Análise Dinâmica do Comportamento de Filas de Mensagens para o Aumento do Paralelismo de Consumo

Eduardo Henrique Teixeira[1], **Aletéia Patrícia Favacho de Araújo**[1]

[1] Departamento de Ciência da Computação – CIC
Universidade de Brasília (UnB) – Brasília, DF – Brasil

edu.henr@gmail.com, aleteia@cic.unb.br

Resumo. *A elasticidade em computação consiste em dimensionar adequadamente os recursos necessários para processar uma aplicação distribuída. Para isso, são necessários mecanismos para evitar o fenômeno do limiar de detecção de elasticidade para cima ou para baixo. Este artigo propõe um middleware para analisar dinamicamente os fluxos de filas de mensagens, e um mecanismo para aumentar o paralelismo de consumo baseado no comportamento da vazão. Dessa forma, é apresentada a arquitetura do middleware IOD (Increase On Demand) com suporte ao aumento e a diminuição de threads, para conter o crescimento de filas de mensagens, utilizando a técnica de heurísticas baseada em limites por um determinado tempo, e o agrupamento de mensagens em subfilas de acordo com um critério de classificação.*

1. Introdução

Elasticidade é a característica de um ambiente que define o grau no qual um sistema é capaz de adaptar-se dinamicamente às mudanças de carga de trabalho, por meio do provisionamento e da liberação de recursos de forma automática [Herbst et al. 2013]. Assim, computação elástica é o provisionamento dinâmico de recursos [Perez et al. 2009].

A computação elástica traz enormes vantagens para os provedores de aplicações, incluindo economia de custos e prevenção de super e sub-provisionamento de recursos de TI. Isso ocorre por meio do monitoramento da demanda e da aquisição dos recursos requeridos pelas aplicações para alcançar um alto nível de qualidade [Leitner et al. 2012].

O objetivo da elasticidade é que a quantidade de recursos alocada para um serviço seja a que ele realmente necessita. Dessa forma, é possível, por exemplo, reduzir o número de servidores necessários para processar as filas de mensagens de um sistema distribuído e, consequentemente, economizar recursos computacionais. Também é possível determinar a carga atual de processamento requerida em um *cluster*, de acordo com a demanda exigida, que pode ser medida em função da vazão de saída em uma fila de mensagens, e que pode ser limitada pelo consumo de CPU, para evitar a saturação dos servidores. Para este fim, em uma plataforma distribuída, é necessário um *middleware*, que seja capaz de implementar, de maneira transparente e dinâmica, o conceito de elasticidade, a fim de garantir que o processamento possa se adaptar ao crescimento das filas de mensagens, evitando que as mensagens se acumulem antes de serem processadas.

Por outro lado, em plataformas distribuídas, além da elasticidade, é fundamental garantir tolerância a falhas, pois, dessa forma, um ou mais processos podem falhar sem prejudicar o restante do sistema [Tanenbaum and Steen 2008]. Falhas são frequentes, e

podem ocorrer devido a erros de *hardware* ou *software* [Smith 1986]. As falhas de *hardware* resultam, geralmente, da degradação física de componentes, enquanto que as falhas de *software* ocorrem devido a erros no projeto ou na implementação, e são também conhecidas como *bugs* [Smith 1986]. Nesse contexto, a tolerância a falhas de *software* é a garantia do comportamento correto da aplicação, independente do número e do tipo de falhas que ocorram [Coulouris et al. 2009].

Assim sendo, este artigo propõe o *middleware* IOD (*Increase On Demand*) que garante elasticidade e tolerância a falhas de *software* em arquiteturas de sistemas distribuídos baseadas em filas de mensagens, como são os *clusters* de alto desempenho e de alta disponibilidade. O *middleware* IOD analisa dinamicamente o comportamento da vazão das filas para determinar a necessidade de aumentar ou de diminuir o número de *threads* responsáveis pelo tratamento das mensagens. O objetivo do *middleware* proposto é evitar o enfileiramento das mensagens a serem processadas. O IOD também analisa o comportamento do consumo de CPU para determinar os limites de escalabilidade dos servidores para evitar a saturação do *cluster*. Além disso, ele utiliza mecanismos para recuperação rápida em caso de falhas, garantindo que os serviços disponibilizados pelo *cluster* sejam disponibilizados rapidamente.

Dessa forma, o foco do *middleware* IOD é utilizar a computação elástica para que os núcleos das unidades de processamento, disponíveis no *cluster*, sejam utilizados mais eficientemente. Para isso, no *middleware* proposto, a demanda necessária de *threads* para tratar as filas de mensagens é calculada, dinamicamente, de acordo com o comportamento da vazão média de saída e do consumo médio de CPU de cada nó computacional. Dessa forma, as aplicações distribuídas que utilizam filas de mensagens como mecanismo de IPC (*Inter Process Communication*) [Gray 2003], podem se beneficiar das técnicas projetadas no *middleware* IOD, proposto neste artigo. Isso é possível porque o dimensionamento da utilização dos recursos disponíveis no *cluster* é realizada em cada nó, evitando períodos de ociosidade de CPU.

