Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Middleware para Coordenar Tolerância a Falhas e Elasticidade em Clusters de Alto Desempenho com Produtores e Consumidores Baseados em Filas de Mensagens

Eduardo Henrique Ferreira Mendes Teixeira

Dissertação apresentada como requisito parcial
para conclusão do Mestrado Profissional em Computação Aplicada

Orientadora
Prof.ª Dr.ª Aletéia P. Favacho de Araújo Von Paumgartten

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado Profissional em Computação Aplicada

Coordenador: Prof. Dr. Marcelo Ladeira

Banca examinadora composta por:

Prof.ª Dr.ª Aletéia P. Favacho de Araújo Von Paumgartten (Orientadora) — CIC/UnB
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo — CIC/UnB
Prof. Dr. Alexandre da Costa Sena — IME/UERJ

CIP — Catalogação Internacional na Publicação

Teixeira, Eduardo Henrique Ferreira Mendes.

Middleware para Coordenar Tolerância a Falhas e Elasticidade em Clusters de Alto Desempenho com Produtores e Consumidores Baseados em Filas de Mensagens / Eduardo Henrique Ferreira Mendes Teixeira. Brasília : UnB, 2014.
112 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2014.

1. *Middleware*, 2. Tolerância a Falhas, 3. Elasticidade, 4. *Cluster*, 5. Alto Desempenho, 6. Filas de Mensagens

CDU 004.272:004.275

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Middleware para Coordenar Tolerância a Falhas e Elasticidade em Clusters de Alto Desempenho com Produtores e Consumidores Baseados em Filas de Mensagens

Eduardo Henrique Ferreira Mendes Teixeira

Dissertação apresentada como requisito parcial
para conclusão do Mestrado Profissional em Computação Aplicada

Prof.ª Dr.ª Aletéia P. Favacho de Araújo Von Paumgartten (Orientadora)
CIC/UnB

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo
CIC/UnB

Prof. Dr. Alexandre da Costa Sena
IME/UERJ

Prof. Dr. Marcelo Ladeira
Coordenador do Mestrado Profissional em Computação Aplicada

Brasília, 7 de Julho de 2014

Dedicatória

Este trabalho é dedicado à minha família e aos meus amigos verdadeiros.

Agradecimentos

Os agradecimentos principais são direcionados à minha amada filha Heloísa, ao meu grande amigo e irmão Benício, minhas amadas mãe Socorro e irmãs Silvana e Flória, ao meu grande amigo e cunhado Vladmir, minha grande amiga Natália e a todos os amigos verdadeiros que me apoiaram nos diversos momentos de dificuldade durante a realização deste trabalho. Um agradecimento muito especial, forte e sincero ao meu pai Benício pelo esforço de sua vida lutando para fazer dos filhos pessoas de bem e honradas.

Agradecimentos especiais à Autotrac, aqui representada por Carlos Henrique, Fernando Feliu, Fernando Mühe, Gilnei Cardoso e Hideraldo Bernardo, pelo apoio em acreditar no sucesso do projeto fornecendo todos os mecanismos necessários para as implementações e testes da solução desenvolvida.

Especialmente à equipe da GPS (Fernando Feliu, Pablo Aranha, Luiz Henrique, Alexandre Frota, Alexandre Baião, Felipe Carneiro), e à equipe da HUB (João Romariz, Alan Olímpio, Alisson Fiel, Wallisson Viana, Rômulo de Azevedo, Thiago Cordeiro, Luciane dos Santos, Pedro Farias) por terem dado suporte e apoio durante o tempo de desenvolvimento do projeto e escrita da dissertação.

Um obrigado muito especial à Prof.ª Dr.ª Aletéia Patrícia Favacho de Araújo pela enorme paciência, orientação e apoio essenciais para fazer deste projeto ambicioso uma realidade.

Resumo

Este trabalho propôs e avaliou um *middleware* com suporte à tolerância a falhas e à elasticidade em um *cluster* de alto desempenho. Para isso, foi construída uma arquitetura elástica para se adaptar dinamicamente ao crescimento da fila de requisições, para que as mensagens não se acumulem, e tolerante a falhas para que eventuais paradas do sistema, por queda ou falha dos serviços, não impactem na operacionalidade do *cluster*. Assim sendo, o *middleware* desenvolvido foi capaz de diminuir o número de servidores necessários para processar as filas de mensagens, liberando recursos da infraestrutura do *cluster* para uso como *failover* do sistema distribuído ou em outras aplicações. Consequentemente, a qualidade dos serviços prestados melhorou, devido a diminuição dos tempos de atualização do sistema por conta de manutenções evolutivas e corretivas.

Palavras-chave: *Middleware*, Tolerância a Falhas, Elasticidade, *Cluster*, Alto Desempenho, Filas de Mensagens

Abstract

This work proposed and evaluated a middleware with support for fault tolerance and elasticity in a high performance cluster. For this purpose, it was constructed an elastic architecture to dynamically adapt to growth in the request queue, so that messages do not accumulate. Also the architeture provides fault-tolerance to system outages, in the cases of failure of service, so these failures do not impact on the operation of the cluster. The middleware developed was able to decrease the number of servers needed to process the message queue, freeing infrastructure resources of the cluster for use as a failover of the distributed system or in other applications. Consequently, the quality of service has improved due to shortened time to update the system on behalf of progressive and corrective maintenance.

Keywords: Middleware, Fault Tolerance, Elasticity, Cluster, High Performance, Message Queues

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Um sistema de computação distribuída consiste de processos distintos que não compartilham memória, e se comunicam assincronamente por passagem de mensagem ou canais de comunicação. Dessa forma, um sistema distribuído lida com os aspectos da computação relacionados à troca de informações através de múltiplos elementos de processamento conectados por redes de comunicação [16].

A computação distribuída começou a surgir com os avanços e as evoluções nas áreas de redes, e com a diminuição do custo dos equipamentos de *hardware*. Assim, estes fatores fizeram com que a computação distribuída, com alto desempenho e tolerância a falhas, se tornasse uma realidade. Entre as vantagens do uso de sistemas distribuídos estão a possibilidade de compartilhamento de recursos e tarefas, bem como a redução do custo mesmo diante de um poder de processamento similar ao de *mainframes*.

Entre os vários tipos de arquitetura para realizar a computação distribuída existe o aglomerado (cluster), que caracteriza-se, geralmente, por possuir uma homogeneidade de hardware e software entre todos os nós conectados [4]. O *cluster*, segundo a taxonomia de Flynn [10], é classificado como uma máquina MIMD (*Multiple Instruction Multiple Data*). Assim sendo, os *clusters* possuem fraco acoplamento, onde cada máquina tem sua memória local, não compartilhada entre outros nós [28], o que é uma característica importante para arquiteturas tolerantes a falhas.

Dessa forma, o *cluster* pode ser visto como uma plataforma que consiste em um conjunto de computadores conectados em uma rede local para trabalharem em conjunto, e que são vistos como um único sistema, apesar de cada nó executar sua própria instância de sistema operacional. O *cluster* pode ser usado para melhorar o desempenho, a disponibilidade e o custo de aplicações por meio do uso de vários computadores simples.

A computação em *cluster* utiliza um gerenciamento centralizado para prover serviços e controla os nós disponíveis como servidores compartilhados, tornando esta abordagem transparente para o usuário da aplicação.

O desenvolvimento de aplicações em arquiteturas distribuídas como o *cluster* tem um custo associado à complexidade de manutenção destes sistemas, pois os *softwares* que executam em um ambiente distribuído devem ser capazes de lidar de uma maneira transparente, com a distribuição das tarefas, dos dados e dos recursos para a execução de atividades concorrentes e compartilhadas, sem a existência de sincronização entre todos os componentes e com possibilidades de falhas isoladas [7].

Para facilitar essas atividades, surgiu o conceito de *middleware*, que é uma camada de *software* que simplifica e abstrai grande parte da complexidade envolvida com as características básicas de funcionamento dos sistemas em ambientes distribuídos.

Nesse cenário, este trabalho analisou um *cluster* de alto desempenho composto por 58 servidores. As manutenções evolutivas do sistema duram cerca de três horas em função da quantidade de servidores que devem ser atualizados. Estas atualizações devem ser realizadas com aviso prévio aos seus usuários e com o agendamento de *downtime*, que é o período de tempo em que um sistema fica indisponível por conta de rotinas de manutenção. Além disso, o sistema também fica inoperante durante problemas de funcionamento por queda ou falha no tratamento das filas de mensagens. Ambas as situações levam a uma diminuição da qualidade dos serviços prestados aos usuários desta plataforma distribuída.

Assim sendo, este trabalho propõe e avalia a implementação de um *middleware* eficiente e transparente aos programadores da aplicação distribuída, que garanta a elasticidade e a tolerância a falhas. O *middleware* proposto deve ser elástico a fim de conter dinamicamente o crescimento das filas, e deve ser tolerante a falhas para suportar também a resistência a quedas e falhas.

É importante ressaltar que nesse sistema, a conexão com os servidores, para o envio e o recebimento de mensagens, consome quase 100% da capacidade de processamento de cada nó do *cluster*, durante um ciclo de conexão. Neste trabalho, um ciclo de conexão é o tempo em que o sistema realiza o envio e o recebimento de mensagens com os servidores que se comunicam com as unidades móveis instaladas nos veículos monitorados, e ele ocorre a cada 80 segundos. Dessa forma, após o término deste ciclo, que é realizado rapidamente (menos de 10 segundos), o servidor fica ocioso esperando iniciar um novo ciclo de 80 segundos. Essa situação pode ser observada na Figura 1.1, onde os servidores utilizam a CPU em rajadas muito rápidas e, em seguida, ficam ociosos a maior parte do tempo, representando um uso ineficiente dos recursos disponíveis nesta plataforma.

Assim, nota-se que é interessante a proposta de um *middleware* para esse *cluster*, a fim de garantir o uso eficiente dos recursos por meio da técnica de elasticidade, bem como garantir segurança e confiabilidade por meio das técnicas de tolerância a falhas.

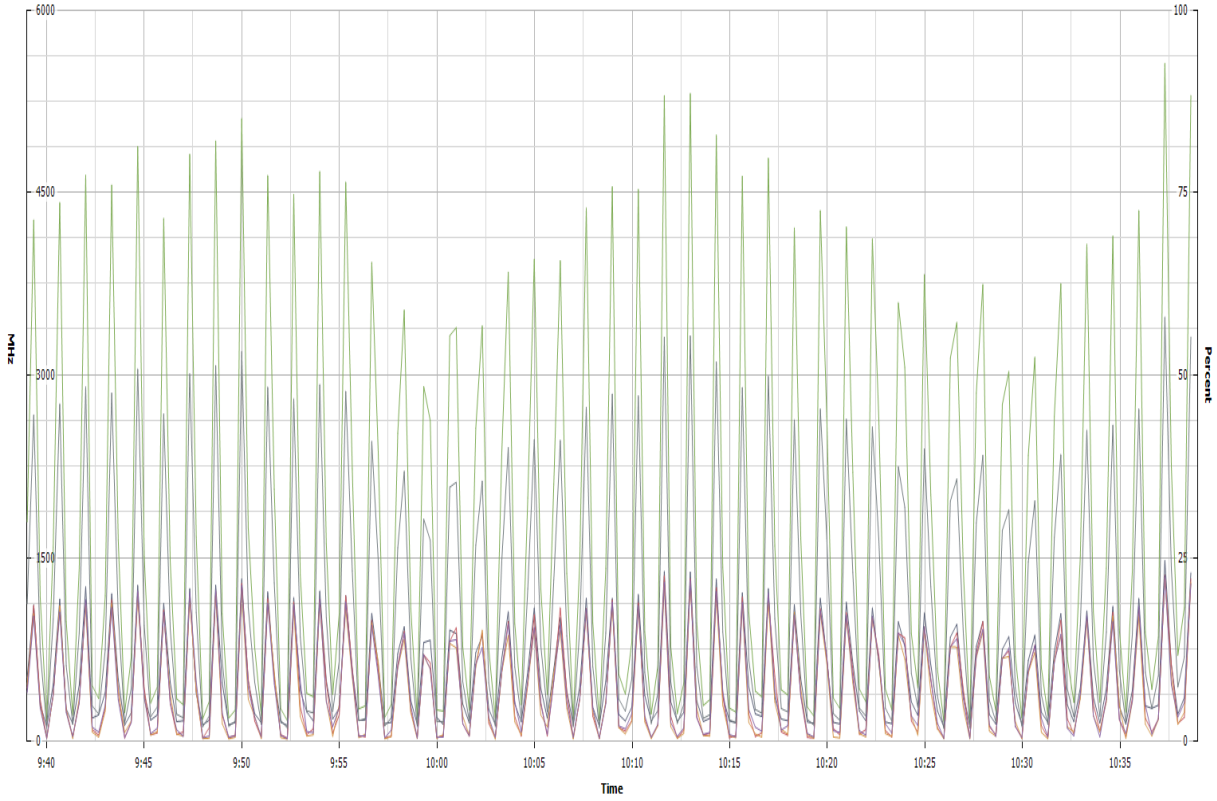


Figura 1.1: Uso de CPU dos Servidores do *Cluster* Durante 1 Hora.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é propor e desenvolver um *middleware* para um *cluster* de alto desempenho, que garante elasticidade e tolerância a falhas de forma transparente aos serviços disponibilizados em uma plataforma distribuída.

1.1.2 Objetivos Específicos

Para cumprir o objetivo geral, este trabalho tem os seguintes objetivos específicos:

- Levantar o estado da arte por meio da análise crítica de artigos científicos recentes, referentes aos temas tolerância a falhas e elasticidade.
- Analisar as abordagens mais relevantes para serem utilizadas na implementação das características de tolerância a falhas e elasticidade no *middleware* proposto.
- Propor o *middleware*.
- Implementar o *middleware* proposto.

- Testar e analisar o *middleware* desenvolvido.
- Validar o *middleware* desenvolvido no ambiente real do sistema distribuído.
- Publicar artigos científicos contendo descobertas relevantes e os resultados dos experimentos realizados.

1.2 Estrutura do Trabalho

O restante deste trabalho está dividido em cinco capítulos. O Capítulo 2 aborda os sistemas distribuídos, os *middlewares*, o ambiente de computação distribuída *cluster* e, em seguida, critica e relaciona os principais artigos científicos em relação aos temas tolerância a falhas e elasticidade. O próximo capítulo, o Capítulo 3, apresenta o cenário de uma aplicação distribuída e o projeto do *middleware* proposto. Em seguida, o Capítulo 4, mostra a implementação de um simulador do servidor de comunicação, necessário para os testes a serem realizados com o *middleware* desenvolvido. O Capítulo 5 mostra os resultados das simulações com o *middleware* proposto e as adaptações e adequações necessárias para o funcionamento no ambiente real da arquitetura. Para finalizar, o Capítulo 6 apresenta as conclusões pertinentes, e alguns trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta brevemente os conceitos fundamentais relacionados ao desenvolvimento deste trabalho. Também é apresentado o cenário de um sistema distribuído em *cluster*, para o qual este trabalho tomou como plataforma de desenvolvimento.

2.1 Sistemas Distribuídos

A existência de aplicações que demandam intenso poder de processamento em sua execução, geraram a necessidade de se utilizar mais de um processador com o objetivo de resolver uma tarefa mais eficientemente. Assim, essa necessidade deu origem a diversos tipos de arquiteturas de sistemas distribuídos, que são uma coleção de computadores autônomos e independentes, que se apresenta ao usuário como um sistema único e consistente [29]. Desta forma, o processamento nessas plataformas distribuídas utiliza computação paralela ou distribuída, executada por duas ou mais entidades independentes, conectadas em rede e que cooperam entre si para resolver um problema complexo.

A redução do custo dos computadores pessoais, aliado a melhoria da tecnologia de comunicação em rede, possibilitou o desenvolvimento das plataformas distribuídas. Existem vários tipos de plataformas distribuídas tais como aglomerado (*cluster*) [4], grade computacional (*grid*) [23] e nuvem computacional (*cloud*) [23]. Neste trabalho será dado ênfase aos *clusters* (aglomerados), por serem a plataforma na qual será desenvolvido o *middleware* proposto.

2.1.1 *Cluster Computing*

Na computação em *cluster*, o *hardware* consiste em uma coleção de estações de trabalho similares, conectadas em redes locais de alta velocidade, em que cada nó executa o mesmo

sistema operacional. É bastante usada para programação paralela, na qual um programa de computação intensiva executa em paralelo em múltiplas máquinas [29].

Nos *clusters*, cada nó de processamento é um computador autônomo, independente e interligado por uma rede de comunicação. Essa platforma é classificada como sendo fracamente acoplada, pois cada nó tem sua própria memória. Assim, caso um elemento falhe (*hardware* ou *software*), não impacta os demais, apesar de todos os computadores serem vistos como um agrupamento único. Dessa forma, uma grande vantagem da arquitetura de *cluster* é a alta disponibilidade, que pode ser alcançada através da tolerância a falhas, pois a queda de um componente não impossibilita o funcionamento do restante do *cluster*. Portanto, o objetivo dessa infraestrutura é fazer com que o conjunto de computadores se pareça como um único recurso com desempenho e capacidade maiores do que o de um único componente isolado.

Uma outra vantagem associada ao uso da infraestrutura de *cluster* está ligada a sua capacidade de poder adicionar e remover componentes do *cluster*, e o mesmo continue operacional. Como os componentes, geralmente, são mais baratos do que os de outras infraestruturas de sistemas distribuídos, a relação custo-benefício tende a ser melhor neste tipo de arquitetura distribuída. Porém, como cada nó do *cluster* é um computador completo e autônomo, a medida que o número de máquinas aumenta, a administração do ambiente tende a ficar mais complexa. Apesar disso, o *cluster* ainda é um dos ambientes de sistemas distribuídos mais usado.

Existem vários tipos de *cluster*, entre eles destacam-se [4]:

- ***Cluster* de Alto Desempenho**
  É projetado de forma que as tarefas executadas utilizem um grande número de recursos disponíveis para obter o melhor desempenho possível, ou seja, funciona permitindo que ocorra uma alta carga de processamento com um grande volume de dados.

- ***Cluster* de Alta Disponibilidade**
  Projetado para que seja utilizado o maior tempo possível sem falhas, ou seja, são *clusters* cujos sistemas conseguem permanecer ativos por um longo período de tempo em condição de uso, sem a necessidade de reinício e de maneira ininterrupta. Além disso, conseguem detectar eficientemente erros, proporcionando que os mesmos sejam tolerantes a falhas.

