor com menor pontuação é selecionado. A função implementada neste trabalho é discutida mais adiante na Seção 4.1.3.

3.5 Escalonamento em ação

Muitos eventos podem desencadear a execução do escalonador. Quando uma *nova requisição* é admitida, o escalonador inicia o valor de sua métrica de QoS com zero e insere suas respectivas instâncias na fila de espera. Da mesma forma, quando *um servidor falha ou é retirado da infraestrutura*, seja para manutenção ou de forma definitiva, o escalonador insere na fila de espera todas as instâncias que estavam em execução neste servidor. Outros even-

tos estão relacionados com o aumento na quantidade de recursos disponíveis. Por exemplo, quando *um servidor recupera-se de falha ou novas máquinas são adicionadas à infraestrutura*. Do mesmo modo, quando *uma requisição completa sua execução*, os recursos alocados para suas instâncias são liberados. Por fim, como a métrica de QoS sofre alteração ao longo do tempo, o escalonador também deve ser executado *periodicamente*.

Sempre que um dos eventos mencionados aciona a execução do escalonador, este recalcula a métrica de QoS associada com cada instância no sistema e ordena a fila de espera. Esta ordenação ocorre em ordem crescente das métricas de QoS associadas com as instâncias enfileiradas, ou seja, do menor valor (início da fila) para o maior (final da fila). O menor valor indica que a requisição da instância está mais distante de recuperar sua QoS (em caso de valor negativo) ou está mais próxima de ter seu SLO violado (em caso de valor positivo), portanto, esta instância deverá ser alocada primeiro. Em seguida, o escalonador processa a fila de espera completa, uma instância por vez, executando os passos de verificação de viabilidade e classificação como descritos anteriormente. O Algoritmo 1 mostra o pseudocódigo do escalonador orientado por QoS.

3.6 Custo do Escalonamento

3.6.1 Complexidade do Escalonamento

A etapa de verificação de viabilidade é a atividade mais custosa do escalonador. A parte central desta verificação tem uma complexidade de tempo na ordem de $\mathcal{O}(ph)$, onde p é o número de instâncias na fila de pendentes e h é o número de servidores no sistema. Como apontado por Verma et al. [51], na prática, este custo pode ser substancialmente diminuído através da redução do número de servidores verificados nesta etapa. Além disso, heurísticas simples podem ser implementadas para evitar que todas as instâncias da fila sejam processadas sempre que o escalonador é executado (por exemplo, não verificando a viabilidade para instâncias que previamente se pode identificar que não serão alocadas). No Capítulo 7 (Seção 7.1.2), algumas otimizações que podem ser implementadas nesse sentido são discutidas e analisadas.

Algoritmo 1: Mecanismo de escalonamento.

On: requisição k foi admitida

begin
- t = TempoAtual()
- $Q_k(t) = 0$
- inserir as instâncias de k na fila de pendentes
- reescalonar = true

On: requisição k completou execução

begin
- liberar recursos alocados para as instâncias de k
- reescalonar = true

On: servidor h foi removido da infraestrutura (*e.g.* falha)

begin
- F = conjunto de instâncias que estavam alocadas em h
- **for** $f \in F$ **do**
  - inserir f na fila de instâncias pendentes
- reescalonar = true

On: servidor h foi adicionado à infraestrutura (*e.g.* recuperação de falha)

begin
- reescalonar = true

On: evento de periodicidade

begin
- reescalonar = true

On: reescalonar = true

begin
- reescalonar = false
- ordena a fila de instâncias pendentes
- k = primeira instância da fila
- **while** $k \neq$ **nil do**
  - E = EtapaVerificaçãoViabilidade(k)
  - **if** $E \neq \emptyset$ **then**
    - h = EtapaClassificação(E)
    - preempçãoSeNecessário(h)
    - aloca k em h
  - k = próxima instância da fila
- reinicia tempo de marcação para evento de periodicidade

3.6.2 Sobrecarga das Preempções

Embora preempções sejam uma forma eficiente para alcançar o provisionamento mais justo e a QoS esperada, elas possuem um custo associado. A quantidade de preempções realizadas é diretamente proporcional ao nível de contenção de recursos no sistema. Se a infraestrutura é super provisionada para uma carga de trabalho particular, preempções raramente devem ocorrer. Dada a mesma carga de trabalho, quanto menor a capacidade da infraestrutura, mais preempções serão realizadas. Quando o número de preempções ocorrendo no sistema é muito alto, uma quantidade significativa de tempo é gasta na alocação de recursos, prejudicando o desempenho do sistema no geral. Os escalonadores baseados em prioridades naturalmente limitam o número de preempções, visto que permitem apenas que instâncias com prioridades mais baixas sejam preemptadas. Este não é o caso do escalonador orientado por QoS.

Desde que as etapas de planejamento de capacidade e controle de admissão sejam executadas de forma adequada, espera-se que os períodos com alta contenção de recursos sejam raros. No entanto, como isso nem sempre pode ser garantido, o escalonador orientado por QoS deve estar preparado para lidar com essas situações. Dessa forma, o escalonador pode incorporar um mecanismo explícito para limitar a sobrecarga associada com preempções. Tal mecanismo pode ser implementado de diversas maneiras. Na Seção 4.1.3 discute-se uma possível implementação para este mecanismo (a implementação utilizada neste trabalho).

Capítulo 4

Metodologia

A avaliação da política de escalonamento orientada por QoS é realizada empiricamente, através de experimentos de simulação e de medição. A avaliação se dá através da comparação do desempenho do escalonador proposto com o de um escalonador baseado em prioridades quando ambos são submetidos às mesmas carga e infraestrutura. Este capítulo descreve os materiais e métodos utilizados ao longo das análises, tais como os modelos de simulação, as amostras de cargas de trabalho e infraestruturas utilizadas, o projeto experimental e as métricas de interesse.

4.1 Modelos de Simulação

O escalonador orientado por QoS é avaliado principalmente através de experimentos de simulação. Para isso, um simulador dirigido por eventos foi desenvolvido na linguagem de programação Erlang [2]. Este simulador[1] executa em uma ferramenta de simulação chamada Sim-Diasca[2], que é uma plataforma para execução de simulações que visa máxima concorrência, contando com um modo de operação paralelo e distribuído. Quando este simulador é executado, a política de escalonamento a ser utilizada (*baseada em prioridade* ou *orien-*

[1]O simulador está disponível para download no repositório https://forge.ericsson.net/plugins/git/ufcg-er/cloudish?a=treehb=experiments-journal-paper. Este doutorado foi desenvolvido como parte de um projeto de pesquisa e desenvolvimento financiado pela Ericsson Telecomunicações, por causa de acordo de confidencialidade com a empresa, o repositório é privado. No entanto, é possível fazer a solicitação de acesso ao código especificando os propósitos. A solicitação pode ser realizada via o próprio repositório.

[2]Informações sobre o Sim-Diasca estão disponíveis em http://sim-diasca.com.

tada por QoS) é informada. A Figura 4.1 apresenta os principais agentes desenvolvidos no simulador e como eles se relacionam.

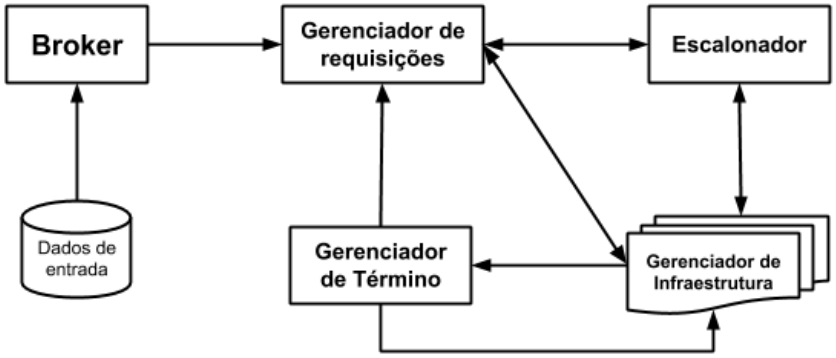


Figura 4.1: Arquitetura do simulador.

Como pode ser observado na Figura 4.1, o *Broker* é responsável por acessar os dados de entrada da simulação (tais como carga de trabalho, infraestrutura e parâmetros de configuração) e submetê-los para o *Gerenciador de Requisições*. Este último gerencia a alocação das requisições e mantém informações sobre a fila de espera. O *Gerenciador de Requisições* processa a fila segundo os eventos que desencadeiam esse processamento, solicitando ao *Escalonador* o servidor mais adequado para alocar cada requisição na fila de espera. Caso nenhum servidor seja adequado para alocação de uma requisição, esta é mantida na fila. O *Escalonador* interage com os *Gerenciadores de Infraestrutura* com o objetivo de escolher o "melhor" servidor para alocar uma requisição específica. Cada *Gerenciador de Infraestrutura* é responsável por lidar com um grupo de servidores da infraestrutura, provendo a verificação de viabilidade desses servidores e executando alocação, preempção e término de instâncias nos mesmos. Uma vez que uma nova instância é alocada, o *Gerenciador de Infraestrutura* responsável pelo servidor notifica o *Gerenciador de Término* sobre quanto tempo falta para a instância completar a duração desejada. Este agente agenda uma verificação de término para o momento em que a instância completará a duração desejada. Neste instante, o *Gerenciador de Término* interage com o *Gerenciador de Infraestrutura* correspondente e verifica se a instância esteve executando pela duração desejada. Em caso afirmativo, o *Gerenciador de Infraestrutura* terminará a instância. Sempre que o término de uma instância ocorre, o *Gerenciador de Término* notifica o *Gerenciador de Requisições*, que inicia um novo processamento da fila de espera.

As instruções para execução do simulador são apresentadas no Apêndice A. Dentre os parâmetros de entrada para uma simulação, destaca-se dois arquivos de descrição: *carga de trabalho* e *infraestrutura*.

4.1.1 Dados de Entrada

O arquivo de descrição da *carga de trabalho* contém informações sobre as requisições a serem processadas. Por conta das características das cargas analisadas, cada requisição admitida está associada com uma única instância e consiste em: (i) quantidades de CPU e memória necessárias para sua instância; (ii) classe de serviço solicitada; e, (iii) tempo necessário para a conclusão da tarefa computacional (*i.e.* por quanto tempo os recursos devem estar alocados para sua instância para que a requisição seja terminada). Observa-se que este último é utilizado para dirigir a simulação no sentido de terminar uma requisição quando este tempo for alcançado, no entanto, este valor é desconhecido pelo escalonador. Ademais, também é possível que as requisições tenham restrições de alocação e afinidade em sua definição.

A descrição da *infraestrutura* possui informações sobre os servidores que compõem a infraestrutura do provedor. Cada servidor é definido por suas capacidades de CPU e memória e um conjunto de atributos no formato "chave=valor". Os atributos são utilizados na etapa de verificação de viabilidade (Seção 3.4.1), enquanto o escalonador checa as restrições de alocação das requisições (se definidas). As quantidades de CPU e memória solicitadas por uma requisição são especificadas na mesma unidade que as capacidades de CPU e memória dos servidores.

Neste trabalho, as cargas de trabalho e as infraestruturas analisadas são obtidas a partir de um rastro de execução real de um *cluster* em produção da Google[3]. Este rastro contém informações sobre as instâncias que estiveram em execução ao longo do período da coleta do rastro. Portanto, neste trabalho, considera-se que cada instância presente no rastro de execução foi solicitada por uma requisição específica. Além disso, também existe o conceito de *job*. Um *job* pode agrupar uma ou mais requisições relacionadas. O conceito de *job* é importante quando restrições de anti-afinidade são especificadas para o mesmo. Essa infor-

[3]O rastro de execução da Google está disponível para download em https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

mação também está disponível nos rastros. Nesse cenário, duas requisições pertencentes a um mesmo *job* não podem ter suas instâncias alocadas em um mesmo servidor. Os métodos de geração dessas amostras são descritos adiante nas Seções 4.2 e 4.3.

4.1.2 Modelo de Simulação do Escalonador Baseado em Prioridade

Com o propósito de comparações, também se fez necessário implementar um escalonador baseado em prioridade. Este simulador implementa o escalonador baseado em prioridade padrão do sistema Kubernetes [5]. Este é um sistema de gerenciamento de contêiner desenvolvido pela Google [11], o qual é bastante popular e tem seu código aberto. Essa característica ajuda a evitar problemas de interpretação e fazer com que as funções do simulador sejam implementadas exatamente como definidas no sistema real. Seu escalonador atribui prioridades para as requisições de acordo com suas respectivas classes de serviço. As prioridades são diretamente proporcionais às metas de QoS (O_i) estabelecidas pelos SLOs das classes das requisições (i), assim, quanto maior a QoS esperada para uma dada requisição, maior é a sua prioridade. A estrutura geral dos dois modelos de escalonamento é essencialmente a mesma. Eles diferem apenas em decisões tomadas em relação a como ordenar a fila de espera, quais instâncias preemptar e como classificar os servidores elegíveis.

As instâncias na fila de espera são ordenadas em ordem decrescente de prioridade. Instâncias de mesma prioridade são ordenadas em ordem crescente de seus respectivos tempos de admissão, consequentemente, considerando cada sub-fila de prioridade, o escalonador opera com base na política FCFS.

Na etapa de verificação de viabilidade, preempções de instâncias com prioridades mais baixas podem ser consideradas apenas em benefício de instâncias com prioridades mais altas. Quando houver a necessidade de preempções, o escalonador interrompe as instâncias de prioridades mais baixas primeiro. Se precisar escolher dentre algumas instâncias de mesma prioridade, a escolhida será aquela com o tempo de admissão mais recente. Isso significa que, em períodos com contenção de recursos, instâncias com prioridades mais baixas são frequentemente preemptadas em benefício de instâncias com prioridades mais altas.

Como discutido anteriormente, quando preempções não são necessárias, o escalonador usa uma *função de pontuação de alocação* para classificar os servidores elegíveis. No Kubernetes, esta função é uma combinação de funções de prioridade. Neste modelo, duas

funções de prioridade padrões do Kubernetes foram implementadas: (i) "Prioridade Menos Solicitada" (do inglês, *Least Requested Priority*), $\mathcal{L}_h(t)$, que favorece servidores com menos recursos solicitados (mais recursos disponíveis), evitando a fragmentação de recursos[4]; e, (ii) "Alocação Balanceada de Recurso" (do inglês, *Balanced Resource Allocation*), $\mathcal{B}_h(t)$, que favorece servidores com a alocação de recursos mais balanceada, evitando o encalhamento de recursos[5].

Considere que h é um servidor elegível para alocar uma instância pendente de uma requisição j no tempo t, e, $c_h^c(t)$ e $a_h^c(t)$ são respectivamente, a capacidade total de CPU do servidor h e a quantidade de CPU de h alocada para as instâncias em execução no tempo t. De forma semelhante, considere $c_h^r(t)$ e $a_h^r(t)$, respectivamente, como a capacidade total e alocada de memória de h em t. As funções de prioridade $\mathcal{L}_h(t)$ e $\mathcal{B}_h(t)$ são definidas a seguir.

$$\mathcal{L}_h(t) = \frac{\frac{c_h^c(t) - a_h^c(t)}{c_h^c(t)} + \frac{c_h^r(t) - a_h^r(t)}{c_h^r(t)}}{2} \tag{4.1}$$

$$\mathcal{B}_h(t) = \left\lfloor 1 - \left| \frac{a_h^c(t)}{c_h^c(t)} - \frac{a_h^r(t)}{c_h^r(t)} \right| \right\rfloor \tag{4.2}$$

Cada umas das funções acima é utilizada para calcular uma pontuação parcial para um servidor elegível h. Essas pontuações têm um valor entre 0 e 10, onde 0 representa o servidor "menos adequado" e 10 representa o servidor "mais adequado" de acordo com cada função de prioridade. Em seguida, a pontuação final do servidor é dada pela média aritmética de suas pontuações parciais. Ao final, o servidor com maior pontuação é selecionado. Caso dois ou mais servidores tenham a mesma pontuação máxima, então um deles é selecionado aleatoriamente.

Caso todos os servidores elegíveis necessitem de preempções, a *função de pontuação de custo de preempção* utilizada para classifica-los favorece os servidores que necessitam da menor quantidade de preempções de instâncias com prioridades mais alta. Quando ne-

[4] A fragmentação de recursos ocorre através de pequenas quantidades de recursos que sobram nos servidores por não serem grandes o suficiente para satisfazer nenhuma das requisições admitidas.

[5] O encalhamento de recursos ocorre quando há quantidades relativamente grandes de recursos que não podem ser alocados porque não há outro tipo de recurso suficiente para serem agrupados juntos em uma instância para atender uma das instâncias pendentes.

cessário, desempates são feitos buscando o número mínimo de preempções de instâncias de outras classes (ordenadas em ordem decrescente de prioridade). Se o empate persistir, a função de pontuação de alocação descrita acima é utilizada para selecionar um dos servidores empatados. Em outras palavras, esta função de pontuação de custo de preempção favorece os servidores onde o menor número de preempções de instâncias das classes mais importantes são necessárias.

4.1.3 Modelo de Simulação do Escalonador Orientado por QoS

O simulador implementa um escalonador orientado por QoS que opera exatamente como descrito no Capítulo 3. Este modelo considera que sempre que uma instância estiver em execução, ela está contribuindo para melhorar a QoS (disponibilidade) de sua requisição. Nesta seção são discutidas algumas situações relacionadas à métrica de QoS no contexto deste trabalho, como também são detalhadas as funções de pontuação de alocação e de custo de preempção implementadas, o mecanismo para limitar a sobrecarga com preempções e os valores de configurações utilizados nos experimentos de simulação.

Métrica de QoS para decisão

Como mencionado anteriormente, este modelo considera que uma instância efetivamente contribui para melhorar a disponibilidade de sua requisição sempre que está em execução. Isso faz com que, para qualquer requisição j, a quantidade de tempo que suas instâncias estiveram em execução sem contribuir para melhorar sua QoS seja nula ($d_j(t) = 0$). Nesse contexto, a equação que calcula o intervalo $\Delta r_j(t)$ (Equação 3.6) — que indica a quantidade de tempo que as instâncias de uma requisição j estiveram pendentes desde que seu SLO passou a ser violado até o instante t —, pode ser simplificada para a Equação 4.3.

$$\Delta r_j(t) = \frac{e_j(t)}{O_i} - (e_j(t) + p_j(t)), \forall j | A_j(t) < O_i. \tag{4.3}$$

onde $e_j(t)$ representa o tempo acumulado em que todas as instâncias da requisição j estiveram em execução efetivamente contribuindo para melhorar a disponibilidade de j até o instante no tempo t (*i.e.* todo o tempo de execução das instâncias); $p_j(t)$ indica o tempo acumulado em que as instâncias da requisição j estiveram pendentes até t; e, O_i é o SLO de

disponibilidade da classe de serviço i, que é a classe da requisição j.

Além disso, considerando que cada requisição está associada com uma única instância (mencionado na Seção 4.1.1), a equação que calcula o intervalo $\Delta t_j(t)$ (Equação 3.3) — que indica o intervalo de tempo que uma requisição j pode permanecer recebendo uma determinada contribuição para sua QoS sem que seu SLO seja violado — também pode ser simplificada. Primeiramente, admitindo que $v_j^r(t)$ e $v_j^w(t)$ são, respectivamente, o número de instâncias da requisição j em execução e pendentes em um instante no tempo t, quando a requisição j está associada com uma única instância, $v_j^r(t) + v_j^w(t) = 1$ para qualquer ponto no tempo t. Além disso, independente do estado da instância sob análise do escalonador, a contribuição que sua requisição j receberá para melhorar sua disponibilidade ao longo de $\Delta t_j(t)$ será sempre nula ($c_j = 0$). Caso a instância de j esteja pendente no tempo t, ela já não está em execução e não contribuirá para a QoS de j ao longo de $\Delta t_j(t)$; se a instância de j estiver em execução em t, quando $\Delta t_j(t)$ é calculado, considera-se que ela será preemptada neste instante, portanto, também não haverá contribuição para a QoS de j durante $\Delta t_j(t)$. Nesse contexto, o intervalo $\Delta t_j(t)$ é definido como segue.

$$\Delta t_j(t) = \frac{e_j(t)}{O_i} - (e_j(t) + p_j(t)), \forall j | A_j(t) \geq O_i. \tag{4.4}$$

Destaca-se que, neste cenário, os intervalos $\Delta t_j(t)$ e $\Delta r_j(t)$ são calculados da mesma forma (Equações 4.3 e 4.4 são iguais), no entanto, $\Delta t_j(t)$ nunca assumirá um valor negativo, enquanto $\Delta r_j(t)$ nunca assumirá um valor positivo. Outras situações e semânticas para a disponibilidade de uma requisição são discutidas no Capítulo 7 (Seção 7.2), nesses cenários, as simplificações não serão possíveis, e, consequentemente, $\Delta t_j(t)$ e $\Delta r_j(t)$ não serão dados pela mesma equação.

Uma vez que há uma relação de 1 para 1 entre requisições e instâncias nas cargas utilizadas nas análises dos Capítulos 5 e 6, os termos *QoS da requisição* e *QoS da instância* podem ser utilizados de forma permutável, visto que não alteram o significado do que é discutido.

Função de pontuação de alocação

Uma vez que nesta etapa nem as prioridades das requisições nem as métricas de QoS são consideradas, este modelo utiliza a mesma função de pontuação de alocação utilizada para o escalonamento baseado em prioridades.

Função de pontuação de custo de preempção

O custo de uma preempção não pode ser facilmente modelado, visto que isso envolve antecipar o impacto que uma preempção teria na QoS entregue pelo sistema. Assim sendo, aplica-se uma heurística que possa estimar este custo, assim como foi feito com o escalonador baseado em prioridades.

A ideia central da heurística implementada é a seguinte. As instâncias associadas com requisições que estão muito próximas de violar ou já estejam violando seus respectivos SLOs têm os maiores custos de preempção, enquanto instâncias de requisições que estejam mais distantes de violar seus SLOs têm custos de preempção mais baixos. Além disso, entre as instâncias que possuem os custos de preempção mais altos, suas importâncias também são consideradas. Dessa forma, as instâncias mais importantes têm custos de preempção ainda mais altos.

Considere que $P_h(t)$ é o conjunto não vazio das instâncias que *precisam ser preemptadas* no servidor h para que uma instância pendente da requisição j seja alocada no tempo t. Este conjunto pode ser dividido em dois conjuntos disjuntos, $P_{h+}(t)$ e $P_{h-}(t)$. O subconjunto $P_{h+}(t)$ representa o conjunto de instâncias que precisam ser preemptadas em h, e que suas requisições não estão tão próximas de violar seus SLOs. Formalmente:

$$P_{h+}(t) = \{k \in P_h(t); Q_k(t) - \sigma_{i_k} \geq 0\},$$

onde σ_{i_k} é a margem de segurança para a classe de serviço i_k da instância k. Assim, uma pontuação parcial do custo de preempção s_+ é calculada como segue:

$$s_+ = \frac{1}{\sum_{k \in P_{h+}(t)} (Q_k(t) - \sigma_{i_k})}$$

Em seguida, o conjunto de instâncias a serem preemptadas em h que já tem suas requisições violando ou próximas de violar seus SLOs ($P_{h-}(t) = P_h(t) - P_{h+}(t)$) é dividido em m conjuntos disjuntos, $P^i_{h-}(t)$, $1 \leq i \leq m$. Um subconjunto $P^i_{h-}(t)$ para cada classe de serviço oferecida no sistema. O conjunto $P^i_{h-}(t)$ contém apenas instâncias da classe i pertencente ao conjunto P_{h-}, as quais precisam ser preemptadas em h para acomodar uma instância de j no tempo t. Assim, as pontuações parciais do custo de preempção s^i_-, $1 \leq i \leq m$, são computadas como segue:

$$s_{-}^{i} = \frac{1}{\sum_{k \in P_{h-}^{i}(t)} (Q_k(t) - \sigma_{i_k})}.$$

Quanto menor o valor de i, mais importante é a classe de serviço. Assim sendo, o custo de preempção de um servidor é dado pela tupla de pontuações parciais $S =< s_{-}^{1}, s_{-}^{2}, ..., s_{-}^{m}, s_{+} >$. É importante lembrar que quando preempções são necessárias, o escalonador seleciona o servidor com o menor custo de preempção. Uma pontuação S é menor que uma pontuação S' ($S < S'$) se há um x tal que o *x-ésimo* elemento de S é menor que o *x-ésimo* elemento de S', e todos os outros elementos y, $y < x$, têm o mesmo valor em S e S'. A Equação 4.5 formaliza esta relação.

$$S < S' \iff \exists x \in [1, m+1] \,|S[x] < S'[x] \wedge \forall y \in [1, x[\,, S[y] = S'[y]. \tag{4.5}$$

De forma similar ao escalonador baseado em prioridade, quando dois ou mais servidores tem o mesmo menor valor para o custo de preempção, então o servidor escolhido será aquele como maior valor para a função de pontuação de alocação dentre os servidores empatados.

Por fim, é importante mencionar que esta é apenas uma das heurísticas que podem ser utilizadas para esse propósito. Uma heurística nem sempre apresenta a melhor escolha para cada caso específico, porém, isso ocorre para qualquer heurística. Em outras palavras, qualquer heurística a ser utilizada para definir a o custo de preempção dos servidores estará propensa a situações específica em que o escalonador toma uma má decisão sob determinado aspecto. Como será discutido mais adiante no Capítulo 6, a heurística implementada levou a resultados satisfatórios. Porém, a avaliação de diferentes heurísticas para definição do custo de preempções não faz parte do escopo deste trabalho.

Mecanismo para limitar a sobrecarga com preempções

O escalonador monitora a sobrecarga relacionada com preempções para cada instância no sistema. A sobrecarga de preempções de uma instância k no tempo t é definida pela Equação 4.6. Neste cenário, n indica o número vezes que a instância k foi preemptada desde a admissão de sua requisição até t, α_k^m representa o tempo de alocação/startup da instância medido quando a instância k foi alocada pela *m-ésima* vez, e, $r_k(t)$ é a quantidade de tempo acumulado que a instância k esteve em execução até o tempo t.

$$C_k(t) = \frac{\sum_{m=1}^{n} \alpha_k^m}{r_k(t) + \sum_{m=1}^{n} \alpha_k^m}. \tag{4.6}$$

Este mecanismo usa um método simples para limitar a frequência com que uma instância pode ser preemptada. A ideia central é ter um limite máximo aceitável para a sobrecarga de preempções por instância. Assim que uma instância atinge este limite, ela não mais pode ser preemptada em benefício de uma outra instância de mesma importância ou importância inferior. Preempções para tais instâncias serão permitidas apenas quando suas respectivas sobrecargas diminuírem para um valor aceitável. As sobrecargas de preempções das instâncias são avaliadas durante a etapa de verificação de viabilidade.

É importante destacar que este mecanismo é configurável. O limite máximo aceitável para sobrecarga de preempções é um dos parâmetros para a simulação. Nesse sentido, a definição de um valor que seja grande o suficiente significa, na prática, que o mecanismo está sendo desligado.

Configurações de simulação

Nos experimentos realizados ao longo deste trabalho, o limite aceitável para a sobrecarga com preempções de uma instância da classe i foi definido para ser $1 - O_i$. Considerando que O_i é a meta de disponibilidade para a classe de serviço i, essencialmente os limites aceitáveis para esta sobrecarga de uma instância ficam limitadas ao tempo máximo que ela pode permanecer na fila mantendo o SLO de sua requisição satisfeito.

Com relação ao tempo máximo entre duas execuções sequencias do escalonador, que define a periodicidade de execução do escalonador (observar linha 29 do Algoritmo 1), 10 segundos foi o valor utilizado nos experimentos. Este valor foi definido empiricamente através da observação do Kubernetes em cenários onde há instâncias na fila de espera e o escalonador tenta alocá-las periodicamente. Para efeito de simplicidade, a margem de segurança (σ_i) para a métrica de QoS também foi definida em 10 segundos para todas as classes de serviço. Destaca-se que esses valores também foram utilizados nos experimentos de medição (detalhados a seguir no Capítulo 5), executados para validação dos modelos de simulação.

4.2 Carga de Trabalho

As cargas de trabalho submetidas nos experimentos de simulação são amostras retiradas de rastros de execução de um *cluster* em produção da Google[6]. Esses rastros abrangem um período de 29 dias de maio de 2011 e consistem em mais de 25 milhões de requisições por recursos do provedor [43], onde cada requisição está associada com uma instância. Apesar de existir outros rastros de execução mais recentes disponíveis publicamente, tais como os da Azure[7], nenhum deles é tão completo (de informações) quanto o da Google. Por exemplo, nos rastros da Azure, não há informação sobre a infraestrutura do *cluster*. Devido ao grande número, diversidade e maturidade das aplicações que executam na Google, acredita-se que esses dados são uma amostra representativa e relevante de ambientes de provedores de nuvem. Análises mais detalhadas sobre os rastros são apresentadas nos trabalhos de Reiss et al. [42] e Abdul-Rahman e Aida [6].

Os rastros da Google possuem informações sobre *jobs* que foram submetidos ao longo dos 29 dias de sua abrangência. Cada *job* é composto por uma ou mais requisições, cada uma associada com uma instância e definida por suas demandas por recursos (CPU e memória), sua duração (*i.e.* por quanto tempo sua instância precisa estar em execução para que a requisição seja terminada) e suas restrições de alocação e/ou afinidade (opcionais). Quando um *job* é composto por múltiplas requisições, tipicamente elas possuem as mesmas demandas, duração e restrições (se definidas). Neste trabalho, cada requisição pertencente a um *job* é considerada como uma requisição independente submetida para o sistema e escalonada de tal forma pelo escalonador.

Os rastros da Google também incluem as capacidades dos servidores da infraestrutura. Essas capacidades são normalizadas como um percentual da capacidade do servidor "mais poderoso" para o recurso em questão. O servidor "mais poderoso" sob o ponto de vista de um recurso é aquele que contém a maior quantidade deste recurso. A requisições nos rastros também usam a mesma escala normalizada para descrever as quantidades de CPU e memória solicitadas, portanto, o escalonamento pode ser executado de forma realista.

[6]Os rastros de execução da Google estão disponíveis para download em https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[7]Os rastros de execução da Azure estão disponíveis para download em https://github.com/Azure/AzurePublicDataset.

As requisições podem ser classificadas de acordo com 12 prioridades que podem ser associadas a cada uma delas (0 a 11). Essas prioridades são usadas para definir as diferentes classes de serviço consideradas nos experimentos de simulação. Com base na descrição dos rastros [43] e em trabalhos anteriores [42; 16] é possível agrupar as requisições em três classes de serviço. Os SLOs de disponibilidade estabelecidos para as classes são os mesmos utilizados por Carvalho et al. [16] quando utilizaram o mesmo rastro da Google para avaliação de um modelo de controle de admissão. As classes de serviço consideradas neste trabalho e seus respectivos SLOs de disponibilidade são descritos abaixo:

- A classe *A* é associada às requisições com prioridades maiores que 8. Essas são as requisições mais importantes na carga de trabalho, visto que supõe-se que suas instâncias nunca são preemptadas. Por esta razão, esta classe promete uma QoS de 100% de disponibilidade em seu SLO. Esta é a classe mais exigente em termos de QoS, portanto, o escalonador baseado em prioridades associa a prioridade mais alta para esta classe, enquanto que o escalonador orientado por QoS considera esta como a classe mais importante (*i.e.* $i = 1$);

- A classe *B* está relacionada com as requisições com prioridades intermediárias (maior que 1 e menor que 9 – $[2, 8]$). Esta classe tem seu SLO de disponibilidade estabelecido em 90%. O escalonador baseado em prioridades associa a segunda prioridade mais alta para esta classe, enquanto o escalonador orientado por QoS estabelece um valor intermediário para sua importância ($i = 2$);

- A classe *C* é a menos exigente em termos de QoS. As requisições com prioridades menores que 2 são associadas com esta classe. As instâncias desta classe são frequentemente preemptadas em benefício de instâncias com prioridades mais altas [42]. Seu SLO de disponibilidade é definido em 50%. Esta é a classe de menor prioridade para o escalonador baseado em prioridade como também a classe menos importante para o escalonador orientado por QoS ($i = 3$).