Para apresentar o *middleware* IOD, o restante deste artigo está dividido em mais 4 seções. A Seção 2 contém uma visão geral sobre trabalhos relacionados à elasticidade e à tolerância a falhas. A Seção 3 descreve a arquitetura do *middleware* proposto IOD. Em seguida, a Seção 4 apresenta a avaliação e a análise dos testes realizados com o *middleware* IOD. Para finalizar, a Seção 5 apresenta algumas conclusões e descreve os próximos passos deste trabalho.

2. Trabalhos Relacionados

Para alcançar a elasticidade de forma transparente e automática, as abordagens citadas em [Tran et al. 2011][Ma et al. 2010] são adequadas quando a integridade sequencial no processamento de mensagens das filas não é um requisito funcional. Assim, as mensagens nas filas podem ser redistribuídas de acordo com a vazão requerida, sem a preocupação com a ordem em que elas são processadas. Entretanto, isso pode ser uma desvantagem quando há a necessidade de manter a integridade sequencial baseada na ordem de entrega para os consumidores das filas. Nesse caso, os eventos gerados pelos consumidores precisam respeitar a restrição de tempo na qual as mensagens foram geradas.

As atribuições feitas pelos *publishers* aos *subscribers* podem gerar assimetrias de distribuição dos dados [Tran et al. 2011][Li et al. 2011][Fang et al. 2011]. Essas assi-

metrias também são comuns em sistemas com múltiplas filas de mensagens e múltiplos consumidores [Tran et al. 2011][Li et al. 2011]. Identificar os melhores candidatos para o consumo das filas é necessário para reduzir a latência e evitar a saturação de consumidores sobrecarregados. Assim, é possível determinar esses candidatos identificando a maior vazão média [Tran et al. 2011][Li et al. 2011]. O menor consumo médio de CPU também pode ser utilizado como métrica de identificação dos melhores candidatos para o consumo de novos itens [Li et al. 2011].

Em sistemas distribuídos com características CPU-*bound*, é possível explorar a transparência de elasticidade para a aplicação [Tran et al. 2011][Ma et al. 2010][Imai et al. 2012]. Segundo [Li et al. 2011][Imai et al. 2012][Sugiki and Kato 2011], essa transparência pode ser alcançada utilizando a métrica de carga exigida em função do consumo de CPU. Entretanto, se o número de mensagens permanecer próximo do limite de detecção de elasticidade, problemas de início e término rápido de *threads* podem ocorrer. Para sanar esse problema e manter a vazão em níveis aceitáveis, [Tran et al. 2011][Li et al. 2011][Imai et al. 2012] propõem o uso de heurísticas com abordagem baseada em limites, por um determinado tempo. Dessa forma, as *threads* criadas são mantidas por mais tempo, o que minimiza o custo de reinício e diminui a latência do consumo de mensagens. Essa abordagem é interessante para sanar o problema do limiar de detecção de elasticidade para cima ou para baixo.

As abordagens [Guo et al. 2012][Imai et al. 2012][Leitner et al. 2012] [Ma et al. 2010][Sugiki and Kato 2011][Tran et al. 2011] citam que a elasticidade deve ser alcançada de forma transparente e automática. Também foi abordado o uso do padrão de projeto *publisher/subscriber* [Fang et al. 2011][Li et al. 2011][Tran et al. 2011] para alcançar a elasticidade e o desacoplamento de produtores e consumidores. Nas abordagens citadas em [Imai et al. 2012][Li et al. 2011][Sugiki and Kato 2011][Leitner et al. 2012] a elasticidade é alcançada de acordo com a demanda de carga exigida, que é medida pelo consumo de CPU. No entanto, em todas as abordagens estudadas não há uma preocupação com a integridade sequencial com que as mensagens assíncronas são processadas, nem com as assimetrias de distribuição dos dados para alcançar o paralelismo de consumo.

Muitas das abordagens de tolerância a falhas utilizam a replicação para que outro conjunto de processos assuma o processamento em caso de queda ou falha [Abbes et al. 2010][Bicer et al. 2010][He et al. 2012][Martins et al. 2010]. No entanto, técnicas como a recuperação rápida de falhas [Bicer et al. 2010][Castro et al. 2012][He et al. 2012][Wang et al. 2009] são interessantes, pois evitam os custos de memória adicional e de comunicação entre os processos replicados para a troca das informações de estados. [Castro et al. 2012][Martins et al. 2010] propõem o uso do suporte à tolerância a falhas de forma transparente no *middleware*. Essa abordagem é atrativa, pois abstrai da aplicação usuária os detalhes de detecção e de recuperação das características implementadas. Para alcançar essa transparência [Bicer et al. 2010][Martins et al. 2010] utilizam a detecção da queda ou falha por meio do monitoramento da conexão entre os processos.