- ***Cluster* para Balanceamento de Carga**
  Esse tipo de *cluster* tem como função controlar a distribuição equilibrada de carga por meio de um monitoramento constante no seu processamento, e em seus mecanismos de redundância.

- ***Cluster* Híbrido**
  Cluster que combina algumas das características dos *clusters* citados anteriormente.

Uma parte importante do *cluster* é formada por uma camada de *software* que possibilita a execução de programas em paralelo, denominada *middleware*. Vários *middlewares* proveêm efetivamente facilidades de comunicação avançada baseada em mensagens [29], para simplificar e abstrair o processo de programação dos sistemas em plataformas distribuídas.

Essa camada de software torna o desenvolvimento mais fácil e ágil, proporcionando o aumento da abstração para os desenvolvedores da aplicação com relação à comunicação, à execução de requisições, à escalabilidade e à heterogeneidade. A seção seguinte explica com detalhes essa camada de *software*, chamada de *middleware*.

2.1.2 *Middleware*

O *middleware* é uma camada de *software* que conduz o sistema distribuído, proporcionando transparência de heterogeneidade ao nível de plataforma [16]. Assim sendo, ele é representado por processos ou objetos em um conjunto de computadores que interagem entre si para implementar a comunicação e o compartilhamento de recursos para aplicações distribuídas [7].

Dessa forma, o *middleware* eleva o nível das atividades de comunicação dos programas aplicativos, por meio do suporte a abstrações como invocação remota de métodos, comunicação entre grupos de processos, notificação de eventos, transmissão e replicação de dados compartilhados. Nesse contexto, o *middleware* é uma camada de software que provê abstração de programação, bem como o mascaramento dos diferentes tipos de redes, *hardware*, sistemas operacionais e linguagens de programação [7].

Atualmente, para se desenvolver um sistema distribuído em *cluster* utiliza-se, geralmente, um *middleware* como CORBA [29], RMI [7], RPC [7], DCOM [16], MPI [5], ou desenvolve-se uma API[1] (*Application Program Interface*) proprietária, tornando a aplicação dependente desta camada.

Assim sendo, muitas aplicações distribuídas dependem inteiramente dos serviços providos pelo *middleware*, para suportar suas necessidades de comunicação e compartilhamento de dados. Os *middlewares*, geralmente, são projetados para funcionar de acordo com as peculiaridades específicas de cada plataforma distribuída, fazendo com que sejam customizados para as características de cada uma destas arquiteturas.

Assim, apesar do *middleware* simplificar a programação de sistemas distribuídos, uma desvantagem é que alguns aspectos de funcionamento desses sistemas ainda requerem

[1] Uma interface consistindo de chamadas de biblioteca [29].

suporte no nível de aplicação, como os mecanismos de correção de erros e as medidas de segurança. As checagens realizadas dentro do *middleware*, somente vão garantir uma parte do funcionamento correto requerido, podendo gerar duplicação de trabalho nas aplicações e desperdício do esforço de programação por meio da adição de complexidade e redundância desnecessárias, que prejudicam o desempenho do sistema distribuído. Ou seja, ainda há muito esforço de programação necessário e externo ao *middleware* para que os sistemas funcionem corretamente. Porém, apesar disso, o *middleware* simplifica a implementação de aplicações e o projeto de acordo com uma arquitetura específica [16].

Nesse contexto, será analisado na próxima seção, como os processos de um sistema distribuído, por meio do *middleware*, comunicam-se para a troca de dados. Esse mecanismo é chamado de IPC (*Interprocess Communication*).

2.1.3 Comunicação Interprocesso

Comunicação Interprocesso - IPC (*Interprocess communication*) é uma abordagem de comunicação implementada para a cooperação entre processos. Sendo assim, ela é o coração de todos os sistemas dsitribuídos [29]. Por exemplo, o IPC de sistemas *unix-like* se utiliza de diversos mecanismos de comunicação [11]:

- ***Pipes***
  São tipos especiais de arquivos que armazenam uma quantidade limitada de dados de uma forma FIFO (*First In First Out*).
- ***Filas de Mensagens***
  As filas de mensagens proveem mecanismos de acesso aos dados de forma assíncrona a partir de múltiplos processos de uma maneira FIFO (*First In First Out*).
- **Semáforos**
  São primitivas atômicas de sistema operacional utilizadas para operações de sincronização quando múltiplos processos acessam um recurso compartilhado, porém de forma exclusiva.
- ***Compartilhamento de Memória***
  É o método mais rápido de IPC, onde a informação é acessada diretamente em um espaço de dados compartilhado. Geralmente, os semáforos são usados para sincronizar o acesso aos segmentos de memória compartilhada.
- ***Sockets***
  São usados para criar um canal de comunicação *full duplex* para enviar e receber informações de processos não relacionados, inclusive em diferentes máquinas.

Na próxima seção serão apresentados diversos trabalhos relacionados na literatura científica, a fim de auxiliar na elaboração de uma proposta para a solução de um *middleware* que implemente tolerância a falhas e elasticidade.

2.2 Trabalhos Relacionados

Esta seção analisa, critica e relaciona diversos artigos científicos associados à tolerância a falhas e à elasticidade em ambiente de *cluster*. Para isso, os trabalhos analisados foram divididos em duas seções, as quais são descritas a seguir.

2.2.1 Tolerância a Falhas

Falhas em sistemas de computação podem ocorrer devido a erros de *hardware* ou *software* [25]. 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* [25]. 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 [7].

Existem diversos tipos de falhas de *software*, entre elas as falhas por omissão, falhas arbitrárias e falhas de temporização [7]. As falhas por omissão podem ocorrer quando um processo sofre queda total ou paralisa parcialmente a execução de uma de suas funcionalidades, e também quando ocorrem falhas de comunicação por conta de erros de transmissão na rede de dados. As falhas arbitrárias ou bizantinas, ocorrem por qualquer tipo de erro devido a procedimentos efetuados de maneira incorreta, como, por exemplo, dados recebendo valores inválidos, conteúdos de mensagens corrompidos e mensagens inexistentes sendo entregues. As falhas de temporização se aplicam a sistemas distribuídos síncronos, e resultam em respostas indisponíveis aos clientes por um certo intervalo de tempo.

Na próxima subseção, serão analisados diversos artigos científicos relacionados à tolerância a falhas de *software* por omissão do processo.

Revisão da Literatura

Para suportar tolerância a falhas de *software*, em [6][33] são propostos mecanismos de gravação e interceptação de *logs* para posterior recuperação. Em [6], além disso, também é utilizado um protocolo de recuperação de falhas transparente. No *framework* HOG (*Hadoop On the Grid*) é utilizada uma arquitetura onde a falha de um nó não afeta a integridade dos demais, e ao ocorrer queda ou travamento de algum processo, a opera-

cionalidade pode ser facilmente recuperada e realizada por outro grupo de serviços que acessam os mesmos dados [13]. A replicação é um ponto chave para que os sistemas distribuídos sejam eficazes em proporcionar tolerância a falhas de *software* [7]. Por outro lado, ao invés do uso de replicação, os trabalhos citados em [3][6][13][33] possuem arquiteturas com suporte a tolerância a falhas por meio de mecanismos rápidos de recuperação.

Em [6][21] é sugerido que as implementações de suporte às características de tolerância a falhas sejam feitas na biblioteca de comunicação, onde será feito o *middleware* e que seja de forma transparente à aplicação.

Por outro lado, a abordagem de implementação de tolerância a falhas citada em [27], onde as aplicações podem impor variações na forma como a tolerância a falhas deve ser suportada, é interessante quando se constrói um *middleware* de propósito genérico, que atenda a vários tipos de projetos com diferentes requisitos de políticas de qualidade de serviço. Em soluções de *middleware*, onde não há necessidade de variações nas políticas de níveis de qualidade, a implementação de suporte a estas características é desnecessária.

A replicação de nós para a tolerância a falhas [1][21] tem um custo de memória necessário para manter os coordenadores auxiliares rodando [1], e de comunicação em que as atualizações devem ser enviadas para todas as réplicas, de forma a manter o estado de *cache* consistente [1]. Uma outra desvantagem é que se a falha ocorrer por causa de um pacote que gera uma exceção, a mesma também ocorrerá nas k réplicas.

A abordagem de *snapshots*, que é o estado de um sistema em um ponto particular no tempo, em períodos de tempo [3], mostra-se interessante por ter uma característica de recuperação rápida em caso de falha. Porém, o trabalho em grupo citado em [3] tem a desvantagem do custo da comunicação de envio dos estados para os outros nós. É interessante o uso de *snapshots* persistentes, para que em caso de detecção de falha algum outro serviço assuma seu processamento, recuperando o último *checkpoint* válido, porém sem envio dos estados para evitar a sobrecarga da comunicação.

Em [35] é usada uma abordagem com o uso de UDP (*User Datagram Protocol*) com uma proposta de protocolo com garantia de entrega confiável e ordenação total. Por outro lado, com o uso do TCP (*Transmission Control Protocol*) é possível detectar quando um nó falha por meio da queda da conexão [3][21] e *pings* periódicos com *timeouts* [21].

Como pôde ser visto, 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 [1][3][13][21]. No entanto, técnicas como a recuperação rápida de falhas [3][6][13][33] são interessantes, pois evitam os custos de memória adicional e de comunicação entre os processos replicados para a troca das informações de estados. O uso do suporte à tolerância a falhas transparente no *middleware* [6][21] e o uso de detecção da queda ou falha por meio do monitoramento da conexão entre os processos [3][21], também são

abordagens atrativas para abstrair da aplicação usuária os detalhes de detecção e de recuperação das características implementadas.

2.2.2 Elasticidade

Elasticidade é definida como o grau no qual um sistema é capaz de adaptar-se às mudanças de carga de trabalho por meio do provisionamento e liberação de recursos de forma automática, de modo que esses recursos disponíveis correspondam, o mais próximo possível, com a demanda atual [14]. Esse provisionamento dinâmico de recursos é definido como computação elástica [23].

Existem dois tipos de elasticidade, a horizontal que é a capacidade de adicionar ou remover as máquinas virtuais atribuídas a um serviço implantado, e a vertical que é a capacidade de alterar a configuração já em execução para aumentar ou diminuir o total de recursos alocados para um serviço sendo executado [2]. Nesse trabalho será utilizada somente a elasticidade vertical, por meio do aumento de recursos necessários para tratar as filas de mensagens dos servidores.

A próxima subseção analisa e critica a elasticidade por meio de diversos artigos científicos relacionados.

Revisão da Literatura

Para suportar elasticidade, a abordagem citada em [19] propõe um mecanismo de migração dinâmica de execução chamado SOD (*Stack-On-Demand*) baseado na pilha de execução da JVM (*Java Virtual Machine*). Assim, o *middleware* SOD migra uma porção mínima do estado da pilha de máquinas virtuais para um sítio de destino para que continue a execução de uma determinada tarefa. Esta abordagem apesar de trazer transparência de paralelismo e de elasticidade, é de difícil implementação pois requer conhecimentos de funcionamento da JVM (*Java Virtual Machine*) e de codificação de baixo nível, além de tornar a aplicação dependente de ser desenvolvida na linguagem *Java*. Porém, pode ser adaptada através da carga dinâmica de códigos que possuem funcionalidades de um grupo de serviços específico.

Assim, qualquer serviço responsável pelo consumo de mensagens pode processar qualquer tipo de mensagem através da sua subscrição em um coordenador e da carga da biblioteca dinâmica que compõe o código que realizará este processamento. Dessa forma, porções de código podem ser ativadas para qualquer processo consumidor, e estes podem estar executando em servidores distintos.

Em [9] é citado que os *middlewares* baseados em *publishers/subscribers*, que é um padrão de projeto que realiza propagação de informações em uma via na qual o *publisher*

pode notificar um ou mais *subscribers* quando o estado de um assunto muda [24], tem se tornado a base da coordenação e da cooperação de sistemas distribuídos, facilitando a disseminação de informação e integração de aplicações. O modelo *publisher/subscriber* é um padrão de projeto de desenvolvimento de *software* onde é utilizado um estilo de cooperação no modelo produtor/consumidor fortemente desacoplados, onde o produtor é suspenso quando o *buffer* está cheio e o consumidor quando está vazio [24]. Na arquitetura proposta em [9] o *middleware* recebe os eventos em um ambiente que é dinâmico e sujeito a mudanças irregulares e inesperadas, habilitando a automação do processamento dos dados pela transmissão da informação dos produtores (*publishers*) para os consumidores (*subscribers*). Um ponto atrativo nessa abordagem é que produtores e consumidores são totalmente desacoplados no tempo, no espaço e no controle de fluxo, significando que pode ser usada uma abordagem de comunicação assíncrona sem perda dos dados e com a flexibilidade elástica do acoplamento dinâmico de clientes.

Na abordagem proposta em [32], os recursos são usados sob demanda permitindo que as aplicações escalem elasticamente de acordo com a carga por meio de mensagens enviadas assincronamente entre aplicações distribuídas por meio de filas de mensagens. Esta abordagem é interessante quando a integridade sequencialno processamento de mensagens não é um requisito funcional. Assim, o sistema escala de forma transparente e automática [32] por meio das mensagens nas filas que podem ser dimensionadas e redistribuídas conforme a vazão requerida, sem se preocupar com a ordem em que as mensagens são processadas. Porém, nesse caso o uso da elasticidade para o aumento da vazão de maneira assíncrona é uma desvantagem, quando há a necessidade de manter a consistência sequencial baseada na ordem de entrega, para que os eventos gerados pelos serviços consumidores preservem a restrição temporal em que foram gerados.

Um sistema com múltiplas filas de mensagens [18] e vários consumidores pode gerar assimetrias de distribuição nas atribuições feitas pelos *publishers* aos *subscribers* [9][18][32]. Sendo assim, é necessário identificar uma correspondência de melhor candidato ao consumo de novas filas de forma a diminuir a latência de consumo e evitar a saturação de consumidores sobrecarregados. Dessa forma, é possível determinar os melhores candidatos ao consumo de novos itens, baseado na maior vazão média identificada [18][32] e no menor consumo médio de CPU [18].

O número médio de novos itens também se mostra uma abordagem pró-ativa interessante para que o sistema possa se preparar previamente, aumentando o número de processos necessários para suportar uma demanda de pico prevista baseada em históricos. Esta característica de provisionamento, para alcançar a qualidade de serviço necessária, representa uma otimização com a diminuição da latência do início de processos que seria feita no modo reativo.

Em [26] é sugerido um protótipo que adota um projeto baseado em *microkernel* de sistemas operacionais (OS *like middleware*) [31], com abstração de recursos alcançada por objetos e funções, oferecendo um ambiente de *script* como interface de programação. A abstração de recursos é explorada para prover acesso uniforme para vários tipos de recursos. São utilizadas técnicas de agendadores de sistemas operacionais para gerenciar e satisfazer a demanda por recursos eficientemente. Nesse trabalho os recursos de máquinas virtuais são encapsulados como objetos de linguagem, por meio de um conjunto de métodos de acesso uniforme para diferentes tipos de recursos. Assim, os recursos podem ser manipulados em um ambiente de *script* de forma que os programas sejam desenvolvidos usando um paradigma orientado a objetos. Nesse caso são disponibilizadas funções que provêem paralelismo e distribuição, para aumentar a escalabilidade usando eficientemente os núcleos disponíveis em cada máquina. A elasticidade é alcançada usando um agendador elástico, que trabalha no lado do usuário gerando *clones* de máquinas virtuais quando mais capacidade é requerida. Se o uso médio de CPU exceder um limite, réplicas de máquinas virtuais são iniciadas para prover elasticidade.

Segundo [17], a computação elástica traz enormes vantagens para provedores de aplicações, incluindo economia de custos, prevenção de super-provisionamento de recursos de TI através do monitoramento de demanda e aquisição dos recursos estritamente requeridos pelas aplicações para alcançar um alto nível de qualidade. Assim, são indicadas métricas de alto nível, tais como a capacidade de resposta de suas aplicações para certas operações de tempo crítico, que podem ser usadas para aumentar a elasticidade de aplicações em *middlewares* existentes. Assim, os desenvolvedores de aplicação escrevem suas políticas de escalabilidade baseadas no significado dessas métricas de alto nível. Porém, métricas como número médio de novos dados processados por segundo e tempo médio (em milisegundos) necessários para processar um único novo item de dados tem uma relevância maior no domínio de negócio da aplicação, e podem dar uma visão mais precisa do desempenho das aplicações do que a carga média de CPU, e também podem definir políticas de comportamento para a escalabilidade.

Os trabalhos [15][19][32] exploram a transparência de elasticidade para a aplicação. Os autores em [15] usam o aumento ou a diminuição do número de VMs de acordo com a demanda de carga exigida, que é medida pelo consumo de CPU [15][18][26]. Esta é uma abordagem interessante quando o objetivo é a contenção do crescimento de filas de mensagens, porém em sistemas distribuídos com características I/O (*Input/Output*) *bound*, que são sistemas que fazem uso intensivo de I/O, pode se mostrar ineficiente.

Dessa forma, se o número de mensagens se mantiver próximo aos limites, podem ocorrer problemas de início e término rápidos de processos. Para manter a vazão em níveis aceitáveis [15][18][32], o uso de heurísticas com abordagem baseada em limites por

um determinado tempo, mostra-se interessante quando aplicada para calcular o número de processos consumidores que serão iniciados [15]. Estes algoritmos podem ser usados para sanar o problema do limiar de detecção de elasticidade para cima ou para baixo. Além disso, a técnica de manter o processo criado por mais tempo, para minimizar o custo de reinício de processos pelo sistema operacional, diminui a latência de consumo de mensagens.

Por último, em [12] é proposta uma abordagem que permite aos desenvolvedores separar a lógica dos programas do controle dos ambientes de computação de forma transparente. Esta é uma característica desejável para reduzir a complexidade das tarefas de gerenciamento que os desenvolvedores precisam lidar no código. Isto também pode ajudar a otimizar o uso de recursos sob restrições de custo para alcançar a qualidade desejada, uma vez que os sistemas serão mais simples de desenvolver e realizar manutenções corretivas e evolutivas. Assim sendo, esta abordagem é interessante em aplicações cujos ambientes possuam diferentes restrições de funcionamento. Quando as restrições de elasticidade são controladas em função de requisitos de qualidade que não necessitam de estratégias diferenciadas, não há a necessidade de uso de diretivas dinâmicas dentro do código dos programas, pois isso só aumentaria o *overhead* a execução da aplicação.