Simular todo o rastro da Google é muito caro em termos de tempo de processamento e recursos necessários. Por esta razão, 10 amostras de cargas de trabalho foram geradas a partir do rastro original.

4.2.1 Geração de amostras da carga de trabalho

As amostras da carga de trabalho foram geradas da seguinte forma. Inicialmente, uma análise de agrupamento dos usuários da Google foi realizada. Esta análise aplicou o algoritmo de agrupamento k-means, levando em consideração, para cada usuário, o número de requisições submetidas e a variância das durações e quantidades de CPU e memória solicitadas nessas requisições. Esta investigação resultou em 6 grupos de usuários. Com o intuito de gerar uma amostra da carga de trabalho, 10% dos usuários de cada um dos grupos foram selecionados aleatoriamente. A amostra da carga resultante consiste de todas as requisições submetidas pelos usuários selecionados. Esse processo foi repetido dez vezes, gerando as 10 amostras utilizadas neste trabalho. Todas as cargas foram geradas a partir das requisições presentes no rastro original, *i.e.* requisições selecionadas para uma amostra também podem ser selecionadas para outras amostras.

As Figuras 4.2 e 4.3 apresentam, respectivamente, as quantidades de CPU e memória alocadas ao longo do tempo (medidas em intervalos de 1 minuto) para cada amostra, quando submetidas em uma infraestrutura hipotética com um único servidor com capacidade infinita para CPU e memória. Em ambas as figuras, a unidade do recurso alocado é o valor normalizado presente no rastro. Por exemplo, uma alocação de 90 CPU ou memória significa que a demanda para o recurso no tempo específico é 90 vezes maior que a capacidade do melhor servidor da infraestrutura para o recurso em questão. Nesses gráficos a demanda da carga de trabalho também é diferenciada pelas três classes de serviço consideradas.

As 10 amostras da carga[8] possuem requisições de todas as classes de serviço e diferem substancialmente entre si; suas formas, a combinação de requisições por classe de serviço, as demandas dos picos de requisições e suas intensidades são diferentes. Essa heterogeneidade das amostras se deve ao fato de diferentes subconjuntos de usuários reais levarem a diferentes agrupamentos de requisições. Ademais, esta variabilidade é importante para analisar o escalonador sob diferentes (e ainda verossímeis) cargas de trabalho.

[8]As amostras da carga utilizadas nos experimentos de simulação e medição ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/workloads. Além disso, instruções sobre como executar os experimentos são apresentadas no Apêndice A.

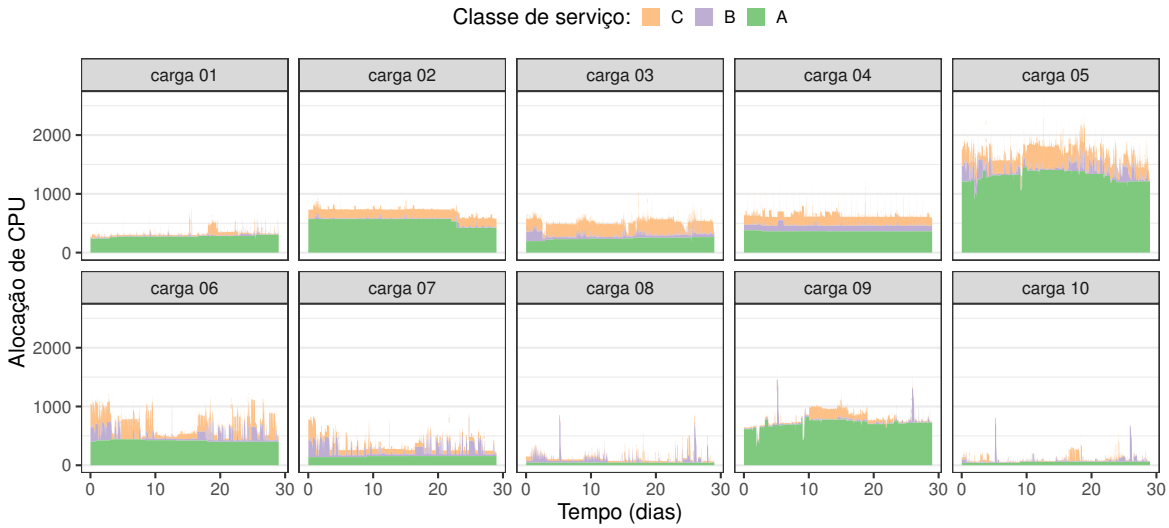


Figura 4.2: Alocação de CPU em intervalos de 1 minuto para as 10 amostras da carga de trabalho geradas.

4.3 Infraestrutura

Alterações na capacidade da infraestrutura afetam o nível de contenção de recursos no sistema. Mantendo-se uma mesma carga de trabalho, quanto maior a infraestrutura, menor é a contenção. A contenção de recursos também é afetada pela demanda que o sistema está sujeito. Uma vez que cada uma das 10 amostras da carga de trabalho leva a diferentes demandas, a infraestrutura a ser considerada para alocação das cargas também deve variar de carga para carga de forma a manter o nível de contenção equiparável entre as várias cargas. Portanto, considera-se uma capacidade N para cada infraestrutura, que é definida de acordo com o pico de demanda de cada carga. Nesse cenário, a contenção de recursos é causada pela fragmentação dos mesmos. Em seguida, gera-se para cada carga uma infraestrutura de capacidade N. Essa infraestrutura é gerada sorteando aleatoriamente um servidor por vez (sem reposição) a partir dos rastros da Google até que a capacidade agregada da infraestrutura sorteada seja N. A seguir descreve-se como o valor N é estabelecido para cada carga de trabalho:

1. Dada uma carga, simula-se a alocação desta em uma infraestrutura hipotética composta por um único servidor com capacidades infinitas para CPU e memória. Não importa qual o escalonador utilizado neste experimento inicial, visto que todas as requisições

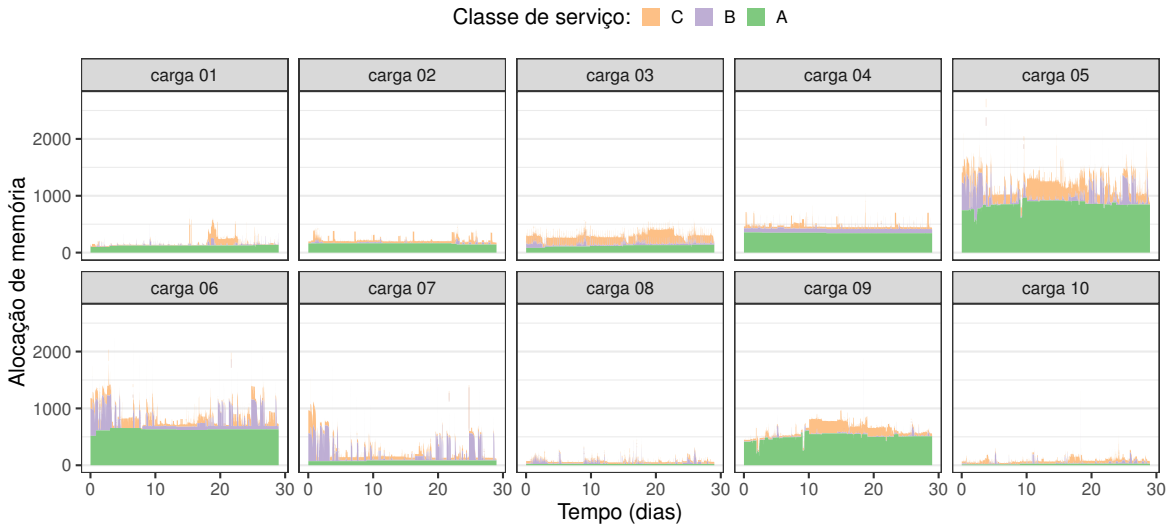


Figura 4.3: Alocação de memória em intervalos de 1 minuto para as 10 amostras da carga de trabalho geradas.

são alocadas imediatamente, sem atrasos na fila ou preempções neste "servidor único";

2. Em seguida, avaliam-se os resultados da simulação realizada no passo (1) para identificar a quantidade máxima de CPU e de memória utilizadas para processar a carga de trabalho em questão. Considere que essas quantidades máximas são N_c e N_m, respectivamente. A capacidade N é definida como o valor máximo entre N_c e N_m, e, a geração da infraestrutura é dirigida pelo recurso com maior pico, *i.e.* CPU se $N_c > N_m$ ou memória se $N_m > N_c$[9].

4.4 Modelo para Definição de Penalidades

Os principais provedores públicos de computação na nuvem (AWS e Azure) definem SLAs cujas penalidades levam em conta o nível de déficit de QoS sofrido por seus consumidores. Dessa forma, as penalidades não são definidas com base apenas no número de requisições que tiveram seus SLOs não atendidos, mas também com base na QoS oferecida para essas

[9]Lembre-se que as quantidades de CPU e memória solicitadas em uma requisição são normalizadas, portanto, a comparação de ambos ocorre em termos percentuais ao servidor mais poderoso para cada recurso. Portanto, se $N_c > N_m$ significa que, considerando o pico da demanda e o servidor com maior quantidade para cada recurso, mais recursos de CPU foram solicitados em comparação com a solicitação de memória.

requisições. Por exemplo, o AWS promete pelo menos $99,99\%$ de disponibilidade para suas instâncias no intervalo de um mês. Se uma instância não receber esta QoS, o cliente terá um crédito de serviço de acordo com a QoS recebida pela instância. Neste caso, o AWS calcula os créditos como um percentual do total pago pelo consumidor pela instância que não recebeu a QoS prometida. Quando a disponibilidade provida é menor que $99,99\%$, mas maior que 99%, o crédito a ser dado ao cliente (penalidade do provedor) é de 10%. Este percentual aumenta para 30% para disponibilidades menores que 99% e maiores que 95%, e, finalmente, para 100% quando a disponibilidade oferecida é menor que 95%[10].

Com base no modelo de penalidades do AWS, este trabalho define um modelo para avaliar como se comporta o escalonador baseado em prioridade e orientado por QoS sob o ponto de vista do custo do provedor com as penalidades decorrentes de violações. Nesse sentido, considera-se os mesmos três níveis de penalidades para cada classe de serviço: 10%, 30% e 100%. As penalidades para a classe *A* seguem as mesmas regras dos serviços do AWS. Para as outras classes (*B* e *C*), utilizam-se regras similares cujos limites são definidos proporcionalmente ao SLO de cada classe. Considere $A_j(t)$ como a disponibilidade da requisição j no tempo t que ela é terminada. A Tabela 4.1 apresenta o intervalo de disponibilidade para cada classe associado com o percentual da penalidade a ser paga pelo provedor quando este entrega uma QoS dentro do intervalo correspondente.

Tabela 4.1: Percentual do bônus a ser creditado para o consumidor de acordo com a classe de serviço e a disponibilidade provida para requisição com SLO não atendido

Classe	crédito de 10%	crédito de 30%	crédito de 100%
A	$99.99\% > A_j(t) \geq 99.00\%$	$99.00\% > A_j(t) \geq 95.00\%$	$A_j(t) < 95.00\%$
B	$90.00\% > A_j(t) \geq 89.11\%$	$89.11\% > A_j(t) \geq 85.56\%$	$A_j(t) < 85.56\%$
C	$50.00\% > A_j(t) \geq 49.50\%$	$49.50\% > A_j(t) \geq 47.50\%$	$A_j(t) < 47.50\%$

Neste modelo, a penalidade a ser paga pelo provedor é definida de acordo com a duração da requisição e sua demanda por CPU. Considere D_j como o déficit de QoS experimentado por uma requisição j ($D_j = O_i - A_j(t)$), d_j como a duração de j e u_j como a quantidade de CPU solicitada para j. O déficit de j em termos de CPU $\cdot$ tempo é dado por:

[10]https://aws.amazon.com/compute/sla/.

$$D_j^* = D_j \cdot d_j \cdot u_j. \tag{4.7}$$

Nesse sentido, D_j^* representa a quantidade adicional de CPU por unidade de tempo que j deveria ter recebido do provedor para ter seu SLO atendido. Como penalidade pela violação de j, define-se que o provedor deve pagar o que deixou de oferecer (D_j^*) incrementado de um bônus baseado na QoS oferecida a j (como apresentado na Tabela 4.1). Considere b_j como o bônus a ser oferecido pelo provedor por causa da violação de j, então a penalidade total associada com a requisição j é dada por:

$$P_j = D_j^* \cdot (1 + b_j). \tag{4.8}$$

4.5 Métricas de Avaliação

O escalonador orientado por QoS é comparado com o escalonador baseado em prioridade através de diferentes métricas. A métrica básica utilizada nesta avaliação é a QoS (*i.e.* a *disponibilidade*) que é entregue para as requisições admitidas, que é computada de acordo com a Equação 3.1 previamente definida. Além disso, o *déficit de QoS* experimentado pelas requisições que tiveram seus respectivos SLOs não satisfeitos também é medido. Esta métrica é calculada como a diferença entre o SLO e a disponibilidade final entregue para essas requisições.

A *satisfação do SLO* também é calculada nos experimentos de simulação. Esta métrica é a proporção entre o número de requisições que tiveram seus SLOs satisfeitos, *i.e.* receberam uma QoS igual ou acima da prometida, e o número total de requisições admitidas. Todas essas métricas são calculadas separadamente para cada uma das três classes de serviço consideradas. Além disso, o *custo total com penalidades* ao qual o provedor fica sujeito por causa de violações também é analisado. Esse custo é dado pela soma das penalidades associadas com as requisições que tiveram seus SLAs não cumpridos ao longo de suas execuções.

Por fim, também avalia-se quão justo são os escalonadores enquanto compartilham os recursos entre as requisições. A ideia central é avaliar a equidade na QoS provida para requisições de mesma classe que estejam ativas aproximadamente no mesmo tempo competindo pelos mesmos recursos. Com o propósito de avaliar a justiça, calcula-se o *coeficiente de*

Gini [9]. Este é um coeficiente muito utilizado para revelar a desigualdade entre os indivíduos de uma população/amostra. O coeficiente de Gini varia no intervalo $[0, 1]$, onde 0 corresponde à equidade perfeita de renda (*i.e.* todos os membros possuem a mesma renda) e 1 corresponde a uma desigualdade perfeita de renda (*i.e.* uma pessoa tem toda a renda enquanto todas as outras não possuem renda). No contexto deste trabalho, quanto menor o valor do coeficiente de Gini para um conjunto de requisições, mais justa foi o provisionamento dos recursos entre essas requisições.

4.6 Projeto Experimental

Os experimentos de simulação seguem um projeto fatorial completo com dois fatores: a política de escalonamento utilizada e a capacidade da infraestrutura. O primeiro tem dois níveis (o escalonamento *baseado em prioridade* e o escalonamento *orientado por QoS*), enquanto que o segundo fator possui três níveis. Alterações neste último fator afetam o nível de contenção de recursos no sistema. Como descrito na Seção 4.3, cada carga tem uma infraestrutura com capacidade N a ser considerada. Para analisar o comportamento dos escalonadores em diferentes níveis de contenção, infraestruturas com outras capacidades são definidas para cada carga. A capacidade N é o primeiro nível deste fator. Os outros dois níveis são configurados em $0,9N$ e $0,8N$, os quais correspondem a infraestruturas que são menores em 10% e 20%, respectivamente. Essas infraestruturas[11] são geradas removendo aleatoriamente um servidor por vez da infraestrutura de tamanho N previamente gerada até que a capacidade desejada seja alcançada.

A partir desses experimentos, a política de escalonamento proposta é avaliada através de sua comparação com uma política baseada em prioridade utilizada no estado-da-prática. Para cada tratamento de dois fatores (política de escalonamento e capacidade da infraestrutura), são executados experimentos com as 10 amostras de cargas geradas conforme descrito na Seção 4.2, levando a 60 diferentes cenários testados. Replicações não são necessárias visto que os resultados são determinísticos quando a mesma carga e infraestrutura são consideradas.

[11] As infraestruturas utilizadas nos experimentos de simulação ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/infrastructures. Além disso, instruções sobre como executar os experimentos são apresentadas no Apêndice A.

Capítulo 5

Validação

Visto que os principais resultados deste trabalho são provenientes de experimentos de simulação, torna-se importante a validação dos modelos de simulação utilizados. Por mais completo que seja, a definição de um modelo de simulação implica em algumas simplificações do sistema real. Por esta razão, este capítulo apresenta como os modelos de simulação utilizados neste trabalho foram validados.

5.1 Metodologia

A validação dos modelos de simulação foi realizada comparando os resultados dos experimentos de medição e de simulação sob as mesmas condições de ambiente — infraestrutura e configuração do escalonador — e carga de trabalho. Esses experimentos seguem um projeto fatorial completo com um único fator: a política de escalonamento utilizada. Os escalonamentos baseado em prioridade e orientado por QoS são os níveis para este fator. Nesse contexto, a métrica avaliada foi a disponibilidade final das requisições submetidas.

O escalonador padrão do sistema Kubernetes foi utilizado como a implementação do escalonador baseado em prioridade. Esse sistema possui um escalonador baseado em prioridade[1] padrão. O Kubernetes foi escolhido não apenas por causa de sua popularidade, mas também pela facilidade de alterar seu escalonador sem necessariamente ter que alterar outras partes do sistema. Já para o escalonador orientado por QoS, um protótipo foi implementado como prova de conceito de acordo com o que foi descrito no Capítulo 3. Este protótipo foi

[1] O escalonador baseado em prioridade é disponibilizado no Kubernetes desde sua versão 1.8.

integrado ao Kubernetes através da substituição de seu escalonador padrão baseado em prioridade. Em ambos os casos, a versão do Kubernetes utilizada nos experimentos foi a 1.9, que era a versão mais recente no momento do início desta validação.

O escalonador orientado por QoS foi configurado da mesma forma em ambos os experimentos de simulação e medição, exatamente como descrito na Seção 4.1.3.

5.2 Protótipo

No contexto do Kubernetes, uma instância pode ser representada por um *pod*. Um *pod* encapsula um ou mais contêiner para uma aplicação, recursos de armazenamento, um IP único na rede e opções que direcionam como a aplicação deve executar. Em outras palavras, um *pod* representa uma instância de uma aplicação no sistema Kubernetes. Além disso, este sistema também disponibiliza um conjunto de abstrações de mais alto nível com o intuito de facilitar o gerenciamento das aplicações. Essas abstrações são chamadas de *controladores*. Dentre os controladores oferecidos, o *Deployment* foi concebido para aplicações que funcionam como serviços. Neste protótipo, considera-se que cada requisição na carga de trabalho consiste de um *Deployment* responsável por gerenciar uma única réplica de um *pod*, ou seja, cada requisição está associada com uma única instância de uma aplicação. Além disso, o nome da aplicação e seu SLO de disponibilidade são atributos que devem ser configurados no arquivo de descrição de seu *Deployment*. Os diferentes controladores suportados pelo Kubernetes bem como seus propósitos são discutidos em detalhes no Capítulo 7 (Seção 7.2.1).

Por ser um escalonador preemptivo, sempre que o escalonador orientado por QoS é executado, um ou mais *pods* podem ser preemptados. Quando existe um *Deployment* responsável por um *pod* e este *pod* é preemptado, um novo *pod* com as mesmas características é criado e associado com a mesma requisição (*Deployment*). Os diferentes *pods* associados com uma mesma requisição representam diferentes encarnações da instância desta requisição. Assim sendo, quando a métrica de QoS de uma requisição j precisa ser calculada no tempo t ($Q_j(t)$), o escalonador requer dados de todas as encarnações da instância de j. Isso ocorre porque o tempo acumulado que a instância de j esteve em execução até t ($e_j(t)$) é dado pela soma dos tempos de execução de todas as encarnações da instância de j. De forma semelhante, o tempo acumulado que a instância da requisição j esteve pendente ($p_j(t)$) é

dado pela agregação dos tempos que todas as encarnações estiveram pendentes. Por esta razão, este protótipo exigiu o uso de dois serviços adicionais: o *kubewatch* e o *prometheus*.

O *kubewatch* é responsável por monitorar os eventos relacionados aos *pods*, tais como criação, alocação, deleção e preempção. Sempre que um desses eventos ocorre, as métricas relacionadas aos tempos em execução e pendente deste *pod* (*i.e.* desta encarnação da instância) são atualizadas. Em um evento de criação, o momento que o *pod* foi criado é registrado, desta forma o escalonador pode inferir a quantidade de tempo que o *pod* está pendente. No caso de um evento de alocação, o *kubewatch* registra o tempo total que o *pod* esteve pendente e o momento que ele foi alocado. Assim sendo, quando necessário, o escalonador pode inferir a quantidade de tempo que o *pod* está em execução. Em um evento de preempção, o *kubewatch* registra o tempo total que o *pod* esteve em execução. Por último, em um evento de deleção, o *kubewtach* registra o tempo total que o *pod* esteve pendente ou em execução desde sua criação. Visto que ambos os serviços (*kubewatch* e escalonador) executam no mesmo servidor, os tempos são calculados por eles com base no mesmo relógio.

Já o *prometheus* é responsável por armazenar os registros realizados pelo *kubewatch* e disponibilizá-los para o escalonador. Dessa forma, sempre que o escalonador é executado e precisa calcular a métrica de QoS de uma requisição j em um instante no tempo t ($Q_j(t)$), ele busca no *prometheus* todos os registros relacionados com as encarnações da instância de j. Com acesso aos registros, o escalonador consegue calcular a métrica de QoS de j adequadamente.

No repositório[2] é possível encontrar instruções sobre como implantar um *cluster* Kubernetes utilizando o escalonador orientado por QoS desenvolvido neste trabalho. Além disso, instruções sobre como reproduzir os experimentos são apresentadas no Apêndice A.

5.3 Projeto experimental

A validação foi realizada em dois testes, com a execução de 2 cargas de trabalho sintéticas na mesma infraestrutura. Em ambos os casos, a infraestrutura consistiu de um *cluster* Kubernetes com 20 servidores homogêneos — máquinas virtuais de um provedor OpenStack

[2]Instruções sobre a implantação de um cluster kubernetes com o escalonador orientado por QoS estão disponíveis em https://github.com/giovannifs/cloudish-kubernetes-experiment/tree/support-multiple-controllers.

— cada um com 4 Gb de memória RAM e 4 vCPUs. Neste *cluster*, o próprio sistema Kubernetes usou aproximadamente $0,25$ Gb da memória em cada servidor para atividades de gerenciamento.

As cargas de trabalho sintéticas[3] foram concebidas de forma que fosse possível antecipar o comportamento esperado dos escalonadores e testá-los em diferentes cenários de estresse. Em ambos os casos, todas as requisições foram submetidas no início do teste, com o intervalo de 1 segundo entre a submissão de duas requisições subsequentes. Os testes executaram por 1 hora e todas as requisições estiveram ativas até o final dos testes, quando as disponibilidades finais das requisições foram calculadas. Todas as requisições solicitaram a mesma quantidade de CPU e memória ($0,375$ Gb de RAM e $0,375$ vCPUs), permitindo a execução de 10 instâncias simultaneamente alocadas em cada servidor. Como já mencionado, nos dois testes, o limite máximo de sobrecarga de preempção aceitável foi configurado em $1 - O_i$ para os experimentos de simulação e medição — *i.e.* 0%, 10% e 50% para as classes *A*, *B* e *C*, respectivamente.

5.4 Testes de validação

No primeiro teste, a carga de trabalho sintética consistiu de 256 requisições, com 80 da classe *A*, 80 da classe *B* e 96 da classe *C*. A ordem de submissão dessas requisições foi definida de forma aleatória. O comportamento esperado é que o escalonador baseado em prioridade ofereça 100% de disponibilidade para todas as requisições *A* e *B*. Com relação as requisições *C*, como a infraestrutura suporta a alocação de 200 requisições simultaneamente, 40 das requisições *C* devem ter disponibilidades próximas de 100% e as outras 56 devem ter disponibilidades próximas de 0%. Por outro lado, espera-se que o escalonador orientado por QoS entregue disponibilidades para todas as requisições bem próximas do SLO de suas respectivas classes (pequenas diferenças são esperadas devido as sobrecargas de preempção e escalonamento envolvidas).

O segundo teste de validação teve como objetivo exercitar o mecanismo adotado para

[3] As cargas utilizadas nos experimentos de medição ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/workloads. Além disso, instruções sobre como reproduzir os experimentos são apresentadas no Apêndice A.

limitar o número de preempções realizadas pelo escalonador orientado por QoS. Para isso, a carga de trabalho sintética usada consistiu de 221 requisições da classe *B*. Visto que a classe *B* tem um SLO alto (90%) e todas as requisições ativas têm a mesma importância, as preempções logo se tornarão muito frequentes e o mecanismo para limitar as preempções terá mais chances de ser acionado. Neste caso, o comportamento esperado é que o escalonador baseado em prioridade alocará as primeiras 200 requisições e deixará as outras 21 requisições na fila de espera. Assim, 200 requisições terão disponibilidades próximas de 100%, enquanto 21 terão disponibilidades de 0%. Com o escalonador orientado por QoS, espera-se que todas as requisições tenham a oportunidade de executar e recebam disponibilidades próximas de seu respectivo SLO. De novo, é possível que algumas requisições tenham pequenos déficits de QoS devido às sobrecargas envolvidas.

5.5 Resultados

A Figura 5.1 apresenta as disponibilidades finais calculadas para as requisições da carga de trabalho do primeiro teste[4]. As disponibilidades finais calculadas nos experimentos de simulação estão em roxo, enquanto que as calculadas nos experimentos de medição estão em verde. Além disso, os resultados obtidos com o escalonador baseado em prioridade estão do lado esquerdo, enquanto os resultados obtidos com o escalonador orientado por QoS estão do lado direito.

Como esperado, o escalonador baseado em prioridade manteve as disponibilidades das requisições com prioridades mais altas em 100%. Além disso, 40 das requisições *C* receberam 100% de disponibilidade porque elas foram submetidas enquanto ainda existiam recursos disponíveis e nunca foram selecionadas para serem preemptadas em benefício de outras requisições. No entanto, o restante das requisições *C* (56) têm disponibilidades abaixo de seus SLOs e próximas de 0%. As requisições *C* que tiveram seus SLOs não atendidos foram submetidas quando a infraestrutura já estava completamente utilizada ou foram preemptadas quando outras requisições de classes com prioridades mais altas foram submetidas. Por sua

[4]Esta figura apresenta os resultados de uma execução desse teste. No entanto, outras 9 cargas de trabalho com as mesmas características foram geradas, submetendo as requisições em ordens diferentes, e os resultados obtidos são muito similares aos apresentados nesta figura. As cargas utilizadas estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/workloads.

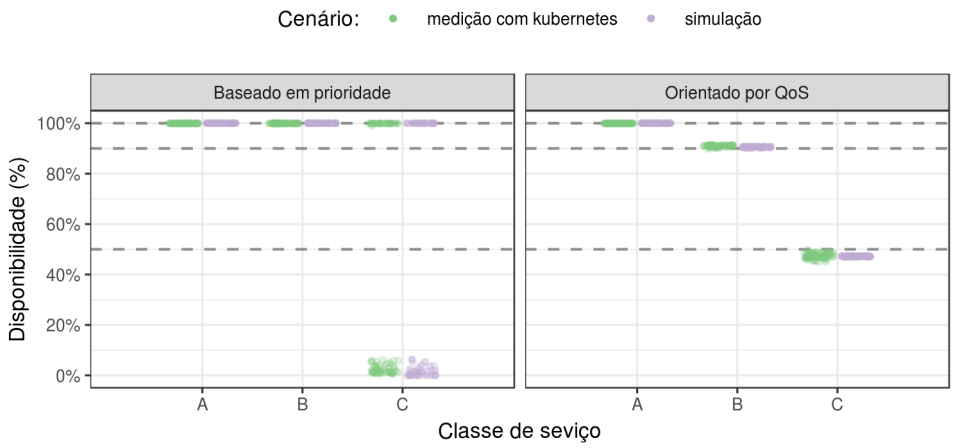


Figura 5.1: Disponibilidades finais das requisições nos experimentos de simulação e medição usando o Kubernetes: requisições de múltiplas classes.

vez, o escalonador orientado por QoS entregou disponibilidades para todas as requisições que são muito próximas de seus respectivos SLOs.

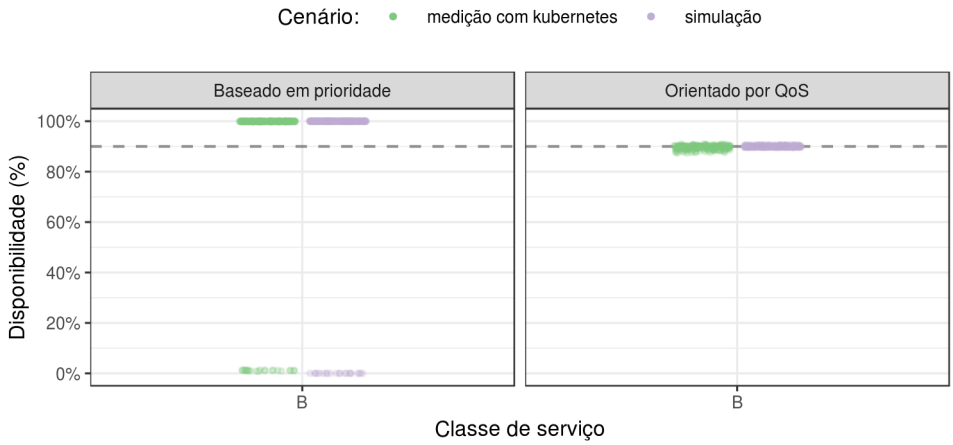


Figura 5.2: Disponibilidades finais das requisições nos experimentos de simulação e medição usando o Kubernetes: requisições de uma única classe.

A Figura 5.2 exibe os resultados para o segundo teste, usando a mesma notação utilizada na Figura 5.1. É possível observar que ambos os escalonadores funcionaram como esperado. O escalonador baseado em prioridade manteve as disponibilidades das requisições que foram alocadas em 100% e as últimas 21 requisições submetidas receberam 0% de disponibilidade. Como a infraestrutura é capaz de alocar 200 requisições simultaneamente, essas últimas requisições foram submetidas quando a infraestrutura já estava completamente

utilizada e nunca foram alocadas. Por sua vez, o escalonador orientado por QoS entregou disponibilidades que são muito próximas de suas respectivas metas de QoS.

Por último, em ambos os testes, as disponibilidades finais calculadas nos experimentos de simulação e medição são muito próximas umas das outras. Os intervalos das disponibilidades foram um pouco mais amplos nos experimentos de medição comparados aqueles obtidos nas simulações. Isso se deve ao fato dos experimentos de medições executarem em um ambiente menos controlado. No entanto, o teste estatístico T revela que não há diferença estatística significante entre os resultados dos experimentos de simulação e medição.