3. *Middleware* Proposto

Para criar ambientes elásticos eficientes, os serviços existentes devem ser estendidos com funcionalidades de computação elástica e com políticas de provisionamento de recursos sob demanda [Marshall et al. 2012]. Dessa forma, o *middleware* IOD propõe o suporte à elasticidade por meio da adaptação dinâmica do número de *threads* para tratamento das mensagens baseado na análise da vazão de entrada e de saída das filas e do consumo de CPU do servidor. O suporte a tolerância a falhas dos processos, no *middleware* IOD, é realizado por meio da detecção da queda e da recuperação com reinício rápido. Assim, as mensagens das filas continuam a ser processadas a partir do ponto em que ocorreu a falha, sem a necessidade de uso da replicação de processos. Essas características serão apresentadas nas próximas seções.

3.1. Arquitetura da Estrutura de Dados

Para alcançar o paralelismo de consumo, é necessário, primeiramente, separar as mensagens de uma fila em grupos distintos, de acordo com um critério de classificação. Assim sendo, cada fila dá origem a várias subfilas, cada uma indexada por uma chave que identifica um grupo. Isto deve ser feito para aumentar o número de *worker threads* que serão responsáveis por processar as mensagens, distribuindo os grupos entre as várias *worker threads* iniciadas e, assim, aumentando a vazão por meio do paralelismo de consumo.

Um sistema com múltiplas filas de consumo pode gerar assimetrias de distribuição [Fang et al. 2011][Li et al. 2011][Tran et al. 2011], o que leva a um desbalanceamento do sistema ao consumir as mensagens dessas subfilas. Dessa forma, é realizada no *middleware* IOD a ordenação dos grupos, no momento em que são criadas ou removidas *worker threads*, iniciando a partir da *thread* que tem o maior número de mensagens até a que tem o menor número. Em seguida, é realizada a distribuição desses grupos entre as *worker threads* utilizando o algoritmo *round robin*. Além disso, em cada rodada de consumo, é realizado o processamento de uma mensagem de cada grupo por meio do FQ (*Fair Queueing*), que é um algoritmo que permite múltiplas filas de mensagens compartilharem a mesma capacidade de processamento, garantindo justiça no consumo e evitando inanição por conta de fluxos pesados. Além disso, também foi utilizado o algoritmo *First Come First Served* [Tanenbaum and Woodhull 2007], para preservar a integridade sequencial no consumo das mensagens que pertencem ao mesmo grupo.

Para alcançar a elasticidade no *middleware* IOD, fez-se necessário realizar a análise da vazão de mensagens na filas, a limitação de criação de *worker threads* por CPU e por desvio padrão da vazão média de saída por grupo, bem como a recuperação de falhas com elasticidade, as quais serão descritas nas próximas subseções.

3.2. Análise da Vazão Média de Entrada e Saída

Durante a rajada de mensagens recebidas, caso uma aplicação distribuída não tenha uma vazão de consumo compatível com a de geração, ocorrem acúmulos nas filas, ou seja, enfileiramentos que geram atrasos no processamento e, consequentemente, redução na qualidade dos serviços prestados. Dessa forma, para obter o número de *worker threads* que serão necessárias para conter esse enfileiramento na presença de rajadas de mensagens, é necessário calcular a vazão média de entrada e saída, e a média da relação entre as mensagens de entrada e saída. Esses cálculos são realizados a partir do momento da

detecção de crescimento da fila, aqui denominada *Growth Detection* (*GD*), até o *Critical Point* (*CP*), que é o momento onde são criadas novas *worker threads*. O *GD* ocorre quando a relação entre as mensagens de entrada e de saída torna-se maior do que um, gerando enfileiramento. O *GD* é detectado pela Equação 1,

$$\textit{GrowthDetection} = \left(\frac{\textit{Input}}{\textit{Output}} > 1 \right) \tag{1}$$

onde, *Input* é o número de mensagens de entrada e *Output* é o número de mensagens de saída.

Assim sendo, novas *worker threads* são criadas no momento em que o acúmulo gerado na fila não pode ser tratado antes do fim do tempo máximo definido para tratamento das mensagens. Esse momento ocorre no *CP* e a ação de aumentar *worker threads* é denominada *Scale Up* (*SU*). O *CP* é encontrado quando o número de mensagens na fila é maior do que o valor dado pela Equação 2,

$$\textit{MaxQueueSize} = (avgTOUT \times (schTIME - (hNOW - hSTART))) \tag{2}$$

onde, a variável *avgTOUT* é a vazão média de saída; *schTIME* é o tempo máximo para o tratamento de mensagens na fila; *hNOW* é o tempo atual; *hSTART* é o tempo de início, onde o valor dado pela Equação 1 se torna verdadeiro.