Como pôde ser visto, a maioria das abordagens citam que a elasticidade deve ser alcançada de forma transparente e automática [12][15][17][19][26][32]. Também foram abordados o uso do padrão de projeto *publisher/subscriber* [9][18][32] para alcançar a escalabilidade e o desacoplamento de produtores e consumidores. Nas abordagens citadas em [15][18][26][17] 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 tampouco com as assimetrias de distribuição dos dados para alcançar o paralelismo de consumo.

O estudo analítico dos trabalhos relacionados aos temas tolerância a falhas e elasticidade constituiu uma importante ferramenta de apoio, necessária para a realização de uma proposta de solução de *middleware*, eficiente e eficaz, que será apresentada no Capítulo 3, juntamente com a descrição de um sistema distribuído em *cluster* de alto desempenho ao qual esse *middleware* se destina.

Capítulo 3

Middleware Proposto

O objetivo do *middleware* proposto é garantir tolerância a falhas e elasticidade para a aplicação distribuída de um *cluster* de alto desempenho que será mostrada na próxima seção. Assim sendo, logo após a descrição do sistema distribuído em estudo, será detalhado o projeto do *middleware* para suportar a tolerância a falhas e a elasticidade, o qual teve forte influência a partir das abordagens tratadas nos artigos científicos analisados no Capítulo 2.

3.1 Cenário em Estudo

Esta seção analisa um sistema distribuído em *cluster* de alto desempenho, que utiliza um *middleware* que realiza IPC por meio de filas de mensagens e *sockets*. Este sistema distribuído se destina ao monitoramento e ao rastreamento, por meio do posicionamento (latitude e longitude), de mais de 365.000 dispositivos embarcados instalados em veículos rodoviários. Neste ambiente, os veículos monitorados possuem dispositivos móveis embarcados capazes de obter a sua localização (latitude, longitude) por meio de satélites GPS (*Global Positioning System*), e se comunicarem via satélite ou rede celular com os servidores de comunicação por meio de um protocolo proprietário utilizando um número de conta e um endereço do veículo único, conforme mostrado na Figura 3.1. O *cluster* em estudo tem uma infraestrutura formada por 58 servidores, cada qual com quatro núcleos e quatro *gigabytes* de memória, totalizando 232 núcleos ao todo e 232 *gigabytes* de memória. Todos os 58 servidores executam o sistema operacional *Microsoft Windows* 2008 R2 64 *bits*, e são virtualizados a partir de quatro servidores BLADE 360c G8, cada um com dois processadores *Intel Xeon Quad Core* 3.0 GHz e com 96 *gigabytes* de memória. O sistema necessita de configuração e inicialização manuais, e possui 10 bases de dados *Microsoft SQLServer*, totalizando 1.7 *terabytes* de informação a partir de 15.031 contas de comunicação. Cada base possui um conjunto de servidores associado estaticamente,

e armazena um histórico de seis meses de dados, com mais de 2 bilhões de registros de posições, conforme mostrado na Tabela 3.1.

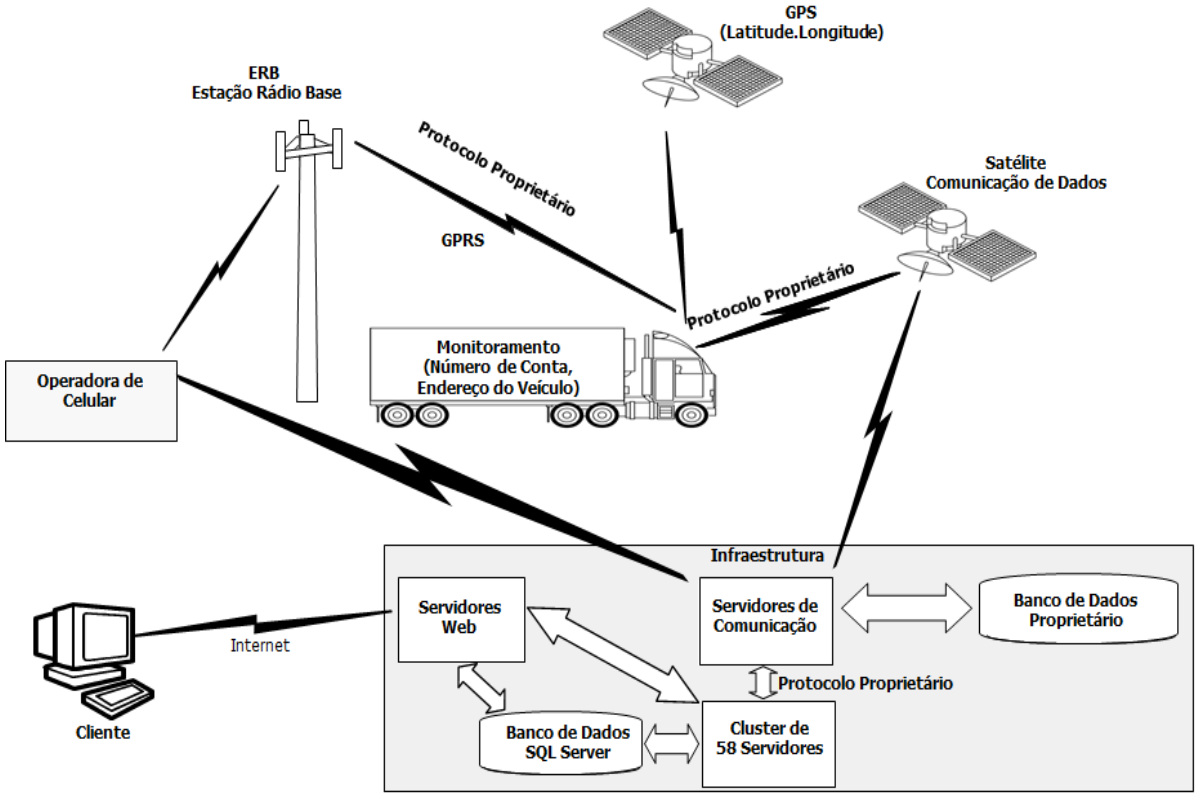


Figura 3.1: Infraestrutura do Sistema Distribuído.

Cada conta possui um sistema, numerado de uma a cinco. O sistema quatro é reservado, e não é utilizado para a troca de dados com o *cluster* de 58 servidores. O conceito de sistemas existe para agrupar equipamentos e dividir a carga de processamento. Dessa forma, como a quantidade de contas é muito grande (15.031), e há a necessidade de aumento da vazão na troca de dados, os sistemas reutilizam o canal de comunicação TCP estabelecido. Sendo assim, são aceitas conexões TCP simultâneas de contas agrupadas pelo seu número de sistema, sem a necessidade de um novo estabelecimento de conexão TCP, por meio do *three-way handshake* [30], entre contas diferentes.

Neste cenário, os servidores de comunicação recebem mensagens contendo o posicionamento dos veículos e as armazenam em uma base de dados proprietária. As mensagens dos veículos ficam associadas à conta que cada veículo se comunica, sendo que, um cliente pode possuir várias contas, e cada conta pode estar associada a vários veículos. As mensagens, que fazem parte da mesma conta com os servidores de comunicação, possuem uma restrição de processamento em que, para que não ocorram erros nas lógicas de regras de negócio que dependem de posicionamento sequencial (latitude e longitude), só podem ser consumidas, pelos processos que realizam seu tratamento, na ordem em que foram geradas

pelos dispositivos móveis. Esta importante característica recebe o nome de restrição de integridade sequencial [29].

Cada servidor do *cluster* realiza a obtenção de dados a partir dos servidores de comunicação por meio de *polling*, que é uma requisição de amostras de estado de maneira síncrona por um cliente. Esta requisição, chamada de cronograma ou ciclo de conexão, ocorre pela solicitação de dados por meio de um protocolo proprietário contendo a informação da conta que se deseja obter as mensagens e posições. Estas requisições ocorrem a cada 80 segundos, pois é a quantidade de tempo mínima para que um dispositivo móvel possa enviar uma mensagem contendo o seu posicionamento. Além disso, a cada *pooling*, são obtidas 300 mensagens de cada conta. Este mecanismo tem o objetivo de evitar *starvation* (inanição) de contas que possuem mais mensagens do que outras. Porém, mesmo com este mecanismo podem ocorrer rajadas de mensagens a partir de algumas contas que possuem um volume de dados maior. Assim, somente após a obtenção dos dados e a confirmação do recebimento e armazenamento na base de dados *SQL Server*, é que os servidores de comunicação apagam as mensagens persistidas no banco de dados proprietário. Esse mecanismo existe para evitar perda de dados não gravados por falhas no *cluster* de 58 servidores.

Tabela 3.1: Bases de Dados com Histórico de Seis Meses do Sistema Distribuído.

Banco de Dados	Tamanho (*Mbytes*)	Instâncias Lógicas	Posições	Contas	Veículos
BD1	294.941,50	8	460.005.267	1.823	57.730
BD2	276.255,88	11	383.133.207	4.141	66.103
BD3	239.518,00	4	236.692.312	1.904	63.406
BD4	225.487,19	10	260.621.400	3.095	70.966
BD5	137.003,75	3	122.688.469	1.056	32.145
BD6	36.864,00	2	36.609.157	221	6.491
BD7	9.695,75	1	178.696	42	249
BD8	161.418,38	5	222.614.618	1.309	22.354
BD9	212.302,75	8	309.203.587	1.791	33.094
BD10	126.976,00	6	66.800.758	1.553	13.069
Total	**1.720.463,20**	**58**	**2.098.547.471**	**15.031**	**365.358**

O tempo de cronograma de conexão por servidor do banco de dados BD2 e, as estatísticas durante um período de 6 horas, são mostrados na Tabela 3.2 e na Figura 3.2. Dessa forma, estes dados mostram que, durante a maior parte dos 80 segundos do cronograma de conexão, os servidores ficam ociosos, esperando para realizar um novo *pooling* de dados nos servidores de comunicação. Logo, é possível observar que existe margem para um melhor aproveitamento dos recursos dos servidores durante os períodos de ociosidade.

Tabela 3.2: Estatísticas do Cronograma de Conexão por Servidor do Banco de Dados BD2 Durante 6 horas.

Servidor	Cronograma (seg.)
espada101svw	5,41
espada102svw	9,14
espada103svw	5,42
espada104svw	5,22
espada105svw	4,18
espada106svw	3,51
espada107svw	4,95
espada108svw	4,66
espada109svw	6,60
espada110svw	4,54
espada111svw	3,56
Média	**5,20**
Mínimo	**0,04**
Máximo	**334,35**
Desvio Padrão	**12,08**

Cada base de dados da Tabela 3.1 possui um subconjunto de servidores responsáveis por seu tratamento. Cada um destes servidores é chamado de instância lógica. Dessa forma, é usada uma configuração em cada banco de dados para identificar este subconjunto de dados que cada instância lógica deve tratar. Esta é uma técnica utilizada para subdividir o volume de dados de uma banco em partes menores, tornando possível o processamento por um só servidor.

Cada servidor do *cluster* executa uma aplicação distribuída com 19 processos, sendo um deles o coordenador (*SharedMemory*), que realiza IPC (*Interprocess Communication*) entre os processos coordenados por meio de *sockets* TCP (*Transmission Control Protocol*) e filas de mensagens MSMQ (*Microsoft Message Queue*). Assim, cada processo é responsável por uma funcionalidade do sistema distribuído. Alguns desses processos possuem mais prioridade, pois tratam funcionalidades relacionadas à segurança, como a detecção de roubo do veículo e pedido de posicionamento do automóvel. A Tabela 3.3 mostra a função e a prioridade dos processos do *cluster*. Os demais processos do sistema possuem prioridade média. Além disso, alguns desses processos mantém um estado interno em memória relacionado à sua lógica de funcionamento, enquanto que outros são *stateless* (sem estado).

Nos processos da aplicação distribuída do *cluster* em estudo, existe uma camada de objetos distribuídos replicados que realiza *cache*, na carga inicial do sistema, dos dados

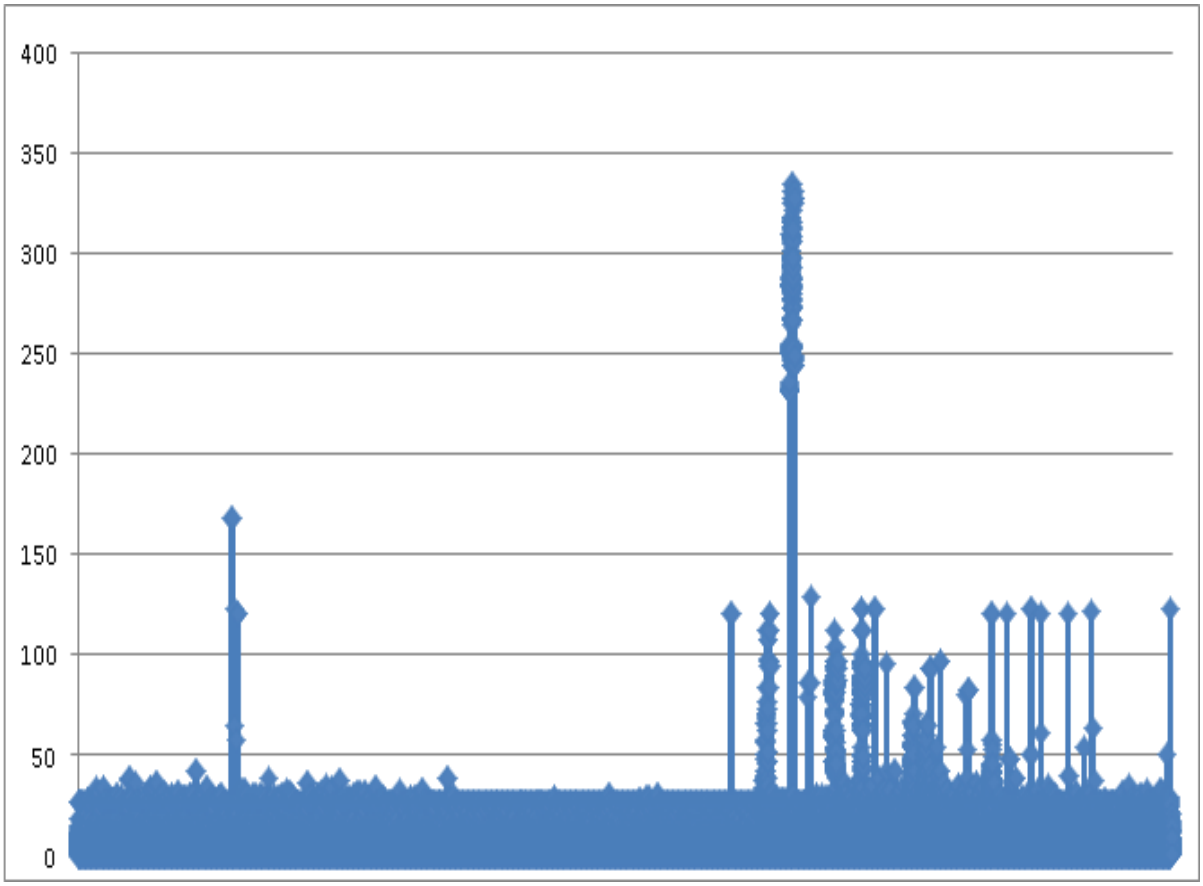


Figura 3.2: Tempo do Cronograma de Conexão dos Servidores do Banco de Dados BD2 Durante 6 Horas.

Tabela 3.3: Prioridades dos Processos do *Cluster*.

Processo	**Função**	**Prioridade**
TripManagement	Controle de Viagens, Inversão de Rota, Fora de Região	Altíssima
VehicleManagement	Controle de Pedido de Posição, Perda de Sinal	Alta
EventManagement	Expiração de Eventos	Média Alta
C2C	Operações entre Clientes	Média Média
SharedMemory	Coordenador	Média Baixa
Demais Processos	Várias Funcionalidades	Média

que os processos acessam do banco de dados. Isso evita que as consultas SQL, durante o funcionamento, onerem o servidor de banco, o que causaria lentidão de acesso pelos servidores *Web*. Dessa forma, as atualizações dos registros da base de dados, que estão na memória dos processos, são realizadas pelo processo coordenador em regiões críticas, por meio de um protocolo proprietário de consistência de *cache*. Assim, o coordenador sempre envia as atualizações realizadas para todas as cópias replicadas, após serem persistidas

no banco de dados.

Além disso, no *cluster* em estudo, quando é necessário realizar manutenções corretivas ou evolutivas é preciso reiniciar todos os servidores do *cluster* manualmente, além de realizar o balanceamento de carga estaticamente em todas as bases de dados. Assim, um outro problema surge quando o sistema sofre uma interrupção por queda de um conjunto de servidores, pois neste caso toda a operação de um subconjunto de usuários é afetada pelo tempo necessário que a carga inicial demorar, pois em média leva-se 20 minutos para isso. Para a manutenção evolutiva ou corretiva é necessário o agendamento de um *downtime* que é reportado a todos os seus usuários. Tanto a interrupção por queda como os *downtimes* agendados afetam o SLA (*Service Level Agreement*) com os usuários do sistema, prejudicando a operação funcional por conta da paralisação total ou parcial do sistema distribuído.

Neste ambiente em estudo, existe uma homogeneidade tanto na configuração de *hardware* dos servidores, quanto na rede e nos sistemas operacionais. As próximas seções descrevem uma proposta de solução de *middleware* que possui seu foco no uso de *clusters* de alto desempenho com filas de mensagens para comunicação entre os processos.

3.2 Tolerância a Falhas

No *middleware* proposto, foi utilizado somente o suporte às características de tolerância a falhas de *software* por omissão do processo, não sendo consideradas as falhas de *hardware*. Segundo [25], a replicação pode ser usada para detectar e mascarar falhas, entretanto, no *middleware* proposto são tratadas somente as falhas dos processos 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. Assim sendo, serão apresentadas a seguir as características do *middleware* proposto para suportar a tolerância a falhas.

3.2.1 Características

No *middleware* proposto foi implementado o suporte à tolerância a falhas de *software* na biblioteca de comunicação e de forma transparente à aplicação [6][21]. Para isso, foi utilizada uma abordagem tal que, quando um nó falha, não é afetada a integridade dos demais nós [6][33], pois os dados são persistidos em filas de mensagens independentes por meio do processo coordenador, que realiza o papel de *publisher* para os processos que são os *subscribers*, sendo sinalizados por meio de eventos de existência de itens a serem consumidos nas filas de mensagens.