Capítulo 6

Avaliação

Este capítulo apresenta e discute os resultados obtidos a partir dos experimentos de simulação realizados. Nesse sentido, analisa-se a QoS oferecida por ambos os escalonadores, bem como o impacto desta nas penalidades as quais o provedor está sujeito em caso de não cumprimento dos SLAs. Além disso, também avalia-se a QoS sob o ponto de vista da justiça no provisionamento de recursos. Por isso, verifica-se quão justo é o provisionamento de recursos gerado por ambos os escalonadores para requisições que estão competindo pelos mesmos recursos.

6.1 QoS provida para as requisições

Com o propósito de comparar a QoS oferecida por cada escalonador, considera-se as disponibilidades finais entregues para cada requisição como um par (x, y), onde x é a disponibilidade provida quando o escalonador baseado em prioridade foi utilizado e y é a disponibilidade provida pelo escalonador orientado por QoS. A Figura 6.1 apresenta os mapas de calor que mostram esses dados de disponibilidade de forma pareada. Os mapas apresentam as disponibilidades finais de todas as requisições de todas as cargas de trabalho analisadas. No eixo x, os mapas apresentam as disponibilidades quando o escalonador baseado em prioridade foi utilizado, enquanto que no eixo y apresentam os valores obtidos quando o escalonador orientado por QoS foi executado. Cada quadrado em um mapa representa um intervalo diferente para a disponibilidade oferecida, com intervalos de tamanhos estritamente inferiores a 5%; por exemplo, o quadrado no canto inferior esquerdo do mapa representa disponibili-

dades no intervalo $[0\%, 5\%)$. A intensidade da cor dos quadrados é proporcional ao número de requisições cujas disponibilidades correspondem ao intervalo do quadrado em questão. Quanto mais escuro é o quadrado, mais requisições estão representadas no mesmo. Uma vez que todas as requisições estão emparelhadas, faz sentido compará-las independentemente da carga de trabalho e do cenário de contenção de recursos no momento em que elas estiveram ativas. Esses mapas de calor são agrupados em três partes diferentes, uma para cada capacidade de infraestrutura testada (N, $0,9N$ e $0,8N$). Para enfatizar a diferença da QoS provida para as diferentes classes de serviço, os resultados também são agrupados por classe de serviço. Portanto, cada mapa de calor apresentado na Figura 6.1 está relacionado a uma classe de serviço e a um tamanho de infraestrutura. Cada mapa de calor divide os dados em 4 quadrantes com significados específicos:

1. O quadrante *superior direito* (em roxo) contém as disponibilidades das requisições que tiveram seus SLOs satisfeitos por ambos os escalonadores.

2. O quadrante *inferior direito* (em azul) contém as disponibilidades das requisições que tiveram seus SLOs atendidos pelo escalonador baseado em prioridade, mas não satisfeitos pelo escalonador orientado por QoS.

3. O quadrante *superior esquerdo* (em verde) contém as disponibilidades das requisições que tiveram seus SLOs satisfeitos pelo escalonador orientado por QoS, mas não atendidos pelo escalonador baseado em prioridade.

4. O quadrante *inferior esquerdo* (em vermelho) contém as disponibilidades das requisições cujos SLOs não foram atendidos por ambos os escalonadores.

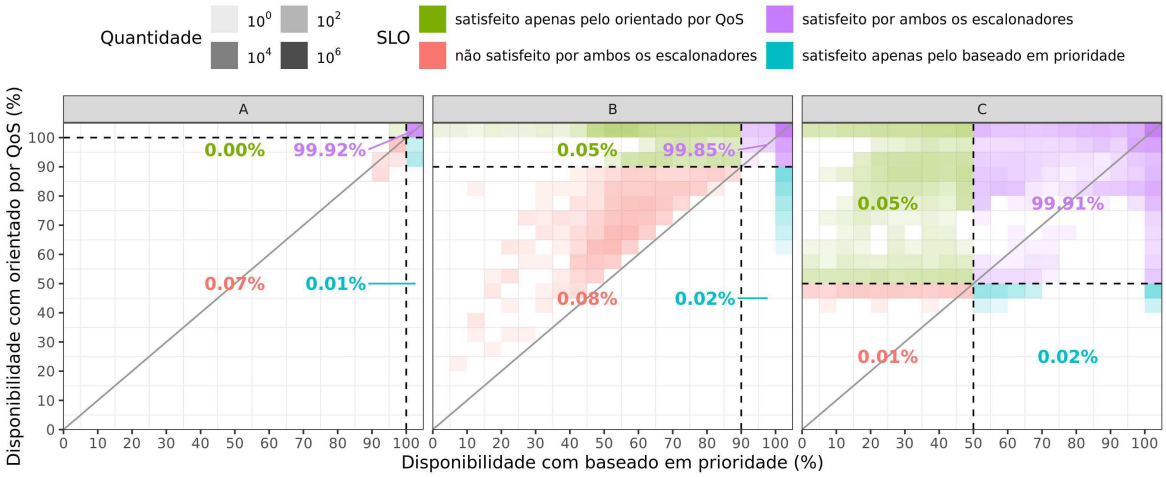
O percentual associado com cada quadrante representa a fração de requisições que se enquadram no quadrante. Assim sendo, para cada classe de serviço e capacidade de infraestrutura, a satisfação do SLO com o escalonador baseado em prioridade pode ser calculado somando os percentuais associados com os quadrantes direitos superior e inferior (roxo e azul). Por outro lado, a satisfação do SLO com o escalonador orientado por QoS pode ser computado somando os percentuais associados com os quadrantes superiores esquerdo e direito (verde e roxo).

Além disso, cada mapa de calor também uma linha de identidade (diagonal). Esta linha auxilia na análise dos resultados. Independente do cenário, cada ponto que está acima da linha de identidade representa uma requisição cuja QoS entregue pelo escalonador orientado por QoS foi maior que a QoS provida pelo escanador baseado em prioridade.

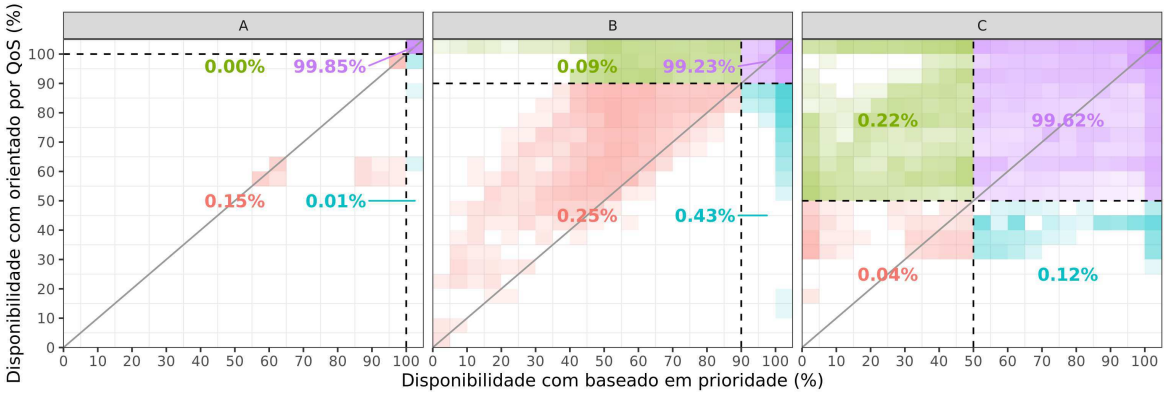
6.1.1 Satisfação do SLO

O escalonador orientado por QoS visa manter a QoS das requisições acima de seus respectivos SLOs. Além disso, em períodos de maior contenção de recursos, este escalonador busca prover QoS semelhante para requisições de mesma classe que estejam competindo pelos mesmos recursos. Isso significa que este escalonador é mais propenso a oferecer QoS mais alta no geral, enquanto que, em períodos de maior contenção, ele prioriza a justiça no provisionamento ao invés da satisfação do SLO.

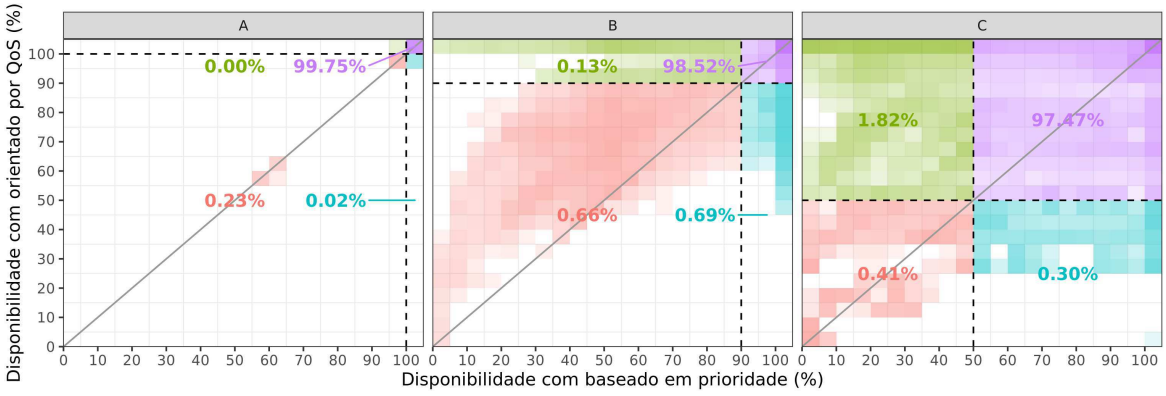
Mantendo-se a mesma carga de trabalho, a diminuição da capacidade da infraestrutura leva a um aumento na contenção de recursos do sistema. Cenários com maior contenção de recursos tornam o escalonamento mais desafiador. Como esperado, quando a capacidade da infraestrutura é reduzida, a satisfação do SLO diminui. Esse comportamento é observado em todos os quadrantes nos mapas de calor apresentados na Figura 6.1. A taxa de atendimento dos SLOs alcançada pelo escalonador orientado por QoS, comparada com a obtida pelo escalonador baseado em prioridade, é essencialmente a mesma para as requisições da classe *A*. Isso ocorre em todas as capacidades de infraestrutura analisadas. Já para para as requisições das classes *B* e *C*, esse comportamento se repete apenas para o cenário com infraestrutura maior. Considerando os outros tamanhos de infraestrutura, a satisfação dos SLOs é um pouco menor para as requisições *B* e um pouco maior para as *C*. A pequena redução no atendimento dos SLOs em alguns casos, resultante da utilização do escalonador orientado por QoS, é explicada pelo fato que, ao tentar prover QoS mais próxima do SLO para todas as requisições, este escalonador pode aumentar o número de requisições com QoS menor que a prometida. No entanto, isso é compensado pela QoS geralmente fornecida melhor e pelo provisionamento mais justo para as requisições concorrentes que estão ativas ao mesmo tempo. Esses benefícios são cuidadosamente analisados a seguir.



(a) Infraestrutura com capacidade N



(b) Infraestrutura com capacidade 0,9N



(c) Infraestrutura com capacidade 0,8N

Figura 6.1: QoS provida para requisições usando os escalonadores orientado por QoS e baseado em prioridade em infraestruturas com diferentes capacidades.

6.1.2 QoS para requisições da classe *A*

No geral, as disponibilidades das requisições da classe *A* não são diferentes quando os escalonadores baseado em prioridade e orientado por QoS são utilizados. Independentemente da capacidade da infraestrutura, quase todas as requisições *A* têm seus objetivos de QoS atendidos por ambos os escalonadores ($99,75\%$ no pior caso). A classe de serviço *A* é a mais exigente em termos de QoS (100%), e, consequentemente, essas requisições são classificadas como as mais importantes por ambos os escalonadores. Por esta razão, quando necessário, os dois escalonadores preemptam todas as instâncias de outras classes (*B* e *C*) em benefício de instâncias *A*. Portanto, já era esperado que os resultados fossem muito similares.

As diferenças para esta classe ocorrem devido aos diferentes mecanismos de empacotamento utilizados pelos escalonadores baseado em prioridade e orientado por QoS. Esta diferença pode ser observada quando se analisa os resultados da carga 02. Como pode ser observado na Figura 4.2, esta carga tem um pico de requisições *A* por volta do dia 5. Nesse instante, 150 requisições desta classe são admitidas de uma única vez, cada uma solicitando aproximadamente 31% de CPU do maior servidor dos rastros da Google. Nenhum dos escalonadores conseguiu alocar recursos para todas as requisições *A*, mesmo considerando preemptar todas as instâncias das classes *B* e *C*. Embora os dois escalonadores decidam pela preempção das instâncias *B* e *C* em benefício de uma instância *A*, as funções de pontuação que eles usam são diferentes. O escalonador orientado por QoS considera a métrica de QoS, enquanto o escalonador baseado em prioridade considera a prioridade da classe e o instante de admissão da requisição. Como resultado, a alocação real executada pelos escalonadores não é a mesma, e, as instâncias *A* podem ser alocadas em diferentes servidores dependendo do escalonador utilizado. Por isso, observa-se algumas instâncias *A* no quadrante inferior direito (azul). Isso acontece porque, para a carga 02, o escalonador baseado em prioridade foi capaz de alocar mais instâncias *A* que o escalonador orientado por QoS. É importante destacar que esta situação é específica para a carga de trabalho usada e ocorreu por acaso. Para outras cargas, o resultado oposto — escalonador orientado por QoS conseguir alocar mais instâncias *A* que o escalonador baseado em prioridade — também seria possível.

6.1.3 QoS para requisições da classe *B*

Analisando a QoS provida para as requisições da classe *B* (mapas de calor no centro da Figura 6.1), é possível observar que, no geral, o escalonador orientado por QoS entregou disponibilidades maiores que o escalonador baseado em prioridade. Este resultado pode ser melhor visualizado quando se observa a linha de identidade (diagonal) nesses gráficos. É possível verificar que a maioria dos pontos está concentrada acima da linha de identidade. Esse comportamento se deve ao fato que ambos os escalonadores atuam de formas diferentes durante períodos com contenção de recursos. Em particular, o escalonador orientado por QoS é capaz de alternar as instâncias em execução, independentemente de suas classes, com base simplesmente na QoS atual provida para as requisições. Em geral, isso faz as instâncias alcançarem uma QoS mais próxima de seus respectivos objetivos. Por outro lado, durante períodos com contenção, o escalonador baseado em prioridade não alterna as instâncias de mesma classe em execução. Ao invés disso, este escalonador mantém algumas instâncias sempre em execução e outras sempre pendentes. Isso é feito com base nos tempos de admissão de suas respectivas requisições. Como resultado deste comportamento: *(i)* muitas instâncias recebem uma QoS muito alta (observa-se pelos quadrados mais escuros na linha vertical de 100% de disponibilidade no quadrante azul); e, *(ii)* muitas instâncias recebem QoS muito menor (observa-se pela distribuição mais dispersa no quadrante verde). Na maioria desses casos, a QoS entregue às instâncias foi maior quando o escalonador orientado por QoS foi executado. Esses resultados evidenciam a utilização mais efetiva dos recursos alcançada pelo escalonador orientado por QoS, o qual visa manter a QoS de cada instância o mais próximo possível de seu objetivo. Nesse contexto, o uso mais efetivo dos recursos significa que os recursos são alocados de forma que mais requisições conseguiram receber QoS igual ou acima da prometida, e, quando não foi possível atender o SLO de todas as requisições, a alocação dos recursos foi executada de forma mais justa (igualitária).

Observando os quadrantes verde e azul, verifica-se os diferente resultados obtidos pelos escalonadores. Como já mencionado, no quadrante azul estão presentes as requisições que tiveram seus SLOs satisfeitos apenas quando o escalonador baseado em prioridade foi utilizado. A maior parte dessas requisições recebe 100% de disponibilidade deste escalonador. A QoS entregue pelo escalonador orientado por QoS para essas requisições raramente atingem valores abaixo de 45%, e, na maioria das vezes não ficou tão distante do objetivo (90%). Por

outro lado, quando se analisa as requisições *B* com SLOs atendidos apenas pelo escalonador orientado por QoS (quadrante verde), constata-se que as disponibilidades providas pelo escalonador baseado em prioridade foram mais dispersas, alcançando disponibilidades muito baixas. Nos casos onde ambos os escalonadores não atendem o SLO (quadrante vermelho), a concentração de pontos acima da linha de identidade é clara. Neste quadrante, também é possível observar cenários onde o escalonador orientado por QoS oferece baixa QoS para algumas requisições. No entanto, é importante enfatizar que para essas mesmas requisições, o escalonador baseado em prioridade também entregou QoS ruim. Isso é uma indicação que o escalonador orientado por QoS oferece QoS muito ruim apenas durante períodos com alta contenção de recursos.

6.1.4 QoS para requisições da classe *C*

Examinando a QoS oferecida para as requisições da classe *C* (gráficos no lado direito da Figura 6.1), verifica-se que, no geral, o escalonador orientado por QoS também entregou disponibilidades maiores que o escalonador baseado em prioridade. Este fato fica evidente quando se considera a linha de identidade nesses gráficos. É possível observar que a maioria dos pontos estão concentrados acima da linha de identidade.

Uma vez que as requisições *C* são as menos exigentes em termos de QoS (50%), elas podem permanecer mais tempo pendentes sem comprometer a satisfação de seus SLOs que as requisições de outras classes. Isso significa que o escalonador orientado por QoS tem mais folga para lidar com essas requisições. Basicamente, este escalonador alterna as instâncias *C* em execução, preemptando-as e iniciando-as de forma controlada para entregar uma disponibilidade que seja a maior e mais justa possível. Por esta razão, o escalonador orientado por QoS aumenta a taxa de atendimento dos SLOs para as requisições *C* em comparação com o escalonador baseado em prioridade, que prioriza as instâncias que foram admitidas mais cedo.

Novamente, verifica-se que no quadrante verde existem muitos casos de disponibilidades muito ruins providas pelo escalonador baseado em prioridade (que não alterna instâncias da mesma classe em execução). Embora essas instâncias estejam recebendo QoS muito ruim, há provavelmente outras com QoS muito alta, muito acima do objetivo. Por outro lado, quando apenas o escalonador baseado em prioridade satisfez o SLO (quadrante azul), as

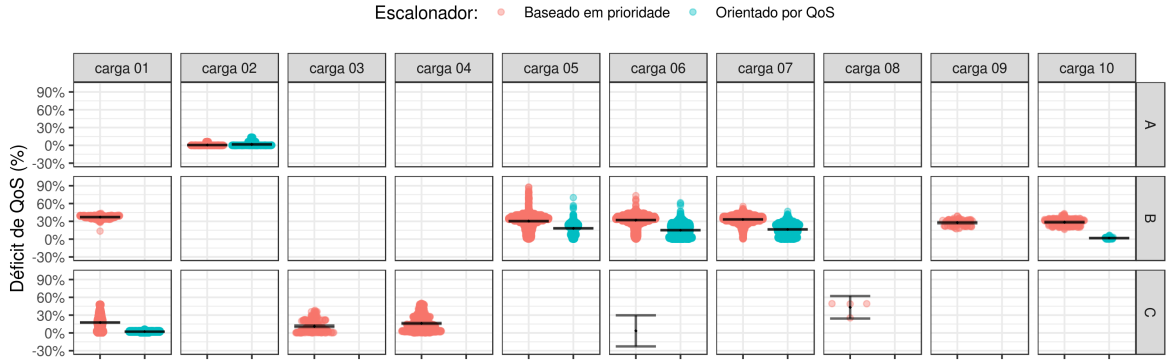
disponibilidades providas pelo escalonador orientado por QoS foram na maioria das vezes acima de 25%. Em casos onde ambos os escalonadores não satisfizeram os SLOs (quadrante vermelho), mais uma vez, o escalonador orientado por QoS entregou QoS muito ruim apenas quando o escalonador baseado em prioridade também fez o mesmo.

6.1.5 Distribuição dos déficits de QoS

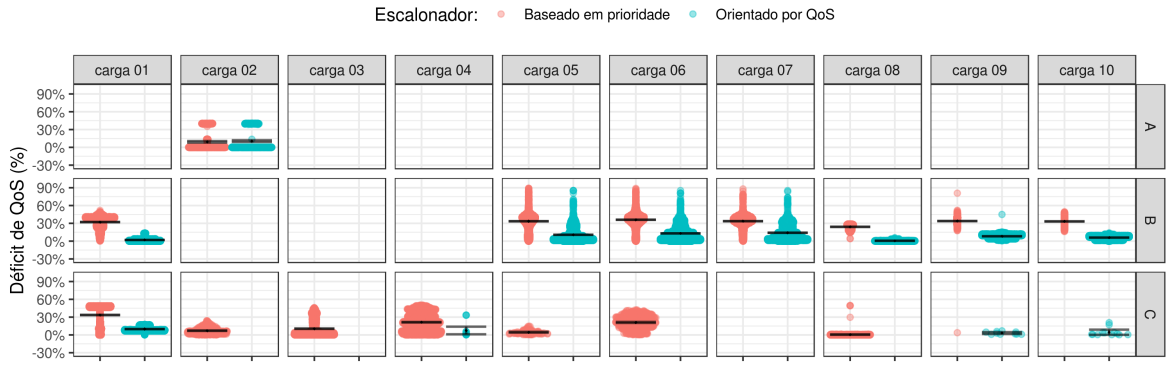
A Figura 6.2 apresenta os déficits de QoS das requisições que tiveram seus respectivos SLOs não atendidos. Esta métrica é calculada por requisição e é definida pela diferença entre a disponibilidade prometida (SLO) e a disponibilidade final oferecida para a requisição. Nesta figura, quanto maior a largura do gráfico, maior a quantidade de pontos com o valor específico. O intervalo de confiança de 95% dos déficits de QoS medido para cada cenário também está plotado nos gráficos. Os sub-gráficos vazios significam que nenhuma violação de SLO ocorreu para os cenários específicos. Os déficits de QoS são apresentados individualmente por carga de trabalho analisada e agrupados por classe de serviço.

No geral, verifica-se déficits de QoS maiores quando o escalonador baseado em prioridade foi utilizado em comparação ao mesmo cenário com o escalonador orientado por QoS. Isso ocorre independentemente da carga de trabalho e da capacidade da infraestrutura. Além disso, há mais cenários onde o escalonador orientado por QoS conseguiu atender todos os SLOs, enquanto que o escalonador baseado em prioridade não conseguiu. Para ambos os escalonadores, o número de requisições com déficit de QoS aumenta quando a capacidade da infraestrutura diminui. Da mesma maneira, a extensão dos déficits também aumenta quando a capacidade da infraestrutura diminui. No entanto, no geral, o escalonador orientado por QoS conseguiu limitar os déficits a um intervalo menor e com valores mais baixos.

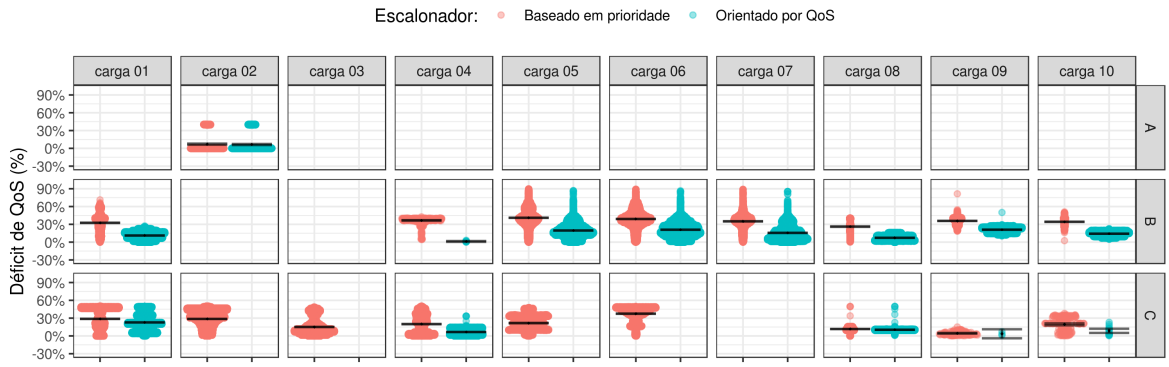
Quando a capacidade da infraestrutura foi N (Figura 6.2a), o escalonador orientado por QoS conseguiu evitar déficits em 4 de cargas de trabalho: 03, 04, 08 e 09. Essas cargas estão entre aquelas com maior pico de requisições sem ser da classe *A* (carga 08) ou com uma boa proporção de demanda pela classe *C* (carga 03 e 04) ou ambos (carga 09) — observar Figuras 4.2 e 4.3. Com base em como as capacidades das infraestruturas foram definidas (detalhado na Seção 4.6), as cargas com altos picos de demandas possuem recursos extras, permitindo que ambos os escalonadores aloquem mais requisições ao longo do tempo. No entanto, o escalonador orientado por QoS tem mais flexibilidade para alternar as instâncias



(a) Infraestrutura com capacidade N



(b) Infraestrutura com capacidade 0,9N



(c) Infraestrutura com capacidade 0,8N

Figura 6.2: Déficit de QoS para as requisições usando os escalonadores baseado em prioridade e orientado por QoS em infraestruturas com diferentes capacidades.

em execução, levando ao uso mais eficiente dos recursos quando comparado com o escalonador baseado em prioridade. O uso eficiente dos recursos significa alocá-los de forma que mais requisições conseguiram receber QoS igual ou acima da prometida, e, quando não foi possível atender o SLO de todas as requisições, a alocação dos recursos foi executada de

forma a produzir déficits menores e menos variáveis.

O escalonador orientado por QoS ainda conseguiu atender 100% dos SLOs para a carga 03 quando a capacidade da infraestrutura foi reduzida para $0,9N$ (Figura 6.2b) e para $0,8N$ (Figura 6.2c). Nesta carga, a demanda por recursos para instâncias da classe *A* é constante ao longo do tempo. Além disso, esta carga tem proporcionalmente uma boa quantidade de demanda para classe *C* (observar Figuras 4.2 e 4.3). Essas instâncias podem executar de forma alternada quando o escalonador orientado por QoS é executado, enquanto que o escalonador baseado em prioridade mantém algumas em execução e outras pendentes. Ao lidar com as instâncias da classe *C*, o escalonador orientado por QoS foi capaz de escalonar toda a carga de maneira mais eficiente, *i.e.* de forma que o SLO de todas as requisições fossem atendidos.

Ainda analisando os déficits de QoS expostos na Figura 6.2, em alguns cenários é possível notar que existem déficits para a classe *B* e não existem para a classe *C* quando o escalonador orientado por QoS é usado (por exemplo, as cargas 05, 06 e 07 em todas as infraestruturas avaliadas). Essas situações podem, erroneamente, dar a impressão de que o escalonador orientado por QoS favoreceu requisições *C* em detrimento de requisições *B*. Em um primeiro momento é possível inferir que este escalonador poderia liberar recursos alocados para as instâncias *C* para entregá-los para instâncias *B*. Na realidade, essas cargas têm alguns picos de demandas para a classe *B* (observar Figuras 4.2 e 4.3). Em alguns desses picos, ambos os escalonadores não conseguem executar todas as instâncias *B* (mesmo considerando a preempção de todas as instâncias *C*). Nesses momentos, o escalonador baseado em prioridade começa a enfileirar as novas requisições *B* admitidas. O escalonador orientado por QoS atua de forma diferente: assim que ele identifica que não há recursos suficientes para executar a demanda da classe *B*, ele começa a preemptar recursos de algumas instâncias *B* (depois de preemptar todas as instâncias *C* do servidor escolhido). A partir desse momento, o escalonador orientado por QoS alterna as requisições *B* em execução de tal forma que a diferença na QoS momentânea provida para essas instâncias seja minimizada. Durante períodos de maior contenção de recursos, todos os recursos foram utilizados para executar instâncias associadas com as classes mais importantes. Após esses períodos, o escalonador orientado por QoS foi mais eficiente em escolher as requisições certas para executar, de tal forma que todas as requisições *C* alcançaram seus respectivos SLOs. Portanto, o escalonador orientado

por QoS conseguiu atender o SLO de todas as requisições da classe C não porque priorizou essas requisições, mas porque ele foi mais eficiente em determinar quais das requisições devem ter suas respectivas instâncias em execução e pendentes ao longo do tempo.

6.2 Impacto das violações nas penalidades

Com o objetivo de comparar os custos relacionados com as penalidades decorrentes de violações, calculou-se a penalidade P_j (definida pela Equação 4.8) para cada requisição j que teve seu SLA violado em cada teste nos experimentos de simulação. Dado um cenário de execução, o custo total do provedor é definido pela soma das penalidades de todas as requisições que não tiveram seus respectivos SLAs cumpridos ao longo deste cenário. A Figura 6.3 apresenta quão maior foi o custo total com as penalidades quando o escalonador baseado em prioridade foi usado, em comparação com quando o escalonador orientado por QoS foi executado. O valor apresentado no gráfico é dado pela diferença entre o custo total das penalidades quando os escalonadores baseado em prioridade e orientado por QoS são utilizados, dividido pelo custo total das penalidades quando o escalonador orientado por QoS é executado. Para enfatizar os resultados para os diferentes cenários, os valores são agrupados por capacidade de infraestrutura analisada.

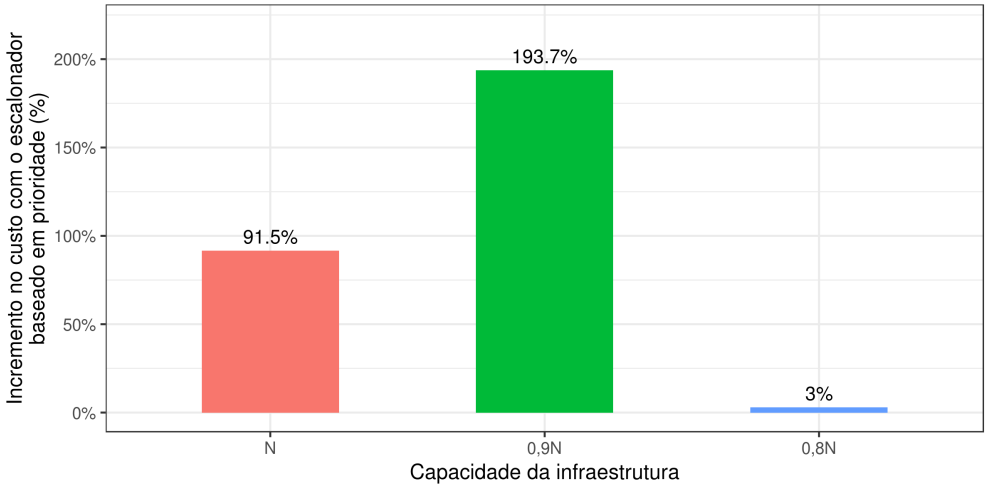


Figura 6.3: Incremento nos custos com penalidades quando o escalonador baseado em prioridade é usado em comparação com o orientado por QoS.

Os resultados mostram que, para todas as infraestruturas analisadas, o escalonador ori-

entado por QoS resultou em um custo total com penalidades menor em relação ao uso do escalonador baseado em prioridade. Quando a capacidade da infraestrutura foi N, o custo das penalidade obtido com o escalonador baseado em prioridade ($91,5\%$ maior) foi quase o dobro daquele resultante do uso do escalonador orientado por QoS. Quando considerou-se a infraestrutura com capacidade de $0,9N$, o custo total das penalidades com o escalonador orientado por QoS foi aproximadamente $1/3$ do custo obtido com o escalonador baseado em prioridade ($193,7\%$ maior). Finalmente, para a menor infraestrutura analisada (capacidade $0,8N$), o custo das penalidades proporcionada pelo escalonador baseado em prioridade não foi tão alta (3%) quanto nos outros cenários.