O *Scale Down* (*SD*) ocorre no *Exit Point*, que é o momento onde as *worker threads* são removidas. Ele é obtido pela Equação 3,

$$QueueSize < \left(\frac{avgTIN}{2} \right) \tag{3}$$

onde, *QueueSize* é o tamanho atual da fila, e *avgTIN* é a vazão média de entrada desde o *CP*. A divisão de *avgTIN* pela metade destina-se a evitar o problema da detecção do limiar de elasticidade para baixo [Imai et al. 2012], o que faz com que a fila de mensagens tenha uma nova tendência de crescimento. Ele foi inspirado pelo decremento multiplicativo utilizado em algoritmos de congestionamento do TCP (*Transmission Control Protocol*), em que, quando há um *timeout*, o limiar é definido como a metade da janela de congestionamento atual [Tanenbaum and Wetherall 2010].

Em adição às *Contention Threads* (*CT*), que são as *worker threads* de contenção de rajadas, um outro conjunto de *worker threads* é necessário, o qual é chamado de *Zero Threads* (*ZT*), para consumir as mensagens que se acumularam na fila desde o *GD* até o *CP*. Dessa forma, são criadas várias *worker threads*, do tipo *ZT* ou *CT*, sendo que cada uma delas está associada a várias subfilas de mensagens separadas por grupos, para que ocorra o aumento da vazão por meio do paralelismo de consumo.

3.3. Análise do Consumo de CPU

Conhecer o consumo médio de CPU por *worker thread* é útil para determinar limites para o *SU* (*Scale Up*). Assim, ao detectar rajadas de mensagens de entrada, pode ser iniciado o processo para a medição de consumo médio de CPU, por *worker thread* e por nó do *cluster*, para determinar se a vazão necessária para conter o crescimento, e zerar a fila pode ser atendido, sem a saturação de uso de CPU do servidor.

3.4. Análise do Desvio Padrão da Vazão Média de Saída por Grupo

As rajadas de mensagens de entrada podem ser originadas a partir de vários grupos. Para atender as rajadas de *n* grupos, o *middleware* IOD decide criar várias *worker threads*, de acordo a vazão média de saída medida durante o crescimento da fila, como descrito na análise da vazão média de entrada e saída, apresentada na Seção 3.2.

No entanto, quando as rajadas são originadas a partir de um pequeno grupo de mensagens, criar várias *worker threads* pode significar um desperdício de recursos e, dependendo da vazão de consumo, pode não ser suficiente para conter o crescimento da fila e ainda gerar *SU* (*Scale Up*) recursivo, sem resolver o problema de crescimento da fila. Para resolver esse problema, antes do *SU* (*Scale Up*) e durante a rajada de mensagens, é realizada a medição da vazão média de saída por grupo, que será utilizada no cálculo do desvio padrão no momento de realizar o *SU*. Uma variável aleatória em uma distribuição normal tem 95% de chance de estar a menos de dois desvios-padrão de sua média [Downing and Clark 2011]. Assim, como a vazão média de saída foi normalizada, em no máximo uma mensagem de cada grupo por rodada por meio do algoritmo FQ, após *n* rodadas, quando a média da vazão de saída se distancia muito do desvio padrão, ou seja, mais de duas vezes, é possível detectar rajadas vindas de grupos específicos. Dessa forma, é criada uma quantidade de *worker threads* limitada pelo número de grupos, cuja vazão média de saída seja maior do que duas vezes o desvio padrão identificado durante a rajada de mensagens. Essa abordagem tem o objetivo de criar *worker threads* para o tratamento específico dos grupos de mensagens nos quais as rajadas foram identificadas.

3.5. Tolerância a Falhas

No *middleware* IOD foi utilizado o suporte às características de tolerância a falhas de *software* por omissão do processo, não sendo consideradas as falhas de *hardware*. Assim, as mensagens das filas continuam a ser processadas a partir do ponto em que ocorreu a falha, sem a necessidade de uso da replicação de processos.

Para suportar os mecanismos de recuperação rápida de falhas [Bicer et al. 2010] [Castro et al. 2012] [He et al. 2012] [Wang et al. 2009], se um processo falha e o número de mensagens na fila é maior do que o *CP* (*Critical Point*), mecanismos de elasticidade no *middleware* são necessários para realizar o *SU* (*Scale Up*). Assim, quando os processos são reiniciados após uma falha, se as rajadas de entrada continuarem, o *SU* ocorre rapidamente por meio da detecção do *CP* maior do que o *QueueSize*. No entanto, nesse momento é necessário um outro mecanismo para proporcionar elasticidade, caso as rajadas tenham terminado após a queda e antes do reinício do processo. Nesta situação, se o número de mensagens na fila estiver acima de *avgTOUT* x *schTIME*, as medidas de vazão, antes de decidir realizar o *SU*, são realizadas por somente 10% do *schTIME*. Isso é necessário para realizar *SU* de uma maneira mais rápida, como é o caso quando as rajadas terminam após a queda, e antes do reinício rápido do processo. O mesmo cenário ocorre quando as rajadas de mensagens não param, mesmo após o reinício rápido do processo.

Assim sendo, nota-se que a implementação eficiente de tolerância a falhas em um ambiente distribuído necessita ser realizada em conjunto com um mecanismo de elasticidade, exatamente como proposto pelo *middleware* IOD.