Para a implementação do *middleware* com tolerância a falhas, também foi usado IPC com *sockets* TCP para obter entrega confiável (retransmissão, controle de fluxo, fragmentação), ordenação total de pacotes [35] e detecção de falhas de nós por meio da queda da conexão [3][21]. Contudo, o uso de *pings* periódicos com *timeouts* [21], foi uma abordagem não utilizada, pois oneraria o canal de comunicação com pacotes, uma vez que a queda do nó já é detectada por meio da queda da conexão TCP. Assim, o coordenador, por meio dessa queda de conexão, detecta a falha de um nó, e reinicia rapidamente o processo.

Além disso, foram utilizados *locks* no processo coordenador ao persistir e rotear os dados dentro de regiões críticas, antes de escrever nas filas de mensagens, para garantir a integridade sequencial e a consistência do *cache* das informações de banco realizadas na carga inicial da aplicação distribuída. Para isto, no *middleware* proposto foi usada a arquitetura de publicação de novos dados pelos produtores, com os dados persistidos pelo coordenador nas filas de mensagens [9], após serem gravados em um banco de dados.

A técnica de replicação de nós também pode ser utilizada para suportar tolerância a falhas e, assim, manter o sistema distribuído funcional, por meio de uma implementação *stateless* com baixo acoplamento e dependência entre os serviços, para que, caso um processo falhe, não afete os demais. Dessa forma, ao ocorrer queda ou travamento de algum processo a operacionalidade da aplicação pode ser facilmente recuperada e realizada por outro grupo de serviços replicados que acessam os mesmos dados [13]. Assim, em caso de falha do coordenador principal, coordenadores auxiliares podem ser iniciados rapidamente [1][3] e responder às requisições dos processos, somente em caso de falha ou queda. Da mesma forma, os processos coordenados, caso detectem uma falha de comunicação com o coordenador, podem esperar até que o novo coordenador inicie e responda às requisições.

Todavia, no cenário da aplicação distribuída em estudo, uma arquitetura onde ocorre maior rapidez durante a carga inicial do sistema, devido às otimizações das consultas no banco de dados, bem como na comunicação entre os processos, torna possível a adoção de mecanismos de reinício rápido dos processos em caso de falha ou queda [3][6][13][33]. Além disso, essa abordagem não possui o custo de memória adicional associado à processos replicados, e nem a latência de inicialização em caso de falha, sendo, portanto, essa a técnica utilizada no *middleware* proposto para suporte a esta característica de tolerância a falhas.

Para isso, inicialmente, foram medidos os tempos de carga atuais dos servidores que tratam as 11 instâncias lógicas do banco de dados BD2 da Tabela 3.1, que são mostrados na Tabela 3.4. Como pode ser observado, para suportar os mecanismos de recuperação rápido de falhas citados é necessário uma melhoria desses tempos. Dessa forma, foram realizadas, no *middleware*, as modificações arquiteturais a seguir, visando a diminuição

desse tempo:

Tabela 3.4: Tempo de Inicialização do *Cluster* antes das Otimizações Implementadas.

Servidor	Tempo do Sistema Atual
*instance*1	4'10"
*instance*2	12'05"
*instance*3	8'59"
*instance*4	15'06"
*instance*5	12'11"
*instance*6	15'37"
*instance*7	15'43"
*instance*8	12'51"
*instance*9	4'56"
*instance*10	4'32"
*instance*11	16'02"
Total	**16'02"**

- Redução do Número de Consultas - diminuição do número de consultas ao banco de dados que possuem resultados pequenos. Dessa forma, é retornado o máximo de registros por meio do menor número de consultas, possibilitando ao banco de dados realizar a transferência de dados sem interrupções causadas por consultas que retornam poucos registros e, assim, aumentando a vazão de preenchimento da memória dos serviços.
- Arquitetura de Cópia de Objetos - alteração na arquitetura de cópia de objetos, por meio da sua criação e preenchimento diretamente a partir dos resultados retornados pelo banco de dados. Desta forma, são evitadas diversas cópias de objetos e criação de listas intermediárias, o que aumenta a vazão de preenchimento do resultado definitivo em memória.
- Serialização dos Objetos - a serialização dos objetos que serão enviados para os processos é feita a medida em que os resultados do banco são preenchidos nos objetos em memória. Dessa forma, não é necessário percorrer posteriormente listas intermediárias com os retornos das consultas para enviar os dados para os processos que possuem *cache* destas informações.

Sendo assim, o objetivo da implementação dessas abordagens no *middleware* é a redução do tempo de inicialização dos serviços da aplicação distribuída, necessário para suportar os mecanismos de reinício rápido de processos como estratégia de tolerância a falhas.

Além disso, para os processos que possuem estado na aplicação distribuída, o uso da abordagem de *snapshots* periódicos, juntamente com a detecção de falha por meio da queda da conexão TCP, entre os processos e o coordenador, é um interessante mecanismo de tolerância a falhas. Os *snapshots* podem ser persistidos em um banco de dados para que, em caso de detecção de falha, algum outro serviço assuma seu processamento recuperando o último *checkpoint* válido, mas sem o envio constante dos estados para os processos replicados, para evitar que ocorra sobrecarga de comunicação. Porém, uma arquitetura que persiste os estados em um banco de dados, limita a vazão do IPC, onerando o desempenho dos serviços em casos de alta carga de processamento.

Assim, optou-se por uma arquitetura com o envio rápido de pacotes por IPC para alimentar as estruturas em memória que contenham as informações necessárias dos estados de cada processo, como suporte a esta característica de tolerância a falhas. Para isso, além da melhoria na serialização dos objetos já descrita, também foi realizada uma otimização na arquitetura de IPC que é realizada por meio de *sockets*. Como consequência, foi criada uma estrutura de dados chamada de *dual buffer*. O *dual buffer* utiliza um mecanismo eficiente de armazenamento de pacotes por meio de dois ponteiros para listas em memória, uma de leitura e outra de escrita. Esses ponteiros são trocados sempre que a lista de leitura está vazia. A escrita dos pacotes no *dual buffer* é realizada na *thread* de leitura das mensagens a partir do *socket*. A leitura de pacotes a partir do *dual buffer* é realizada na *thread* de tratamento das mensagens, e realiza a troca dos ponteiros das listas de leitura e escrita, caso a lista de leitura esteja vazia. Somente neste momento ocorre a serialização da estrutura de dados por meio de *lock*. Como o tratamento da mensagem requer processamento, tende a ser mais lento do que a leitura do pacote a partir do *socket* e, portanto, os momentos de serialização são pequenos, pois são realizados somente quando a lista de leitura está vazia. Dessa forma, o *dual buffer* é uma estrutura de dados quase *lock free* [34]. O acesso concorrente às estruturas de dados *lock free* é realizado sem serializações, enquanto que nas estruturas quase *lock free* os momentos de serialização existem, porém ocorrem em seções críticas muito curtas. Dessa forma, são evitados, nos casos de picos de comunicação com muitas trocas de pacotes, os excessos de sinalização de espera por escrita nos *buffers* da conexão TCP, o que gera diminuição da vazão de comunicação entre os processos.

As características com suporte à elasticidade que foram implementadas no *middleware* serão vistas na próxima seção.

3.3 Elasticidade

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 [20]. Dessa forma, para alcançar esse objetivo no *middleware* proposto, foram utilizadas diversas características tendo por base os artigos relacionados na seção 2.2. Essas características serão apresentadas a seguir.

3.3.1 Características

O uso da carga dinâmica de códigos que possuem funcionalidades de um grupo de serviços específico [19], é uma abordagem interessante nos casos em que qualquer serviço pode processar qualquer tipo de mensagem por meio da sua subscrição no coordenador, e da carga da DLL (*Dynamic Link Library*) que possui o código que realizará este processamento. Assim, porções de códigos podem ser ativadas para qualquer processo consumidor e estes podem estar executando em servidores distintos. Quando uma DLL é carregada, pode ser criada uma nova *worker thread*, que é um processo consumidor que processa e remove requisições de uma fila [24], para processar o novo conjunto de filas associado. Os processos consumidores podem agrupar porções de código distintas que tratam o mesmo grupo de mensagens. Isso reduz a latência de início de novos processos, além de representar uma economia no número total de processos necessários para rodar o sistema distribuído.

Todavia, utilizar vários *handlers*, que são manipuladores que fornecem um contexto para a execução de instruções do programa [24], do mesmo tipo em diferentes *threads*, pode gerar serializações e aumentar a latência no tratamento das mensagens. Assim, no *middleware* proposto, foi utilizada a abordagem de escrever em filas diferentes, mesmo possuindo *handlers* para o mesmo tipo de objeto e tratando funcionalidades distintas. Essa abordagem foi utilizada por se mostrar uma arquitetura que favorece a elasticidade, permitindo paralelizar o processamento das mensagens, por meio do aumento de *worker threads*, tornando o tratamento da demanda mais rápido.

Dessa forma, é possível explorar essa característica de várias filas de mensagens separadas por conta, de forma que, ao iniciar vários processos consumidores, ocorra o aumento da vazão devido ao paralelismo de consumo. Porém, criar filas por conta tornam o gerenciamento e o monitoramento onerosos, pois no sistema estudado há um grande número de contas existentes. Como são 19 filas, uma para cada processo consumidor, existiriam 57.000 filas de mensagens para 3.000 contas, em cada um dos servidores do *cluster* da aplicação distribuída, por exemplo.

Nesse cenário, o uso de subfilas se tornam uma opção arquitetural útil, pois são utilizadas em processamento ordenado de itens que podem ser agrupados para melhorar a eficiência. De acordo com a vazão de consumo, as filas de mensagens podem ser dinamicamente distribuídas em novas instâncias de subfilas, as quais podem ser criadas para escalar quando forem requeridas, ou redistribuídas em um subconjunto menor quando a carga diminuir. Também podem ser utilizadas para a manipulação de mensagens que apresentam exceções em seu tratamento. Dessa forma, a aplicação pode mover essas mensagens para a subfila de falhas para análise posterior. Assim sendo, foi implementado no *middleware* proposto somente a característica de subfilas para o aumento da eficiência.

Assim sendo, suponha que uma determinada fila possua 9.000 mensagens, e que o coordenador detecte a necessidade de ativar três processos para alcançar a vazão necessária, para que não ocorra o enfileiramento de mensagens. Nesta fila, se encontram 2.000 mensagens da conta A, 3.000 mensagens da conta B, e 4.000 mensagens da conta C. Assim, o coordenador pode utilizar essa abordagem para redistribuir os dados em três filas distintas, uma para cada novo processo, sem perder a integridade sequencial, e mantendo o aumento de vazão no consumo.

Dessa forma, por meio da criação de processos, os melhores candidatos ao consumo de novos itens podem se basear no menor consumo médio de CPU e na maior vazão média identificada [18]. O parâmetro utilizado pelo *middleware* para a implementação dessa abordagem foi o tempo médio de consumo de cada item de dado na fila de mensagens [17], por conta de comunicação. Porém, a coleta de estatísticas da vazão média de mensagens e do consumo de CPU gera uma comunicação intensa entre o coordenador e os processos consumidores. Para resolver isso, foi implementado no *middleware* uma arquitetura em que são criadas subfilas por conta, e *worker threads* em alternativa à filas de mensagens por conta associadas a processos consumidores. Essa arquitetura é mostrada na Figura 3.3.

Diante do exposto, optou-se por implementar no *middleware* proposto, um algoritmo que se adapta dinamicamente à carga, isolando os produtores e os consumidores mais importantes, e redireciona-os para novas instâncias de subfilas criadas, a partir dos mesmos dados com o objetivo de aumentar a vazão. O *middleware* é o responsável pelo agrupamento das mensagens em subfilas correlacionadas e pela atribuição das filas às *worker threads* que realizam o seu consumo.

Como citado em [9][18][32], um sistema com múltiplas filas de consumo pode gerar assimetrias de distribuição, o que leva a um desbalanceamento do sistema ao consumir as mensagens dessas subfilas. Dessa forma, foi codificado no *middleware* proposto a ordenação das contas, no momento em que são criadas ou removidas *worker threads*, iniciando na que tem o maior número de mensagens até a que tem o menor número. Em seguida é

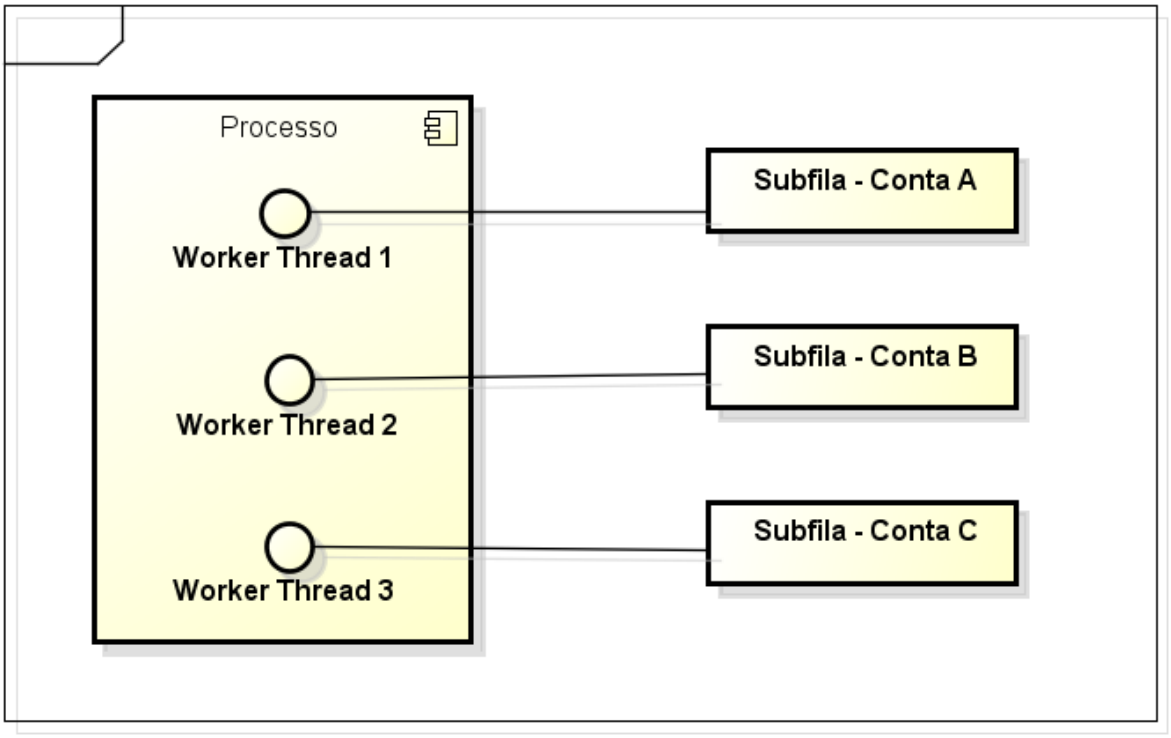


Figura 3.3: *Worker Threads* para Consumo das Mensagens das Subfilas Separadas por Conta.

realizada a distribuição dessas contas entre as *worker threads* utilizando o algoritmo *round robin*. Além disso, o consumo na mesma *worker thread* foi implementado utilizando 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. Também foi utilizado o algoritmo *First Come First Served* [31], para preservar a integridade sequencial no consumo das mensagens que pertencem à mesma conta de comunicação. Dessa forma, foi implementado no *middleware* um algoritmo para consumir as mensagens, atendendo as seguintes características arquiteturais:

- *First Come First Served* - os pacotes são ordenados por ordem de chegada com *timestamp* em milisegundos. Assim, os pacotes mais antigos são tratados primeiro.
- *Round Robin* - um pacote de cada conta por rodada. Caso não haja pacotes da conta na rodada, é processado o de uma conta que não tenha sido tratada ainda nesta rodada.
- *Fair Queueing* - tratamento justo, onde cada conta só tem um pacote tratado por rodada independente de quantos pacotes da mesma conta estejam enfileirados.
- *Subqueues* - pacotes separados por conta em subfilas distintas.

- *Worker Threads* - criação de *worker threads* para aumentar a vazão e o paralelismo de consumo.

Assim, o consumo justo por ordem de chegada e com *round robin* tem o objetivo de realizar o tratamento de no máximo uma mensagem de cada conta por rodada. Essa arquitetura de estrutura de dados visa transformar o número de mensagens das contas em uma distribuição normal, para que seja possível detectar rajadas de mensagens vindas de contas específicas por meio do distanciamento da vazão média em relação ao desvio padrão.

Além disso, o processo coordenador, baseado no número de mensagens das filas e no consumo médio de cada mensagem, determina se a fila será totalmente consumida no intervalo do tempo do ciclo de conexão, e inicia mais *worker threads*, se necessário, para consumir os dados em paralelo. O objetivo dessa ação é conter o crescimento e zerar a fila de mensagens. Porém, o aumento de *worker threads* só ocorre caso o servidor tenha CPU disponível.

Nas próximas seções, para demonstrar as características de análise da vazão implementadas no *middleware* desenvolvido, foi utilizado um estudo de caso de enfileiramento de mensagem, onde foi necessário aumentar o número de *worker threads* para conter rajadas. Da mesma forma, após a diminuição do número de mensagens, é mostrado como o *middleware* realiza a diminuição do número de *worker threads*. Também serão explicadas as características de implementação da limitação de criação de *worker threads*, por CPU e desvio padrão da vazão média de saída por conta, bem como a recuperação de falhas com elasticidade.

3.3.2 Análise da Vazão Média de Entrada e Saída

Para obter o número de *worker threads* que serão necessárias para conter o crescimento e zerar as subfilas, é necessário calcular a vazão média de entrada e saída, e a média da relação entre as mensagens de entrada e saída, a partir do momento da detecção de crescimento da fila até o *Entry Point* (EP), que é o momento onde são criadas novas *worker threads*.

Outra importante definição é o *Growth Detection* (GD), que é a detecção do crescimento da fila de mensagens. Isso ocorre quando a relação entre as mensagens de entrada e de saída torna-se maior do que um, gerando enfileiramento. Ela é detectada pela Fórmula 3.1,

$$\left(\frac{Input}{Output} > 1\right) \tag{3.1}$$

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

O *Scale Up* (SU) ocorre no EP, e corresponde ao momento em que novas *worker threads* são criadas para conter uma rajada de mensagens que gera enfileiramento, e que não pode ser tratada antes do fim do tempo máximo de tratamento de mensagens, que é o cronograma de conexão. O EP pode ser encontrado quando o número de mensagens na fila é maior que o valor dado pela Fórmula 3.2,

$$(avgTOUT \times (schTIME - (hNOW - hSTART))) \tag{3.2}$$

onde, a variável $avgTOUT$ é a vazão média de saída, $schTIME$ é o tempo máximo para o tratamento de mensagens na fila (cronograma de conexão), $hNOW$ é o tempo atual, $hSTART$ é o tempo de início onde o valor dado pela Formula 3.1 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 Fórmula 3.3,

$$QueueSize < \left(\frac{avgTIN}{2}\right) \tag{3.3}$$

onde, $QueueSize$ é o tamanho atual da fila, e $avgTIN$ é a vazão média de entrada desde o EP.