Portanto, nos cenários com infraestruturas N e $0,9N$, o comportamento do escalonador orientado por QoS (produzindo déficits menores e menos variáveis) impactou substancialmente no custo total das penalidades do provedor. Já no caso com infraestrutura menor (*i.e.* maior contenção de recursos), a contenção de recursos durante períodos críticos foi bastante alta, deixando pouca margem de manobra para operação do escalonador orientado por QoS. Por isso, o custo com as penalidades resultante do uso do escalonador baseado em prioridade não foi tão mais alta quanto nos cenários anteriores. No entanto, é importante destacar que desde que as etapas de planejamento de capacidade e controle de admissão sejam executadas de forma adequada, espera-se que a probabilidade de ocorrência de tais situações seja bastante reduzida.

6.3 Justiça no provisionamento de recursos

Como discutido na Seção 6.1, no geral, o escalonador orientado por QoS proporciona déficits de QoS que são menores e menos variados que os obtidos com o escalonador baseado em prioridade para os mesmos cenários. No entanto, é importante avaliar a justiça do provisionamento de recursos promovida pelos escalonadores em períodos de tempo mais curtos. Nesse sentido, apenas as requisições que estiveram ativas ao longo deste período devem ser consideradas ao invés de toda a carga de trabalho.

Dada a variabilidade das demandas das cargas ao longo do tempo (observar Figuras 4.2 e 4.3), o sistema pode estar sujeito a diferentes níveis de contenção em diferentes períodos de um mesmo cenário de simulação. Por exemplo, não há contenção de recursos em períodos

onde todas as requisições admitidas podem ser alocadas e recebem 100% de disponibilidade. Por outro lado, há alta contenção de recursos quando o provedor não consegue alocar as requisições ativas de forma que todas elas possam momentaneamente receber a disponibilidade prometida. No geral, infraestruturas menores tendem a resultar em uma quantidade maior de períodos com alta contenção de recursos. Da mesma forma, infraestruturas maiores geralmente resultam em um número maior de períodos de baixa (ou nenhuma) contenção de recursos. Todavia, períodos de baixa e alta contenção de recursos podem ocorrer em infraestruturas com as três capacidades analisadas, dependendo de como a demanda da carga de trabalho varia ao longo do tempo.

Em períodos com contenção de recursos, o escalonador orientado por QoS decrementa igualmente a QoS das requisições de uma mesma classe, provendo QoS similar para essas requisições. Por esta razão, a QoS entregue para uma requisição em um ponto no tempo específico está diretamente relacionada ao nível de contenção de recursos neste mesmo ponto no tempo. Com o propósito de avaliar como se comportam os escalonadores de acordo com os diferentes níveis de contenção, a execução dos experimentos foi dividida em intervalos mais curtos de 10 minutos. Portanto, cada teste do experimento que abrange 29 dias foi dividido em 4.176 intervalos de 10 minutos.

Cada um dos intervalos de cada execução do experimento foi classificado levando em conta o nível de contenção de recursos. Primeiramente, identificou-se as requisições que estiveram ativas ao longo do intervalo. Uma requisição esteve ativa no intervalo se ela foi admitida durante este intervalo ou se ela foi admitida e não foi terminada antes do início do mesmo. Uma vez que as requisições ativas tenham sido identificadas, a disponibilidade parcial de cada uma delas foi calculada de acordo com a Equação 3.1. Para as requisições que concluíram durante o intervalo, o tempo t considerado no cálculo da disponibilidade foi o instante de seu término (*i.e.* esta é a disponibilidade final da requisição). No caso das requisições que não tenham terminado durante o intervalo, a disponibilidade foi calculada considerando o tempo t do final do intervalo (*i.e.* a disponibilidade da requisição ao fim do intervalo).

Em seguida, os resultados obtidos com o escalonador baseado em prioridade são utilizados para classificar um intervalo segundo seu nível de contenção como segue:

1. *sem contenção*: há mais recursos que o necessário para alocar todas as requisições ad-

mitidas. Por isso, todas as requisições de cada classe conseguem ter recursos alocados durante o intervalo de forma que todas elas recebem 100% de disponibilidade;

2. *baixa contenção*: o escalonador baseado em prioridade consegue atender o SLO de todas as requisições admitidas. Neste cenário, todas as requisições das classes *A* e *B* recebem 100% de disponibilidade, enquanto as requisições da classe *C* recebem disponibilidade maior ou igual a prometida (50%), mas abaixo de 100%;

3. *média contenção*: o escalonador baseado em prioridade não consegue entregar a QoS prometida para todas as requisições da classe *C*, mas entrega 100% de disponibilidade para as requisições das classes *A* e *B*;

4. *alta contenção*: não há recursos suficientes para o escalonador baseado em prioridade atender o SLO para todas as requisições das classes *A* e *B*.

Considerando como os intervalos foram classificados, não há diferença entre os resultados obtidos pelos escalonadores em intervalos sem contenção. Uma vez que a quantidade de recursos é maior que a demanda, ambos os escalonadores conseguem alocar as requisições de tal forma que todas elas recebam 100% de disponibilidade. Para os intervalos onde houve contenção de recursos, a avaliação dos escalonadores se deu através de três métricas baseadas nas disponibilidades parciais das requisições que estiveram ativas em cada intervalo:

(i) *Desigualdade das disponibilidades:* dá uma ideia da justiça no provisionamento de recursos promovido pelo escalonador. Esta métrica é definida pelo coeficiente de Gini [9] das disponibilidades das requisições ativas no intervalo. Quanto menor é a desigualdade das disponibilidades oferecidas para requisições de uma classe particular, mais justo é o provisionamento de recursos da política de escalonamento;

(ii) *Disponibilidade mínima:* representa a menor disponibilidade entregue para uma requisição que esteve ativa no intervalo. Esta métrica indica o pior caso para a disponibilidade, considerando as requisições ativas no intervalo;

(iii) *Satisfação parcial do SLO:* indica a proporção de requisições ativas no intervalo que estão momentaneamente atendendo seus respectivos SLOs. É importante destacar que a disponibilidade final da requisição pode ser incrementada ou reduzida ao longo de

sua execução. Essa alteração pode afetar a QoS da requisição de forma a atender ou não seu SLO ao final de sua execução.

A Figura 6.4 apresenta os resultados obtidos para essas três métricas considerando os intervalos de todos os experimentos executados. No lado esquerdo estão os resultados para os intervalos com baixa contenção de recursos, ao centro estão os obtidos para os intervalos com média contenção e à direita para os intervalos com alta contenção. Os intervalos de confiança de 95% dos resultados obtidos para as diferentes métricas também estão plotados nos gráficos. Para enfatizar as diferenças, os valores também estão agrupados por classe de serviço.

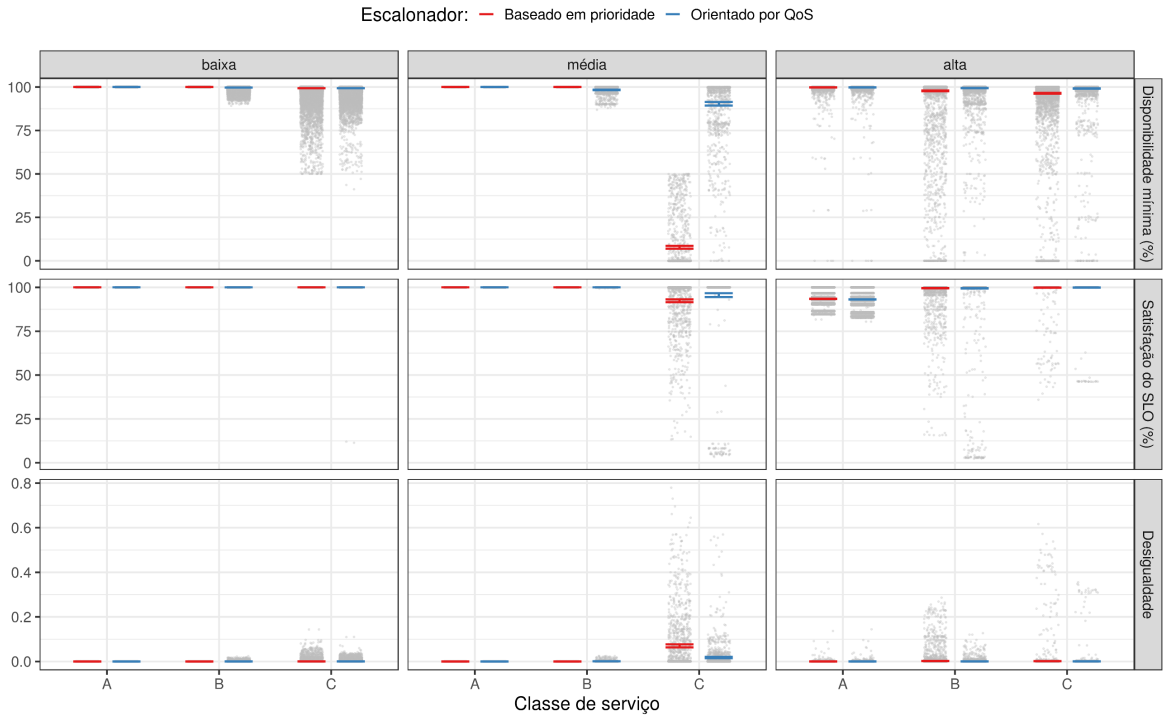


Figura 6.4: Intervalos de confiança da disponibilidade mínima, satisfação parcial do SLO e desigualdade das disponibilidades das requisições ativas por classe de serviço em intervalos de 10 minutos com baixa, média e alta contenção de recursos.

Analisando a Figura 6.4, é possível verificar que, para os intervalos com baixa contenção, não há diferença estatística significativa entre os resultados obtidos com os escalonadores baseado em prioridade e orientado por QoS. Destaca-se que o escalonador orientado por QoS conseguiu concentrar as disponibilidades mínimas das requisições da classe *C* mais distantes

(positivamente) da QoS prometida (50%). Além disso, este escalonador também concentrou a desigualdade das disponibilidades das requisições *C* mais próximas de 0, indicando um provisionamento mais justo. No entanto, ambos os escalonadores conseguiram satisfazer os SLOs das requisições admitidas e não há diferença significativa entre seus resultados.

Considerando os intervalos com média contenção, o escalonador orientado por QoS reduziu um pouco a QoS excedente das requisições da classe *B* com o intuito de melhorar a QoS provida para as requisições da classe *C*. Este resultado pode ser visualizado pela redução das disponibilidades mínimas das requisições *B* e o incremento substancial desta métrica para as requisições *C*, quando o escalonador orientado por QoS foi usado. Como já discutido, este escalonador também pode alternar as instâncias de mesma classe em execução. Ao permitir essas preempções, o escalonador proposto aumentou significantemente a satisfação de SLO para a classe *C* sem afetar negativamente esta métrica para as classes mais importantes (*A* e *B*). Além disso, como esperado, também verificou-se que o escalonador orientado por QoS elevou um pouco a desigualdade das disponibilidades das requisições *B* enquanto diminuiu significativamente esta métrica para as requisições *C*. Isso ocorreu porque, quando o escalonador baseado em prioridade foi utilizado, todas as requisições da classe *B* receberam 100% de disponibilidade nesses intervalos, levando a um coeficiente de Gini igual a 0 (igualdade perfeita). Por esta razão, quando o escalonador orientado por QoS preempta algumas instâncias *B* em benefício de outras instâncias *B* ou *C*, ele está introduzindo desigualdade para as disponibilidades da classe *B*. No entanto, uma vez que isso não afeta negativamente a satisfação dos SLOs para esta classe, no geral, esse comportamento não é um problema neste cenário. Finalmente, o escalonador orientado por QoS também conseguiu escalonar as instâncias de tal forma que a QoS provida para as requisições *C* foram mais similares (menos desiguais).

Examinando os intervalos com alta contenção, constatou-se que o escalonador orientado por QoS incrementou as disponibilidades mínimas para as requisições das classes *B* e *C*. Isso ocorreu porque este escalonador pôde alternar as instâncias de mesma classe em execução. Consequentemente, as desigualdades das disponibilidades entre as requisições de uma mesma classe foram reduzidas. No entanto, este comportamento não foi suficiente para melhorar a satisfação de SLO para nenhuma das classes.

Além disso, com base em como os intervalos foram classificados, todos os intervalos

onde ao menos uma requisição da classe *A* não conseguiu ter sua instância alocada no momento de admissão são classificados como intervalos com alta contenção. Isso ocorre porque uma requisição *A* necessita estar sempre em execução para que tenha seu SLO (100%) satisfeito. No entanto, como discutido anteriormente (Seção 6.1.2), requisições da classe *A* não satisfazem seu SLO apenas quando há um pico na demanda desta classe. Além disso, diferenças raramente ocorrem na QoS provida para uma requisição da classe *A* quando ambos os escalonadores são executados, visto que ambos consideram esta classe como a mais importante. Esse comportamento é confirmado quando, na Figura 6.4, analisa-se a satisfação do SLO para as requisições da classe *A* nos intervalos com alta contenção.

Em resumo, os intervalos com média contenção são aqueles onde os ganhos em usar o escalonador orientado por QoS são mais evidentes em comparação com o escalonador baseado em prioridade. Ressalta-se que as etapas de planejamento de capacidade e controle de admissão operam em conjunto com o escalonamento para evitar cenários de super provisionamento (para manter os custos o mais baixo possível) e sub provisionamento (para manter níveis adequados de QoS), portanto, nesse contexto, um escalonador que opera bem em cenários com contenção moderada de recurso é muito útil. Além disso, o escalonador orientado por QoS também entregou QoS de forma mais justa que o escalonador baseado em prioridade, inclusive em períodos com nível de contenção mais alto. Isso é interessante porque permite que os usuários tenham maior previsibilidade da indisponibilidade esperada para suas requisições.

Capítulo 7

Aspectos Práticos

As análises discutidas nos capítulos anteriores avaliam o escalonador orientado por QoS sob a perspectiva da QoS provida. No entanto, também é importante avaliar aspectos práticos relacionados à implementação deste escalonador em um sistema real. Este capítulo apresenta duas análises realizadas sob este ponto de vista. Em um primeiro momento, avalia-se o desempenho do escalonador sob a perspectiva da quantidade de processamento necessário para que ele tome decisões. Em seguida, com base em um sistema real, caracteriza-se a forma como normalmente um consumidor submete suas cargas de trabalho para este sistema, e, define-se como o escalonador orientado por QoS pode ser implementado nesse contexto.

7.1 Análise de Desempenho

A análise de desempenho do escalonador ocorre através da comparação das quantidades de processamento necessário para que ambos os escalonadores (baseado em prioridade e orientado por QoS) tomem decisões ao longo do tempo sob as mesmas condições (infraestrutura e carga de trabalho). Nesse contexto, é importante destacar que, quando há contenção de recursos, é esperado que o escalonador orientado por QoS leve a um custo operacional maior que o escalonador baseado em prioridade. Isso ocorre por dois motivos: *processamento periódico da fila* de espera e *decisões baseadas na métrica de QoS*.

Além dos eventos que desencadeiam a execução do escalonador baseado em prioridade, o escalonador orientado por QoS também *processa a fila periodicamente* (ver Seção 3.5). Além disso, como consequência da contenção de recursos, a fila de espera sempre possui

instâncias que não conseguiram ser alocadas quando o escalonador foi executado. Por estas razões, espera-se que a fila (não vazia) seja processada mais vezes quando o escalonador orientado por QoS é executado. Esse comportamento leva a um custo operacional maior em comparação ao custo do escalonador baseado em prioridade.

Com relação ao *uso da métrica de QoS* na tomada de decisões, isso eleva o custo operacional por causa da característica dinâmica desta métrica. Como a métrica de QoS ($Q_j(t)$) sofre alteração ao longo do tempo (TTV e *recoverability* de uma requisição mudam com o passar do tempo), dado o mesmo conjunto de requisições ativas no sistema, o escalonador orientado por QoS pode tomar decisões diferentes quando executado em dois instantes de tempo distintos. Esse comportamento não ocorre com o escalonador baseado em prioridade, que toma decisões com base em métricas tipicamente estáticas (prioridade e tempo de admissão). Portanto, dado o mesmo conjunto de requisições no sistema, as decisões sobre as alocações são as mesmas se o escalonador for executado em dois instantes de tempo diferentes. Essa característica facilita a reutilização de decisões anteriores (por exemplo, a implementação de *cache*) para o escalonador baseado em prioridade e dificulta para o escalonador orientado por QoS, consequentemente, elevando o custo operacional para este último.

Portanto, faz-se necessário avaliar se o custo de operação adicional do escalonador orientado por QoS torna sua implementação inviável. Nesse sentido, este estudo analisa a quantidade de operações realizadas por ambos os escalonadores quando submetidos às mesmas condições. A ideia central é, via experimentos de simulação, submeter uma carga de trabalho para ser escalonada em uma mesma infraestrutura duas vezes. Uma vez executando o escalonador orientado por QoS e outra usando um escalonador baseado em prioridade. Ao final da execução de cada teste do experimento, os números de operações realizadas são contabilizados. Considera-se a execução da verificação de viabilidade e o cálculo da pontuação de um servidor (descritas nas Seções 3.4.1 e 3.4.2) para uma instância pendente como uma operação executada pelo escalonador.

Com o intuito de tornar a solução de escalonamento proposta neste trabalho menos custosa, algumas otimizações foram implementadas. Para manter a justiça de comparação, otimizações também são implementadas para o escalonador baseado em prioridades. Essas otimizações foram implementadas nos modelos de simulação para fins de análise. As descrições dessas otimizações para cada modelo de simulação, bem como a metodologia, o projeto

experimental e os resultados deste estudo são detalhados nas seções a seguir.

7.1.1 Otimizações para o Escalonador Baseado em Prioridade

As otimizações implementadas no simulador do escalonador baseado em prioridade são as mesmas apresentadas por Verma et al. [51]. Essas são as otimizações implementadas no escalonador do sistema *Borg* da Google. O *Borg* é um sistema de gerenciamento de contêiner desenvolvido e utilizado na Google para gerenciar, escalonar e monitorar requisições de diferentes classes de serviço [11]. O escalonador do *Borg*, que é baseado em prioridade, opera com alta confiabilidade e disponibilidade. Ademais, este escalonador também é altamente escalável, permitindo executar uma carga de trabalho com milhares requisições em dezenas de milhares de servidores [51]. O *Borg* é tido como um dos primeiros sistemas de gerenciamento de contêiner em larga escala a funcionar com o grau de resiliência e completude necessários [11; 51]. Por último, esse sistema também teve forte influência no desenvolvimento dos sistemas de gerenciamento de contêiner desenvolvidos em seguida na Google, tais como o Omega [45] e o Kubernetes [5]. Desses sistemas, apenas o Kubernetes possui o seu código aberto.

No sentido de aumentar a escalabilidade do escalonador do *Borg*, três otimizações foram implementadas [51] neste sistema:

(i) *Score Caching:* As etapas de verificação de viabilidade e o cálculo da pontuação de um servidor (seja pontuação de alocação ou de custo de preempção) são atividades custosas para o escalonador. Esta otimização implementa uma *cache* para a pontuação de um servidor. Portanto, uma vez que o escalonador tenha tentado alocar uma requisição pendente sem sucesso, isso significa que este escalonador verificou a viabilidade de todos os servidores para esta requisição. Com esta otimização, novas verificações para a mesma requisição serão necessárias apenas quando alguma propriedade de um servidor (por exemplo, uma instância alocada foi terminada ou um atributo modificado) ou da requisição pendente (por exemplo, requisitos ou demandas foram modificados) for alterada. Com isso, a quantidade de verificações de viabilidade a serem executadas pelo escalonador é reduzida quando a fila de espera é processada novamente;

(ii) *Equivalence class:* As requisições pertencentes a um mesmo *job* tipicamente possuem

os mesmos requisitos (demandas e restrições). Ao invés do escalonador verificar a viabilidade e calcular a pontuação de cada servidor para cada requisição individualmente, esta otimização agrupa as requisições em classes de equivalência. As requisições em uma mesma classe de equivalência possuem requisitos e prioridades iguais. Dessa forma, o escalonador verifica a viabilidade e calcula a pontuação para uma única requisição por classe de equivalência, utilizando os resultados para todas as requisições da classe;

(iii) *Relaxed Randomization:* Quanto maior a infraestrutura, mais custoso é a execução da etapa de verificação de viabilidade para a alocar recursos a uma requisição. Isso ocorre porque mais servidores precisam ser analisados. Esta otimização permite reduzir a quantidade de servidores a serem checados neste etapa. A ideia principal é que o escalonador verifique a viabilidade dos servidores em ordem aleatória até que uma quantidade "suficiente" de servidores seja considerada elegível, ou não existam mais servidores a serem avaliados. A partir daí, não mais será analisada a viabilidade de outros servidores e o servidor escolhido estará entre os elegíveis. Por permitir verificar, geralmente, uma quantidade menor de servidores, esta otimização também reduz a quantidade de *caches* inválidas. Por *cache* inválida entende-se que o valor armazenado em *cache* para um servidor em questão não está válido por causa de alguma alteração nas propriedades do mesmo ou da requisição desde o momento que o valor foi calculado anteriormente. Por esta razão, a verificação e o cálculo da pontuação precisam ser feitos novamente.

Visto que o sistema Borg é bastante consolidado na Google [51], essas otimizações, na prática, representam melhorias viáveis implementadas em um sistema real. Como o modelo de simulação do escalonamento baseado em prioridade implementa o escalonador do Kubernetes, o qual é baseado no Borg [11], essas otimizações foram implementadas no simulador sem grandes dificuldades.

7.1.2 Otimizações para o Escalonador Orientado por QoS

Para o modelo de simulação do escalonador orientado por QoS, nem todas as otimizações discutidas na seção anterior podem ser desenvolvidas da mesma forma porque o simulador

orientado por QoS usa métrica de QoS dinâmica para a tomada de decisão. A dinamicidade dessa métrica interfere em algumas otimizações. Por exemplo, o *score caching*, como definido na seção anterior, não seria útil para o escalonador orientado por QoS uma vez que a métrica de QoS sofre alteração com o passar do tempo. Sempre que o escalonador for executado em instantes diferentes, a *cache* estará inválida. Uma alternativa seria considerar um período de validade para a *cache*. No entanto, isso faz com que o escalonador tome decisões com base na métrica desatualizada em alguns momentos, podendo impactar na QoS provida pelo mesmo. Por esta razão, a otimização *score caching* não foi implementada para este escalonador neste estudo inicial.

As otimizações implementadas para o escalonador orientado por QoS são detalhadas a seguir:

(i) *Equivalence Class:* Esta otimização é implementada de forma semelhante à desenvolvida para o escalonador baseado em prioridade. A principal diferença é que, para o escalonador orientado por QoS, a métrica de QoS é considerada na definição da classe de equivalência. Portanto, as requisições de uma mesma classe de equivalência possuem requisitos e métricas iguais. Isso reduz a probabilidade de formação de classes com uma grande quantidade de requisições. A exceção ocorre quando um *job* com muitas requisições é submetido. Por exemplo, considere que um *job* com 200 requisições é submetido, todas elas com os mesmos requisitos. No momento da admissão, todas as requisições possuem os mesmos tempos em execução (0) e pendentes (0), portanto, as métricas de QoS $(Q_j(t))$ de todas elas são iguais. No entanto, a partir do momento que algumas dessas requisições são alocadas e outras não, os tempos em execução e pendentes das requisições do *job* diferem. Consequentemente, as métricas de QoS passam a ser diferentes e essas requisições não mais fazem parte de uma mesma classe de equivalência. No caso do escalonador baseado em prioridade, esse comportamento não ocorre. Como a prioridade e a demanda das requisições são tipicamente estáticas, as requisições do *job* utilizado como exemplo sempre farão parte da mesma classe de equivalência;

(ii) *Relaxed Randomization:* Esta otimização é implementada para o escalonador orientado por QoS exatamente da mesma forma que foi implementada para o escalonador

baseado em prioridade.

(iii) *Pending Queue Pruning:* Sabe-se que as requisições são ordenadas na fila de espera de acordo com suas respectivas métricas de QoS. As requisições que o escalonador deve tentar alocar primeiro aparecem no início da fila. Uma vez que o escalonador não consegue alocar uma requisição, é possível antecipar a possibilidade de alocação das próximas. Como descrito no Capítulo 3, visto que todas as próximas requisições na fila possuem métricas de QoS maiores ou iguais às da requisição cuja alocação falhou, elas só conseguirão ser alocadas se solicitarem menos recursos. Com esta otimização, sempre que o escalonador não conseguir alocar uma requisição, esta é utilizada como referência para comparação. Portanto, antes de tentar alocar uma requisição, o escalonador checa se ela solicita menos CPU ou memória que a requisição de referência. Em caso negativo, o escalonador nem tenta alocá-la, reduzindo o número de operações executadas. Caso contrário, o escalonador inicia a etapa de verificação de viabilidade para a requisição. Em caso de nova falha na alocação, esta passa a ser a referência para comparação. É importante destacar que a referência vale para um único processamento da fila. A primeira requisição cuja alocação tenha falhado será a referência para um novo processamento da fila. Em resumo, esta otimização evita que o escalonador tente alocar requisições cuja impossibilidade de alocação pode ser antecipada. Destaca-se que esta otimização também faria sentido para o escalonador baseado em prioridades, no entanto, por não ser uma das otimizações implementadas no escalonador usado como referência, este trabalho não considerou sua implementação no modelo de simulação baseado em prioridades.

7.1.3 Metodologia

A análise de desempenho dos escalonadores foi realizada através da comparação da quantidade de operações realizadas por ambos (baseado em prioridade e orientado por QoS) quando submetidos às mesmas carga e infraestrutura. Para isso, experimentos de simulação com diferentes níveis de contenção foram executados. Para realizar esta análise, as otimizações descritas nas seções anteriores foram implementadas nos modelos de simulação apresentados nas Seções 4.1.2 e 4.1.3.

O nível de contenção de recursos é definido pelo par carga de trabalho e infraestrutura. Mantendo-se uma mesma carga, quanto maior a infraestrutura, menor é a contenção. Para os resultados discutidos no Capítulo 6, a alteração da capacidade da infraestrutura (como detalhado na Seção 4.6) foi a abordagem utilizada para analisar cenários com diferentes níveis de contenção. Neste estudo, a abordagem adotada foi diferente. A infraestrutura foi mantida entre os diferentes testes do experimento e o nível de contenção foi definido pela variação das cargas submetidas. Nesse sentido, três amostras da carga dos rastros de execução da Google[1] foram geradas da forma descrita adiante (Seção 7.1.4). Cada amostra possui uma demanda diferente, representado três níveis de contenção de recursos. Quanto maior a demanda da carga, maior a contenção de recursos. Esta abordagem foi utilizada com o propósito de gerar cenários mais desafiadores para o escalonador, onde a contenção fosse menos influenciada pelos picos de demandas.

É importante destacar que as otimizações implementadas nos escalonadores podem afetar o escalonamento sob duas perspectivas: a QoS provida pelo escalonador e o número de operações executadas. Considerando as otimizações discutidas nas seções anteriores, apenas a *relaxed randomization* pode afetar a QoS provida pelo escalonador, uma vez que ela verifica os servidores em ordem aleatória e permite que nem todos os servidores sejam verificados. Assim, a alocação real será provavelmente diferente para duas execuções de um mesmo cenário. Essa possível diferença na alocação pode fazer com que o escalonador entregue QoSs diferentes para uma mesma requisição em duas execuções de um mesmo cenário. Com o uso desta otimização, não necessariamente o "melhor" servidor da infraestrutura é escolhido para alocação de uma requisição em um dado momento, mas sim o "melhor" dentre os elegíveis. Além disso, o valor configurado para esta otimização também pode interferir na QoS das requisições. Por exemplo, quando configura-se o valor de 10% para esta otimização, significa que o escalonador para de verificar outros servidores quando encontrar um número total de servidores elegíveis que é 10% do total de servidores da infraestrutura. Por outro lado, quando configura-se 50%, ele verifica servidores até que a metade da infraestrutura seja elegível. Quanto maior o valor configurado, mais servidores precisam ser verificados, elevando a probabilidade do "melhor" servidor ser selecionado. Por isso, faz-se necessário examinar

[1]Os rastros da Google estão disponíveis para download em https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

o impacto causado na QoS provida pelos escalonadores com diferentes configurações para esta otimização.

As outras otimizações implementadas (*score caching* e *equivalence class* para o escalonador baseado em prioridade, e, *equivalence class* e *pending queue pruning* para o escalonador orientado por QoS) não afetam a QoS provida pelos escalonadores. Essas otimizações resultam apenas na redução da quantidade de operações realizadas por cada escalonador. Assim sendo, não há contra indicação na utilização dessas otimizações pelos escalonadores.

Portanto, uma vez que o cenário com melhor custo/benefício para cada escalonador for identificado (*i.e.* melhor configuração para *relaxed randomization*), as quantidades de operações necessárias por cada escalonador são contabilizadas e comparadas. As seções a seguir descrevem o projeto dos experimentos de simulação, como as amostras de infraestrutura e carga de trabalho foram selecionadas a partir dos rastros de execução da Google e os resultados desses experimentos.

7.1.4 Projeto Experimental

Os experimentos de simulação seguem um projeto fatorial completo com três fatores: a política de escalonamento utilizada, a carga de trabalho e o valor configurado para a otimização *relaxed randomization*. O primeiro tem dois níveis: o escalonamento *baseado em prioridade* e o escalonamento *orientado por QoS*. O segundo fator possui três níveis que serão descritos adiante nesta seção. O terceiro fator possui cinco níveis: 5%, 10%, 25%, 50% e 100%. Quanto maior o valor deste fator, maior a quantidade de servidores que precisam ser verificados. Na prática, o valor de 100% indica que esta otimização está "desligada". Este valor estabelece que o escalonador pare de verificar servidores quando encontrar um total de servidores elegíveis equivalente a 100% dos servidores da infraestrutura , o que obriga o escalonador a verificar todos os servidores da infraestrutura. Obviamente, mesmo que nem todos os servidores sejam elegíveis para a alocação da requisição, todos os servidores da infraestrutura são verificados.

A partir desses experimentos, analisa-se para cada escalonador qual a configuração da *relaxed randomization* resulta no melhor custo/benefício. Quanto menor o valor configurado, menor a quantidade de operações necessárias, portanto, sob este ponto de vista, busca-se diminuir ao máximo o valor configurado. Por outro lado, a avaliação de um número menor

de servidores pode trazer impactos à QoS do escalonador. Por esta razão, examina-se como esses valores impactam na QoS provida por cada escalonador. Neste estudo, a QoS provida é analisada através da satisfação do SLO e do custo das penalidades. Uma vez que a "melhor" configuração seja identificada para cada escalonador, as quantidades de operações executadas são contabilizadas e comparadas.

Cada um dos testes do experimento foi executado 25 vezes e as análises são feitas através dos intervalos de confiança de 95% das métricas de interesse. As subseções a seguir descrevem a definição das amostras (infraestrutura e carga de trabalho) analisadas.

Infraestrutura

A infraestrutura utilizada consistiu de um conjunto de servidores selecionados aleatoriamente (sem reposição) dos rastros da Google. Um servidor por vez foi sorteado até que a capacidade agregada da infraestrutura sorteada atingisse 5% da capacidade total dos rastros da Google. Esse processo resultou em 620 servidores heterogêneos.