4. Resultados

Para os testes do *middleware* IOD foi criado um simulador, onde foram codificadas duas *threads*: uma produtora que escreve pacotes na fila de mensagens, e outra consumidora que lê a partir da fila e entrega a mensagem para o *middleware* IOD. Para esses testes, foi utilizado um tempo máximo de tratamento de mensagens de 80 segundos, e a *thread* produtora foi programada para gerar mensagens a uma taxa cinco vezes maior do que a *thread* consumidora. O objetivo dessa configuração é gerar o enfileiramento de mensagens e demonstrar o funcionamento da elasticidade proposta pelo *middleware* IOD.

Assim sendo, as subseções a seguir descrevem os testes realizados para verificar o aumento e a diminuição dinâmica das *worker threads* em função da vazão de entrada e saída; o aumento das *worker threads* com limite em função do consumo de CPU; e a tolerância a falhas com elasticidade.

4.1. Criação e Remoção Dinâmica de *Threads*

Os testes apresentados nesta seção têm o objetivo de verificar o funcionamento da criação e da remoção de *worker threads*, dinamicamente, em função da vazão média de entrada e de saída de mensagens.

Para isso, foram feitos testes com rajadas de geração de pacotes a cada 10 milisegundos e consumo a cada 50 milisegundos, ou seja, geração de mensagens a uma taxa cinco vezes maior do que o consumo. O objetivo foi testar o comportamento do *middleware* IOD diante da necessidade de criar dinamicamente *threads*. Assim sendo, os experimentos mostraram que ocorreu o aumento adequado do número de *worker threads* para incrementar a vazão de consumo, como mostra a Figura 1, na qual o eixo x corresponde ao tempo decorrido e o eixo y corresponde ao número de mensagens na fila (*QueueSize*), ao valor da Equação 2 (*MaxQueueSize*), a vazão média de entrada (*avgTIN*) ou a vazão média de saída (*avgTOUT*), dependendo da variável analisada. No cenário apresentado na Figura 1, o *CP* foi detectado com 1.130 mensagens na fila, aos 14 segundos após o início da rajada de mensagens, como mostrado na Tabela 1. Aos 48 segundos após o *SU*, o algoritmo decide remover cinco *threads* de forma que a vazão média de saída fique compatível com a vazão média de entrada. Assim, o *middleware* IOD se mostrou satisfatório quanto à criação e remoção de *worker threads* em função da vazão, pois ele foi capaz de criar 12 *worker threads*, e encerrar com o cenário de gargalo da fila em um período de 23 segundos, ou seja, dentro do limite máximo de tratamento de mensagens definido em 80 segundos. O número de mensagens ficou em um nível controlado, pois a estabilização do algoritmo ocorreu com 33 mensagens de entrada e saída, ou seja, um valor pequeno e controlável.

Tabela 1. Novos Tempos do Algoritmo de Elasticidade.

Momento	Duração (segundos)	Entrada	Saída	*Threads*	Enfileiramento
Início	14	99	19	1	0
Scale Up	9	99	197	12	1.130
Scale Down	57	99	99	7	33

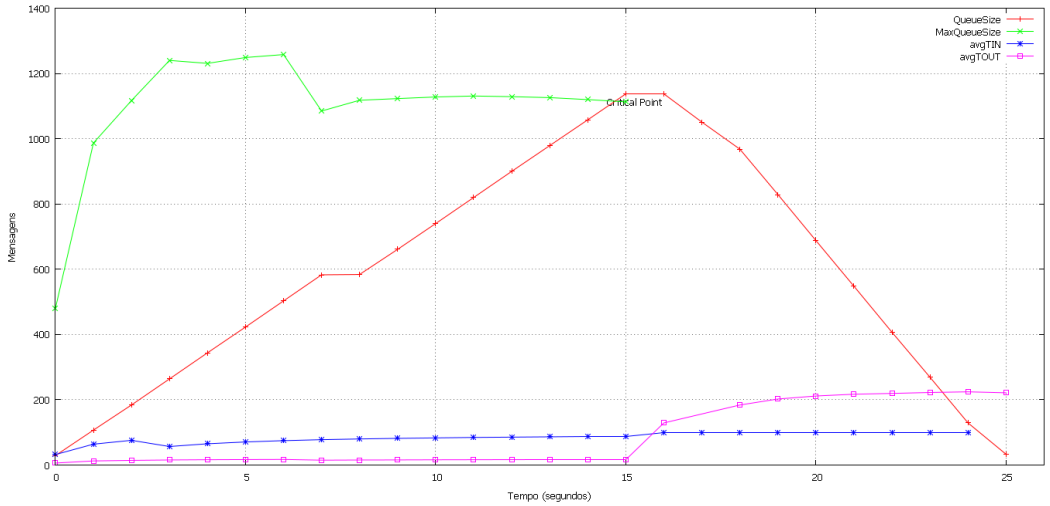

Figura 1. Fila de Msgs com a Criação e a Remoção de *Worker Threads*.