Em adição às *Contention Threads* (CT), que são as *worker threads* de contenção de rajadas, um outro conjunto de *worker threads* é necessário, as *Zero Threads* (ZT), para consumir as mensagens que se acumularam na fila desde o GD até o EP, como mostrado na Figura 3.5 e na Tabela 3.6. Dessa forma, são criadas várias *worker threads*, do tipo ZT ou CT, sendo que cada uma delas está associada a várias subfilas de mensagens, para que ocorra o aumento da vazão de consumo por meio do paralelismo de consumo. Esta arquitetura é mostrada na Figura 3.4, onde são criadas quatro *worker threads* do tipo CT e uma do tipo ZT.

Scale Up

Na Tabela 3.5, a coluna *Time* corresponde ao tempo, em segundos, de cada medição; $avgTIN$ é a vazão média de entrada; $avgTOUT$ é a vazão média de saída; $avgIO$ é a média da relação entre as mensagens de entrada e de saída; $QueueSize$ é o número de mensagens na fila; *Entry Point* corresponde ao valor dado pela Fórmula 3.2.

O cálculo do número total de *worker threads* inciadas no SU, é mostrado no Algoritmo 1. Quando o $QueueSize$ é maior que o EP, linha 2, o valor de $avgIO$ é usado para encontrar o número de CT *worker threads*, linha 3. A variável $avgTOUT$ dividida pelo número total de *worker threads* resulta na vazão média por *thread*, linha 4, que é utilizada como a taxa

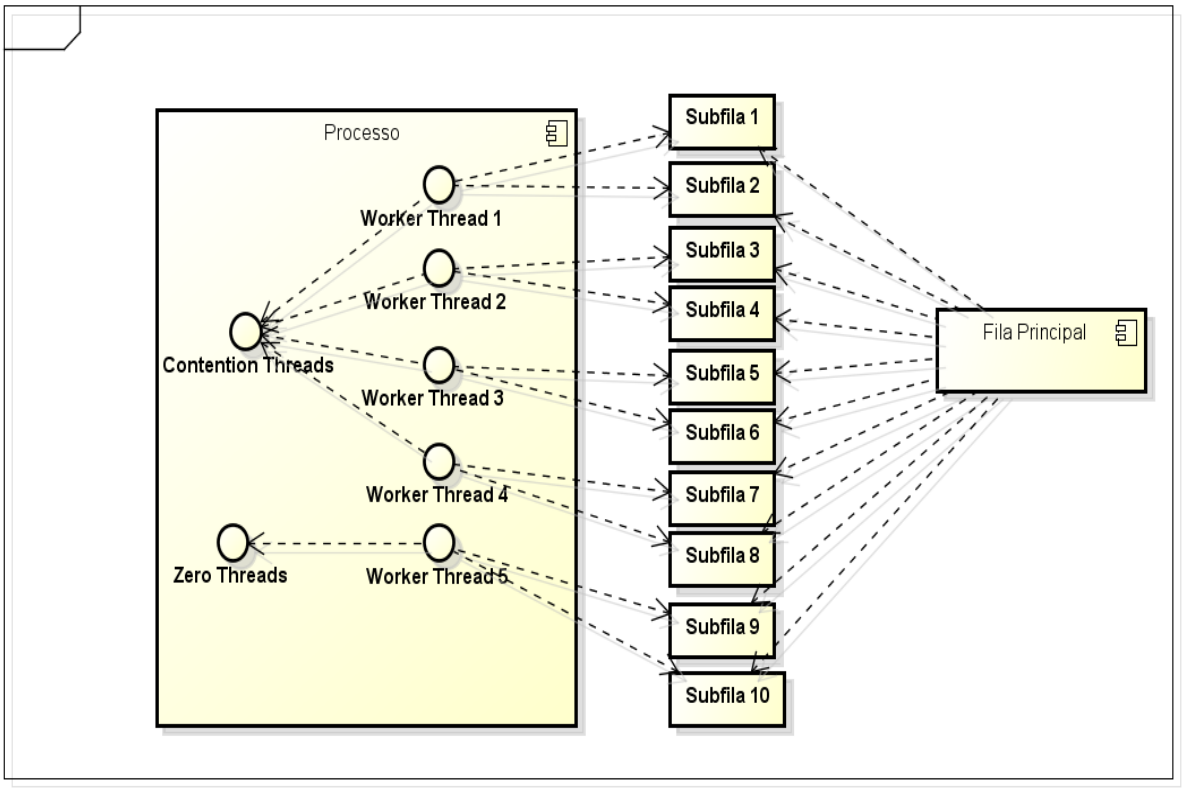


Figura 3.4: *Worker Threads* do Tipo *ZT* e *CT* Associadas às Subfilas.

de consumo, linha 5, para obter o número de *ZT worker threads*, linha 6, necessárias para zerar as mensagens que se acumularam.

Algoritmo 1: Cálculo do Total de *Worker Threads* Iniciadas no *SU*.

```
1 EntryPoint = ( avgTOUT ) * ( schTIME - ( hNOW - hSTART ) );
2 if QueueSize > EntryPoint then
3 |   ContentionThreads = avgIO;
4 |   ThroughPutByWorkerThread = ( avgTOUT / TotalWorkerThreads );
5 |   ConsumptionRate = QueueSize / ( schTIME - ( now - hSTART ) );
6 |   ZeroThreads = ConsumptionRate / ThroughPutByWorkerThread;
7 |   TotalThreadsToStart = ContentionThreads + ZeroThreads + 1;
8 end if
```

Para evitar o problema do limiar de detecção de elasticidade para cima [15], o *middleware* inicia uma *worker thread* além do somatório de *CT* com *ZT*, linha 7. Assim, como mostrado na Tabela 3.6, são iniciadas seis *worker threads*. O momento do *SU* é mostrado na Figura 3.6.

Tabela 3.5: Antes do *Scale Up*.

Time	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*	*Entry Point*
0	18.00	4.00	4.50	17	320
1	40.50	9.50	4.35	63	750
2	50.25	12.25	4.17	110	955
3	55.63	13.63	4.12	156	1049
4	58.31	14.31	4.09	202	1087
5	60.66	14.66	4.15	250	1098
6	61.83	14.83	4.17	298	1097
7	62.41	14.91	4.19	346	1088
8	62.71	14.96	4.19	394	1076
9	62.85	14.98	4.20	442	1063
10	63.43	14.99	4.23	490	1049
11	63.21	14.99	4.22	538	1034
12	63.11	15.00	4.21	586	1019
13	63.05	15.00	4.20	634	1004
14	62.03	14.50	4.28	679	956
15	62.51	14.75	4.24	727	957
16	62.76	14.87	4.22	775	951
17	62.88	14.94	4.21	823	940
18	62.94	14.97	4.21	871	927
19	**62.94**	**14.97**	**4.20252**	**919**	**912**

Tabela 3.6: Scale Up.

Start Time	*Reaction Time*	*ZeroThreads*	*ContentionThreads*	*TotalThreadsToStart*
17:08:19	17:08:38	1.00542	4.20252	6

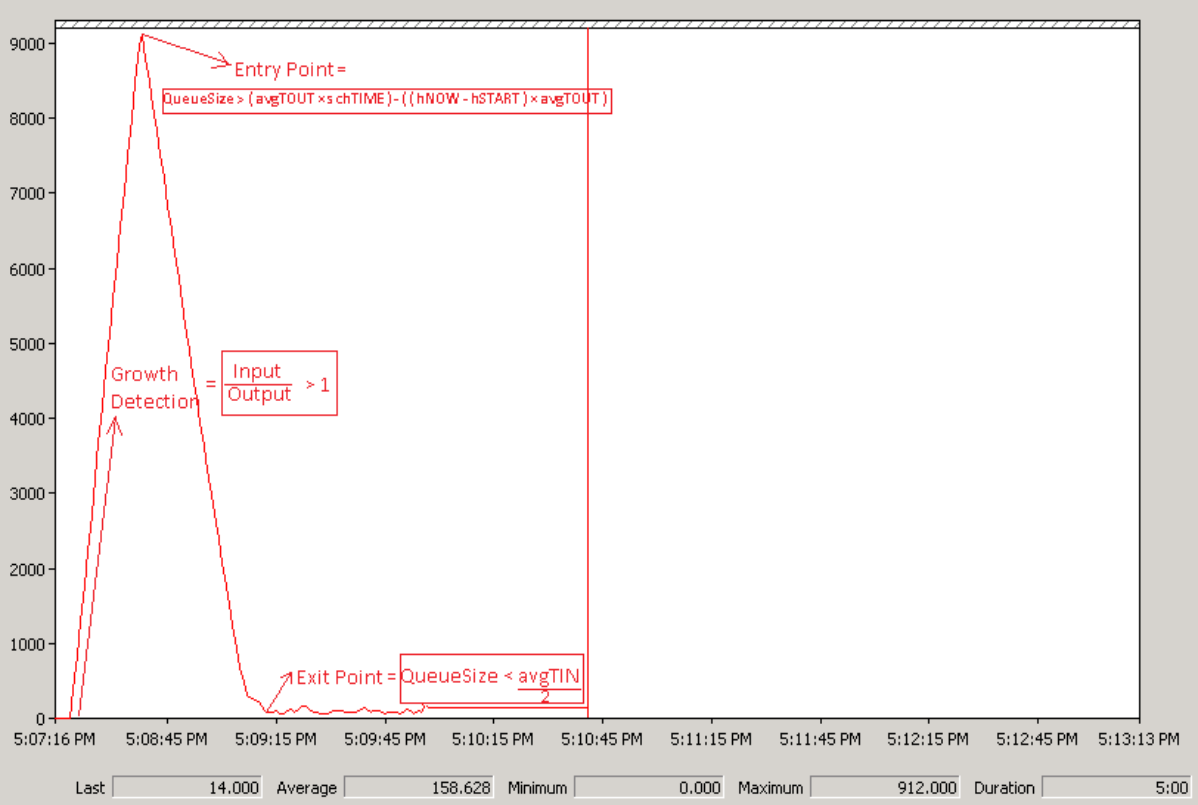


Figura 3.5: Análise da Vazão Média de Entrada e Saída na Fila de Mensagens.

Scale Down

Na Tabela 3.7, a coluna *Time* corresponde ao tempo, em segundos, de cada medição; *Input* é o número de mensagens de entrada; *Output* é o número de mensagens de saída; *avgTIN* é a vazão média de entrada; *avgTOUT* é a vazão média de saída; *avgIO* é a média da razão entre as mensagens de entrada e saída; *QueueSize* é o número de mensagens na fila.

O cálculo do número de *threads* que são removidas no *SD*, é mostrado no Algoritmo 2. Para realizar o *SD*, é necessário que o valor dado pela Fórmula 3.3 seja verdadeiro, conforme mostrado na linha 1. A divisão de *avgTIN* por dois destina-se a evitar o problema da detecção do limiar de elasticidade para baixo [15], o que faz com que as filas tenham uma nova tendência crescimento. Em seguida, é calculada a vazão por *thread*, linha 2, que é utilizada no cálculo de *threads* a serem removidas, linha 3. Novamente, para evitar o fenômeno do limiar de elasticidade para baixo, é removida uma *thread* a menos, linha 4. O momento de *SD* pode ser visto na Figura 3.5.

De acordo com a Tabela 3.7, após o *SU* visto na seção anterior, a vazão média de saída é maior do que a vazão média de entrada, fazendo com que o crescimento da fila seja contido, e diminua para um valor próximo de zero. Após 49 segundos, foi detectado

Figura 3.6: Análise dos Momentos de *Scale Up* e *Scale Down*.

que o *QueueSize* (26) é menor do que o *avgTIN* (63,00), dividido por dois (31,50), tal como na Fórmula 3.3. Assim, de acordo com a Tabela 3.8, foi decidido realizar *SD* retirando uma *worker thread*, com base na vazão média de saída por *thread*, como mostrado na Figura 3.6.

Algoritmo 2: Cálculo do Total de *Threads* Removidas no *SD*.

```
1 if QueueSize < ( avgTIN / 2 ) then
2 |   ThroughPutByThread = (avgTOUT / TotalWorkerThreads);
3 |   ThreadsToRemove = TotalWorkerThreads - (avgTIN / ThroughPutByThread);
4 |   ThreadsToRemove = ThreadsToRemove - 1;
5 end if
```

3.3.3 Análise do Consumo de CPU

Conhecer o consumo médio de CPU por *worker thread* pode ser útil para determinar limites para o *SU*. 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

Tabela 3.7: Antes do Momento de *Scale Down*.

Time	*Input*	*Output*	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*
20	63	91	63.00	91.00	0.69	889
21	63	95	63.00	93.00	0.68	857
22	63	95	63.00	94.00	0.67	825
23	63	95	63.00	94.50	0.67	793
24	63	95	63.00	94.75	0.66	761
25	63	95	63.00	94.88	0.66	729
26	63	95	63.00	94.94	0.66	697
27	63	95	63.00	94.97	0.66	665
28	63	95	63.00	94.98	0.66	633
29	63	95	63.00	94.99	0.66	601
30	64	95	63.50	95.00	0.67	569
31	63	95	63.25	95.00	0.67	537
32	63	95	63.13	95.00	0.66	505
33	63	95	63.06	95.00	0.66	473
34	63	95	63.03	95.00	0.66	441
35	63	95	63.02	95.00	0.66	409
36	63	95	63.01	95.00	0.66	377
37	63	95	63.00	95.00	0.66	345
38	63	95	63.00	95.00	0.66	313
39	61	89	62.00	92.00	0.67	283
40	64	101	63.00	96.50	0.65	249
41	63	95	63.00	95.75	0.66	217
42	63	95	63.00	95.37	0.66	185
43	63	90	63.00	92.69	0.68	162
44	63	90	63.00	91.34	0.69	132
45	63	87	63.00	89.17	0.71	108
46	63	92	63.00	90.59	0.70	79
47	63	81	63.00	85.79	0.74	63
48	63	88	63.00	86.90	0.73	39
49	**63**	**74**	**63.00**	**80.45**	**0.79**	**26**

Tabela 3.8: Scale Down.

StartTime	*ReactionTime*	*avgTIN*	*avgTOUT*	*Throughput*	*RemoveThreads*
17:08:19	17:09:08	63.50	96	16.08	1

pode ser atendido, sem a saturação de uso de CPU do servidor. Esta situação é mostrada no Algoritmo 3.

Primeiramente, baseado no número de *threads*, que o *middleware* decidiu criar para atender a vazão necessária, e no consumo médio de CPU por *thread*, é calculado o aumento previsto do consumo de CPU, linha 1. Em seguida, é calculada a CPU residual, baseada no consumo médio de CPU do servidor desde o início da rajada de mensagens, e em um limite configurado (*GetCpuThreshold*), mostrado na linha 2. Caso o valor da previsão do crescimento de CPU, pelo aumento de *worker threads*, seja maior do que a CPU residual calculada, a criação de *worker threads* é limitada pela divisão entre a CPU residual e a média de uso por *thread*, conforme mostrado nas linhas 3 e 4.

Algoritmo 3: Aumento de *Threads* Limitado pelo Consumo de CPU.

```
1 CpuToGrow = ( TotalThreadsToStart * GetThreadsMeanCpuUsage() );
2 ResidualCpu = GetCpuThreshold() - GetMeanCpuUsageSince( hSTART );
3 if CpuToGrow > ResidualCpu then
4 |   TotalThreadsToStart = ResidualCpu / GetThreadsMeanCpuUsage();
5 end if
```

3.3.4 Análise do Desvio Padrão da Vazão Média de Saída por Conta

As rajadas de mensagens de entrada podem vir de várias contas, bem como somente de algumas. Para atender as rajadas de n contas, o *middleware* decide criar várias *worker threads* de acordo a vazão média de saída, medida durante o crescimento da fila.

No entanto, quando as rajadas vem de poucas contas, 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* recursivo, sem resolver o problema de crescimento da fila.

Para resolver este problema, antes do *SU*, durante a rajada é realizada a medição da vazão média de saída por conta, que será utilizado no cálculo do desvio padrão no momento de realizar *SU*. Uma variável aleatória em uma distribuição normal tem 95% de chance de estar a menos de dois desvios-padrão de sua média [8], conforme mostrado na Figura 3.7. Assim, como a vazão média de saída foi normalizada em no máximo uma mensagem para cada conta por rodada, após n rodadas quando a média se distancia muito do desvio padrão, ou seja, mais de duas vezes, é possível detectar rajadas vindas de contas específicas. Esta situação é mostrada no Algoritmo 4.

Inicialmente, é calculado o número de contas que tenham vazão média de saída duas vezes maior que o desvio padrão, na linha 1. Dessa forma, se o número de *worker threads*

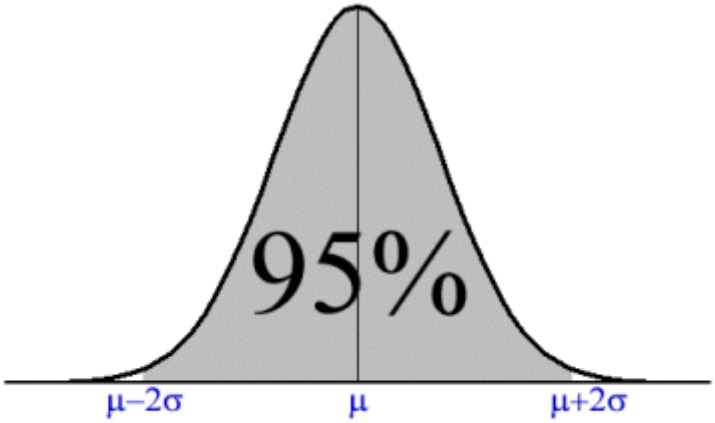


Figura 3.7: Probabilidade de 95% da Média estar no Máximo a Dois Desvios-Padrão.

a serem criadas com base na vazão, for maior do que o o valor de *accountsAboveStdDev*, linha 2, é criada uma quantidade menor de *worker threads* para tratamento específico das contas com rajadas de mensagens, conforme mostrado na linha 3.

Além disso, as contas que geraram o aumento de *worker threads* são armazenadas no *middleware*. Assim, se ocorrer uma nova rajada para a mesma conta, não ocorrerá *SU* recursivo. Dessa forma, se a fila continuar a crescer após o *SU*, este é um problema de otimização do algoritmo de tratamento para cada mensagem.