Carga de Trabalho

A geração das amostras da carga ocorreu através da utilização de um controle de admissão simples. O controle implementado funciona de forma *offline*, ou seja, a carga foi gerada sem a necessidade de execução do simulador. As entradas para este controle são: as requisições do rastro da Google, os servidores da infraestrutura utilizada e um percentual indicando o limite dos recursos da infraestrutura a ser considerado para alocação. Como resultado, o controle entrega um subconjunto das requisições capaz de ser escalonado na infraestrutura informada. Neste modelo, o controle de admissão não considera fragmentação de recursos, *i.e.* examina a capacidade agregada da infraestrutura como se fosse um único servidor, ao invés de um conjunto de servidores menores.

O controle de admissão filtra as requisições dos rastros, admitindo ou rejeitando as mesmas. A ideia é que o controle admita as requisições até que as requisições ativas no sistema alcancem um limite (configurável) da capacidade da infraestrutura — o percentual informado como entrada do controle. Uma vez que uma requisição é admitida, ela é considerada ativa no sistema ao longo de seu tempo de duração a partir do momento de sua admissão. Isso indica que este controle não considera possíveis períodos da requisição em fila. Uma

requisição que passa algum período na fila de espera necessariamente estará no sistema por uma quantidade de tempo maior que sua duração. Isso ocorre porque ela levará mais tempo para alcançar a quantidade de tempo desejada com recursos alocados. Uma nova requisição é admitida se: (i) existir servidor na infraestrutura que atende suas restrições de alocação e afinidade (se definidas); e, (ii) as quantidades de recursos por ela solicitadas somadas às quantidades agregadas das requisições ativas não excederem o limite de alocação para nenhum tipo de recurso.

Com o intuito de gerar cargas com diferentes demandas, o controle de admissão foi utilizado com três limites de alocação distintos:

(i) 100% *da capacidade da infraestrutura (maior contenção):* o controle admite requisições enquanto a demanda das requisições ativas não exceder a capacidade da infraestrutura. Visto que o controle de admissão não considera o tempo das requisições em fila nem o desperdício com a fragmentação de recursos, considerar 100% da capacidade da infraestrutura já leva a uma carga com nível de contenção considerável. Este foi o maior limite analisado, portanto, gerou a carga com maior nível de contenção;

(ii) 95% *da capacidade da infraestrutura (média contenção):* o controle admite requisições enquanto a demanda das requisições ativas não exceder 95% da capacidade da infraestrutura. Este foi o segundo maior limite examinado, portanto, gerou a carga com o nível de contenção médio;

(iii) 90% *da capacidade da infraestrutura (menor contenção):* o controle admite requisições enquanto a demanda das requisições ativas não exceder 90% da capacidade da infraestrutura. Este foi o menor limite analisado, gerando a carga com menor nível de contenção.

Para permitir que as simulações executassem rapidamente (no máximo $1,5$ hora) com os recursos disponíveis e viabilizar a execução de uma maior quantidade de repetições para cada teste, esta análise se deu considerando a primeira 1 hora dos rastros de execução. Portanto, o conjunto de requisições submetidas ao longo da primeira hora do rastros foi submetido ao controle de admissão. Este controle filtrou as cargas considerando os limites acima mencionados. É importante destacar que o realismo da carga de trabalho é mais importante para

as questões relacionadas com avaliação da QoS provida pelo escalonador. O objetivo desta análise é entender como um escalonador se compara ao outro (em termos de número de operações) em diferentes níveis de contenção. Portanto, apenas a configuração da otimização *relaxed randomization* (que pode afetar a QoS do escalonador) poderia ser potencialmente afetada caso uma carga de trabalho menos realista fosse utilizada nesta análise.

Ainda é importante destacar algumas características dos rastros da Google [43]. Primeiramente, a Google realiza *overbooking*, portanto, as quantidades de recursos solicitadas pelas requisições nos rastros são maiores que as capacidades da infraestrutura descrita no mesmo. Além disso, o primeiro tempo disponível nos rastros (tempo 0) contém todas as requisições que estavam executando no sistema antes do início do mesmo. Algumas dessas estão associadas com tarefas de longa duração e permanecem no sistema até o final do rastros de execução. Caso o controle de admissão considere o limite total para alocação desde o tempo 0, as requisições admitidas neste instante já ocupam toda a infraestrutura durante o período completo de análise. Consequentemente, novas requisições não seriam admitidas ao longo do tempo. Como o objetivo deste estudo é analisar o desempenho dos escalonadores em um cenário onde requisições são admitidas e terminadas ao longo do tempo, o controle de admissão tem um tratamento especial para o início do *trace*. Particularmente no tempo 0, o controle admite requisições enquanto a quantidade agregada de recursos solicitados não exceder o limite de alocação reduzido em 20%. Em outras palavras, no tempo 0, o controle de admissão considera 80%, 75% e 70% da capacidade como limites para admissão nos cenários com maior, média e menor contenção, respectivamente. Isso garante que ao menos 20% da capacidade alocável da infraestrutura esteja destinada às requisições submetidas ao longo do tempo.

Por fim, as requisições das cargas geradas são categorizadas nas classes de serviço *A*, *B* e *C* seguindo o mesmo método descrito na Seção 4.2.

7.1.5 Resultados e Discussão

Impacto da configuração da otimização *relaxed randomization* na QoS

A Figura 7.1 apresenta os intervalos de confiança de 95% para o atendimento do SLO quando diferentes configurações para a otimização *relaxed randomization* são utilizadas no escalo-

nador baseado em prioridade. Esses resultados são agrupados por classe de serviço e por nível de contenção analisados (menor, médio, maior). Já a Figura 7.2 mostra os intervalos de confiança de 95% para o custo total com as penalidades para os mesmos cenários de simulação.

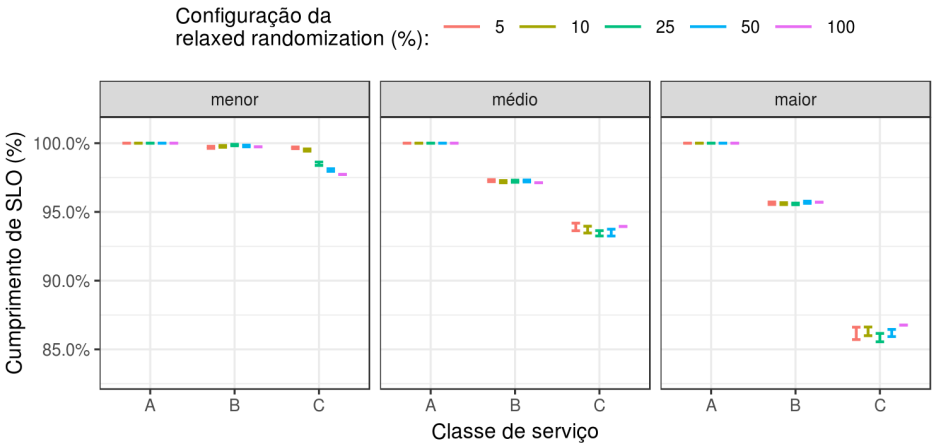


Figura 7.1: Intervalos de confiança do atendimento do SLO com diferentes configurações para a otimização *relaxed randomization* no escalonador baseado em prioridade.

Analisando as Figuras 7.1 e 7.2, verifica-se que a utilização de valores menores para a otimização *relaxed randomization* não necessariamente resulta na degradação das métricas de QoS. Destaca-se que no cenário com menor contenção de recursos, menores valores para *relaxed randomization* resultaram em um melhor atendimento do SLO para as requisições da classe *C* (Figura 7.1). No entanto, isso pode ter ocorrido por acaso devido à aleatoriedade na escolha dos servidores a serem verificados. Na medida que o nível de contenção é aumentado, torna-se menos provável que exista diferença entre os resultados obtidos com as diversas configurações. Em um ambiente com maior nível de contenção, provavelmente mais servidores precisam ser verificados para que se encontre o $x\%$ (onde x é o valor configurado para a otimização) de servidores elegíveis para alocar uma instância pendente. Além disso, também destaca-se que no cenário com maior nível de contenção, valores intermediários para *relaxed randomization* resultaram em uma penalidade um pouco maior em comparação ao cenário onde a otimização está desligada — 100% — (Figura 7.2). Isso ocorre porque a quantidade de recursos solicitados e o déficit de QoS impactam no cálculo da penalidade (definida pelo Equação 4.8). Como já discutido, a aleatoriedade na verificação dos servidores

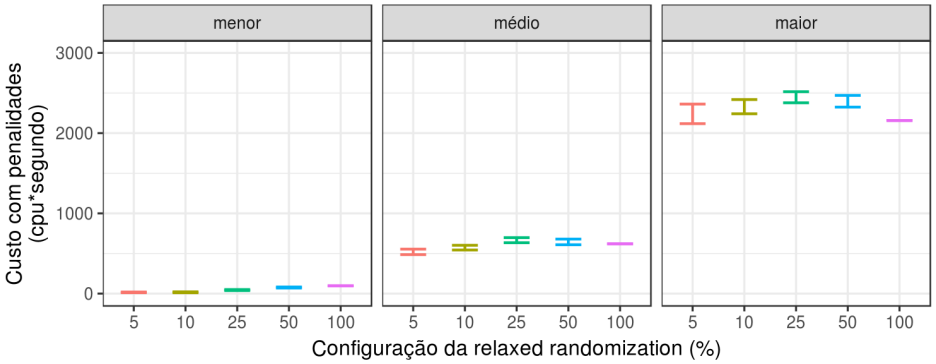


Figura 7.2: Intervalos de confiança do custo das penalidades com diferentes configurações para a otimização *relaxed randomization* no escalonador baseado em prioridade.

pode levar a diferentes alocações reais, consequentemente, tanto as requisições que tiveram seus SLOs violados, como os déficits experimentados por elas podem ser diferentes. Isso leva a variações nas penalidades tanto entre execuções de um mesmo cenário como também entre diferentes cenários.

No entanto, no geral, diferentes configurações para *relaxed randomization* não impactaram significativamente na QoS provida pelo escalonador baseado em prioridade. Até quando ocorreu diferença estatisticamente significante, na prática, esta diferença não foi substancial. Considerando que esta otimização é utilizada pelo sistema *Borg*, essas pequenas variações são admitidas na prática. Portanto, considerando os valores e as cargas analisadas, o escalonador baseado em prioridade pode ter 5% como o valor configurado para esta otimização.

As Figuras 7.3 e 7.4 apresentam, respectivamente, os intervalos de confiança de 95% para o atendimento do SLO e o custo com as penalidades quando diferentes configurações para a otimização *relaxed randomization* são utilizadas no escalonador orientado por QoS. O *layout* dessas figuras é o mesmo das Figuras 7.1 e 7.2.

Com base nos resultados exibidos nas Figuras 7.3 e 7.4, é possível verificar que, no geral, também não houve diferenças significativas na QoS provida pelo escalonador orientado por QoS quando a configuração da *relaxed randomization* foi variada. Em cenários onde a diferença foi estatisticamente significante, na prática, a diferença foi muito pequena. Como esta otimização é utilizada em um sistema real viável (*Borg*), essas pequenas variações são aceitas. Portanto, o valor de 5% também pode ser utilizado para a otimização *relaxed randomization* no escalonador orientado por QoS, sem impacto substancial na QoS provida pelo

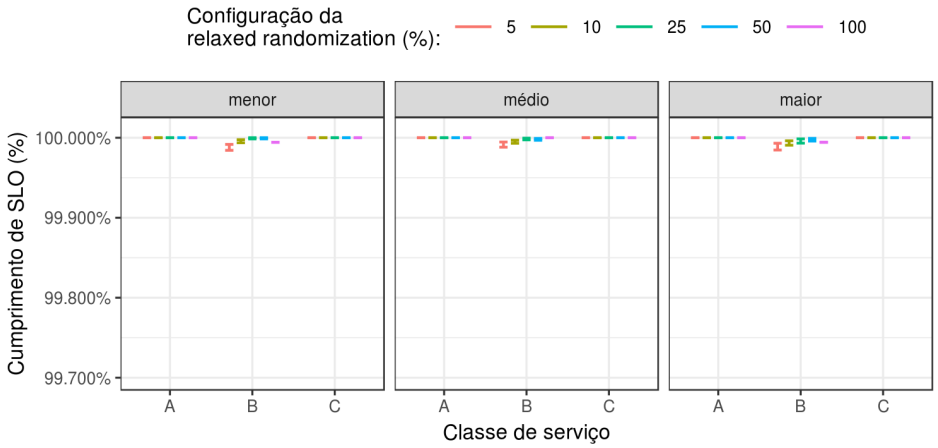


Figura 7.3: Intervalos de confiança do atendimento do SLO com diferentes configurações para a otimização *relaxed randomization* no escalonador orientado por QoS.

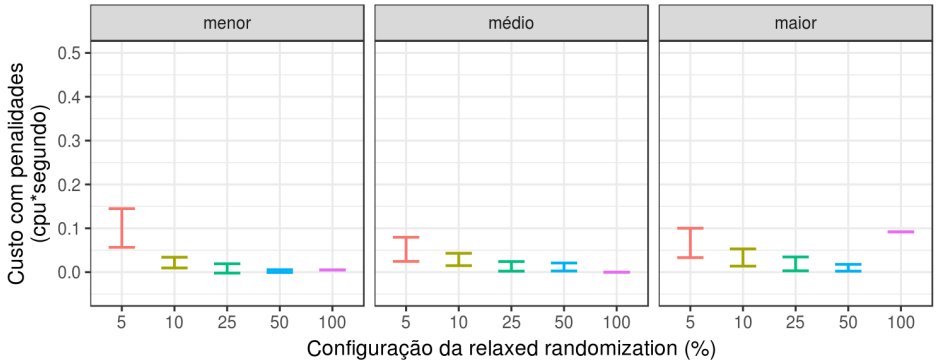


Figura 7.4: Intervalos de confiança do custo das penalidades com diferentes configurações para a otimização *relaxed randomization* no escalonador orientado por QoS.

escalonador.

Apesar do objetivo deste estudo ser a análise do impacto do valor utilizado para a otimização *relaxed randomization* para cada escalonador, é interessante enfatizar que, como esperado, o uso do escalonador orientado por QoS resultou em um maior satisfação do SLO e menor custo com penalidades. É possível visualizar esses resultados quando compara-se os resultados obtidos por cada cenário de simulação para o atendimento do SLO (Figuras 7.1 e 7.3) e para o custo com as penalidades (Figuras 7.2 e 7.4) quando ambos os escalonadores foram utilizados. Ao analisar as diferentes figuras, deve-se levar em conta que as escalas das mesmas são diferentes. Isso ocorre com o intuito de facilitar a visualização das diferenças entre as configurações avaliadas para cada escalonador.

Análise do Número de Operações

Uma vez que as melhores configurações para *relaxed randomization* foram identificadas na análise anterior, compara-se o desempenho dos escalonadores em termos do número de operações executadas. Relembrando que uma operação é contabilizada pela execução da verificação de viabilidade e o cálculo da pontuação de um servidor (descritas nas Seções 3.4.1 e 3.4.2) para uma instância pendente. A Figura 7.5 apresenta os intervalos de confiança de 95% para o número de operações executadas por cada escalonador, considerando as 25 execuções de cada cenário. Os resultados são exibidos agrupados por nível de contenção. Com o intuito de evidenciar o ganho obtido com as otimizações no escalonador orientado por QoS, além dos resultados para os escalonadores baseado em prioridade e orientado por QoS otimizados (configurados com 5% para *relaxed randomization*), a Figura 7.5 também apresenta os resultados para o escalonador orientado por QoS sem utilizar nenhuma das otimizações.

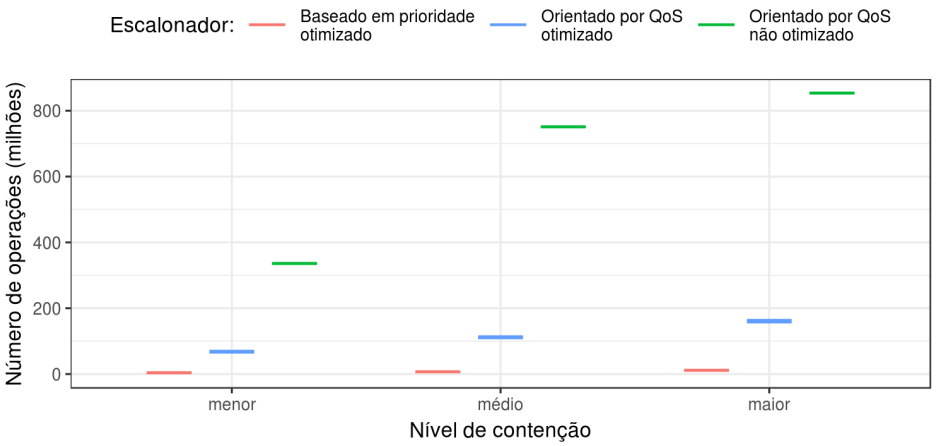


Figura 7.5: Intervalos de confiança da quantidade de operações executadas quando diferentes escalonadores são utilizados.

Com o intuito de facilitar a visualização das diferenças, a Tabela 7.1 apresenta a quantidade média de operações executadas (em milhões) para cada um dos níveis de contenção e escalonadores analisados.

Examinando a Figura 7.5 e a Tabela 7.1, verifica-se que as otimizações reduziram substancialmente a quantidade de operações executadas pelo escalonador orientado por QoS. Isso pode ser visualizado quando compara-se os resultados obtidos por sua versão otimizada

Tabela 7.1: Número médio de operações executadas (em milhões) por cada escalonador e nível de contenção analisados

Escalonador	menor	médio	maior
Baseado em prioridade otimizado	$3,87$	$7,16$	$11,4$
Orientado por QoS otimizado	$67,9$	112	161
Orientado por Qos não otimizado	336	751	854

e não otimizada, independente do nível de contenção analisado. Em média, o número de operações é reduzido em aproximadamente 82% quando o escalonador orientado por QoS é executado com as otimizações. No entanto, quando sua execução otimizada é comparada com o escalonador baseado em prioridade otimizado, o escalonador orientado por QoS ainda executa, em média, aproximadamente $15,5$ vezes mais operações que o escalonador baseado em prioridade.

Como já discutido, um custo operacional maior já era esperado para o escalonador orientado por QoS. Todavia, algumas avaliações adicionais foram realizadas para entender a causa desse número de operações tão maior em comparação ao escalonador baseado em prioridade. Nesse sentido, visto que o escalonador orientado por QoS pode potencialmente processar a fila mais vezes que o escalonador baseado em prioridade, analisou-se tanto a quantidade de vezes que cada escalonador processou a fila de espera como também o tamanho médio da fila processada. Além disso, o escalonador baseado em prioridade possui a otimização *score caching* implementada enquanto o escalonador orientado por QoS não. Por isso, uma outra análise foi realizada com o intuito de identificar o impacto desta otimização no número de operações executadas pelo escalonador baseado em prioridade.

Considerando que ambos os escalonadores processam a fila quando novas requisições são admitidas e quando as instâncias em execução são terminadas, investigou-se a quantidade de vezes em que esses eventos ocorrem nas simulações executadas. Todas as cargas utilizadas possuem admissões em aproximadamente 360 instantes de tempo diferentes. Além disso, em todos os cenários ocorreram eventos de admissão e/ou término de requisições em aproximadamente 2 mil instantes diferentes. Portanto, espera-se que a fila deve ser processada ao menos 2 mil vezes por ambos os escalonadores nas simulações executadas. Obviamente, a fila de espera é processada apenas quando não estiver vazia.

A Figura 7.6 apresenta os intervalos de confiança de 95% para o número de vezes que a fila foi processada e para o tamanho médio da fila. Com o objetivo de focar a análise nas melhores configurações dos escalonadores, a figura apresenta os resultados obtidos pela versão otimizada de cada escalonador[2].

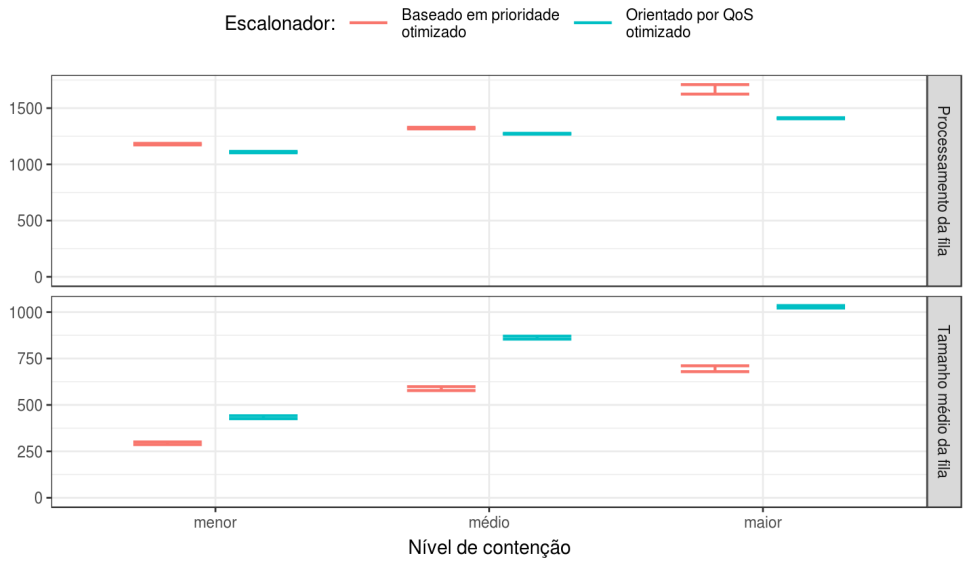


Figura 7.6: Intervalos de confiança do número de vezes que a fila é processada e do tamanho médio da fila quando os diferentes escalonadores são utilizados.

Analisando a Figura 7.6, observa-se que em nenhum dos cenários de simulação a fila foi processada 2 mil vezes. Isso significa que a fila esteve vazia durante alguns instantes em que deveria ter sido processada. Além disso, a fila foi processada um pouco mais quando o escalonador baseado em prioridade foi usado. Apesar da diferença na quantidade de processamentos da fila não ser substancial, houve uma diferença estatisticamente significante. Considerando os diferentes níveis de contenção, em média, a fila foi processada 1390 vezes pelo escalonador baseado em prioridade, enquanto que o escalonador orientado por QoS processou em média 1264 vezes. Esse resultado indica que o escalonador orientado por QoS conseguiu esvaziar a fila mais rapidamente em alguns cenários. Por exemplo, em momentos com baixa contenção (quando a fila está pequena), o escalonador orientado por QoS, através da alternância das instâncias em execução, conseguiu esvaziar a fila mais rapidamente. Essa alternância permitiu que as instâncias de menor duração terminassem mais rapidamente.

[2]Apesar de não apresentados na Figura 7.6, não há diferença substancial entre os resultados obtidos pelas versões otimizada e não otimizada do escalonador orientado por QoS para as métricas analisadas.

Por outro lado, ainda considerando a Figura 7.6, constata-se que o uso do escalonador orientado por QoS levou a filas de espera, em média, maiores em comparação ao escalonador baseado em prioridade. Isso ocorre porque o escalonador orientado por QoS, em períodos com maior contenção, faz com que as requisições permaneçam mais tempo no sistema. Por este escalonador alternar as instâncias em execução, elas levem mais tempo para alcançarem a duração desejada. No entanto, a fila que em média é $1,4$ vezes maior não causa uma quantidade tão maior de operações quando o escalonador orientado por QoS é executado.

Por isso, a influência da otimização *score caching* na quantidade de operações do escalonador baseado em prioridade também foi examinada. A Figura 7.7 apresenta o número de operações executadas por cada escalonador de acordo com o tamanho da fila processada. Além das versões otimizadas dos escalonadores baseado em prioridade (gráfico à esquerda) e orientado por QoS (gráfico à direita), esta figura também apresenta os resultados obtidos quando o escalonador baseado em prioridade é executado sem a otimização de *score caching* (gráfico central).

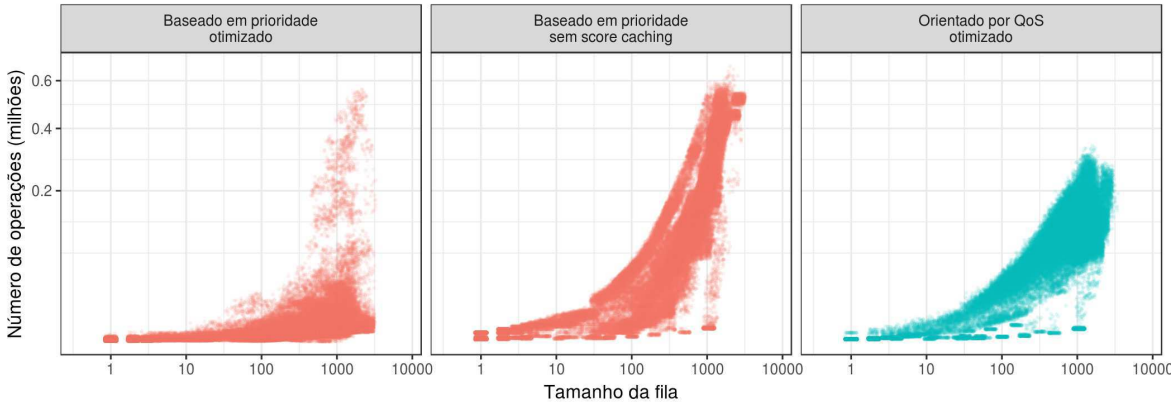


Figura 7.7: Número de operações executadas quando os diferentes escalonadores são utilizados.

Com base na Figura 7.7, verifica-se que, no geral, como esperado, quanto maior a fila de espera, mais operações são necessárias ao escalonador. No entanto, quando compara-se os resultados do escalonador baseado em prioridade otimizado (esquerda) com os de sua versão sem o *score caching* (centro), nota-se que a otimização geralmente mantém a quantidade de operações concentrada em valores mais baixos. A diferença é ainda mais evidente observando os resultados para as filas maiores. No caso do escalonador orientado por QoS

(direita), por não suportar *score caching*, ele mantém a tendência de realizar muitas operações quando as filas são maiores.

O número médio de operações executadas pelo escalonador baseado em prioridade sem a otimização *score caching* foi, respectivamente, 56, 162 e 293 milhões para os cenários com menor, média e maior contenção de recursos. Esta métrica para o escalonador baseado em prioridade otimizado está apresentada na Tabela 7.1. Considerando os cenários com menor, média e maior contenção, respectivamente, observa-se que a ausência da otimização levou a um aumento de aproximadamente 14, 23 e 26 vezes no número médio de operações executadas. Isso indica que a otimização *score caching* reduz consideravelmente o número de operações necessárias ao escalonador baseado em prioridade.

Na verdade, esse resultado não chega a ser surpreendente. Considere um cenário com o escalonador baseado em prioridade com *score caching*. Quando uma requisição estiver na fila (em um instante diferente de sua admissão), significa que todos os servidores da infraestrutura já foram verificados para ela. Isso ocorre independentemente da configuração para *relaxed randomization*, visto que se a requisição estiver na fila, nenhum servidor foi elegível para sua alocação. A partir desse momento, sempre que a fila for processada, apenas servidores que sofreram alguma alteração desde a última verificação precisam ser verificados novamente. O servidor sofre alteração, por exemplo, quando uma requisição é alocada ou terminada no mesmo. Neste caso, quando a fila for processada, o servidor é verificado novamente para a requisição e a *cache* é atualizada. Já em um cenário com o escalonador orientado por QoS, sem *score caching*, sempre que a fila é processada os servidores precisam ser verificados novamente. No pior caso (quando não há servidores elegíveis), todos os servidores da infraestrutura são verificados. Por exemplo, suponha uma infraestrutura com 620 servidores (quantidade utilizada neste estudo) e uma requisição na fila. Quando esta fila é processada por causa do término de uma requisição, apenas o servidor onde a requisição estava alocada é verificado pelo escalonador baseado em prioridade, enquanto todos os 620 servidores podem ser verificados novamente pelo escalonador orientado por QoS.

Portanto, a otimização *score caching* reduz consideravelmente o número de operações necessárias ao escalonador baseado em prioridade. A ausência desta otimização é o principal motivo para o número de operações executadas pelo escalonador orientado por QoS ser tão maior.

7.1.6 Alternativas para melhorar o desempenho

Como discutido na seção anterior, a quantidade de operações executadas por ambos os escalonadores é substancialmente diferente. No entanto, é possível que o tempo de processamento das operações não sejam significativamente diferente. Por esta razão, fazer uma análise do tempo de execução poderia indicar se realmente seria necessário buscar por mais otimizações para o escalonador orientado por QoS ou não. Em caso de necessidade de mais otimizações, a implementação de um mecanismo de *cache* no escalonador proposto seria uma alternativa a ser considerada. Isso porque a ausência desta otimização impacta consideravelmente no número de operações realizadas pelo escalonador.

Como discutido na Seção 7.1.2, a otimização *score caching* da forma como implementada para o escalonador baseado em prioridade não faz sentido para o escalonador orientado por QoS porque a métrica utilizada por este último sofre alteração com o passar do tempo. Nesse cenário, a *cache* estaria sempre inválida quando a fila fosse processada em diferentes momentos.

Entretanto, outras técnicas de *caching* podem ser implementadas para o escalonador proposto. Uma alternativa é considerar um período de validade para a informação na *cache*. Essa opção faz com que o escalonador tome decisões com base em métricas de QoS ($Q_j(t)$) desatualizadas em alguns momentos, podendo impactar na QoS provida pelo mesmo. Nesse contexto, é necessário a realização de uma análise sobre o impacto do período da validade da *cache* na QoS entregue pelo escalonador.

Uma outra opção para a implementação da *cache* está relacionado com a informação a ser armazenada na mesma. Por exemplo, quando um servidor fosse verificado para uma requisição pendente e não fosse elegível, o escalonador poderia armazenar quando este servidor se tornará elegível para a requisição em questão. Esse instante é calculado com base nas métricas de QoS das requisições alocadas no servidor e da requisição pendente. Assim sendo, este servidor seria verificado para a mesma requisição apenas: (i) se houvesse alocação ou término de requisições no mesmo; ou (ii) após o tempo definido na *cache*, visto que antes desse instante saberia-se que o servidor não se tornaria elegível. Esse mecanismo para a *cache* não permitiria que o escalonador considerasse a métrica desatualizada para tomada de decisão, logo, não impactaria na QoS provida pelo escalonador.

Uma solução híbrida também poderia ser utilizada para o escalonamento. A ideia central

é fazer o escalonador orientado por QoS operar apenas com o nível de contenção para o qual foi planejado. Assim sendo, o escalonador poderia ser adaptado para alterar seu modo de operação entre a política baseada em prioridade e orientada por QoS. Essa troca no modo de operação seria feita com base no nível de contenção momentâneo do sistema. Por exemplo, quando o tamanho da fila exceder um dado limite configurável, o escalonador deve operar segundo a política baseada em prioridade; caso contrário, quando o tamanho da fila for menor que este limite, o escalonador opera de acordo com a política orientada por QoS.

Por último, a otimização *equivalence class* também pode ser relaxada visando melhorar o desempenho do escalonador. Da forma como está implementada, duas requisições pertencem a uma mesma classe de equivalência se possuem requisitos, demandas e métricas de QoS iguais. Como esta última é um valor real, a probabilidade de existir duas requisições com métricas exatamente iguais ao longo do tempo é bem reduzida. Portanto, esta otimização poderia ser relaxada para considerar métricas de QoS equivalentes se a diferença entre elas for menor que um valor configurável. Essa modificação também pode impactar na QoS provida pelo escalonador, e, por isso, requer uma análise.