4.2. Criação de *Threads* Limitada pelo Consumo Médio de CPU

O objetivo deste teste é validar se o algoritmo de elasticidade evita a saturação do consumo de CPU dos servidores, por meio da verificação do comportamento do aumento de *worker threads* limitado pelo consumo médio de CPU.

Para isso, foi utilizado no simulador a técnica de *busywait* nas *worker threads* de consumo, para simular um alto uso de CPU. Os testes foram realizados com o produtor a 10 milisegundos, e com o consumidor a 50 milisegundos, de forma que ocorra o enfileiramento das mensagens. Os resultados da fila de mensagens são mostrados na Figura 2, onde o eixo *x* corresponde ao tempo decorrido e o eixo *y* corresponde ao número de mensagens na fila (*QueueSize*), ao valor da Equação 2 (*MaxQueueSize*), a vazão média de entrada (*avgTIN*) ou a vazão média de saída (*avgTOUT*), dependendo da variável analisada.

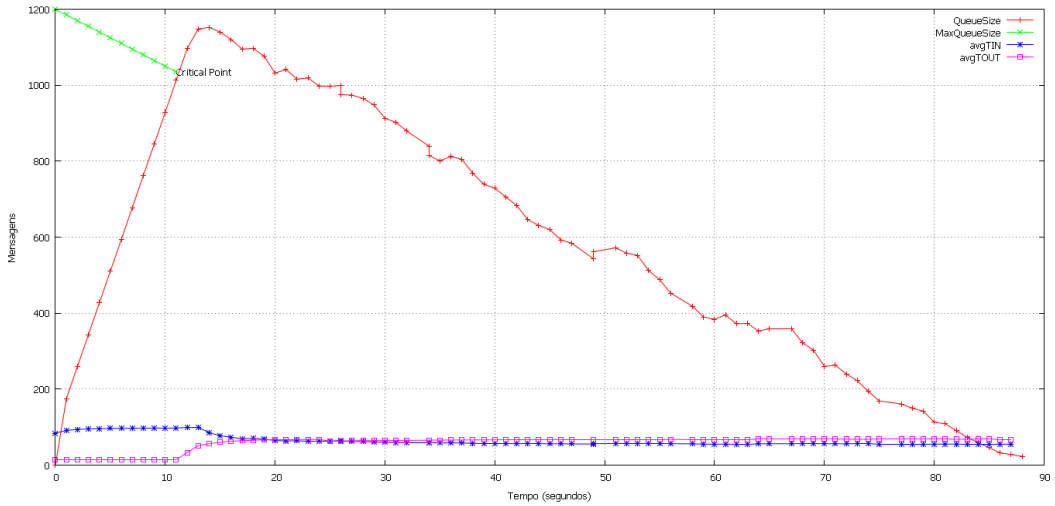

Figura 2. Comportamento da Fila de Mensagens com Limites de CPU.

Como pode ser observado na Figura 2, a diminuição do número de mensagens ocorre de maneira mais lenta, pois o número de *threads* é limitado em função do uso de CPU médio das *worker threads*, de forma que não ultrapasse 100% do servidor. Além disso, a diminuição não é linear como mostrado na Figura 1, pois não foram criadas todas as *worker threads* necessárias para que a diminuição das mensagens nas filas ocorresse no tempo requerido.

Tabela 2. Comportamento do *Middleware* IOD com Limitação de CPU.

Momento	*Threads* Necessárias	*Threads* Criadas	*Threads* Removidas
Scale Up	8	6	N/A
Scale Down	N/A	N/A	0

Tabela 3. Comportamento da CPU no Algoritmo de Elasticidade com Limites.

Momento	Média por *Worker Thread*	Total Necessária	Total Residual do Servidor
Scale Up	11.30%	90.47%	73.89%
ScaleDown	N/A	N/A	N/A

De acordo com a Tabela 2, o *middleware* deveria ter criado oito *worker threads*, baseado na vazão média de entrada. No entanto, como mostrado na Tabela 3, o consumo médio de CPU por *worker thread* foi de 11.30%, e ultrapassaria o limite de CPU residual do servidor, que era de 73.89%. Assim, o algoritmo de elasticidade criou apenas seis *worker threads* (2 a menos), para evitar a saturação do uso de CPU da máquina. No momento de escalar para baixo, a vazão média de saída ficou em 68 mensagens por segundo e, portanto, abaixo da vazão média de entrada que era de 85.33 mensagens por segundo. Dessa forma, o algoritmo de elasticidade não detectou a necessidade de remover *worker threads*.

Considerar a saturação do ambiente é importante para que a aplicação distribuída seja capaz de identificar dinamicamente os limites de carga que é capaz de processar de acordo com a infraestrutura para a qual foi dimensionada. Dessa forma, o sistema ao identifcar que necessita de mais processamento e não aumentou o número *worker threads* devido a uma limitação em sua capacidade de processamento, pode gerar alarmes para seja realizado o correto dimensionamento de acordo com a carga exigida.