Algoritmo 4: Aumento de *Threads* Limitado pelo Desvio Padrão da Vazão Média de Saída por Conta.

```
1 accountsAboveStdDev = GetAccount2TimesAboveStdDev();
2 if TotalThreadsToStart > accountsAboveStdDev then
3 |   CreateThreads( accountsAboveStdDev );
4 else
5 |   CreateThreads( TotalThreadsToStart );
```

A próxima seção detalha como funciona o algoritmo de elasticidade, que utiliza a adaptação dinâmica de *worker threads* do *SU* e do *SD*, com limites de aumento baseados na análise de CPU e do desvio padrão, mostrados nas seções anteriores.

3.3.5 O Algoritmo de Elasticidade

Para o funcionamento do algoritmo de elasticidade foram desenvolvidas quatro classes no *middleware* proposto, como mostrado na Figura 3.8. Essas classes são estruturas de dados necessárias para o funcionamento de quatro tipos de *threads*, sendo elas: a *thread* principal que lê da fila de mensagens do processo, e a envia para uma *worker thread*; A

worker thread que trata as mensagens; A *thread* de coleta da vazão instantânea de entrada e de saída; E a *thread* de monitoramento da vazão média de entrada e de saída. A medida que mais ou menos vazão é requerida para o tratamento das mensagens, são iniciadas ou removidas *worker threads*. A quantidade das outras *threads* não sofre aumento ou diminuição. As próximas subseções explicam o funcionamento de cada uma das classes da Figura 3.8, bem como das *threads* do *middleware* proposto.

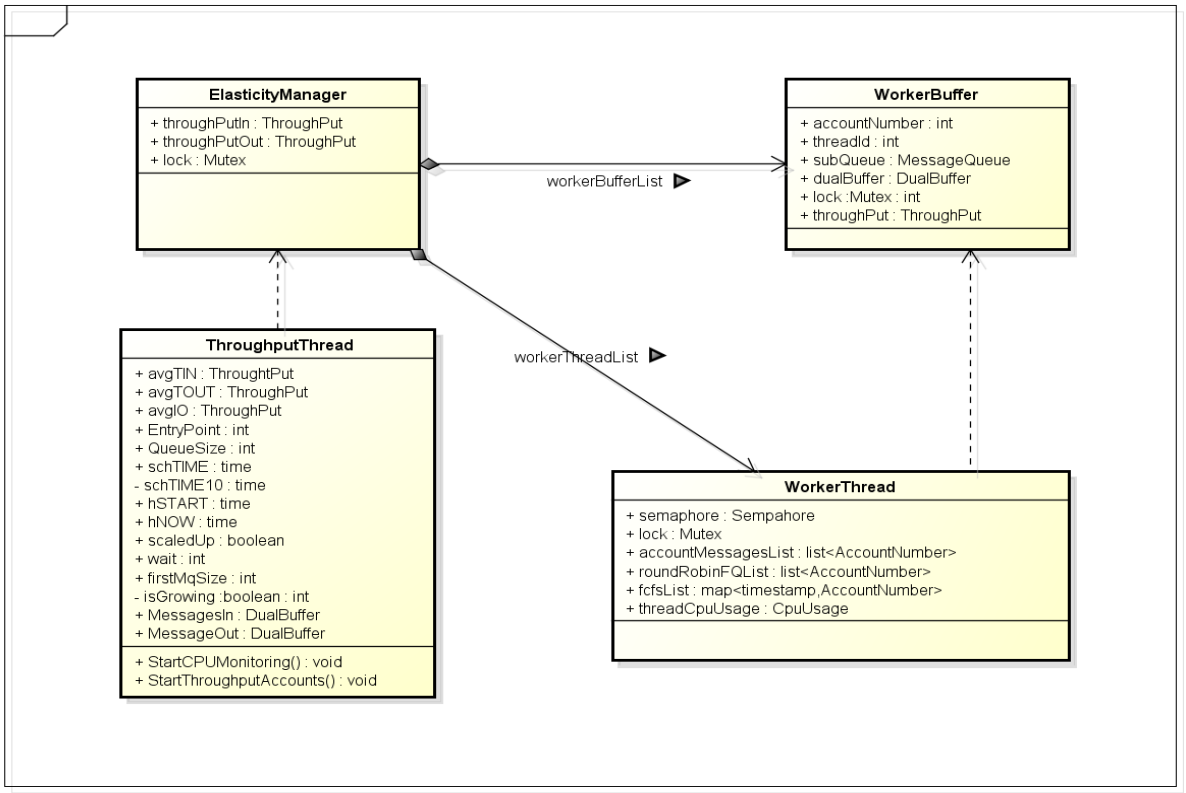


Figura 3.8: Diagrama de Classes para Suporte à Elasticidade.

Classe *WorkerBuffer*

A classe *WorkerBuffer* é responsável por guardar as informações relativas à conta, armazenada na variável *accountNumber*. Assim, para que a *worker thread* responsável pelo seu tratamento possa ser encontrada rapidamente, foi criado o campo *threadId*. Dessa forma, após ser encontrado o *WorkerBuffer*, as mensagens, que são lidas a partir da fila de mensagens principal, são movidas para a *subQueue* e escritas no *dualBuffer*, que será responsável por ler as mensagens. Nesse contexto, a *subQueue* existe apenas como um repositório persistente para os casos de queda ou falha do processo, pois a leitura efetiva da mensagem será feita pelo *dualBuffer*, por ser uma estrutura de leitura de mensagens mais eficiente. Assim, somente após a leitura e o tratamento da mensagem sem erros, a

mensagem é removida da *subQueue*. Esta classe também possui um *lock* para proteger a alteração de mudança de *thread*, e uma variável responsável pela medição da vazão de saída associada a conta, que é utilizada para os cálculos de desvio padrão da vazão média de saída. Na próxima subseção será visto como é realizado o tratamento das mensagens pela classe *WorkerThread*.

Classe *WorkerThread*

A classe *WorkerThread* é responsável pelo tratamento das mensagens que foram movidas para o *WorkerBuffer*. Para isso, ela possui um semáforo (variável *semaphore*) para sinalizar a necessidade de tratamento de pacotes pertencentes aos *WorkerBuffers*, por meio de uma lista de contas (*accountMessagesList*). Também possui um *lock* para a proteção da seção crítica de mudança da lista de contas tratadas. Dessa forma, ao ser sinalizada da existência de novos pacotes, primeiramente, são tratados os pacotes mais antigos por meio do algoritmo FCFS que são sinalizados em *fcfsList*. Além disso, para que ocorra um consumo justo, apenas uma mensagem de cada conta é processada por rodada, ou seja, utilizando o algoritmo *round robin* com *fair queueing* por meio do armazenamento da conta tratada em *roundRobinFQList*. Por fim, a variável *threadCPUUsage* é responsável pelas medições de consumo de CPU da *thread*, para uso nos limites de aumento de *threads* do *SU*. Em seguida será vista a classe *ElasticityManager*, que utiliza as classes *WorkerBuffer* e *WorkerThread*.

Classe *ElasticityManager*

A classe *ElasticityManager* é um *singleton*, que garante que a classe tenha somente uma instância por meio de um ponto único e global de acesso [24]. Ela possue as listas de todos os *WorkerBuffers* (*workerBufferList*) e das *WorkerThreads* (*workerThreadList*), bem como um *lock* para a proteger esses dados. Também possui as medidas de vazão geral de saída (*throughPutOut*) e de entrada (*throughPutIn*) para o processo. Os dados dessa classe são utilizados pela classe *ThroughputThread*.

Classe *ThroughputThread*

A classe *ThroughputThread* é responsável por conter os dados que são necessários para realizar o aumento ou a diminuição de *worker threads*. Para isso, ela contém as medições das vazões médias de entrada ($avgTIN$) e de saída ($avgTOUT$), e das médias da relação entre a entrada e a saída de mensagens ($avgIO$). Também é responsável pelo armazenamento do *EntryPoint*, que é recalculado a cada medição de vazão pela Fórmula 3.2, do tamanho da fila ($QueueSize$), e do tempo de cronograma de conexão ($schTIME$),

que é o tempo máximo para tratamento da fila de mensagens. Além disso, possui a hora de início do crescimento da fila ($hSTART$), quando o valor dado pela Fórmula 3.1 se torna verdadeiro. Também possui a variável $hNOW$, que armazena a hora atual utilizada para os cálculos do $EntryPoint$, a variável $scaledUp$, para indicar que foi realizado um SU, a variável $wait$, para sinalizar espera de três medições de vazão antes de realizar SU recursivo, e as variáveis $firstMQSize$ e $schTIME10$ (10% do $schTIME$), ambas utilizadas para recuperação rápida de falhas com elasticidade. A variável $isGrowing$ é utilizada para indicar o reinício das medições de vazão de saída por conta (método $StartCPUMonitoring$), e das medições de uso de CPU por *thread* (método $StartThroughputAccounts$). Por fim, a classe $ThroughputThread$ possui duas variáveis de $DualBuffer$, $MessagesIn$ e $MessagesOut$, que serão utilizadas para a coleta de vazão instantânea de entrada e de saída. Dessa forma, serão vistas nas próximas subseções como essas estruturas de dados são utilizadas nos quatro tipos de *threads* para suportar a elasticidade no *middleware* proposto.

Thread Principal

A *thread* principal do *middleware* realiza a leitura dos pacotes a partir da fila de mensagens do processo, conforme mostrado no Algoritmo 5. Após ler a mensagem da fila, linha 2, a *thread* principal obtém a conta a qual pertence essa mensagem, linha 3. Em seguida, obtém o *scoped lock*[1] de escrita do *singleton* $ElasticityManager$, linha 4, para poder utilizar as listas de $WorkerBuffer$ e de $WorkerThread$, que não são *thread safe*. Após entrar na região crítica, protegida pelo *scoped lock*, o *middleware* procura o $WorkerBuffer$ e a $WorkerThread$, linhas 5 e 7, associados à mensagem.

Encontradas essas estruturas, é realizada a escrita no $DualBuffer$ correspondente, linha 9, bem como é movida a mensagem para a subfila associada à conta, linha 10. Por fim, o *middleware* obtém o *scoped lock* de leitura da $WorkerThread$ para poder inserir na estrutura de consumo de pacotes por ordem de chegada, linha 12, e incrementar o semáforo, linha 13, para sinalizar que a *worker thread* tem uma nova mensagem para consumir. A *thread* principal também é responsável pela geração dos dados de vazão de entrada total, como pode ser visto na linha 16. A seguir, será descrito como a *Worker Thread* realiza o consumo da mensagem que foi sinalizada pela *thread* principal.

Worker Thread

A *worker thread* é responsável pelo consumo das mensagens que foram geradas pela *thread* principal, e é mostrada no Algoritmo 6. Para realizar um consumo justo, por

[1]Garante que o *lock* é adquirido quando entra em um escopo e liberado automaticamente quando deixa o escopo, independente do caminho de retorno do escopo [24]. Nesse contexto, um *scoped lock* de escrita é exclusivo da *thread* que o adquiriu, e o de leitura é compartilhado entre várias *threads* de leitura.

Algoritmo 5: Thread Principal.

```
1  while !stop do
2      message = MessageQueue.Read();
3      accountNumber = GetAccountNumber( message );
4      ScopedWriteLock( ElasticityManager::lock );
5      workerBuffer = workerBufferList.find( accountNumber );
6      if workerBuffer != workerBufferList.end() then
7          workerThread = workerThreadList.find( workerBuffer->threadId );
8          if workerThread != workerThreadList.end() then
9              workerBuffer->dualBuffer.Write( message );
10             MessageQueue.Move( message, workerBuffer->subQueue );
11             ScopedReadLock( workerThread->lock );
12             workerThread->fcfsList.insert( timeStamp, accountNumber );
13             workerThread->semaphore.Up();
14         end if
15     end if
16     ++( ElasticityManager::throughPutIn );
17 end while
```

ordem de chegada e com *round robin* entre as contas tratadas pelo processo, é realizada a busca de uma conta que ainda não tenha sido tratada na rodada atual, como mostrado nas linhas de 6 a 27. Após escolher a próxima conta, a *worker thread* encontra a estrutura de *WorkerBuffer* associada, para que possa ler a mensagem do *DualBuffer*, tratar a mensagem e remover da subfila, como mostrado nas linhas de 29 a 37. É possível notar também, que a *worker thread* é responsável pelas medidas de vazão de saída total, linha 36, e vazão de saída por conta, linha 34. A próxima seção mostra como é realizada a coleta de vazão instantânea de entrada e de saída, que foram geradas pela *thread* principal e pela *worker thread* respectivamente.

Thread de Coleta da Vazão Instantânea de Entrada e Saída

A *thread* de coleta de vazão instantânea de entrada e de saída realiza a obtenção dos dados de vazão, como mostrado no Algoritmo 7. Como mostrado nas linhas 3 e 4, essas informações são lidas a partir do *ElasticityManager*, pois a *thread* principal e a *worker thread* foram responsáveis por escrever essas informações nessa estrutura de dados. Após a leitura da vazão instantânea, a *thread* de coleta realiza a escrita dos dados em nas estruturas de *DualBuffer* da classe *ThroughputThread*, uma de entrada (*MessagesIn*) e outra de saída (*MessageOut*), como mostrado nas linhas 5 e 6. Assim, para obter a vazão instantânea de entrada e de saída, essa coleta é realizada a cada segundo, como mostrado

Algoritmo 6: *Worker Thread.*

```
1  begin
2      while semaphore.Down() do
3          ScopedReadLock( ElasticityManager::lock );
4          begin
5              ScopedWriteLock( lock );
6              if roundRobinFQList.size() >= accountMessagesList.size() then
7                  roundRobinFQList.clear();
8              end if
9              accountNumber = 0;
10             while accountNumber == 0 do
11                 fcfs = fcfsList.begin();
12                 found = false;
13                 while fcfs != fcfsList.end() do
14                     accountTreated = roundRobinFQList.find( fcfs->accountNumber
                       );
15                     if accountTreated == roundRobinFQList.end() then
16                         found = true;
17                         roundRobinFQList.insert( fcfs->accountNumber );
18                         fcfsList.erase( fcfs );
19                         break;
20                     end if
21                     ++fcfs;
22                 end while
23                 if !found then
24                     roundRobinFQList.clear();
25                 else
26                     break;
27             end while
28         end
29         workerBuffer = workerBufferList.find( accountNumber );
30         if workerBuffer != workerBufferList.end() then
31             message = workerBuffer->dualBuffer.Read();
32             Treat( message );
33             if isGrowing then
34                 ++( workerBuffer->throughPut );
35             end if
36             ++( ElasticityManager::throughPutOut ) );
37             workerBuffer->subQueue.Read();
38             break;
39         end if
40     end while
41 end
```

na linha 7. Esta abordagem é utilizada porque os dados instantâneos coletados serão utilizados pela *thread* de monitoramento de vazão, que será mostrada a seguir.

Algoritmo 7: *Thread* de Coleta da Vazão de Entrada e Saída.

```
1 begin
2     while !stop do
3         throughPutIn = ElasticityManager::throughPutIn.GetMean();
4         throughPutOut = ElasticityManager::throughPutOut.GetMean();
5         MessagesIn.Write( throughPutIn );
6         MessagesOut.Write( throughPutOut );
7         Sleep( 1000 );
8     end while
9 end
```

Thread de Monitoramento

A *thread* de monitoramento da vazão média de entrada e de saída é responsável por realizar *SU* e *SD*, conforme mostrado nos Algoritmos 8 e 9. Dessa forma, a *thread* de monitoramento do processo consumidor da fila de mensagens realiza o aumento e a dimunuição das *CT* e *ZT threads* conforme mostrado na Figura 3.9.

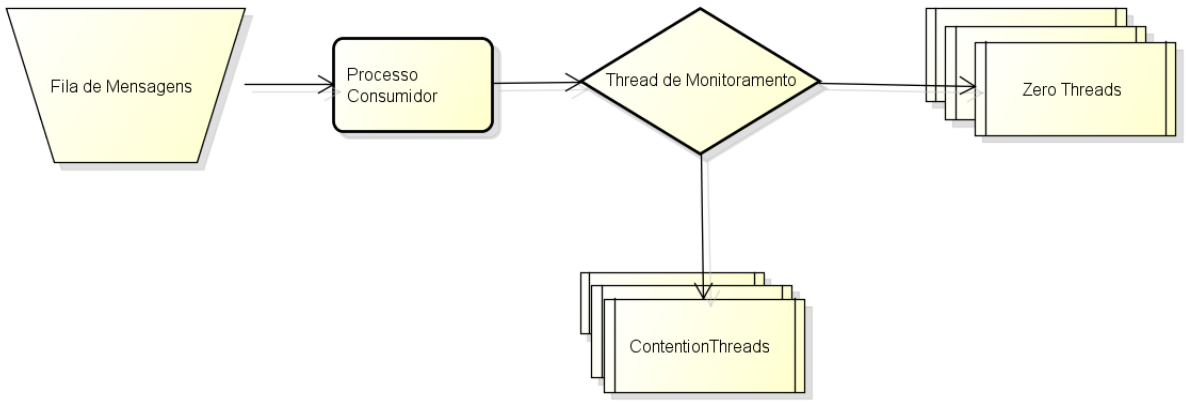


Figura 3.9: Gerenciamento das *CT* e *ZT Threads* da *Thread* de Monitoramento.

Essa *thread* é tratada pela classe *ThroughputThread* da Figura 3.8. Primeiramente, é realizada a leitura das vazões instantâneas de entrada e de saída que foram geradas na *thread* de coleta, mostrado na linha 2 do Algoritmo 8. Em seguida, o valor da Fórmula 3.1 é armazenado na variável *ThroughPut*, linha 3. Também é armazenado o tamanho da fila, linha 27, e atualizadas as médias de *avgTOUT*, *avgTIN* e *avgIO* na linha 5.

A *thread* de monitoramento possui dois estados. Um deles é utilizado para detectar quando a vazão de entrada é maior do que a de saída, conforme a linha 6 do Algoritmo 8.

O outro é utilizado para detectar quando a vazão de saída é maior ou igual a de entrada, linha 6 do Algoritmo 9.

A primeira situação é utilizada para a detecção de crescimento da fila, e destina-se a aumentar o número de *worker threads* para conter rajadas de mensagens, conforme mostrado no Algoritmo 8. Assim que essa situação é detectada, são iniciadas as medições de uso de CPU e vazão de saída por conta nas *worker trheads*, conforme mostrados nas linhas 11 a 14. Esses valores de medições serão utilizados na análise do consumo de CPU e do desvio padrão da vazão média de saída por conta, no momento de realizar o *SU*.