7.2 Viabilidade de implementação

Esta seção tem como objetivo analisar a viabilidade de implementação da política de escalonamento orientada por QoS em um sistema real. Nesse sentido, decidiu-se estender um sistema existente de ampla utilização na atualidade (Kubernetes) para suportar o escalonamento orientado por QoS no contexto que tipicamente seus usuários interagem com o sistema.

7.2.1 Submissão de cargas de trabalho no Kubernetes

No contexto do Kubernetes, uma instância pode ser representada por um *pod*. Um *pod* é a menor abstração gerenciável no sistema e é capaz de encapsular um ou mais contêineres de uma aplicação, recursos de armazenamento, um IP único na rede e opções que direcionam como o(s) contêiner(es) da aplicação deve(m) executar. Em outras palavras, um pod representa uma instância de uma aplicação no sistema Kubernetes.

No caso de uso mais comum, um pod executa um único contêiner, mas também é possível

que o pod execute múltiplos contêineres que precisem trabalhar em conjunto na aplicação a ser executada. Quando o usuário cria um pod diretamente no sistema, ele é adicionado em uma fila de espera que é processada pelo escalonador. Para cada pod na lista de espera, o escalonador seleciona o servidor mais adequado para prover recursos de acordo com sua política de escalonamento. Caso o escalonador não consiga selecionar um servidor para um dado pod, este continuará na fila até que o escalonador consiga alocar recursos para o mesmo no futuro. Após um pod ser alocado em um servidor, ele permanece executando no mesmo até que sua tarefa computacional seja concluída, o servidor falhe ou o pod seja deletado ou preemptado. Após um desses eventos, o pod é finalizado, e, caso não seja gerenciado por outras abstrações, não será reescalonado.

Existem duas formas básicas de criar instâncias para aplicações no sistema Kubernetes. Na primeira, os usuários podem criar objetos pods diretamente. Neste caso, os pods são chamados de *bare pods* e funcionam como descrito no parágrafo anterior. Uma instância de uma aplicação estará no sistema (pendente ou em execução) enquanto o pod desta instância não for finalizado.

No entanto, espera-se que um usuário raramente crie pods individuais diretamente no Kubernetes. Este sistema oferece um conjunto de abstrações de mais alto nível, denominadas de *controladores*. Um controlador tem como objetivo gerenciar um conjunto de pods, manipular replicações e atualizações de um pod e fornecer recursos de tolerância a falhas no escopo do sistema. Por exemplo, se um servidor falhar ou um pod for preemptado, o controlador pode substituir automaticamente o pod finalizado por meio da criação de um outro pod idêntico, a ser escalonado em um servidor diferente. Neste cenário, apesar de serem dois objetos diferentes, ambos os pods estão relacionados com uma mesma instância da aplicação. O segundo pod é idêntico ao primeiro e foi criado apenas porque o primeiro foi finalizado, *i.e.* foi criado para que a instância da aplicação possa continuar sua execução.

No geral, os controladores usam um modelo especificado pelo usuário para criar os pods pelos quais eles são responsáveis. Os principais controladores oferecidos pelo Kubernetes são:

- ***Deployment*:** o propósito deste controlador é assegurar que um número específico de réplicas de um pod esteja em execução ao longo do tempo. Em outras palavras, este controlador visa assegurar que um número específico de instâncias da aplicação

esteja em execução ao longo do tempo. Um *deployment* gerencia pods que executam por tempo indeterminado, sem a expectativa de serem finalizados — por exemplo, serviços interativos na Web. Quando um pod é finalizado (por exemplo, por causa de preempção ou falha do servidor) e um outro pod é criado com o intuito de manter o número de réplicas (instâncias) da aplicação no sistema, o estado da aplicação não é mantido entre os diferentes pods. Portanto, este controlador é destinado a aplicações *stateless*.

- ***StatefulSet*:** este controlador tem objetivo semelhante ao *Deployment*. Ele também visa assegurar que um número específico de réplicas de um pod (*i.e.* instância) esteja em execução ao longo do tempo. No entanto, este controlador mantém o estado entre dois pods criados em sequência para a mesma réplica, mantendo um identificador e armazenamento persistente entre esses pods. Além disso, o *StatefulSet* também garante a ordem de implantação das réplicas por ele gerenciadas. Por exemplo, considerando que um controlador *StatefulSet* é responsável por manter três réplicas em execução, a réplica 1 sempre será a primeira a ser alocada, e, em seguida, as réplicas 2 e 3 nesta ordem.

- ***DaemonSet*:** este controlador tem como meta assegurar que uma réplica de um pod execute em todos ou em um conjunto específico dos servidores da infraestrutura. Quando um *DaemonSet* é criado, é possível que o usuário especifique restrições de alocação para as réplicas por ele gerenciadas, tais como $vCPU > 2$ e $RAM < 2GB$. Neste caso, uma réplica será alocada em cada servidor que satisfaça suas restrições. Caso nenhuma restrição seja definida pelo usuário, uma réplica deve ser alocada em cada servidor da infraestrutura. Assim sendo, quando um novo servidor é adicionado à infraestrutura, uma nova réplica é criada para cada *DaemonSet* no sistema que tenha suas restrições de alocação atendidas pelo novo servidor. Quando servidores são removidos, as réplicas de um *DaemonSet* que estavam executando nos mesmos não são criadas em outros servidores.

- ***Job*:** este controlador cria uma ou mais cópias de um pod e assegura que um número específico dessas cópias sejam finalizadas com sucesso. Diferentemente dos controladores apresentados até então, um *Job* gerencia pods que têm a expectativa de serem

finalizados após a execução de uma tarefa computacional — por exemplo, aplicações do tipo "saco-de-tarefas" (BoT, do inglês *Bag-of-Tasks*). Quando um número específico de cópias finalizadas é alcançado, o *Job* está completo. Além disso, um *Job* pode executar mais de uma cópia do pod em paralelo, este limite é configurado no momento de criação do *Job*. Nesse sentido, quando uma das cópias (instâncias) é finalizada com sucesso, uma nova cópia é criada caso ainda seja necessário.

- ***CronJob*:** este controlador é responsável por criar um *Job* específico periodicamente. Um objeto *CronJob* é como uma linha de um arquivo *crontab*[3], que tem o objetivo de executar um *Job* periodicamente em um dado escalonador. Como exemplo de caso de uso deste controlador, suponha um *Job* responsável pela realização de um *backup* que precisa ser realizado todos os dias em um horário específico. Ao invés de criar um *Job* para essa atividade todos os dias individualmente, um *CronJob* pode ser criado para escalonar esse *Job* todos os dias no mesmo horário específico.

Nesse contexto, a submissão de um controlador à um *cluster* Kubernetes pode ser interpretada como a submissão de uma requisição que pode estar associada com mais de uma instância. Por exemplo, suponha que um usuário submete um *Deployment* que deve criar e manter 3 réplicas de uma aplicação específica, então considera-se este *Deployment* como uma requisição que está associada a 3 instâncias no *cluster*.

7.2.2 Caracterização dos controladores do Kubernetes

Como discutido na seção anterior, os controladores são abstrações oferecidas pelo Kubernetes com o objetivo de facilitar o gerenciamento de aplicações por parte de seus usuários. Cada um dos controladores disponíveis no sistema foi projetado e implementado com base em um tipo diferente de aplicação. De acordo com suas descrições e propósitos, é possível categorizar os controladores suportados pelo Kubernetes em duas classes: *de Serviço* e *Batch*. Segue uma breve descrição para cada uma dessas classes.

[3] O crontab é um programa que edita o arquivo onde são especificados os comandos a serem executados e a hora e dia de execução pelo cron, que é um serviço que executa comandos agendados nos sistemas operacionais do tipo Unix.

Controlador de Serviço

Um controlador de serviço tem como meta assegurar que um número específico de réplicas de um pod esteja em execução ao longo do tempo. O pod a ser replicado é definido pelo usuário no momento da criação do controlador por meio de um *template*. Além disso, as réplicas gerenciadas por um controlador de serviço não têm a expectativa de serem finalizadas, *i.e.* elas executam réplicas de um serviço. A partir deste ponto, uma réplica gerenciada por um controlador desta classe será referenciada como uma instância. Portanto, se um controlador é responsável por manter em execução, por exemplo, duas réplicas de um *template* definido pelo usuário, significa que duas instâncias do *template* devem estar em execução ao longo do tempo.

Esta classe de controlador gerencia um conjunto finito de réplicas (instâncias) $C_s = \{p_1, ..., p_r\}$, onde r é o número máximo de instâncias do *template*. Cada instância $p_i, p_i \in C_s$, tem uma ou mais encarnações. Sempre que um pod é criado pelo controlador C_s, este pod é considerado como uma nova encarnação de uma das instâncias deste controlador. Portanto, quando uma instância p_i falha (por exemplo, por causa de preempção ou falha do servidor), uma nova encarnação de p_i (*i.e.* um novo pod) é criada e adicionada à fila de espera para escalonamento. Considere $p_i = \{p_i^1, ..., p_i^m\}$ como o conjunto de encarnações da instância p_i e m como o número de encarnações de p_i, então p_i^e representa a *e-ésima* encarnação da instância p_i.

Em qualquer instante no tempo t, uma encarnação p_i^e de uma instância p_i de um controlador de serviço C_s está associada a um único estado. Considere que $s(p_i^e, t)$ indica o estado de p_i^e no tempo t, então $s(p_i^e, t) \in \{Esperando, Executando, Falha\}$. Uma encarnação de instância qualquer inicia no estado *Esperando* e seguirá para o estado *Executando* quando o escalonador conseguir alocar recursos para a mesma. Do estado *Executando*, a encarnação pode seguir para o estado *Falha*, no caso dela ser finalizada por causa de falha do servidor ou devido a atuação da política de escalonamento (que pode decidir preemptá-la). Destaca-se que uma encarnação de instância falha não mais será considerada para escalonamento. Este é o estado final para a encarnação em questão.

Considere que $W_{C_s}(t), R_{C_s}(t)$, e $F_{C_s}(t)$ são os conjuntos das encarnações das instâncias do controlador de serviço C_s que, no tempo t, estão, respectivamente, nos estados *Esperando*, *Executando* e *Falha*. O conjunto $I_{C_s}(t) = W_{C_s}(t) \cup R_{C_s}(t) \cup F_{C_s}(t)$ representa o conjunto de

todas as encarnações das instâncias de C_s no tempo t. Note que uma mesma encarnação não poderá estar em dois desses conjuntos no instante t, no entanto, uma mesma instância pode ter encarnações em mais de um conjunto no instante t. Por exemplo, quando uma terceira encarnação de uma instância p_i é criada no tempo t, significa que duas outras encarnações desta instância já falharam até t, portanto, essas encarnações que já falharam estarão no conjunto $F_{p_i}(t)$, enquanto a nova encarnação estará no conjunto $W_{p_i}(t)$.

No contexto dos controladores do Kubernetes (discutidos na Seção 7.2.1), é possível classificar o *Deployment*, o *StatefulSet* e o *DaemonSet* como controladores de serviço. Todos esses controladores são semelhantes no sentido que eles atuam para manter instâncias de um *template* em execução ao longo do tempo. O objetivo tanto do *Deployment* como do *StatefulSet* é assegurar que um número específico de instâncias estejam em execução ao longo do tempo. O primeiro é projetado para aplicações *stateless* enquanto que o segundo para aplicações *stateful*. O *DaemonSet* também visa manter um conjunto de instâncias em execução, no entanto, há restrições envolvidas na alocação dessas instâncias (cada instância deve executar em um servidor diferente).

Controlador *Batch*

Diferentemente de um controlador de serviço, um controlador *Batch* é responsável por gerenciar instâncias que têm a expectativa de serem finalizadas com sucesso. Uma instância é finalizada com sucesso quando sua tarefa computacional é concluída sem interrupções ao longo de sua execução. Nesse sentido, o usuário especifica um *template* para a instância, definindo também a tarefa de computação a ser executada na mesma. Um controlador desta classe é responsável por criar uma ou mais cópias do *template* e assegurar que um número específico dessas cópias sejam finalizadas com sucesso. A partir deste ponto, uma cópia gerenciada por um controlador *Batch* será referenciada como uma instância. Portanto, se um controlador é responsável por concluir com sucesso, por exemplo, três cópias de um *template* definido pelo usuário, significa que três instâncias desse *template* devem ser finalizadas com sucesso.

Um controlador *Batch* gerencia um conjunto finito de cópias (instâncias) $C_b = \{p_1, ..., p_n\}$, onde n é o número total de instâncias do *template* que precisam ser finalizadas com sucesso. É possível que o controlador crie mais de uma instância para executar

paralelamente. Considere C_b^r como o número máximo de instâncias de um controlador C_b que pode executar em paralelo. Caso falte menos instâncias que C_b^r para que o controlador tenha as n instâncias concluídas com sucesso, o controlador criará apenas a quantidade necessária de instâncias para que o número n de instâncias concluídas seja alcançado.

De forma semelhante ao controlador de serviço, cada instância $p_i, p_i \in C_b$, tem uma ou mais encarnações. Sempre que um pod é criado pelo controlador C_b, este pod é considerado como uma nova encarnação de uma das instâncias deste controlador. Portanto, quanto a instância p_i falha, uma nova encarnação da instância p_i (*i.e.* um novo pod) é criada e adicionada à fila de espera para escalonamento. Considere $p_i = \{p_i^1, ..., p_i^m\}$ como o conjunto de encarnações da instância p_i e m como o número de encarnações de p_i, então p_i^e indica a *e-ésima* encarnação da instância p_i.

Em qualquer instante no tempo t, uma encarnação de uma instância p_i^e de um controlador *Batch* C_b está associada a um estado. Considere que $s(p_i^e, t)$ representa o estado de p_i^e no tempo t, $s(p_i^e, t) \in \{Esperando, Executando, Falha, Completa\}$. Além dos estados possíveis para uma encarnação de instância de serviço (*Esperando*, *Executando* e *Falha*), uma encarnação de uma instância *batch* também pode estar no estado *Completa*. Este estado representa quando uma encarnação conseguiu concluir sua computação com sucesso. Portanto, uma encarnação inicia no estado *Esperando* e seguirá para o estado *Executando* quando o escalonador conseguir alocar recursos para a mesma. Do estado *Executando*, a encarnação pode seguir para um dos dois estados: *Completa* (se a execução for concluída com sucesso) ou *Falha* (se a encarnação for finalizada sem sucesso, por exemplo, por causa da atuação do escalonador ou falha do servidor). Novamente, é importante destacar que uma vez que uma encarnação de um instância falhar, esta encarnação não mais será considerada para escalonamento. Como discutido anteriormente, uma nova encarnação para a mesma instância é criada, e, esta sim será considerada para ser preemptada. Uma encarnação que é completada também não mais será completada para escalonamento. Esses (Falha e Completa) são os estados finais para as encarnações em questão.

Considere que $W_{C_b}(t), R_{C_b}(t), F_{C_b}(t)$ e $C_{C_b}(t)$ são os conjuntos de encarnações das instâncias de um controlador *Batch* C_b que, no tempo t, estão, respectivamente, nos estados *Esperando*, *Executando*, *Falha* e *Completa*. O conjunto $I_{C_b}(t) = W_{C_b}(t) \cup R_{C_b}(t) \cup F_{C_b}(t) \cup C_{C_b}(t)$ indica o conjunto de todas as encarnações de todas as instâncias de C_b no tempo t. De

forma equivalente, $W_{p_i}(t), R_{p_i}(t)$, $F_{p_i}(t)$ e $C_{p_i}(t)$ são os conjuntos das encarnações de uma instância p_i que, no tempo t, estão, respectivamente, nos estados *Esperando*, *Executando*, *Falha* e *Completa*. Assim, $I_{p_i}(t) = W_{p_i}(t) \cup R_{p_i}(t) \cup F_{p_i}(t) \cup C_{p_i}(t)$ é o conjunto de todas as encarnações da instância p_i no tempo t.

Analisando os controladores oferecidos pelo Kubernetes (discutidos na Seção 7.2.1), é possível classificar o *Job* e *CronJob* como controladores *Batch*. Esses controladores têm características semelhantes, ambos gerenciam instâncias que têm a expectativa de serem finalizadas com sucesso. O *Job* é usado pelo usuário quando a tarefa computacional a ser executada deve ser concluída e não se repetirá de forma previamente conhecida, enquanto que o *CronJob* cria *Jobs* a serem executados periodicamente no sistema.

7.2.3 Abordagens para medir QoS de um controlador

Uma vez que o uso de controladores é a forma mais comum de submeter cargas de trabalho ao sistema Kubernetes [4], faz sentido pensar que a QoS fornecida pelo provedor nesse contexto poderia estar associada aos controladores. Como mencionado antes, um controlador Kubernetes pode ser interpretado como uma requisição associada com uma ou mais instâncias. Nesse sentido, o provedor deve medir a QoS entregue para a requisição (controlador) como um todo, ao invés de medir a QoS fornecida para cada instância no sistema individualmente. No entanto, cada controlador suportado pelo Kubernetes foi projetado para um tipo específico de aplicação, que pode ter necessidades específicas para sua execução. Por esta razão, a forma como a QoS de um controlador deve ser medida está diretamente relacionada com o tipo de aplicação para a qual ele foi projetado, como também com as necessidades da aplicação do usuário. Este trabalho propõe três abordagens possíveis para medir a QoS de um controlador, *i.e.* três diferente semânticas para a QoS de um controlador:

- **Independente:** nesta abordagem a QoS é calculada individualmente para cada instância p_i gerenciada por um controlador C. Neste trabalho, visto que cada requisição nas cargas analisadas nos experimentos de simulação e medição está associada com uma única instância, a QoS das instâncias foi calculada individualmente. Portanto, esta foi a abordagem utilizada para medir os resultados apresentados nos Capítulos 5 e 6. No contexto de controladores, a QoS de um controlador C é definida como a menor QoS

dentre todas as instâncias p_i gerenciadas por C. Neste caso, se ao menos uma das instâncias de C recebe QoS menor que a estabelecida por seu respectivo SLO, então C tem seu SLO não satisfeito. Caso contrário, o controlador C tem seu SLO atendido;

- **Concorrente:** nesta abordagem o provedor considera que o serviço está sendo entregue para um controlador C no tempo t apenas se cada instância p_i gerenciada por C tem uma encarnação p_i^e no tempo t onde $s(p_i^e, t) = Executando$. Em outras palavras, considera-se que o serviço está sendo entregue para C apenas quando todas as instâncias sob a responsabilidade de C estão executando simultaneamente. Assim sendo, é possível que existam períodos onde uma ou mais instâncias de C estejam em execução, mas sem contribuir para aumentar a QoS de C, dado que pelo menos uma instância de C não está em execução. Dessa forma, a QoS do controlador é definida levando em conta conjuntamente todas as instâncias gerenciadas pelo mesmo. Uma vez que a QoS do controlador tenha sido calculada, este tem seu SLO satisfeito se sua QoS for maior ou igual à meta estabelecida pelo seu respectivo SLO;

- **Agregada:** nesta abordagem a QoS de um controlador também é calculada levando em consideração conjuntamente todas as instâncias sob sua responsabilidade. Esta abordagem pode agregar as instâncias de um controlador de duas formas: tempo ou tarefas computadas. No primeiro caso, considera-se que sempre que uma instância executa ela está contribuindo para melhorar a QoS de seu controlador. Assim, a QoS de um controlador C é dada pela média das QoSs das instâncias $p_i, p_i \in C$. Na agregação por tarefas computadas, considera-se que a instância contribui para melhorar a QoS do controlador apenas quando contribui para que o mesmo torne-se completo. Considerando aplicações *batch stateless*, as encarnações que falharam têm sua computação desperdiçada. Nesse contexto, apenas as encarnações completas e atualmente em execução contribuem para melhorar a QoS de um controlador. Em ambos os casos, uma vez que a QoS do controlador tenha sido calculada, este tem seu SLO satisfeito se sua QoS for maior ou igual à meta estabelecida por seu SLO. Destaca-se que um controlador C pode ter seu SLO satisfeito inclusive se uma ou mais de suas instâncias p_i não receberem, individualmente, a QoS mínima esperada.

7.2.4 Escalonamento em operação no contexto de controladores

No contexto de controladores, o escalonamento orientado por QoS continua operando exatamente como descrito na Seção 3.5. Embora a QoS de um controlador seja medida de acordo com uma das abordagens propostas na seção anterior, o escalonador continua tomando decisões sobre alocação e preempção de instâncias individualmente. Por esta razão, cada instância no sistema deve ter uma métrica de QoS ($Q_j(t)$) associada. Como detalhado no Capítulo 3, esta métrica é utilizada pelo escalonador em suas decisões, e, neste trabalho, a métrica de QoS é definida com base nas métricas *TTV* (Equação 3.4) e *recoverability* (Equação 3.7) de uma requisição. Em um cenário com uso de controladores (*i.e.* requisições que podem estar associadas com mais de uma instância), a métrica de QoS associada a uma instância de uma requisição j depende: (i) do estado desta instância; (ii) da classe do controlador que é responsável por esta instância; e, (iii) da abordagem de medição de QoS escolhida pelo usuário para seu controlador.

Ao calcular a métrica de QoS ($Q_j(t)$) para uma instância de uma requisição (controlador) j no tempo t, considera-se que esta instância estará pendente a partir de t. A ideia central é verificar, em caso de necessidade de preempções, quais as instâncias mais apropriadas para ficarem pendentes. O **estado da instância** no instante t é importante no cálculo desta métrica, pois ele impacta na contribuição que a requisição j estará recebendo para a sua QoS a partir de t. Quando o escalonador, no tempo t, calcula a métrica $Q_j(t)$ a ser associada com uma instância de j na fila de espera (estado *Esperando*), ele considera que a instância continuará *Esperando* após t. Portanto, as instâncias do controlador j que já estão em execução continuarão *Executando* após t, enquanto que as instâncias do mesmo controlador que estão *Esperando* em t continuarão pendentes após t. Neste caso, j continuará recebendo contribuição para melhorar sua QoS das mesmas instâncias que já estão contribuindo. Por outro lado, quando o escalonador calcula $Q_j(t)$ a ser associada com uma instância de j no estado *Executando*, este considera que a instância será preemptada no instante t. Neste cenário, o número de instâncias de j em execução e pendentes após t será, respectivamente, decrementado e incrementado em 1 unidade.

A **classe de seu controlador** também é importante no cálculo da métrica de QoS a ser associada com uma instância. Esta classe define quais as encarnações da instância de fato contribuem para a QoS do controlador. Por exemplo, considerando o contexto de controla-

dores no Kubernetes, uma encarnação de instância no estado *Falha* não contribui para a QoS de um controlador *Batch*, pois isso significa que esta encarnação foi interrompida antes de concluir sua computação. Por outro lado, uma outra encarnação no mesmo estado pode contribuir para a QoS de um controlador de serviço, pois enquanto a mesma esteve em execução ela pode ter contribuído para o serviço de seu controlador. Essas situações serão discutidas em detalhes mais adiante.

Por último, a **abordagem de medição** escolhida define por quanto tempo cada encarnação de instância contribui para melhorar a QoS de seu controlador. Por exemplo, quando a abordagem agregada é utilizada em um controlador de serviço, uma encarnação de instância contribui para melhorar a QoS de seu controlador por todo o período de sua execução. Por outro lado, quando a abordagem concorrente é utilizada, uma encarnação de instância contribuiu para incrementar a QoS de seu controlador apenas enquanto esteve executando simultaneamente com todas as outras instâncias do mesmo controlador.

7.2.5 Calculando as métricas de QoS de um controlador

Como descrito no Capítulo 3, neste trabalho, a disponibilidade de uma requisição é considerada como a métrica de QoS de interesse para o escalonador. Basicamente, dada uma requisição (controlador) j, a métrica de QoS $Q_j(t)$ utilizada para decisões é definida com base em sua disponibilidade (Equação 3.1), seu TTV (Equação 3.4) e/ou sua *recoverability* (Equação 3.7). As funções $e_j(t)$, $d_j(t)$ e $p_j(t)$ utilizadas nessas equações são definidas de acordo com a abordagem de medição de QoS estabelecida para j. Cada abordagem proposta estabelece uma semântica diferente para a QoS entregue ao controlador (discutido na Seção 7.2.3). Além disso, a contribuição que j continuará recebendo para melhorar sua QoS (c_j) ao longo do intervalo do TTV ($\Delta t^*_j(t)$) também é definida de acordo com a abordagem de medição usada por j e pelo estado da instância sob análise do escalonador ($Executando$ ou $Esperando$). As subseções a seguir definem as funções utilizadas para calcular as métricas de interesse ($e_j(t)$, $d_j(t)$ e $p_j(t)$) para cada classe de controlador e abordagem de medição proposta. Além disso, também são discutidas particularidades relacionadas com cada um dos cenários.

Controlador de Serviço

Como discutido anteriormente, sempre que uma encarnação de uma instância p_i sob responsabilidade de um controlador de serviço C_s falha, uma nova encarnação de p_i é criada e adicionada à fila de espera para escalonamento. A Figura 7.8 apresenta uma visão geral dos instantes de tempo associados com uma instância e suas encarnações.

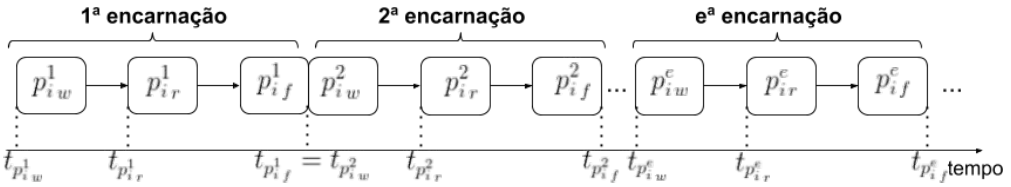


Figura 7.8: Visão geral dos tempos associados com uma instância p_i de um controlador de serviço.

Admitindo que p_i^e é a *e-ésima* encarnação da instância p_i de um controlador de serviço C_s, p_{iw}^e, p_{ir}^e e p_{if}^e representam a encarnação p_i^e nos estados *Esperando*, *Executando* e *Falha*, respectivamente. Assim sendo, os instantes de tempo $t_{p_{iw}^e}$, $t_{p_{ir}^e}$ e $t_{p_{if}^e}$ são os momentos em que p_i^e vai, respectivamente, para os estados *Esperando*, *Executando* e *Falha*. Neste trabalho, assume-se que uma nova encarnação de uma instância é criada no mesmo instante que a encarnação anterior falhou, portanto, $t_{p_{if}^e} = t_{p_i^{e+1}{}_w}$.

Considere que $w(p_i^e, t)$ e $r(p_i^e, t)$ indicam, respectivamente, as quantidades de tempo que a encarnação p_i^e esteve *Esperando* e *Executando* até o tempo t. Os valores retornados por essas funções são calculados com base no estado de p_i^e no tempo t.

(i) Quando $p_i^e \in W_{C_s}(t)$:

$w(p_i^e, t) = t - t_{p_{iw}^e}$

$r(p_i^e, t) = 0$

(ii) Quando $p_i^e \in R_{C_s}(t)$:

$w(p_i^e, t) = t_{p_{ir}^e} - t_{p_{iw}^e}$

$r(p_i^e, t) = t - t_{p_{ir}^e}$

(iii) Quando $p_i^e \in F_{C_s}(t)$:

$w(p_i^e, t) = t_{p_{ir}^e} - t_{p_{ir}^e}$

$r(p_i^e, t) = t_{p_{if}^e} - t_{p_{ir}^e}$

Este trabalho assume que cada controlador de serviço C_s submetido por um usuário é associado com uma classe de serviço i oferecida pelo provedor. Caso o usuário não especifique uma das classes de serviço para o controlador, a classe configurada como a classe padrão do provedor é associada ao mesmo. As classes de serviço têm metas de QoS definidas pelos SLOs que são estabelecidos em seus SLAs. Considere $O_{i_{C_s}}$ como a QoS prometida pelo provedor para a classe de serviço do controlador C_s. Cada instância p_i gerenciada por C_s herda a meta de QoS de C_s — $O_{i_{p_i}} = O_{i_{C_s}}$. Seguem as definições das funções para cada abordagem de medição proposta.

Independente. Nesta abordagem as métricas são calculadas individualmente para cada instância $p_i, p_i \in C_s$. Por esta razão, considere que cada instância p_i do controlador C_s está associada a uma requisição independente de mesmo nome. Neste cenário, a disponibilidade de um controlador C_s no tempo t, $A_{C_s}(t)$, é dada pela menor disponibilidade dentre todas as instâncias p_i, $A_{p_i}(t)$, sob sua responsabilidade.

Portanto, para calcular a disponibilidade de um controlador C_s no tempo t é necessário calcular a disponibilidade $A_{p_i}(t)$ de cada instância p_i, $p_i \in C_s$. Nesse contexto, sempre que uma encarnação p_i^e estiver em execução, ela estará contribuindo para a QoS da instância p_i. Isso implica que $e_{p_i}(t) = \sum_{p_i^e \in I_{pi}(t)} r(p_i^e, t)$ e $d_{p_i}(t) = 0$. Ou seja, não há tempo em execução de encarnações de p_i que não tenham efetivamente contribuído para melhorar sua disponibilidade. Já o tempo total em que as encarnações da instância p_i estiveram pendentes até t é dado por $p_{p_i}(t) = \sum_{p_i^e \in I_{pi}(t)} w(p_i^e, t)$. Assim sendo, substituindo-se as funções na Equação 3.1, a disponibilidade de uma instância p_i no tempo t é dada por:

$$A_{p_i}(t) = \frac{\sum_{p_i^e \in I_{pi}(t)} r(p_i^e, t)}{\sum_{p_i^e \in I_{pi}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{pi}(t)} w(p_i^e, t)} \tag{7.1}$$

De forma semelhante, substituindo as funções definidas para esta abordagem na Equação 3.6, a quantidade de tempo que a instância p_i esteve pendente desde que seu SLO passou a ser violado até o instante t, $\Delta r_{p_i}(t)$, é definido como segue.

$$\Delta r_{p_i}(t) = \frac{\sum_{p_i^e \in I_{pi}(t)} r(p_i^e, t)}{O_{i_{pi}}} - \Big(\sum_{p_i^e \in I_{pi}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{pi}(t)} w(p_i^e, t) \Big). \tag{7.2}$$

No caso do cálculo do intervalo $\Delta t_{p_i}(t)$ de uma instância p_i no tempo t, torna-se necessário discutir algumas particularidades desta abordagem de medição. $\Delta t_{p_i}(t)$ representa o

intervalo de tempo que a instância p_i pode permanecer sem receber contribuição para sua QoS sem que seu SLO seja violado. Como discutido antes, uma nova encarnação de uma instância é criada apenas quando a encarnação anterior falha. Assim sendo, por construção, é impossível que uma instância p_i tenha, no mesmo instante t, uma encarnação pendente e uma outra em execução. Por esta razão, o número de encarnações de p_i ativas no sistema sempre será 1. Admitindo que $v^r_{p_i}(t)$ e $v^w_{p_i}(t)$ são, respectivamente, o número de encarnações de p_i em execução e pendente em um instante no tempo t, $v^r_{p_i}(t) + v^w_{p_i}(t) = 1$ para qualquer ponto no tempo t.