4.3. Ativação da Elasticidade após o Reinício Rápido do Processo

O objetivo deste teste é validar se, após a queda de uma processo durante o tratamento de mensagens, a elasticidade é iniciada sem a presença de rajadas de pacotes, e após o reinício rápido do processo. Dessa forma, foi realizada a simulação da queda do processo e o seu reinício com a fila contendo mensagens com um número acima do *CP*.

Na Figura 3 o eixo *x* corresponde ao tempo decorrido e o eixo *y* corresponde ao número de mensagens na fila (*QueueSize*), ao valor da Equação 2 (*MaxQueueSize*), a vazão média de entrada (*avgTIN*) ou a vazão média de saída (*avgTOUT*), dependendo da variável analisada. Assim, alguns segundos após o reinício do processo, a inclinação do

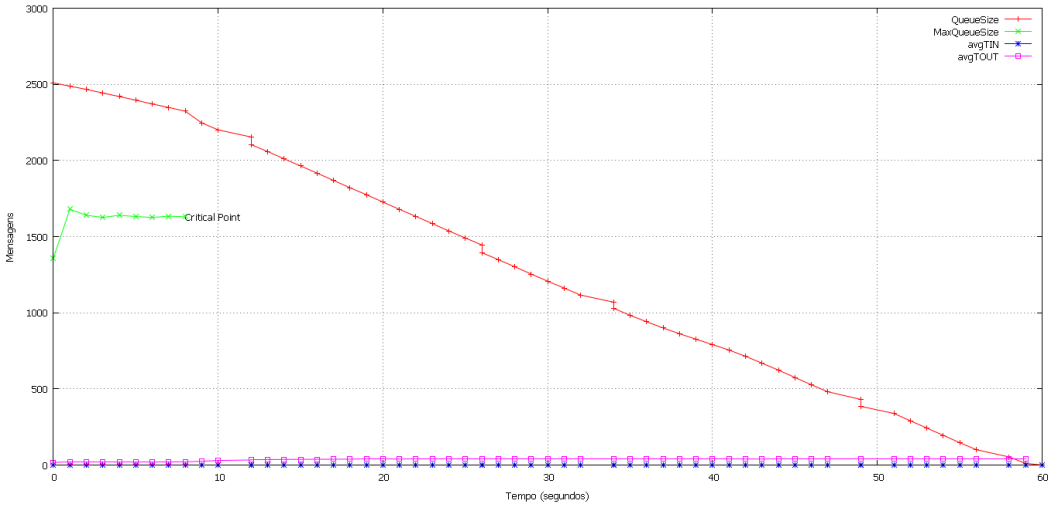

Figura 3. Comportamento da Fila de Mensagens após o Reinício Rápido.

consumo de mensagens aumenta, indicando o incremento do número de *worker threads* para consumir as mensagens antes do término do tempo máximo de tratamento de mensagens. Este comportamento demonstra que, mesmo com falha no processo, o *middleware* consegue ser elástico diante de uma demanda gerada anteriormente, e após o reinício rápido do processo.

Assim sendo, diante dos testes realizados, foi possível notar que o *middleware* IOD proposto é capaz de implementar elasticidade em uma plataforma distribuída. Além disso, o uso de filas de mensagens por processo, mostrou-se um método tolerante a falhas, pois a queda de um processo não impacta em funcionalidades de outros processos [Wang et al. 2009][Castro et al. 2012]. Assim, escrever em filas diferentes com funcionalidades distintas, paraleliza o tratamento, escala o sistema e torna o processamento mais rápido.

Outra importante contribuição foi utilizar o paralelismo de consumo em filas de mensagens, por meio do agrupamento de mensagens correlacionadas em subfilas, pois ele proporcionou a diminuição dos custos com a infraestrutura necessária para manipular as filas de mensagens. Dessa forma, criar mecanismos que proveem o aumento da vazão de consumo, melhoram significativamente a qualidade dos serviços prestados, uma vez que a mesma quantidade de massa de dados pode ser processada em menos tempo e por uma quantidade menor de servidores.

5. Conclusões e Trabalhos Futuros

O custo cada vez mais alto de equipamentos de infraestrutura mostra que é essencial utilizar os recursos disponíveis de maneira otimizada e eficiente. Dessa forma, o dimensionamento dinâmico de recursos mais próximo da quantidade de processamento necessária para prover serviços, mostra-se cada vez mais importante, pois significa economia de custos sem perda da qualidade dos serviços prestados. Assim, o *middleware* IOD mostrou ser possível alcançar os requisitos de elasticidade, adaptando-se as condições da carga de processamento, aumentando ou diminuindo o número de *worker threads* de acordo com a demanda requerida. Além disso, o *middleware* IOD mostrou como otimizar o uso de

recursos de TI por meio do paralelismo de tarefas, usando múltiplas *worker threads* para processar as filas de mensagens.