Em seguida, a *thread* de monitoramento armazena a hora de início do crescimento da fila em *hNOW*, como mostrado nas linhas 15 a 17. Após isso, o valor da Fórmula 3.2 é calculado e armazenado na variável *EntryPoint*, linha 18, para que seja possível detectar o ponto de entrada de criação de *worker threads*. Dessa forma, caso o *EntryPoint* seja maior do que o número de mensagens na fila, a *thread* de monitoramento cria mais *worker threads* para conter as rajadas de mensagens, conforme mostrado na linhas 32 a 34.

Para evitar o problema de *SU* recursivo em períodos próximos, foi colocado um fator de multiplicação por dois no tamanho da fila, linha 32, e modificado o valor da variável *wait*, linha 33, para sinalizar a necessidade de três medições de vazão antes de realizar um novo *SU*, linhas 8 a 10.

O segundo estado da *thread* de monitoramento é responsável por realizar o *SD*, e *SU* sem presença de rajadas de mensagens. Para o primeiro caso, é calculado o *ExitPoint* de acordo com a Fórmula 3.3, como mostrado na linha 10 do Algoritmo 9. Assim, caso o número de mensagens seja menor do que o *ExitPoint*, linha 11, são removidas *worker threads*, linha 12. O segundo caso, *SU* sem presença de rajadas de mensagens, está relacionado à recuperação rápida de falhas, e será visto a seguir.

Para suportar os mecanismos de recuperação rápida de falhas [3] [6] [13] [33], se um processo falha e o número de mensagens na fila é maior do que o *EP*, mecanismos de elasticidade no *middleware* são necessários para realizar o *SU*. Assim, quando os processos são reiniciados, se as rajadas de entrada continuam, o *SU* ocorre rapidamente por meio da detecção do *EP* maior do que o *QueueSize*, conforme mostrado na linha 32 do Algoritmo 8. No entanto, 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 está acima de $avgTOUT$ vezes o *schTIME*, linhas 16 e 17, as medidas de vazão, antes de decidir realizar o *SU*, são realizadas por somente 10% do *schTIME*, como mostrado na linha 19 do Algoritmo 9. Isso é necessário para realizar *SU* de uma maneira mais rápida, como é o caso onde as rajadas terminaram 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

Algoritmo 8: *Thread* de Monitoramento - *Scale Up* com Rajadas.

```
1  while !stop do
2      Input = MessagesIn.Read(); Output = MessagesOut.Read();
3      Throughput = Input / Output;
4      QueueSize = MessageQueue.GetMessagesCount();
5      UpdateAverages(Input, Output, Throughput);
6      if Throughput > 1 then
7          hNOW = now;
8          if wait > 0 then
9              wait = wait - 1; continue;
10         end if
11         if isGrowing == false then
12             StartCPUMonitoring();StartThroughputAccounts();
13             isGrowing = true;
14         end if
15         if ( hSTART == 0 ) or ( ( hNOW - hSTART ) > schTIME ) then
16             hSTART = hNOW;
17         end if
18         EntryPoint = ( avgTOUT ) * ( schTIME - ( hNOW - hSTART ) );
19         IsFirst = false;
20         if firstQueueSize == 0 then
21             firstQueueSize = QueueSize;
22             IsFirst = true;
23         end if
24         if QueueSize > EntryPoint then
25             if ( schTIME10 != 0 ) and ( hNOW >= schTIME10) then
26                 ScaleUp(); scaledUp = true; wait = 3;
27                 firstQueueSize = QueueSize;
28                 schTIME10 = 0;
29             else
30                 if IsFirst then
31                     schTIME10 = now + ( schTIME * 0.10 );
32                 else if QueueSize > ( firstQueueSize * 2 ) then
33                     ScaleUp(); scaledUp = true; wait = 3;
34                     firstQueueSize = QueueSize;
35             end if
36         end if
37     end if
38 end while
```

do processo. Essa situação é mostrada nas linhas 25 a 31 do Algoritmo 8, por meio do agendamento do próximo *SU*.

Algoritmo 9: *Thread* de Monitoramento - *Scale Up* sem Rajadas, e *Scale Down*.

```
1  while !stop do
2  |  Input = MessagesIn.Read(); Output = MessagesOut.Read();
3  |  Throughput = Input / Output;
4  |  QueueSize = MessageQueue.GetMessagesCount();
5  |  UpdateAverages(Input, Output, Throughput);
6  |  if Throughput <= 1 then
7  |  |  isGrowing = false;
8  |  |  firstQueueSize = 0;
9  |  |  if scaledUp then
10 |  |  |  ExitPoint = ( avgTIN / 2 );
11 |  |  |  if ( QueueSize < ExitPoint ) or ( QueueSize == 0 ) then
12 |  |  |  |  ScaleDown(); scaledUp = false;;
13 |  |  |  |  ZeroSchedule();
14 |  |  |  end if
15 |  |  else
16 |  |  |  EntryPoint = avgTOUT * schTIME;
17 |  |  |  if ( avgTOUT != 0.0 ) and ( QueueSize > EntryPoint ) then
18 |  |  |  |  if schTIME10 == 0 then
19 |  |  |  |  |  schTIME10 = now + ( schTIME * 0.10 );
20 |  |  |  |  else if now >= schTIME10 then
   |  |  |  |  |  // Scale Up com recuperação rápida sem rajadas
21 |  |  |  |  |  hSTART = now;
22 |  |  |  |  |  ScaleUp(); scaledUp = true; wait = 3;
23 |  |  |  |  |  schTIME10 = 0;
24 |  |  |  end if
25 end while
```

3.4 Considerações Finais

Após as implementações das características de tolerância a falhas e elasticidade no *middleware* proposto vistas neste capítulo, é necessária a configuração de um ambiente para a realização dos testes de validação. Assim sendo, no Capítulo 4 será descrita a implementação de um simulador do servidor de comunicação, mostrado na Figura 3.1, que será usado para os testes de validação das implementações apresentadas.

Capítulo 4

Simulador do Servidor de Comunicação

Este capítulo descreve a implementação de um servidor de comunicação TCP, chamado de SERVSIM, que simula o funcionamento do servidor do ambiente de produção, para que possa ser utilizado nos testes do *middleware* proposto neste trabalho.

Inicialmente, para a realização das simulações de comunicação foi montado um ambiente com um servidor de produção desativado, que atendia aos requisitos de vazão necessários para os testes do *middleware*.

Entretanto, verificou-se que a arquitetura de recuperação de dados para que o servidor estivesse em um estado com uma quantidade grande de pacotes de todas as contas tornou-se inviável, pois para isso seria necessário uma parada total do sistema de produção para gravação de *logs* e, posterior, recuperação no servidor de testes. Assim sendo, foi construído um simulador de pacotes com a implementação do protocolo do servidor de produção. Este simulador possui alta vazão e atende a milhares de requisições, em paralelo, por meio da multiplexação de conexões em *multithread*.

Para realizar os experimentos com o SERVSIM, primeiramente, foram capturados 32.054 pacotes de uma conta do servidor de produção, que trocou cerca de 128.000 posições de veículos monitorados. Em seguida, esses pacotes foram carregados para o SERVSIM. Nas próximas seções serão descritos diversos experimentos que visam testar e adequar o simulador às características de funcionamento do servidor de produção, proporcionando testes com dados reais no *middleware* proposto.

4.1 Uma Instância Lógica de Banco de Dados em um Servidor Físico

Primeiramente, montou-se um ambiente com o primeiro dos 11 servidores do banco de dados BD2 da Tabela 3.1. O SERVSIM foi configurado para sempre entregar mensagens

para todas as contas que solicitarem dados em todos os ciclos de conexão.

Para evitar *starvation* de uma conta, que sempre possui dados em relação às outras, o serviço que realiza *polling* no servidor, ao obter 300 mensagens, interrompe o processo de obtenção de dados. Desta forma, foi realizada a medição do consumo de CPU que chegou a cerca de 75% de média, como mostra a Figura 4.1. Os vales mostrados nessa figura correspondem ao tempo em que os processos dormem para que chegue o novo período de conexão que ocorre a cada 80 segundos, e demonstram que os processos conseguiram realizar um *polling* completo de 300 mensagens para todas as contas sendo tratadas.

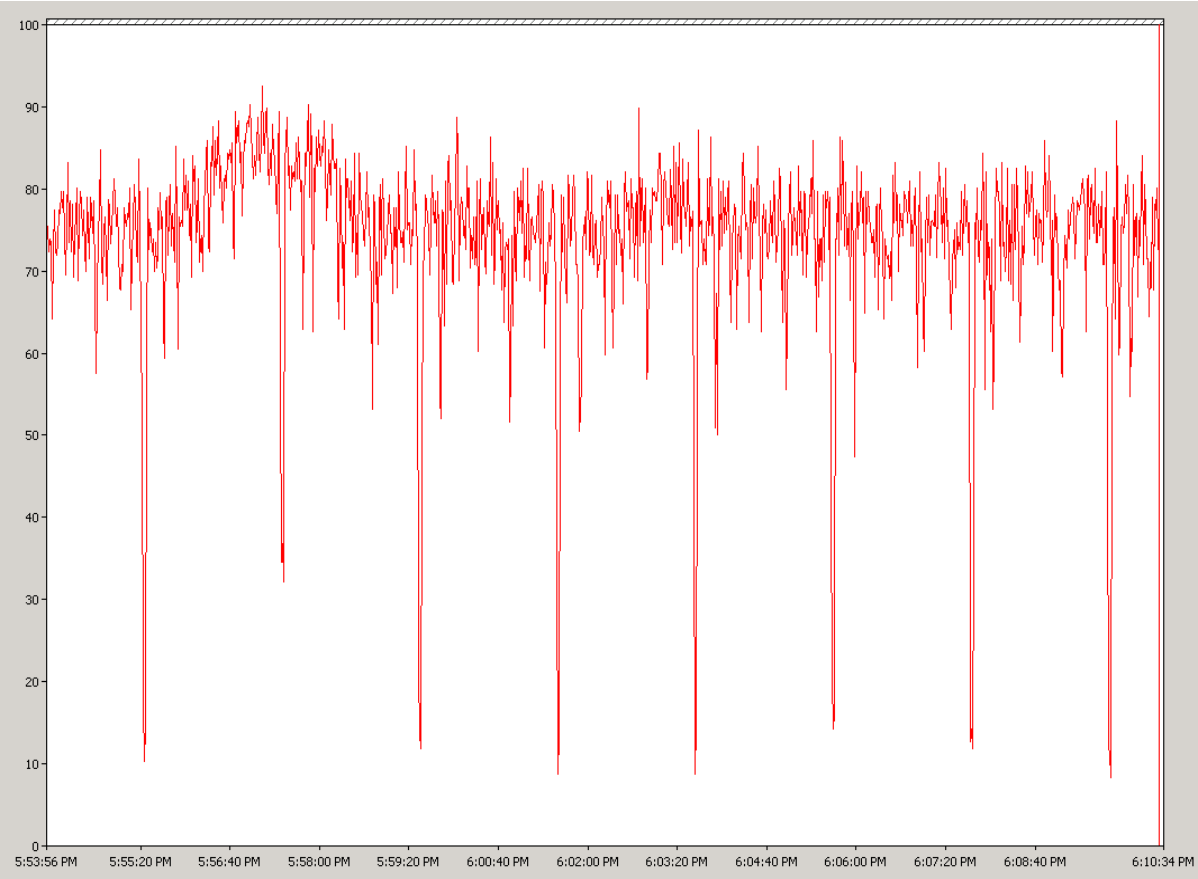


Figura 4.1: Consumo de CPU do Primeiro Servidor da Instância 2.

Também foi realizada a medição do número de mensagens nas filas dos processos, conforme mostrado na Figura 4.2. Como pode ser observado, o número de mensagens eventualmente passa de seis em cada uma das 19 filas de mensagens. Isso ocorre porque a vazão de entrada de mensagens nas filas está compatível com a velocidade de tratamento pelo servidor, sem gerar enfileiramentos.

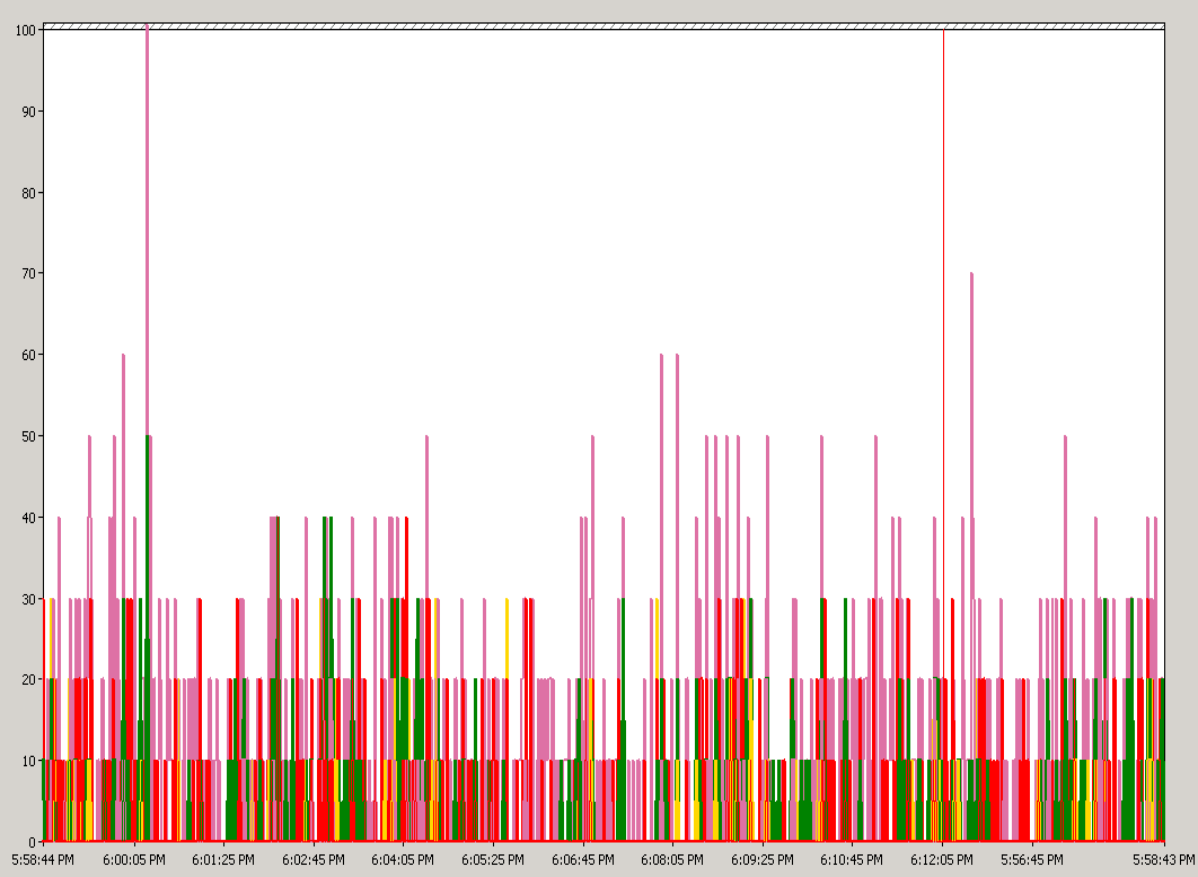


Figura 4.2: Número de Mensagens nas Filas do Primeiro Servidor da Instância 2.

4.2 Onze Instâncias Lógicas de Banco de Dados Agrupadas em Um Servidor Físico

Cada instância lógica de servidor estabelece quatro conexões TCP paralelas, uma para cada um dos sistemas de comunicação (1, 2, 3 e 5) com o SERVSIM. Portanto, ao agrupar o número de instâncias lógicas, também é necessário aumentar o número de conexões paralelas para que a vazão não seja comprometida. Assim, foram criados, *n pools* de conexões, cada um contendo quatro conexões TCP, para manter o mesmo número de conexões TCP obtido com 11 servidores físicos conectando em paralelo no servidor de comunicação.

Dessa forma, para o experimento com 11 instâncias lógicas agrupadas em um servidor com 4.141 contas, foram utilizadas 44 conexões TCP paralelas. Apesar do SERVSIM ter suportado o aumento de conexões TCP, o uso de rede não passou de 0,02%, como mostra a Figura 4.3. Sendo assim, nota-se que existe uma necessidade de aumento da vazão durante a troca de pacotes por parte do SERVSIM.

Também é possível verificar, conforme indicado na Figura 4.4, que existe margem de

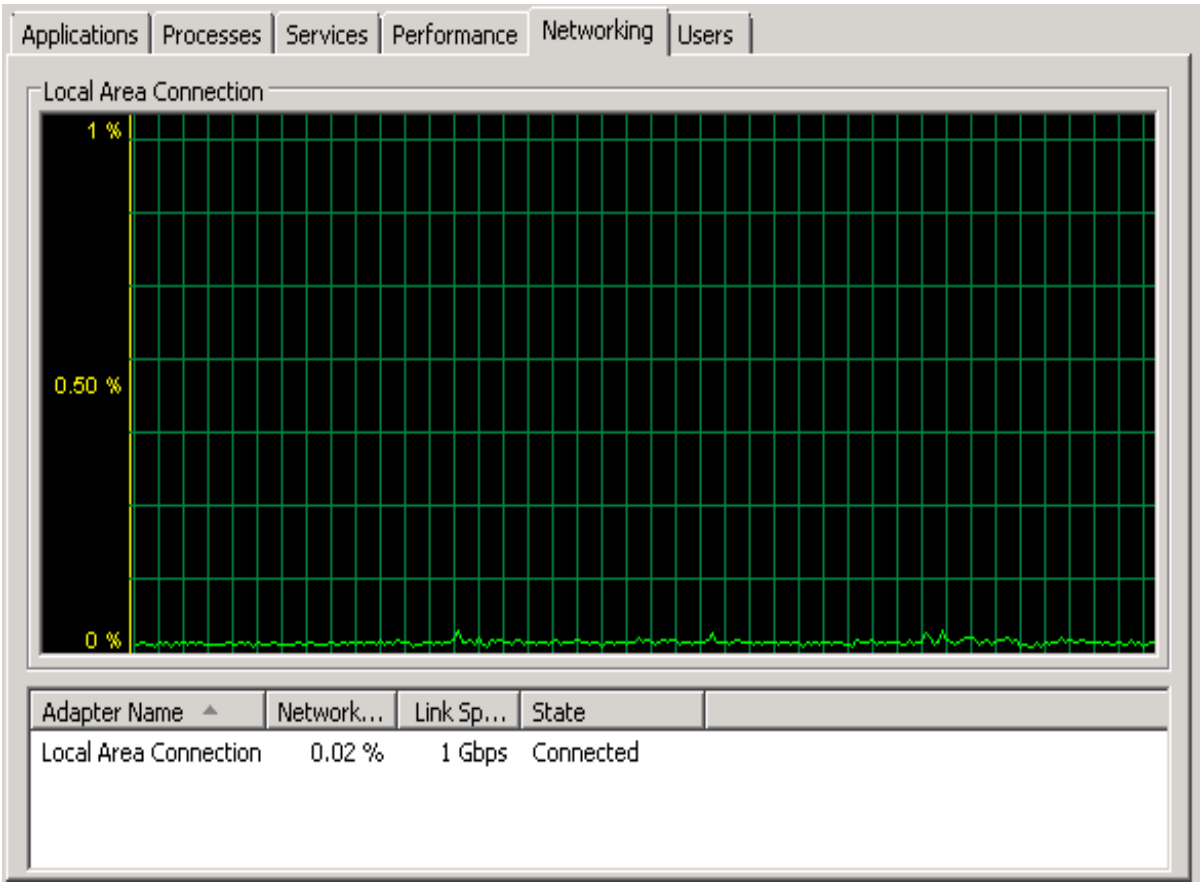


Figura 4.3: Uso de Rede do SERVSIM com 44 Conexões TCP Simultâneas.