Além disso, independente do estado da instância p_i analisada pelo escalonador, a contribuição que p_i receberá ao longo do intervalo $\Delta t_{p_i}(t)$ para elevar sua disponibilidade será sempre nula ($c_{p_i} = 0$). Considere que p^e_i é a encarnação sob análise do escalonador, se $s(p^e_i, t) =$ *Esperando* ($v^w_{p_i}(t) = 1$), o escalonador considera manter esta encarnação esperando; caso contrário, se $s(p^e_i, t) =$ *Executando* ($v^r_{p_i}(t) = 1$), o escalonador considera preemptar p_i. Uma vez que $v^r_{p_i}(t) + v^w_{p_i}(t) = 1$, em ambos os casos considera-se que não haverá encarnação executando após t, portanto, não haverá nenhuma encarnação de p_i contribuindo para melhorar sua QoS ao longo do intervalo $\Delta t_{p_i}(t)$ ($c_{p_i} = 0$). Portanto, fazendo as devidas substituições na Equação 3.3, o $\Delta t(p_i, t)$, é definido como segue.

$$\Delta t_{p_i}(t) = \frac{\sum_{p^e_i \in I_{pi}(t)} r(p^e_i, t)}{O_{i_{p_i}}} - \Big(\sum_{p^e_i \in I_{pi}(t)} r(p^e_i, t) + \sum_{p^e_i \in I_{pi}(t)} w(p^e_i, t) \Big). \tag{7.3}$$

Destaca-se que, nesta abordagem, como discutido no Capítulo 4 (Seção 4.1.3), a métrica de QoS ($Q_{p_i}(t)$) a ser associada a uma instância p_i ($\Delta r^*_{p_i}(t)$ ou $\Delta t^*_{p_i}(t)$) é dada pela mesma equação (as Equações 7.2 e 7.3 são iguais).

Concorrente. Nesta abordagem o provedor considera que está entregando serviço para um controlador C_s apenas quando todas as instâncias p_i, $p_i \in C_s$, estão executando concorrentemente. De acordo com esta abordagem, sempre que o escalonador decidir sobre alocação/preempção de uma encarnação de uma instância p_i de um controlador C_s, as métricas de QoS a serem associadas com p_i são definidas pelas métricas de seu controlador ($A_{p_i}(t) = A_{C_s}(t)$ e $Q_{p_i}(t) = Q_{C_s}(t)$).

Considere que $c(p^e_i, t)$ representa a quantidade de tempo que uma encarnação p^e_i esteve executando simultaneamente com encarnações de todas as outras instâncias gerenciadas pelo mesmo controlador. Dessa forma, o tempo total que as encarnações das instân-

cias de C_s efetivamente contribuem para melhorar a QoS de C_s até t é dado por $e_{C_s}(t) = \sum_{p_i^e \in I_{Cs}(t)} c(p_i^e, t)$. Por outro lado, é possível que uma ou mais instâncias de C_s tenham executado por algum período, mas essa execução não contribuiu para melhorar a QoS de C_s, logo $d_{C_s}(t) = \sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) - \sum_{p_i^e \in I_{Cs}(t)} c(p_i^e, t)$. Já o tempo que as encarnações das instâncias de C_s estiveram pendentes até o instante t é dado por $p_{C_s}(t) = \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t)$. Assim, com base na Equação 3.1, a disponibilidade do controlador C_s no tempo t é definida como segue.

$$A_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} c(p_i^e, t)}{\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t)} \tag{7.4}$$

Substituindo as funções definidas para esta abordagem na Equação 3.6, o intervalo $\Delta r_{C_s}(t)$ do controlador C_s no tempo t é dado como segue.

$$\Delta r_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} c(p_i^e, t)}{O_{i_{C_s}}} - \Big(\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t) \Big) \tag{7.5}$$

Com relação ao cálculo do intervalo $\Delta t_{C_s}(t)$, destaca-se que também não há contribuição para melhorar a QoS de C_s ao longo deste período ($c_{C_s} = 0$). Considere que o escalonador está analisando a encarnação p_i^e no tempo t. Se $s(p_i^e, t) = Esperando$, o escalonador considera manter esta encarnação na fila e nenhuma outra instância estará contribuindo para a QoS de C_s — visto que o provedor só contabiliza QoS quando todas as instâncias executam concorrentemente —; caso contrário, se $s(p_i^e, t) = Executando$, o escalonador considera que esta encarnação será preemptada, e, consequentemente, nenhuma outra instância contribuirá para a QoS de C_s. Dessa forma, substituindo adequadamente as funções definidas para esta abordagem na Equação 3.3, o intervalo $\Delta t_{C_s}(t)$ é definido como segue.

$$\Delta t_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} c(p_i^e, t)}{(v_{C_s}^r(t) + v_{C_s}^w(t)) O_{i_{Cs}}} - \frac{(\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t))}{v_{C_s}^r(t) + v_{C_s}^w(t)} \tag{7.6}$$

Visto que esta abordagem possibilita a existência de instâncias em execução no sistema sem que elas contribuam para aumentar a QoS de seus controladores, sugere-se que essas instâncias sejam as primeiras a serem interrompidas quando preempções forem necessárias. Além disso, a preempção dessas instâncias pode não ter um custo associado, visto que sua preempção não prejudicará na QoS entregue aos seus respectivos controladores.

Agregada. Nesta abordagem, a disponibilidade do controlador C_s é dada pelo tempo médio em que todas as instâncias p_i, $p_i \in C_s$, estiveram em execução. Novamente, sempre que o escalonador decidir sobre alocação/preempção de uma encarnação de uma instância p_i de um controlador C_s, as métricas de QoS a serem associadas com p_i são definidas pelas métricas de seu controlador ($A_{p_i}(t) = A_{C_s}(t)$ e $Q_{p_i}(t) = Q_{C_s}(t)$).

Neste cenário, sempre que uma encarnação de instância p_i^e estiver em execução, ela está contribuindo para melhorar a disponibilidade de seu controlador C_s. Portanto, isso implica que, no instante t, $e_{C_s}(t) = \sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t)$ e $d_{C_s}(t) = 0$. Já o tempo total que as encarnações de C_s estiveram pendentes até instante t é dado por $p_{C_s}(t) = \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t)$. Dessa forma, de acordo com a Equação 3.1, a disponibilidade do controlador C_s no tempo t é definida como segue.

$$A_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t)}{\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t)} \tag{7.7}$$

Com base na Equação 3.6 e nas funções definidas para esta abordagem, o intervalo $\Delta r_{C_s}(t)$ é estabelecido como segue.

$$\Delta r_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t)}{O_{iC_s}} - \Big(\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t) \Big). \tag{7.8}$$

Já com relação ao cálculo do intervalo $\Delta t_{C_s}(t)$, é possível que o controlador C_s receba alguma contribuição para melhoria de sua disponibilidade c_{C_s} ao longo do intervalo $\Delta t_{C_s}(t)$. Considere que $v^r_{C_s}(t)$ é a quantidade de instâncias de C_s em execução no tempo t e que p_j^e é a encarnação sob análise do escalonador. Se $s(p_j^e, t) = Esperando$, o escalonador considera manter esta encarnação esperando. Neste caso, as instâncias de C_s que já estão em execução em t continuarão executando após t (*i.e.*, contribuindo para aumentar sua disponibilidade), portanto, $c_{C_s} = v^r_{C_s}(t)$. No cenário onde $s(p_j^e, t) = Executando$, o escalonador considera preemptar esta instância em t. Neste caso, a quantidade de instâncias que contribuirão para a disponibilidade de C_s após t é dada pela quantidade de instâncias que já contribuem para a QoS de C_s em t decrementada em 1 unidade, portanto, $c_{C_s} = v^r_{C_s}(t) - 1$. Assim sendo, substituindo as funções definidas para esta abordagem na Equação 3.3, o intervalo $\Delta t_{C_s}(t)$ do controlador C_s no tempo t é dado por:

$$\Delta t_{C_s}(t) = \frac{\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) - O_{i_{C_s}}(\sum_{p_i^e \in I_{Cs}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cs}(t)} w(p_i^e, t))}{(v_{C_s}^r(t) + v_{C_s}^w(t))O_{i_{C_s}} - c_{C_s}}. \quad (7.9)$$

Controlador *Batch*

Semelhante ao controlador de serviço, sempre que uma encarnação de instância p_i gerenciada por um controlador *Batch* C_b vai para o estado *Falha* (por exemplo, por causa da atuação do escalonador ou falha de servidor), uma nova encarnação de p_i é criada e adicionada à fila de espera para escalonamento. No entanto, diferentemente de um controlador de serviço, espera-se que uma encarnação de cada instância p_i, em algum momento, vá para o estado *Completa*. Isso ocorre quando a tarefa computacional executada na instância em questão termina com sucesso. Neste cenário, esta será a última encarnação da instância p_i, consequentemente, nenhuma nova encarnação de p_i é criada. A Figura 7.9 apresenta uma visão geral dos instantes de tempos associados com uma instância p_i de um controlador *Batch*.

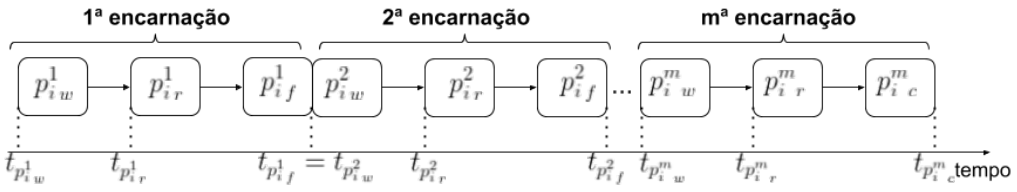


Figura 7.9: Visão geral dos tempos associados com uma instância p_i de um controlador *Batch*.

Considerando que p_i^e é a *e-ésima* encarnação da instância p_i de um controlador *Batch* C_b, admita que $p_{i\,w}^e$, $p_{i\,r}^e$, $p_{i\,f}^e$ e $p_{i\,c}^e$ representam, respectivamente, a encarnação p_i^e nos estados *Esperando, Executando, Falha* e *Completa*. Nesse sentido, os instantes de tempo $t_{p_{i\,w}^e}$, $t_{p_{i\,r}^e}$, $t_{p_{i\,f}^e}$ e $t_{p_{i\,c}^e}$ indicam os momentos em que p_i^e vai, respectivamente, para os estados *Esperando, Executando, Falha* e *Completa*. Neste estudo, também assume-se que uma nova encarnação de uma instância que falhou é criada no mesmo instante da falha, portanto, $t_{p_{i\,f}^e} = t_{p_i^{e+1}\,w}$.

Ainda admita que $w(p_i^e, t)$ e $r(p_i^e, t)$ são, respectivamente, os tempos totais que uma encarnação p_i^e esteve *Esperando* e *Executando* até o tempo t. Os valores retornados por estas funções são computados com base no estado de p_i^e no tempo t. Segue a definição dessas funções para cada estado possível de p_i^e.

(i) Quando $p_i^e \in W_{C_s}(t)$:

$w(p_i^e, t) = t - t_{p_{i\,w}^e}$

$r(p_i^e, t) = 0$

(ii) Quando $p_i^e \in R_{C_s}(t)$:

$w(p_i^e, t) = t_{p_{i\,r}^e} - t_{p_{i\,w}^e}$

$r(p_i^e, t) = t - t_{p_{i\,r}^e}$

(iii) Quando $p_i^e \in F_{C_s}(t)$:

$w(p_i^e, t) = t_{p_{i\,r}^e} - t_{p_{i\,w}^e}$

$r(p_i^e, t) = t_{p_{i\,f}^e} - t_{p_{i\,r}^e}$

(iv) Quando $p_i^e \in C_{C_b}(t)$:

$w(p_i^e, t) = t_{p_{i\,r}^e} - t_{p_{i\,w}^e}$

$r(p_i^e, t) = t_{p_{i\,c}^e} - t_{p_{i\,r}^e}$

Nota-se que $w(p_i^e, t)$ e $r(p_i^e, t)$ são definidas da mesma forma para os estados *Esperando*, *Executando* e *Falha* para encarnações de ambas as classes de controladores. A principal diferença entre as classes é que uma encarnação de instância de um controlador *Batch* possui um estado adicional (*Completa*).

É importante destacar que os controladores *Batch* foram projetados para executar aplicações *batch*, por exemplo, aplicações do tipo "saco-de-tarefas" (BoT, do inglês *Bag-of-Tasks*). Por esta razão, esta classe de controlador gerencia um conjunto finito de instâncias que precisam ser finalizadas com sucesso. De acordo com a literatura, o *throughput* é uma das métricas de QoS mais apropriadas para avaliar aplicações desse tipo. Esta métrica pode ser calculada de diversas formas. Neste trabalho, considera-se que o *throughput* de um controlador *Batch* C_b é definido pela quantidade de instruções processadas por suas instâncias que contribuem para completar a execução de C_b por unidade de tempo. Um controlador C_b conclui quando todas as instâncias por ele gerenciadas são concluídas com sucesso. Portanto, em outra palavras, o *throughput* de C_b é o tempo que suas instâncias estiveram em execução contribuindo para alcançar sua conclusão em relação ao tempo total que estiveram no sistema. Nesse sentido, o *throughput* do controlador pode ser relacionado com sua disponibilidade de acordo com a Equação 3.1.

Enquanto a disponibilidade de um controlador *Batch* C_b é calculada, o provedor pode considerar que a alocação de recursos está efetivamente contribuindo para melhorar a disponibilidade de C_b apenas quando esta alocação estiver contribuindo para concluir C_b. Neste cenário, um controlador *Batch* com 50% como meta de disponibilidade significa que este controlador deve ter recursos alocados contribuindo para sua finalização durante pelo menos 50% do tempo desde sua admissão.

Com base no sistema Kubernetes, este trabalho assume que as encarnações de uma instância gerenciada por um controlador *Batch* são *stateless*. Isso significa que os processamentos realizados por uma encarnação no estado *Falha* são descartados, portanto, não contribuem para a conclusão de seu controlador.

Sob o ponto de vista das abordagens propostas para medição de QoS de um controlador, nem todas são adequadas para serem utilizadas com controladores *Batch*. Por exemplo, a abordagem independente calcula a QoS para as instâncias de um controlador individualmente, no entanto, um controlador *Batch* espera ter suas tarefas computacionais concluídas, independente da QoS de suas instâncias individualmente. Por esta razão, a abordagem independente não é adequada para o cenário com controladores *Batch*. Além disso, uma aplicação *batch* tem como característica executar tarefas computacionais independentes umas das outras. Por isso, não necessariamente suas instâncias devem existir e executar simultaneamente no sistema. Assim como um controlador C_b pode ter que completar uma quantidade de instâncias maior do que a quantidade que ele pode executar paralelamente. Por esses motivos, a abordagem concorrente também não é adequada para o cenário com controladores *Batch*. Portanto, apenas a abordagem de medição agregada é apropriada para esta classe de controladores.

Agregada. Nesta abordagem, a disponibilidade de um controlador C_b é dada pela proporção do tempo em que suas instâncias p_i, $p_i \in C_b$, estiverem contribuindo para a conclusão de C_b. Assim como ocorre quando controladores de serviço utilizam essa abordagem, sempre que o escalonador decidir sobre alocação/preempção de uma encarnação de uma instância p_i de um controlador C_b, as métricas de QoS a serem associadas com p_i são definidas pelas métricas de seu controlador ($A_{p_i}(t) = A_{C_b}(t)$ e $Q_{p_i}(t) = Q_{C_b}(t)$).

Neste cenário, uma encarnação de instância p_i^e do controlador C_b está contribuindo para melhorar a disponibilidade de C_b no tempo t se ela estiver em execução ($p_i^e \in R_{C_b}(t)$)

ou se ela já tiver sido concluída com sucesso ($p_i^e \in C_{C_b}(t)$). Considere $e'_{C_b}(t) = \sum_{p_i^e \in R_{C_b}(t)} r(p_i^e, t) + \sum_{p_i^e \in C_{C_b}(t)} r(p_i^e, t)$ como o tempo total que as instâncias de C_b contribuíram para melhorar sua QoS até t. Como o estado dos processos não são mantidos entre diferentes encarnações, o tempo total que as encarnações falhas estiveram em execução é descartado, portanto, considere $d'_{C_b}(t) = \sum_{p_i^e \in F_{Cb}(t)} r(p_i^e, t)$ como este tempo total de execução descartado até t. O tempo total que as instâncias de C_b estiveram pendentes até t é dado por $p_{C_b}(t) = \sum_{p_i^e \in I_{Cb}(t)} w(p_i^e, t)$. Além disso, o tempo total que todas as instâncias de C_b estiveram executando, independente de contribuírem ou não para sua disponibilidade, é dado por $\sum_{p_i^e \in I_{C_b}(t)} r(p_i^e, t) = \sum_{p_i^e \in R_{C_b}(t)} r(p_i^e, t) + \sum_{p_i^e \in C_{C_b}(t)} r(p_i^e, t) + \sum_{p_i^e \in F_{C_b}(t)} r(p_i^e, t)$. Assim sendo, com base na Equação 3.1 e considerando $e_{C_b}(t) = e'_{C_b}(t)$ e $d_{C_b}(t) = d'_{C_b}(t)$, a disponibilidade do controlador C_b no tempo t é definida como segue.

$$A_{C_b}(t) = \frac{\sum_{p_i^e \in R_{Cb}(t)} r(p_i^e, t) + \sum_{p_i^e \in C_{Cb}(t)} r(p_i^e, t)}{\sum_{p_i^e \in I_{Cb}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cb}(t)} w(p_i^e, t)} \tag{7.10}$$

A diferença principal no uso desta abordagem com os controladores de serviço e *Batch* é que, para o primeiro, sempre que uma encarnação estiver em execução, ela estará contribuindo para melhorar a disponibilidade de seu controlador. Já no caso de controladores *Batch*, nem sempre a execução da encarnação contribui para a QoS do controlador. Neste cenário, as encarnações no estado *Falha* não contribuem aumentar para a disponibilidade do controlador, *i.e.* o período que estas encarnações estiveram em execução é descartado no cálculo da disponibilidade de seu controlador.

Portanto, considere que p_j^e é uma encarnação de instância de C_b sob análise do escalonador no tempo t. Se $s(p_j^e, t) = Esperando$, o escalonador considera mantê-la pendente. Neste caso, $e_{C_b}(t) = e'_{C_b}(t)$ e $d_{(C_b}(t) = d'_{C_b}(t)$, visto que nenhum tempo de execução adicional será descartado. Caso contrário, se $s(p_j^e, t) = Executando$, o escalonador considera preemptá-la no tempo t. Neste cenário, o tempo de execução de p_j^e deve ser descartado no cálculo da métrica de QoS ($Q_{C_b}(t)$). Consequentemente, $e_{C_b}(t) = e'_{C_b}(t) - r(p_j^e, t)$ e $d_{C_b}(t) = d'_{C_b}(t) + r(p_j^e, t)$. Assim sendo, com base na Equação 3.6 e no estado da encarnação sob análise do escalonador, o $\Delta r_{C_b}(t)$ do controlador C_b no tempo t é dado por:

$$\Delta r_{C_b}(t) = \frac{e_{C_b}(t)}{O_{i_{C_b}}} - \Big(\sum_{p_i^e \in I_{Cb}(t)} r(p_i^e, t) + \sum_{p_i^e \in I_{Cb}(t)} w(p_i^e, t) \Big) \tag{7.11}$$

De forma similar ao que ocorre com controladores de serviço, também é possível que um controlador *Batch* receba alguma contribuição para melhoria de sua disponibilidade ao longo do intervalo $\Delta t_{C_b}(t)$. Considere $v^r_{C_b}(t)$ como o número de instâncias do controlador C_b em execução no tempo t e que p^e_j é a encarnação sob análise do escalonador no tempo t. Se $s(p^e_j, t) = Esperando$, o escalonador considera mantê-la pendente após t. Neste cenário, as instâncias de C_b que já estão em execução em t continuarão executando após t, isto é, essas instâncias contribuirão para aumentar a disponibilidade de C_b após t, portanto, $c_{C_b} = v^r_{C_b}(t)$. Neste caso, não há tempo de execução adicional a ser descartado, então $e_{C_b}(t) = e'_{C_b}(t)$ e $d_{C_b}(t) = d'_{C_b}(t)$. Por outro lado, se $s(p^e_j, t) = Executando$, o escalonador considera preemptar esta instância em t. Neste caso, é necessário decrementar a quantidade de instâncias que contribuirão para a disponibilidade de C_b ao longo de $\Delta t_{C_b}(t)$, portanto $c_{C_b} = v^r_{C_b}(t) - 1$. Ademais, o tempo de execução de p^e_j precisa ser descartado, isto é, $e_{C_b}(t) = e'C_b(t) - r(p^e_j, t)$ e $d_{C_b}(t) = d'_{C_b}(t) + r(p^e_j, t)$. Dessa forma, com base na Equação 3.3 e no estado da instância sob análise pelo escalonador, o intervalo $\Delta t_{C_b}(t)$ do controlador *Batch* C_b no tempo t é definido como segue:

$$\Delta t_{C_b}(t) = \frac{e_{C_b}(t) - O_{i_{C_b}}(\sum_{p^e_i \in I_{Cb}(t)} r(p^e_i, t) + \sum_{p^e_i \in I_{Cb}(t)} w(p^e_i, t))}{(v^r_{C_b}(t) + v^w_{C_b}(t))O_{i_{C_b}} - c_{C_b}}. \tag{7.12}$$

7.2.6 Avaliação Preliminar

Neste trabalho, a avaliação de viabilidade de implementação da política orientada por QoS se deu através de experimentos de medição. O protótipo implementado (descrito na Seção 5.2) foi evoluído de forma a suportar o cálculo das métricas de QoS (disponibilidade, TTV e *recoverability*) de acordo com as diferentes abordagens propostas anteriormente. Nesse cenário, experimentos de medição foram executados com propósito de comparar os resultados obtidos pelo protótipo do escalonador proposto com os obtidos pelo escalonador padrão baseado em prioridade do Kubernetes. Destaca-se que as otimizações discutidas nas Seções 7.1.1 e 7.1.2 não foram implementadas e consideradas nesses experimentos de medição. O objetivo principal é verificar a viabilidade de implementação do escalonador proposto no contexto de controladores no Kubernetes, que é a forma de uso mais comum desse sistema. A seguir são apresentados a carga de trabalho, infraestrutura e projeto experimental dos experimentos de medição executados.

Carga de Trabalho

Esta carga de trabalho sintética foi concebida de forma que fosse possível antecipar o comportamento esperado dos escalonadores. Todos os controladores foram submetidos no início do teste, com o intervalo de 1 segundo entre a submissão de dois controladores subsequentes. Os testes executaram por 1 hora e todos os controladores estiveram ativos até o final dos testes, quando suas disponibilidades finais foram calculadas. Nesta avaliação preliminar, a carga de trabalho utilizada é homogênea em termo da classe dos controladores. Todos os controladores submetidos são da categoria de serviço, especificamente do tipo *Deployment*. No total, a carga de trabalho é composta por 53 controladores de 3 serviços diferentes (20 da classe de serviço 1, 16 da classe 2 e 17 da classe 3). Neste experimento, as metas de disponibilidades dos serviços 1, 2 e 3 foram, respectivamente, 100%, 95% e 90%. Cada controlador submetido é responsável por gerenciar 2 instâncias, portanto, há 106 instâncias a serem escalonadas no sistema. Além disso, todas as instâncias demandam a mesma quantidade de recursos ($0,187$ GB de memória RAM e $0,187$ vCPUs).

Infraestrutura

A infraestrutura utilizada nesse experimento para implantação do *cluster* consistiu de 5 servidores homogêneos — máquinas virtuais de um provedor OpenStack — cada um com 4 GB de memória RAM e 4 vCPUs. Neste *cluster*, o próprio sistema Kubernetes usou aproximadamente $0,25$ GB da memória em cada servidor para atividades de gerenciamento.

Projeto Experimental

Este experimento avalia o impacto das diferentes abordagens de medição de QoS na disponibilidade final dos controladores. Ele segue um projeto fatorial completo com dois fatores: o escalonador e a abordagem de medição de QoS utilizados. O primeiro fator tem dois níveis: o escalonador baseado em prioridade e o escalonador orientado por QoS. O segundo fator varia em três níveis de acordo com as abordagens de medição de QoS propostas anteriormente: independente, concorrente e agregada. Ao final da execução, a métrica de avaliação é a disponibilidade final do controlador.

Por fim, o escalonador orientado por QoS foi configurado com os mesmos parâmetros

utilizados na validação dos modelos de simulação (Capítulo 5). O limite aceitável para a sobrecarga com preempções de uma instância da classe i foi definido para ser $1-O_i$, onde O_i é o SLO de disponibilidade para a classe de serviço i. Além disso, as margens de segurança para todas as classes foram configuradas em 10 segundos.

Resultados e Discussão

A Figura 7.10 apresenta a disponibilidade final de cada controlador submetido. À esquerda estão os resultados obtidos quando a abordagem independente foi utilizada para medir a QoS do controlador, ao centro os resultados obtidos com a abordagem concorrente, e, à direita os resultados obtidos com a abordagem agregada. As disponibilidades resultantes do escalonador baseado em prioridade estão em vermelho, enquanto àquelas obtidas com o escalonador orientado por QoS estão em azul.

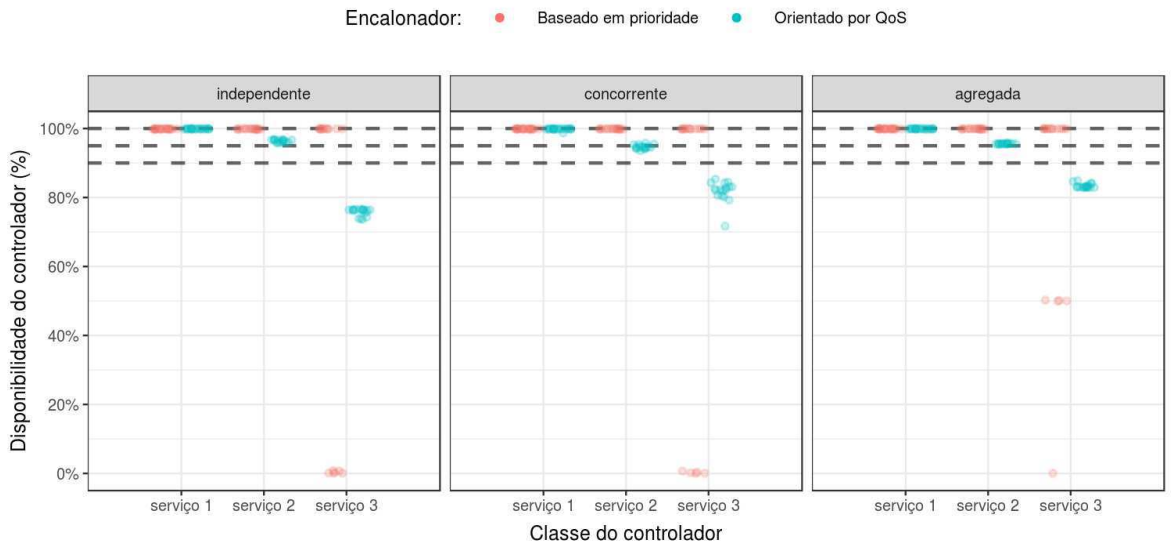


Figura 7.10: Disponibilidades finais dos controladores submetidos ao sistema Kubernetes.

Como pode ser observado na Figura 7.10, como esperado, no geral, o escalonador orientado por QoS conseguiu entregar disponibilidades mais próximas das metas estabelecidas. Isso ocorreu independente da abordagem de medição utilizada. Pequenas diferenças são esperadas devido às sobrecargas de preempção e escalonamento envolvidas.

O escalonador baseado em prioridade alocou recursos para os controladores mais importantes (serviços 1 e 2) de forma a oferecer 100% de disponibilidade para estes, independente

da abordagem de medição de QoS utilizada. Durante períodos com contenção de recursos, o escalonador baseado em prioridade não alterna instâncias da mesma classe em execução. Ao invés disso, o escalonador mantém algumas instâncias sempre em execução e outras sempre pendentes. Por esta razão, houve uma alta variabilidade nas disponibilidades fornecidas para os controladores do serviço 3 quando este escalonador foi usado. Lembrando que cada controlador cria e gerencia 2 instâncias, assim sendo, as abordagens independente e concorrente entregam disponibilidades de 100% para aqueles controladores que conseguiram ter suas 2 instâncias alocadas, enquanto os outros que tiveram ao menos uma das instâncias pendentes receberam disponibilidade de 0%. Na abordagem independente, isso se deve ao fato da disponibilidade do controlador ser definida pela menor disponibilidade dentre as disponibilidades de suas instâncias. Já a abordagem concorrente contabiliza a disponibilidade do controlador apenas quando todas as suas instâncias estão em execução simultaneamente, portanto, se ao menos uma das instâncias de um controlador nunca executou, o controlador recebe 0% de disponibilidade. Por outro lado, quando a abordagem agregada foi usada, os controladores do serviço 3 podem ser classificados em 3 grupos. Os controladores que tiveram suas 2 instâncias alocadas receberam 100% de disponibilidade; aqueles que tiveram apenas uma instância alocada receberam 50% de disponibilidade; enquanto os outros que não tiveram nenhuma das instâncias alocadas receberam 0% de disponibilidade. Neste caso, sempre que uma instância esteve em execução, ela esteve contribuindo para melhorar a disponibilidade de seu controlador.

Analisando a disponibilidade média geral entregue por cada escalonador, o escalonador orientado por QoS entregou uma disponibilidade geral melhor que o escalonador baseado em prioridade quando as abordagens independente ($91,1\%$ vs $88,6\%$) e concorrente ($92,5\%$ vs $90,5\%$) foram utilizadas. Quando a abordagem agregada foi usada, a disponibilidade média geral foi um pouco menor com o escalonador orientado por QoS ($93,3\%$ vs $94,3\%$). No entanto, como discutido no Capítulo 6, o escalonador orientado por QoS gerou déficits de QoS que são menores e menos variados que os obtidos com o escalonador baseado em prioridade. Isso pode ser visualizado na Figura 7.10 observando as diferenças entre as disponibilidades dos controladores e as metas de disponibilidades para aqueles controladores que tiveram seus respectivos SLOs não satisfeitos.

Apesar de simples, esses experimentos de medição inicias indicam a viabilidade de im-

plementação da política de escalonamento orientada por QoS em um sistema real amplamente utilizado.

Capítulo 8

Considerações Finais

Neste capítulo, são apresentadas as considerações finais deste trabalho, descrevendo as suas principais conclusões e listando possíveis trabalhos futuros.

8.1 Conclusões

Este trabalho apresentou uma visão geral das etapas envolvidas com o gerenciamento de recursos de um provedor de computação na nuvem: planejamento de capacidade, controle de admissão e escalonamento. Apesar do objeto de estudo ter sido a etapa de escalonamento, destacou-se que esta etapa não pode ser vista de forma completamente isolada. É importante ressaltar que as atividades de planejamento de capacidade e controle de admissão operam em conjunto com o escalonamento com o objetivo de evitar cenários de super provisionamento (para manter os custos o mais baixo possível) e sub provisionamento (para manter a QoS em níveis aceitáveis). Nesse contexto, pode-se inferir que o provedor tem como interesse operar em cenários com contenção moderada, onde a infraestrutura e carga de trabalho são dimensionadas de forma a manter alta utilização e ainda entregar a QoS prometida para as requisições. No entanto, caso ocorra períodos com alta contenção devido a imprecisões no controle de admissão e/ou no planejamento de capacidade, o escalonador precisa lidar com essas situações.