Uma contribuição alcançada neste artigo foi a criação das Equações 2 e 3 para determinar os pontos críticos de entrada e saída para o tratamento paralelo das filas de mensagens, dinamicamente, de acordo com a vazão de entrada e de saída e a detecção de crescimento dada pela Equação 1. Estas equações são baseadas no comportamento dinâmico de cada servidor e nos limites de tempo impostos para o processamento de mensagens, os quais estão diretamente relacionados com os parâmetros de QoS (*Quality of Service*) definidos com cada cliente da aplicação distribuída que utiliza fila de mensagens. Todavia, uma limitação atual do *middleware* IOD está ligada ao fato de não ser possível paralelizar o tratamento de mensagens associadas ao mesmo grupo, por conta da restrição de integridade sequencial que impõe ordenação no processamento.

Assim sendo, entre os trabalhos futuros estão os estudos de *pipelining* para a execução paralela de instruções que podem ser exploradas para o adiantamento do processamento de mensagens associadas ao mesmo grupo, sem a geração de conflito de integridade sequencial.

Referências

Abbes, H., Cerin, C., Jemni, M., and Walid.Saad (2010). Fault tolerance based on the publish-subscribe paradigm for the bonjourgrid middleware. *11th IEEE/ACM International Conference on Grid Computing*, pages 57–64.

Bicer, T., Jiang, W., and Agrawal, G. (2010). Supporting fault tolerance in a data-intensive computing middleware. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12.

Castro, M., Rexachs, D., and Luque, E. (2012). Transparent fault tolerance middleware at user level. *2012 International Conference on High Performance Computing and Simulation (HPCS)*, pages 566–572.

Coulouris, G., Dollimore, J., and Kindberg, T. (2009). *Distributed Systems Concepts and Design*. Editora Pearson Prentice Hall, fourth edition.

Downing, D. and Clark, J. (2011). *EstatÌstica Aplicada*. Editora Saraiva, third edition.

Fang, W., Jin, B., Zhang, B., Yang, Y., and Qin, Z. (2011). Design and evaluation of a pub/sub service in the cloud. *International Conference on Cloud and Service Computing*, pages 32–39.

Gray, J. S. (2003). *Interprocess Communications in LinuxÆ: The Nooks Crannies*. Editora Pearson Prentice Hall, first edition.

Guo, Y., Hanm, R., Satzger, B., and Truong, H.-L. (2012). Programming directives for elastic computing. *Internet Computing and IEEE*, pages 72–77.

He, C., Weitzel, D., Swanson, D., and Lu, Y. (2012). Hog: Distributed hadoop mapreduce on the grid. *SC Companion: High Performance Computing and Networking Storage and Analysis*, pages 1276–1283.

Herbst, N. R., Kounev, S., and Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA. USENIX.

Imai, S., Chestna, T., and Varela, C. A. (2012). Elastic scalable cloud computing using application-level migration. *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 91–98.

Leitner, P., Inzinger, C., Hummer, W., Satzger, B., and Dustdar, S. (2012). Application-level performance monitoring of cloud services based on the complex event processing paradigm. *5th IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–8.

Li, M., Ye, F., Kim, M., Chen, H., and Lei, H. (2011). A scalable and elastic publish/-subscribe service. *IEEE International Parallel & Distributed Processing Symposium*, pages 1254–1265.

Ma, R. K., Lam, K. T., Wang, C.-L., and Zhang, C. (2010). A stack-on-demand execution model for elastic computing. *39th International Conference on Parallel Processing*, pages 208–217.

Marshall, P., Tufo, H., and Keahey, K. (2012). Provisioning policies for elastic computing environments. *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forums*, pages 1085–1094.

Martins, R., Narasimhan, P., Lopes, L., and Silva, F. (2010). Lightweight fault-tolerance for peer-to-peer middleware. *29th IEEE International Symposium on Reliable Distributed Systems*, pages 313–317.

Perez, J., Germain-Renaud, C., Kégl, B., and Loomis, C. (2009). Responsive elastic computing. *ACM/IEEE Conference on International Conference on Autonomic Computing*, pages 55–64.

Smith, J. M. (1986). A survey of software fault tolerance techniques. pages 165–170.

Sugiki, A. and Kato, K. (2011). An extensible cloud platform inspired by operating systems. *Fourth IEEE International Conference on Utility and Cloud Computing*, pages 306–311.

Tanenbaum, A. S. and Steen, M. V. (2008). *Distributed systems: principles and paradigms*. Editora Pearson Prentice Hall, second edition.

Tanenbaum, A. S. and Wetherall, D. J. (2010). *Computer Networks*. Editora Pearson Prentice Hall, fifth edition.

Tanenbaum, A. S. and Woodhull, A. S. (2007). *Operating Systems Design and Implementation*. Editora Pearson Prentice Hall, third edition.

Tran, N.-L., Skhiri, S., and Zimányi, E. (2011). Eqs: an elastic and scalable message queue for the cloud. *Third IEEE International Conference on Cloud Computing Technology and Science*, pages 391–398.

Wang, J., Jian-Wen Chen, Y. D., and Zheng, D. (2009). Research of the middleware based fault tolerance for the complex distributed simulation applications. *International Conference on Computational Intelligence and Software Engineering (CiSE)*, pages 1–4.