Através da revisão da literatura, apresentada no Capítulo 2, foi constatado que a estratégia de escalonamento comumente utilizada pelos principais provedores de larga escala toma decisões ineficientes em cenários com contenção de recursos. Apesar de usar a prioridade

como um *proxy* para a QoS prometida, o escalonamento baseado em prioridade não foi concebido para respeitar a QoS prometida para todas as classes de serviço simultaneamente. Em períodos com contenção de recursos, esta política sempre preemptará recursos de instâncias com prioridades mais baixas em benefício de outra com prioridade mais alta, isso ocorre independente das QoSs atuais recebidas pelas instâncias envolvidas. Além disso, nesses períodos, esta política também não provisiona recursos de forma justa entre requisições de uma mesma classe (para todas as classes). Uma vez que esta política não alterna as instâncias em execução, algumas permanecem sempre em execução e outras sempre pendentes, causando alta variabilidade na QoS entregue para essas requisições.

Nesse contexto, o Escalonamento Orientado por QoS foi descrito no Capítulo 3. Um escalonador que implementa esta política toma decisões levando em conta a QoS entregue para cada requisição no sistema no momento da tomada de decisão. Além da QoS atual, as respectivas metas de QoS também são consideradas pelo escalonador. Essas metas são definidas tipicamente pelos SLOs estabelecidos no SLA da classe de serviço da requisição. Assim sendo, o escalonamento orientado por QoS tem como objetivo cumprir os SLAs das requisições admitidas, independentemente de suas classes de serviço, ao mesmo tempo que promove provisionamento mais justo entre requisições de uma mesma classe, especialmente em períodos com contenção de recursos. Por provisionamento justo, entende-se que as requisições de uma mesma classe que competem pelo mesmo recurso devem receber QoSs semelhantes, inclusive quando não há recursos suficientes para entregar a QoS prometida. Isso ocorre pelo uso de um mecanismo que permite preempções de instâncias com base na métrica de QoS. A ideia principal é que, em períodos com contenção de recursos, as requisições cuja QoS estejam excedendo suas respectivas metas sejam preemptadas em benefício de outras com QoS abaixo da meta (ou que estejam mais próximas de violar seus SLOs).

Apesar da nova política de escalonamento poder ser utilizada com base em uma ou mais métricas de QoS, este trabalho considerou a disponibilidade como a métrica de QoS de interesse. Esta escolha se deu pelo fato da disponibilidade ser uma das principais preocupações de consumidores de computação em nuvem quando os SLAs são negociados [46; 41]. Portanto, tanto os modelos de simulação quanto o protótipo desenvolvidos calculam métricas de QoS associadas com a disponibilidade entregue para a requisição.

A avaliação da política proposta se deu comparando-se sua performance com a de um

escalonamento baseado em prioridade representando o estado-da-prática. Esta avaliação foi realizada empiricamente, através de experimentos de simulação e de medição. No Capítulo 4 foram apresentados os materiais e métodos adotados ao longo da avaliação. Amostras da carga de trabalho e da infraestrutura foram extraídas de rastros de execução de um sistema real e foram utilizadas nas análises. Essas amostras foram geradas de forma a representarem cenários com diferentes níveis de contenção. Em um primeiro momento, a análise ocorreu sob a perspectiva da QoS provida pelos escalonadores, analisando a QoS entregue pelos escalonadores e métricas derivadas desta QoS, tais como satisfação do SLO, custo com penalidades e justiça no provisionamento de recursos.

Os modelos de simulação utilizados foram validados através de experimentos de medição (Capítulo 5). Para validar o modelo do escalonador baseado em prioridade, o escalonador deste tipo padrão do sistema Kubernetes foi utilizado. O Kubernetes foi escolhido por ser bastante popular e por permitir a customização do escalonador sem necessariamente ter que alterar outros componentes do sistema. No caso do modelo do escalonador orientado por QoS, um protótipo foi desenvolvido como prova de conceito para a nova política. Este protótipo foi integrado ao Kubernetes através da substituição de seu escalonador padrão.

O Capítulo 6 apresentou e discutiu os resultados obtidos com os experimentos de simulação. As cargas utilizadas na avaliação abrangem 29 dias de um mês do provedor. Quando cada cenário é analisado por completo, no geral, verificou-se que o escalonador orientado por QoS alocou os recursos de forma que mais requisições conseguissem receber QoS igual ou acima da prometida. Isso ocorreu devido ao escalonador proposto ter mais flexibilidade para alternar as instâncias em execução. Quando não foi possível satisfazer o SLO de todas as requisições, no geral, o escalonador proposto alocou os recursos de forma a produzir déficits de QoS menores e menos variáveis que os produzidos pelo escalonador baseado em prioridade. Além disso, também verificou-se que que as penalidades geradas pelo uso do escalonador baseado em prioridade podem ser, em média, até 193% maiores quando comparadas com as penalidades produzidas pelo uso do escalonador orientado por QoS.

Os resultados de simulação também mostraram que, para cenários extremos onde o nível de contenção foi muito alto ou muito baixo (ou até mesmo sem contenção), o escalonador orientado por QoS teve comportamento similar ao do escalonador baseado em prioridade. No entanto, quando o nível de contenção foi moderado, o escalonador orientado por QoS

elevou substancialmente a QoS provida para as requisições com metas menores. Isso ocorre por causa da preempção de instâncias cuja QoS está excedendo a meta de seu SLO. Neste cenário, as requisições da classe *C* tiveram um aumento considerável na média da disponibilidade mínima fornecida (de 7% para 90%), como também no ganho médio da satisfação do SLO. Isso veio com uma pequena variação na QoS das requisições da classe *B*, o que não comprometeu a satisfação do SLO desta classe. Considerando que o planejamento de capacidade e o controle de admissão operam em conjunto com o escalonamento para evitar cenários de super provisionamento e sub provisionamento, um escalonador que opera bem em cenários com contenção moderada é muito útil. Ademais, o escalonador orientado por QoS também forneceu QoS de forma mais justa que o escalonador baseado em prioridade, inclusive em períodos com nível de contenção mais alto. Isso é interessante por permitir que os usuários tenham maior previsibilidade da indisponibilidade esperada para suas requisições em momentos com contenção de recursos.

Além das avaliações sob a perspectiva da QoS provida pelos escalonadores, este trabalho também apresentou duas análises sob um ponto de vista mais prático. Ambos os estudos foram apresentados no Capítulo 7. Inicialmente, uma análise de desempenho foi realizada para a nova política de escalonamento. A análise de desempenho se deu através da comparação da quantidade de processamento realizado por ambos os escalonadores. A quantidade de processamento foi definida em termos de operações executadas pelo escalonador. A verificação de viabilidade de um servidor para uma instância pendente foi contabilizada como uma operação executada pelo escalonador. Nesse sentido, algumas otimizações foram implementadas com o propósito de diminuir os custos de operação. Os resultados mostraram que as otimizações reduziram substancialmente a quantidade de operações executadas pelo escalonador orientado por QoS (82% em média). No entanto, como esperado, o custo operacional deste escalonador é maior que o baseado em prioridade (cerca $15,5$ vezes). A ausência de um mecanismo de *cache* é o principal motivo para a quantidade de operações executadas com o escalonador orientado por QoS ser tão maior.

Por último, este trabalho também analisou a viabilidade de implementação da nova política no contexto de um sistema real. Novamente, utilizou-se o sistema Kubernetes para a avaliação. Este sistema disponibiliza uma abstração de mais alto nível, denominada de controlador, para lidar com um grupo de instâncias homogêneas de uma mesma aplicação.

O controlador pode ser interpretado como uma requisição que pode estar associada com mais de uma instância. A submissão de controladores é a forma de uso mais comum do Kubernetes. Portanto, este trabalho caracterizou os principais controladores suportados pelo Kubernetes e propôs diferentes abordagens para calcular as métricas de QoS usadas pelo escalonador. Para cada classe de controlador, uma abordagem de medição pode calcular as métricas de formas diferentes com base na semântica de QoS definida na abordagem. Por isso, para cada abordagem de medição proposta, este trabalho definiu como cada métrica de QoS utilizada pelo escalonador pode ser calculada. Esta especificação permitiu que o protótipo do escalonador orientado por QoS fosse evoluído de forma a suportar a submissão de múltiplos controladores responsáveis por múltiplas instâncias. Através de experimentos de medição, observou-se que o escalonador orientado por QoS, como esperado, entregou QoSs mais próximas das respectivas metas de SLO dos controladores, independente de suas classes de serviço. Por outro lado, o escalonador baseado em prioridade entregou uma QoS maior que a prometida para as requisições da classe *B* enquanto que as disponibilidades (e sua variabilidade) para as requisições da classe *C* dependeram da abordagem de medição utilizada.

8.2 Trabalhos Futuros

A partir dos resultados obtidos nesse trabalho, observa-se possíveis pontos de evolução e aprimoramento da área de pesquisa abordada por essa tese de doutorado. As seguintes atividades de pesquisa são sugeridas como trabalhos futuros:

- Analisar o tempo de processamento das operações executadas pelos escalonadores baseado em prioridade e orientado por QoS. Como discutido na Seção 7.1.5, a quantidade de operações executadas por ambos os escalonadores é substancialmente diferente. No entanto, é possível que os tempos de execução das operações de ambos não sejam significativamente diferentes. Por esta razão, fazer uma análise do tempo de execução poderia indicar se realmente seria necessário buscar por mais otimizações para o escalonador orientado por QoS ou não.

- Implementar e avaliar as alternativas para melhorar a performance do escalonador ori-

entado por QoS. Na Seção 7.1.6 foram discutidas algumas possibilidades para que o escalonador proposto possa ter seu desempenho melhorado. Nesse sentido, a implementação de um mecanismo de *cache* é primordial para diminuir a quantidade de operações realizadas por este escalonador. Na Seção 7.1.6, algumas opções para o mecanismo de *caching* são discutidas e podem ser implementadas. Além disso, o relaxamento da otimização *equivalence class* também pode ajudar nessa direção. Portanto, essas e outras alternativas podem ser implementadas e analisadas em relação ao impacto na QoS provida e no número de operações executadas pelo escalonador;

- Elaborar e avaliar um método de escalonamento híbrido. Nesse contexto, o escalonador poderia ser adaptado para alterar seu modo de operação entre a política baseada em prioridade e orientada por QoS. A ideia central é permitir que a política orientada por QoS opere até um nível de contenção aceitável. Nesse sentido, a troca no modo de operação do escalonador pode ser feita com base no nível de contenção momentâneo do sistema. Por exemplo, quando o tamanho da fila exceder um dado limite, o escalonador deve operar segundo a política baseada em prioridade; caso contrário, quando o tamanho da fila for menor que este limite, o escalonador deve operar de acordo com a política orientada por QoS. Um novo modelo de simulação pode ser definido e implementado para este método de escalonamento híbrido. Em seguida, o impacto deste método híbrido na QoS provida e no desempenho do escalonamento deve ser analisado através de experimentos;

- Avaliar o comportamento dos escalonadores quando outras cargas de trabalho são submetidas. Apesar dos resultados apresentados no Capítulo 6 considerarem a execução de 10 amostras bastante diversas dos rastros de execução da Google, pode-se executar os mesmos experimentos de simulação para uma quantidade maior de amostras. Além disso, também é possível considerar outros rastros de execução para a geração das amostras, tais como os rastros da Azure[1]. Embora esses rastros não possuam informações sobre os servidores da infraestrutura, faz sentido definir a infraestrutura de forma semelhante a descrita na Seção 4.3 a partir dos rastros da Google já utilizados;

[1]Os rastros de execução da Azure estão disponíveis para download em https://github.com/Azure/AzurePublicDataset.

- Evoluir a análise do comportamento das diferentes abordagens de medição de QoS no contexto do Kubernetes. Nesse sentido, experimentos de medição com variadas cargas de trabalho podem ser realizados para verificar o impacto tanto da abordagem de medição como das classes dos controladores submetidos na QoS provida pelos escalonadores;

- Avaliar o impacto das diferentes políticas de escalonamento nas aplicações submetidas pelos usuários no contexto de um sistema real. As avaliações apresentadas ao longo deste trabalho medem a QoS oferecida pelo provedor às requisições dos usuários. No entanto, também é interessante analisar como o escalonamento executado pelas diferentes políticas impactam na QoS das aplicações que executam nos recursos provisionados. Nesse sentido, algumas métricas das aplicações poderiam ser coletadas e comparadas quando os diferentes escalonadores são executados. Esta análise poderia se dar através de experimentos de medição. O tempo de resposta (para aplicações de serviço) e o *thoughput* (para aplicações *batch*) são exemplos de métricas de QoS da aplicação que podem ser coletadas para esta análise;

- Definir e avaliar outras heurísticas para a função de pontuação de custo de preempção dos servidores elegíveis que necessitam de preempções. Apesar da heurística implementada neste trabalho ter proporcionado resultados satisfatórios, é possível avaliar quais seriam os resultados obtidos se outras heurísticas fosse implementadas. Nesse sentido, novas funções precisam ser implementadas de acordo com as novas heurísticas e a avaliação pode ocorrer através da reexecução dos mesmos experimentos de simulação;

- Avaliar a política de escalonamento orientada por QoS com base em outras métricas de QoS. Este trabalho considerou a disponibilidade como métrica de QoS de interesse, no entanto, outras métricas para o provisionamento de recursos também poderiam ter sido utilizadas. O isolamento de recursos e a segurança são exemplos de métricas que podem ser usadas para medir a QoS das requisições.

Bibliografia

[1] Amazon ec2 - instances pricing. https://aws.amazon.com/ec2/pricing/. Acessado: 28-11-2019.

[2] Erlang language. http://www.erlang.org/. Acessado: 10-07-2018.

[3] Google compute engine - preemptible instances. https://cloud.google.com/compute/docs/instances/preemptible. Acessado: 15-12-2019.

[4] Kubernetes - pods and controllers. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/pods-and-controllers. Acessado: 20-11-2019.

[5] Kubernetes - production-grade container orchestration. https://kubernetes.io/. Acessado: 10-07-2018.

[6] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of google cloud users: the mice and elephants phenomenon. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 272–277. IEEE, 2014.

[7] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad. Cloud computing pricing models: a survey. *International Journal of Grid and Distributed Computing*, 6(5):93–106, 2013.

[8] A. Barros, F. Brasileiro, G. Farias, F. Germano, M. Nóbrega, A. C. Ribeiro, I. Silva, and L. Teixeira. Using fogbow to federate private clouds. *Salao de Ferramentas do XXXIII SBRC*, 2015.

[9] L. Bellu and P. Liberati. "inequality analysis: The gini index". "Food and Agriculture Organization of the United Nations, FAO", 2005.

[10] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, volume 14, pages 285–300, 2014.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[12] M. Carvalho. *Gerência de Nuvens Computacionais Considerando Diferentes Classes de Serviço*. PhD thesis, Universidade Federal de Campina Grande, 03 2016.

[13] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term slos for reclaimed cloud computing resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 20:1–20:13, New York, NY, USA, 2014. ACM.

[14] M. Carvalho, G. Farias, D. Turull, F. Brasileiro, and R. Lopes. Method and resource manager for scheduling of instances in a data centre, 04 2017.

[15] M. Carvalho, D. Menascé, and F. Brasileiro. Prediction-based admission control for iaas clouds with multiple service classes. In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 82–90. IEEE, 2015.

[16] M. Carvalho, D. A. Menascé, and F. Brasileiro. Capacity planning for iaas cloud providers offering multiple service classes. *Future Generation Computer Systems*, 77:97 – 111, 2017.

[17] M. Cirinei and T. P. Baker. Edzl scheduling analysis. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 9–18. IEEE, 2007.

[18] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[19] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*, number CONF. USENIX Association, 2015.

[20] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[21] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[22] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[23] S. Dubey and S. Agrawal. Qos driven task scheduling in cloud computing. *Int. J. Comput. Appl. Technol. Res*, 2(5):595–600, 2013.

[24] G. Farias, F. Brasileiro, R. Lopes, M. Carvalho, F. Morais, and D. Turull. On the efficiency gains of using disaggregated hardware to build warehouse-scale clusters. In *Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on*, pages 239–246. IEEE, 2017.

[25] G. Farias, V. B. da Silva, F. Brasileiro, R. Lopes, and D. Turull. Availability-driven scheduling in kubernetes. Available at http://www.lsd.ufcg.edu.br/~giovanni/papers/Availability-driven-scheduling-in-Kubernetes.pdf, 2020. Submitted to IEEE CLOUD.

[26] G. Farias, R. Lopes, F. Brasileiro, F. Morais, M. Carvalho, J. Mafra, and D. Turull. Qos-driven scheduling in the cloud. Available at http://www.lsd.ufcg.edu.br/~giovanni/papers/QoS-driven-scheduling-in-the-cloud.pdf, 2020. Submitted to Future Generation Computer Systems.

[27] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer, 1997.

[28] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[29] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. Usenix, 2016.

[30] I. Goiri, F. Julia, R. Nou, J. L. Berral, J. Guitart, and J. Torres. Energy-aware scheduling in virtualized datacenters. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 58–67. IEEE, 2010.

[31] X. He, X. Sun, and G. Von Laszewski. Qos guided min-min heuristic for grid task scheduling. *Journal of Computer Science and Technology*, 18(4):442–451, 2003.

[32] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[33] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[34] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*, pages 485–497, 2015.

[35] X. Kong, C. Lin, Y. Jiang, W. Yan, and X. Chu. Efficient dynamic task scheduling in virtualized data centers with fuzzy prediction. *Journal of network and Computer Applications*, 34(4):1068–1077, 2011.

[36] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[37] R. V. Lopes and D. Menascé. A taxonomy of job scheduling on distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3412–3428, Dec 2016.

[38] P. Marshall, K. Keahey, and T. Freeman. Improving utilization of infrastructure clouds. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud*

and Grid Computing, CCGRID '11, pages 205–214, Washington, DC, USA, 2011. IEEE Computer Society.

[39] A. K.-L. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[40] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[41] W. Pan, J. Rowe, and G. Barlaoura. Records in the cloud (ric) user survey report. Technical report, 2013.

[42] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symp. on Cloud Computing, SOCC '12, San Jose, CA, USA*, page 7, 2012.

[43] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Posted at: https://github.com/google/cluster-data/blob/master/ClusterData2011$_2$.$mdtrace - data, November 2014$.

[44] R. Rosen. Linux containers and the future cloud. *Linux J*, 240(4):86–95, 2014.

[45] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[46] M. Shahrad and D. Wentzlaff. Availability knob: Flexible user-defined availability in the cloud. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 42–56. ACM, 2016.

[47] G. Silva, R. Lopes, F. Brasileiro, M. Carvalho, F. Morais, J. Mafra, and D. Turull. Escalonamento justo em infraestruturas de nuvem com múltiplas classes de serviço. In *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 636–649, Porto Alegre, RS, Brasil, 2019. SBC.

[48] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[49] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[51] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, New York, NY, USA, 2015. ACM.

[52] A. Weiss. Computing in the clouds. *networker*, 11(4):16–25, 2007.

[53] X. Wu, M. Deng, R. Zhang, B. Zeng, and S. Zhou. A task scheduling algorithm based on qos-driven in cloud computing. *Procedia Computer Science*, 17:1162–1169, 2013.

[54] J. Xu and C. Zhu. Optimal pricing and capacity planning of a new economy cloud computing service class. In *2015 International Conference on Cloud and Autonomic Computing*, pages 149–157. IEEE, 2015.

Apêndice A

Instruções para reprodução dos experimentos

Este apêndice descreve como os experimentos de simulação e medição executados ao longo deste trabalho podem ser reproduzidos. Portanto, as instruções de como executar o simulador e o protótipo implementado, os parâmetros de entrada para um teste de experimento, bem como os formatos dos arquivos utilizados são descritos a seguir.

A.1 Experimentos de Simulação

O simulador desenvolvido é capaz de emular as políticas de escalonamento baseada em prioridade e orientada por QoS (de acordo com as Seções 4.1.2 e 4.1.3). Este simulador[1] foi desenvolvido em Erlang e executa em uma ferramenta de simulação chamada Sim-Diasca, que é um mecanismo de simulação que visa aumentar a simultaneidade, paralelismo e distribuição da execução. Após o download do código do simulador, para executar um teste de simulação, é necessário entrar no diretório *cloudish* e executar o seguinte comando:

bash run_simulation.sh $SIM_DURATION $SCHEDULING_POLICY $RELAXED_-RANDOMIZATION $CHECK_CONSTRAINTS_FLAG $WORKLOAD_FILE $INFRAS-

[1] O simulador está disponível para download no repositório https://forge.ericsson.net/plugins/git/ufcg-er/cloudish?a=treehb=experiments-journal-paper. Apesar deste repositório ser privado por causa de acordo de confidencialidade com a Ericsson, é possível fazer a solicitação de acesso ao código especificando os propósitos. A solicitação pode ser realizada via o próprio repositório.

TRUCTURE_FILE 3 *$SAFTEY_MARGIN $PERIODICITY false $PREEMPTION_-OVERHEAD_FILE $MIGRATION_OVERHEAD_FILE* 1 *$LIMITING_PREEMPTION_-FLAG $EXTRA_OVERHEAD $QUEUE_PRUNING_FLAG $EQUIVALENCE_CLASS_-FLAG $CACHING_FLAG*

Segue definição de cada parâmetro do comando acima:

- *SIM_DURATION* representa a duração do teste de simulação em segundos.
- *SCHEDULING_POLICY* é a política de escalonamento a ser simulada no teste {ttv | priority}
- *RELAXED_RANDOMIZATION* representa o valor a ser considerado para a otimização *relaxed randomization*. Caso o valor 1 seja utilizado, isso significa que esta otimização não estará sendo utilizada na execução do teste.
- *CHECK_CONSTRAINTS_FLAG* representa a *flag* que indica se o escalonamento deve considerar as restrições de alocação e/ou afinidade da requisição enquanto estiver tomando decisões {*true* | *false*}.
- *WORKLOAD_FILE* é o arquivo de descrição da carga de trabalho. Este arquivo deve estar localizado no diretório *data*. O formato deste arquivo será descrito a seguir.
- *INFRASTRUCTURE_FILE* é o nome do arquivo que descreve a infraestrutura. Este arquivo também deve está localizado no diretório *data* e seu formato será descrito a seguir.
- *SAFTEY_MARGIN* representa a margem de segurança (em segundos) a ser considerada pelo escalonador orientado por QoS para todas as classes de serviço enquanto toma decisões.
- *PERIODICITY* indica o tempo máximo entre dois processamentos sequenciais da fila de espera.
- *PREEMPTION_OVERHEAD_FILE* é o nome do arquivo com um conjunto de valores que representam sobrecargas de preempções. Esses valores são considerados quando

o escalonador aloca uma instância no mesmo servidor que ela estava alocada previamente. Este arquivo contém um valor em cada linha e deve está localizado no diretório *data*.

- *MIGRATION_OVERHEAD_FILE* é o nome do arquivo com um conjunto de valores que representam sobrecargas de migrações. Esses valores são considerados quando o escalonador aloca uma instância em um servidor diferente do que ela estava alocada previamente. Este arquivo contém um valor em cada linha e deve está localizado no diretório *data*.

- *LIMITING_PREEMPTION_FLAG* representa a *flag* que indica se o mecanismo de limitação de premepção deve ser utilizado ou não no teste {*true* | *false*}.

- *EXTRA_OVERHEAD* indica o valor extra para ser incrementado ao valor padrão de $1 - SLO$ a ser considerado como o valor máximo aceitável para sobrecarga com preempção para uma requisição. Por exemplo, o valor 0 indica que o limite aceitável para a sobrecarga está configurado com $1 - SLO$.

- *QUEUE_PRUNING_FLAG* representa a *flag* que indica se a otimização *Pending Queue Pruning* deve ser considerada no teste {*true* | *false*}.

- *EQUIVALENCE_CLASS_FLAG* representa a *flag* que indica se a otimização *Equivalence Class* deve ser considerada no teste {*true* | *false*}.

- *CACHING_FLAG* representa a *flag* que indica se a otimização *Score Caching* deve ser considerada no teste {*true* | *false*}.

A.1.1 Dados de Entrada

Carga de Trabalho

Ao longo deste trabalho, as cargas de trabalho utilizadas foram passadas ao simulador através de um arquivo de descrição[2]. Um arquivo de descrição de carga de trabalho deve conter os

[2]As cargas utilizadas nos experimentos de simulação e medição ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/workloads.

seguintes dados:

1. *timestamp* (tempo de admissão da requisição)
2. identificador da requisição
3. identificador do job
4. duração da requisição
5. quantidade de CPU solicitada
6. quantidade de memória RAM solicitada
7. prioridade da requisição
8. classe de serviço da requisição
9. SLO de disponibilidade da classe de serviço solicitada
10. restrição de afinidade (o valor 1 indica que o *job* tem a restrição de anti afinidade, que significa que não mais que uma requisição do *job* pode ser alocada em um mesmo servidor)
11. uma lista de restrições de alocação

Infraestrutura

A infraestrutura também é informada ao simulador através de um arquivo de descrição[3]. Para cada servidor da infraestrutura deve existir os seguintes dados:

1. identificador do servidor
2. nome do servidor
3. capacidade de CPU do servidor
4. capacidade de memória RAM do servidor
5. uma lista de atributos no formato "chave=valor"

[3]As infraestruturas utilizadas nos experimentos de simulação ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/infrastructures.

Sobrecargas de alocação

Neste trabalho, os tempos de alocação utilizados nos experimentos de simulação foram medidos a partir de experimentos de medição. Portanto, inicialmente alguns testes de experimentos de medição foram executados com o intuito de medir o tempo de alocação de uma instância quando a instância já havia sido executada previamente em um servidor e também quando a instância nunca havia executado no servidor. Nesse sentido, quando o escalonador simulado decide alocar uma requisição em um servidor, um tempo de alocação é selecionado aleatoriamente do conjunto com tempos de alocação (caso a instância tenha executado anteriormente no servidor) ou do conjunto com tempos de migração (caso a instância não tenha executado no servidor anteriormente). Os tempos de alocação utilizados para preempção e migração estão disponibilizados no repositório[4].

A.1.2 Resultados

Os resultados dos experimentos de simulação obtidos ao longo deste trabalho também estão disponibilizados no repositório[5]. Eles estão divididos em duas partes por nível de contenção de recursos. Cada parte contém resultados para as duas políticas de escalonamento (baseada em prioridade e orientada por QoS) de cinco das amostras analisadas. A parte 01 contém os resultados das cargas 1 até 5, e a parte 02 contém os resultados das cargas 6 até 10. Uma vez que o arquivo seja descompactado, cada arquivo de resultado de simulação tem as seguintes colunas:

1. identificador da requisição
2. tempo de execução (a duração da requisição, *i.e.* a quantidade de tempo que a requisição esteve em execução até ser terminada)
3. classe de serviço
4. disponibilidade final da requisição

[4]Os tempos utilizados para representar as sobrecargas de alocação estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/overheads

[5]Os resultados dos experimentos de simulação estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/results/simulation.

5. tempo de admissão
6. política de escalonamento
7. quantidade de CPU solicitada
8. carga de trabalho

A.2 Experimentos de Medição

Com o intuito de automatizar a execução dos experimentos de medição com o protótipo do Kubernetes, uma outa aplicação foi desenvolvida para submeter requisições para o Kubernetes. Esta aplicação, denominada de *broker*, submete as requisições com base em um arquivo de descrição da carga de trabalho. Embora o conteúdo dos arquivos seja essencialmente o mesmo para os experimentos de simulação e medição, há algumas diferenças no formato dos arquivos. Um arquivo de carga de trabalho submetida para o *broker* contém os seguintes dados:

1. tempo decorrido desde o início do experimento (tempo de admissão da requisição)
2. identificador da requisição
3. identificador do *job*
4. usuário
5. restrição de afinidade
6. prioridade da requisição
7. duração da requisição
8. quantidade de CPU solicitada
9. quantidade da memória RAM solicitada
10. classe de serviço
11. SLO de disponibilidade da classe de serviço solicitada

O *broker*, bem como as instruções de como executá-lo, está disponível no repositório[6]. Além disso, o mesmo repositório tem instruções sobre como implantar um *cluster* Kubernetes utilizando o protótipo do escalonador orientado por QoS desenvolvido.

A.2.1 Dados de Entrada

As cargas de trabalho utilizadas nos experimentos de medição estão disponíveis no repositório[7]. No caso da infraestrutura, ela é definida pela adição/remoção de servidores no *cluster* do Kubernetes. No repositório[8] é possível encontrar instruções sobre como implantar um *cluster* Kubernetes utilizando o escalonador orientado por QoS desenvolvido neste trabalho.

A.2.2 Resultados

Após a execução de um experimento de medição, os resultados desta execução estarão armazenados no diretório */data* dentro do diretório *broker/* em formato CSV. Cada experimento gera quatro arquivos como resultado com informações sobre alocação, tempo de espera, tempo de execução e tempo de alocação das instâncias e controladores. Os formatos dos arquivos são explicados a seguir.

O arquivo *waiting.dat* possui informações sobre o momento que as instâncias foram para a fila de espera. Este arquivo contém os seguintes dados:

- *timestamp*, que indica o tempo que ocorreu o evento de criação do pod
- nome do pod, *i.e.* o nome da instância que foi criada
- nome do controlador, que representa o nome do controlador responsável pelo pod em questão

O arquivo *starting.dat* possui informações sobre a quantidade de tempo que os pods levaram para ser alocados após a decisão do escalonador, *i.e.* o tempo de alocação da instância.

[6] A aplicação *broker* está disponível para download em https://github.com/giovannifs/cloudish-kubernetes-experiment/tree/support-multiple-controllers.

[7] As cargas utilizadas nos experimentos de medição ao longo deste trabalho estão disponíveis para download em https://github.com/giovannifs/qos-driven-scheduling-experiments/tree/phd-thesis/workloads.

[8] Instruções sobre a implantação de um cluster kubernetes com o escalonador orientado por QoS estão disponíveis em https://github.com/giovannifs/cloudish-kubernetes-experiment/tree/support-multiple-controllers.

O arquivo contém os seguintes dados:

- *timestamp*, que indica o tempo que ocorreu o evento de alocação
- nome do pod, *i.e.* o nome da instância que foi alocada
- nome do controlador, que representa o nome do controlador responsável pelo pod em questão
- nome do servidor, que indica o servidor onde a instância foi alocada
- tempo de alocação, que indica a quantidade de tempo que a instância levou para iniciar a execução no servidor após a decisão do escalonador

Já o arquivo *allocation.dat* possui informações sobre o momento que os pods foram alocados. O arquivo contém os seguintes dados:

- *timestamp*, que indica o tempo que ocorreu o evento de alocação
- nome do pod, *i.e.* o nome da instância que foi alocada
- nome do controlador, que representa o nome do controlador responsável pelo pod em questão
- nome do servidor, que indica o servidor onde a instância foi alocada
- tempo de espera, que indica a quantidade de tempo que a instância esteve esperando até ser alocada

Por fim, o arquivo *termination.dat* possui informações sobre o momento que os pods foram terminados, seja por fim da execução ou por preempção. Este arquivo contém os seguintes dados:

- *timestamp*, que indica o tempo que ocorreu o evento de alocação
- nome do pod, *i.e.* o nome da instância que foi alocada
- nome do controlador, que representa o nome do controlador responsável pelo pod em questão
- tempo de execução, que indica a quantidade de tempo que a instância esteve executando desde que foi alocada