CPU para um uso mais eficiente, pois a média de consumo ficou em 35%. Por outro lado, pela Figura 4.5, é possível observar que as filas de mensagens não sofrem aumento constante, apenas picos esporádicos acima de 50 mensagens. Isso ocorre porque o SERVSIM não está apresentando vazão compatível com a do servidor de comunicação de produção.

A Tabela 4.1 compara o desempenho da vazão dos servidores de produção e o SERVSIM, para a entrega de 300 mensagens. É possível observar que o desempenho do SERVSIM é quatro vezes menor e, que há a necessidade de uma melhoria da sua vazão. Com isto será possível verificar o comportamento do sistema em relação à CPU e às filas de mensagens, em caso de entrega de pacotes mais eficiente. As adequações necessárias para a melhoria de desempenho do SERVSIM serão detalhadas na próxima seção.

Tabela 4.1: Tempo de Consumo para 300 Mensagens.

Servidor	Tempo
Produção	1'17"
SERVSIM	5'8"

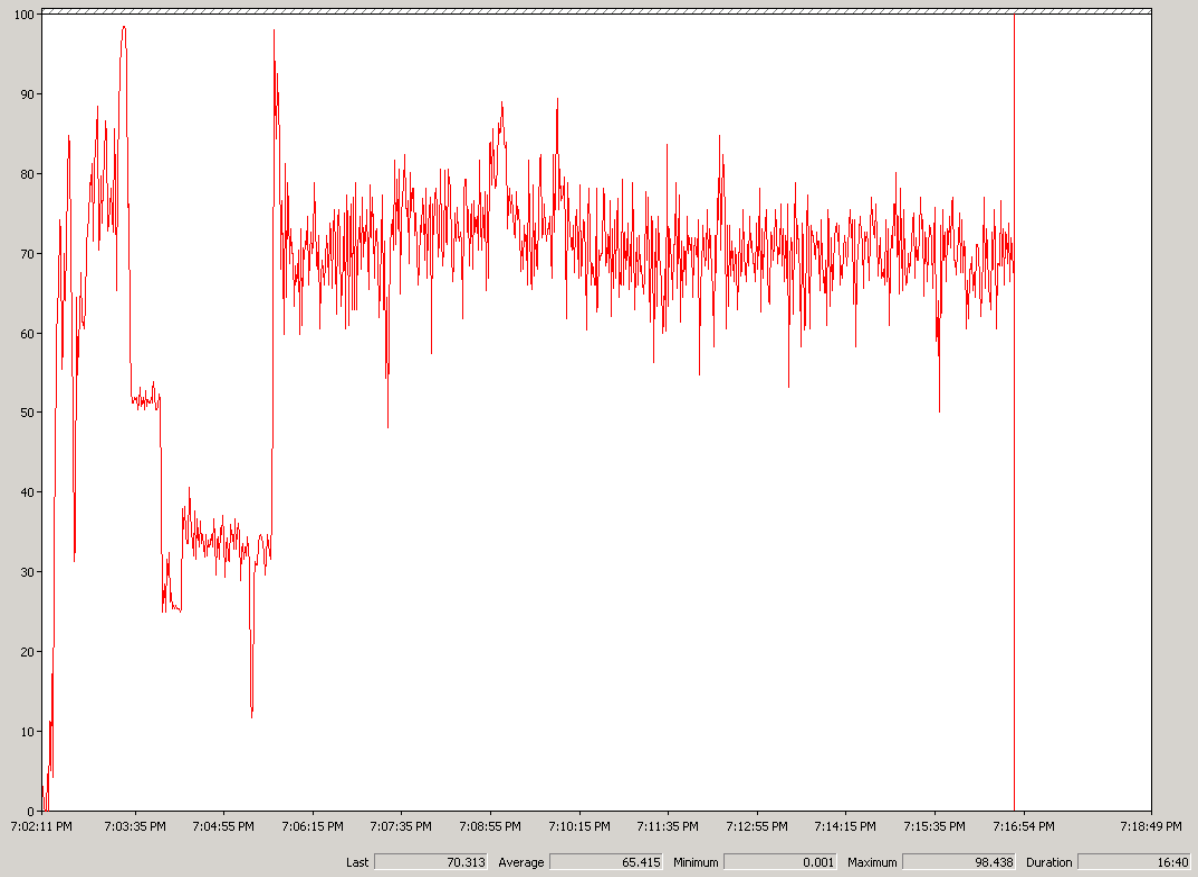


Figura 4.4: Consumo de CPU com 11 Instâncias Lógicas Agrupadas em 1 Servidor.

4.3 Adequações no SERVSIM para Melhoria de Desempenho

Diversas mudanças foram realizadas no SERVSIM para uma melhoria no desempenho, e, por consequência, verificar o comportamento da CPU e das filas de mensagens, quando ocorre o agrupamento do número de instâncias lógicas em um servidor físico. Essas adequações serão vistas nas próximas subseções.

4.3.1 Otimizações para Aumento da Vazão

Diversas otimizações foram realizadas a fim de aumentar a vazão de entrega de pacotes. Entre as mudanças realizadas estão a carga de pacotes para a memória de forma a evitar I/O de disco, e o uso de estruturas de recuperação rápida dos pacotes em memória, para responder as requisições mais eficientemente. A Tabela 4.2 resume os resultados alcançadas no servidor SERVSIM, diante destas mudanças.

Como pode ser observado na Tabela 4.2, foi possível reduzir o tempo 5 minutos e 8 segundos para 13 segundos, após as mudanças implementadas.

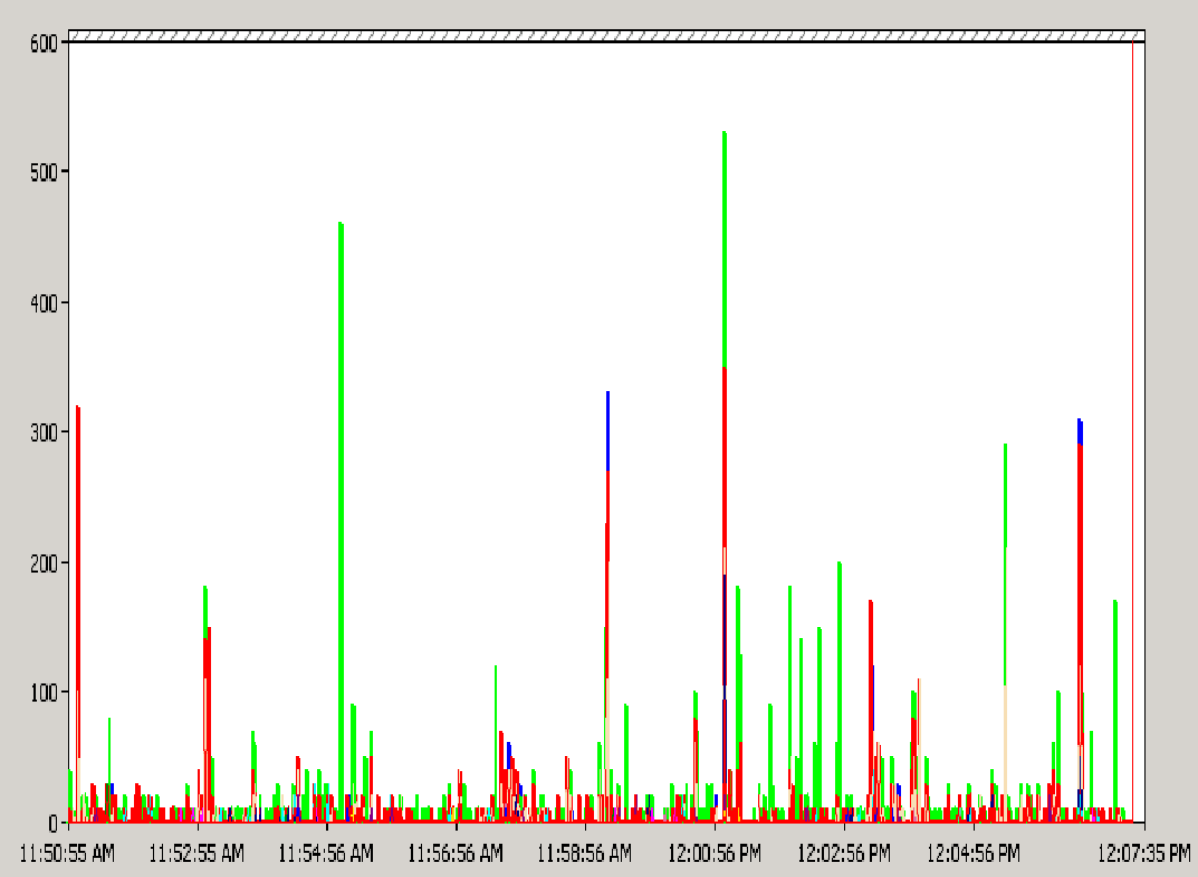


Figura 4.5: Mensagens nas Filas com 11 Instâncias Lógicas Agrupadas em Um Servidor.

Tabela 4.2: Otimizações Realizadas para Aumento da Vazão.

Versão do SERVSIM	Tempo
Sem Otimizações	5'8"
Carga de Pacotes em Memória	3'51"
Retirada da Estrutura *DualBuffer*	41"
Uso da Estrutura de *Iterators* C++	13"

4.3.2 Compatibilização com a Vazão do Servidor de Produção

O tempo de resposta médio do servidor de produção foi calculado baseado nos *logs* de troca de pacotes, e também do SERVSIM, considerando as mesmas mensagens. Nesse cenário, foram analisadas 300 mensagens e considerado o tempo entre a resposta do servidor e da próxima requisição de pacote, e, em seguida, calculada a média. Esta abordagem foi utilizada porque calcula a latência do cliente durante o tratamento das mensagens.

Os tempos de resposta médio de ambos os servidores estão na Tabela 4.3. Assim, para que as vazões sejam compatibilizadas foram colocados *sleeps* de 40 milisegundos entre os

pacotes no SERVSIM, de maneira que o simulador se comporte com uma vazão compatível com o servidor de produção.

Tabela 4.3: Tempos de Resposta dos Servidores de Produção e SERVSIM (300 mensagens).

Servidor	Tempo de Resposta Médio (ms)	*Sleep* (ms)
Servidor de Produção	81.130435	N/A
SERVSIM	41.113712	40.016723

4.3.3 Compatibilização do Número de Mensagens por Cronograma de Conexão

A compatibilização do número de mensagens que o servidor de produção envia, a cada cronograma de conexão (80 segundos), possibilita o cálculo do número de conexões paralelas necessárias para que a vazão esteja em um nível em que o número médio de mensagens históricas consiga ser tratado, mesmo com o agrupamento do número de instâncias lógicas em um servidor físico.

Como o simulador sempre envia 300 mensagens para todas as contas utilizadas no teste, este ambiente não representa o cenário real do ambiente de produção. Assim, foi obtido o número médio de mensagens trocadas entre os cronogramas de conexão para as contas que estão sendo utilizadas na simulação, de forma que o SERVSIM envie o mesmo número de mensagens e simule uma carga real do ambiente de produção.

Para isso, foi utilizada a consulta SQL (*Structured Query Language*), mostrada na Figura 4.6, para obter o número médio de mensagens recebidas por conta em um cronograma de conexão de 80 segundos. Em seguida, o SERVSIM foi modificado com este valor em substituição às 300 mensagens que não refletem uma condição real de funcionamento.

A consulta foi executada em todo o banco de dados BD2 da Tabela 3.1, que contém um histórico de seis meses de dados. Em seguida, foram realizadas as estatísticas em relação às médias das posições da bases de dados. Como pode ser observado na Tabela 4.4 e na Figura 4.7, a média não ficou muito distante do desvio padrão e, portanto, pode ser usada como referência para o número de mensagens a serem enviadas a cada ciclo de conexão pelo SERVSIM.

Tabela 4.4: Estatísticas do Número de Posições das Contas por Cronograma de Conexão.

Mínimo	Máximo	Média	Desvio Padrão
0	100	15,74	9,33

```
select COUNT( * ) as ctmsg, acc.ACC_Num_Number,
( ( datediff(s, Min( PHS_Dtm_ReceiveTime ), MAX( PHS_Dtm_ReceiveTime ) ) / 80 ) )
  as NumberOfSchedules
from PositionHistory_PHS as phs with( nolock )
inner join Vehicle_VEH as veh with( nolock )
  on veh.VEH_Cod_Vehicle = phs.VEH_Cod_Vehicle
inner join Account_ACC as acc with( nolock )
  on acc.ACC_Cod_Account = veh.ACC_Cod_Account
where
acc.acc_idt_isactive = 1 and acc.fam_cod_family = 1
group by acc.ACC_Num_Number
order by  ctmsg desc
```

Figura 4.6: Consulta SQL para o Número Médio de Mensagens Recebidas por Conta.

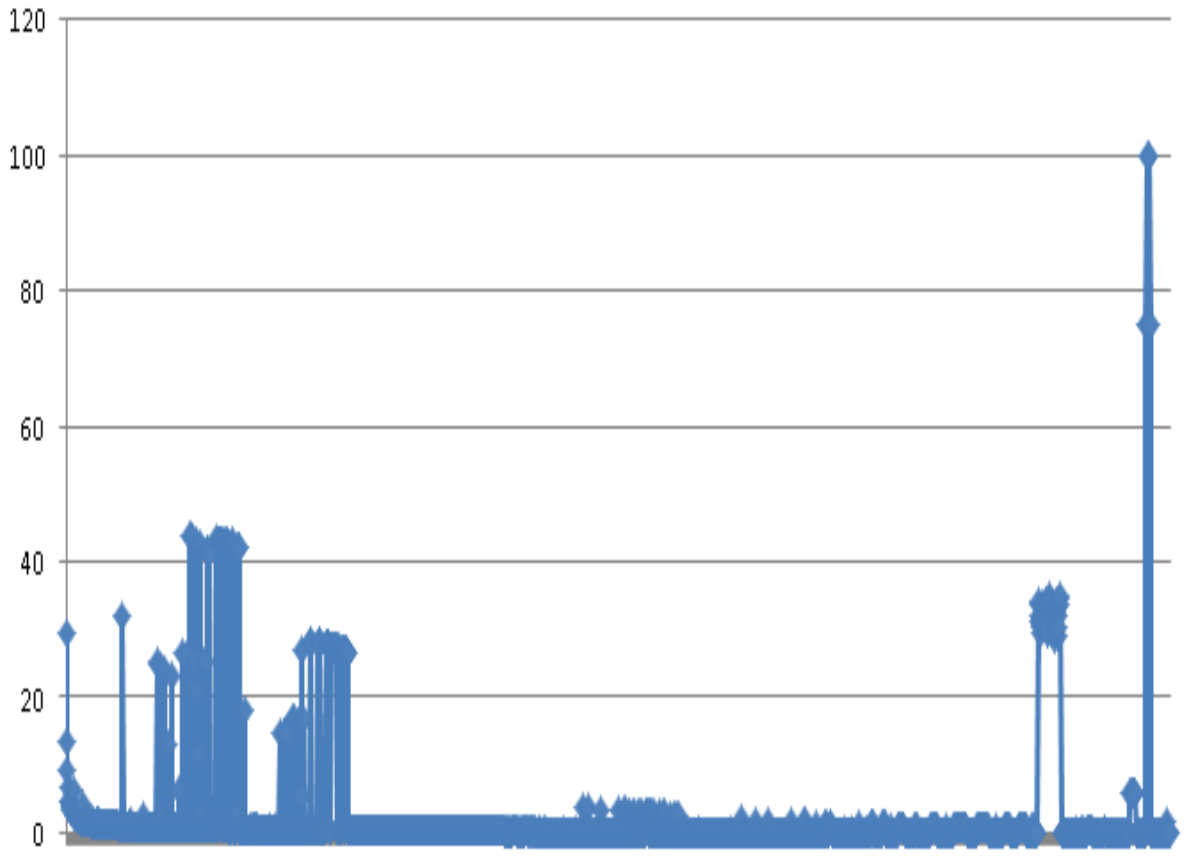


Figura 4.7: Média de Posições das Contas por Cronograma em 6 Meses de Dados.

Assim sendo, o número de mensagens por conta e por cronograma de conexão dever ser de 15, e não de 300, para que o funcionamento do SERVSIM fique compatível com o servidor de produção.

4.4 SERVSIM Compatibilizado com o Servidor de Produção

Após as compatibilizações de vazão e de número de mensagens, os experimentos foram realizados novamente para verificar se um servidor com a carga de 11 instâncias lógicas de banco de dados agrupadas em um servidor físico, e com conexões para 4.101 contas, suporta a troca de 15 mensagens por ciclo de conexão de 80 segundos.

Como pode ser observado na Figura 4.8, ocorreu o enfileiramento de mensagens nas filas de 2 serviços. Uma, em azul, ocorreu por queda sem reinício, o que justifica o crescimento linear da fila e, constata a necessidade de algoritmos em nível de *middleware* para suporte à tolerância a falhas. A outra, em vermelho, ocorreu por rajada de pacotes que não conseguem ser tratados em um ciclo de conexão e geram enfileiramento residual entre cronogramas de 80 segundos.

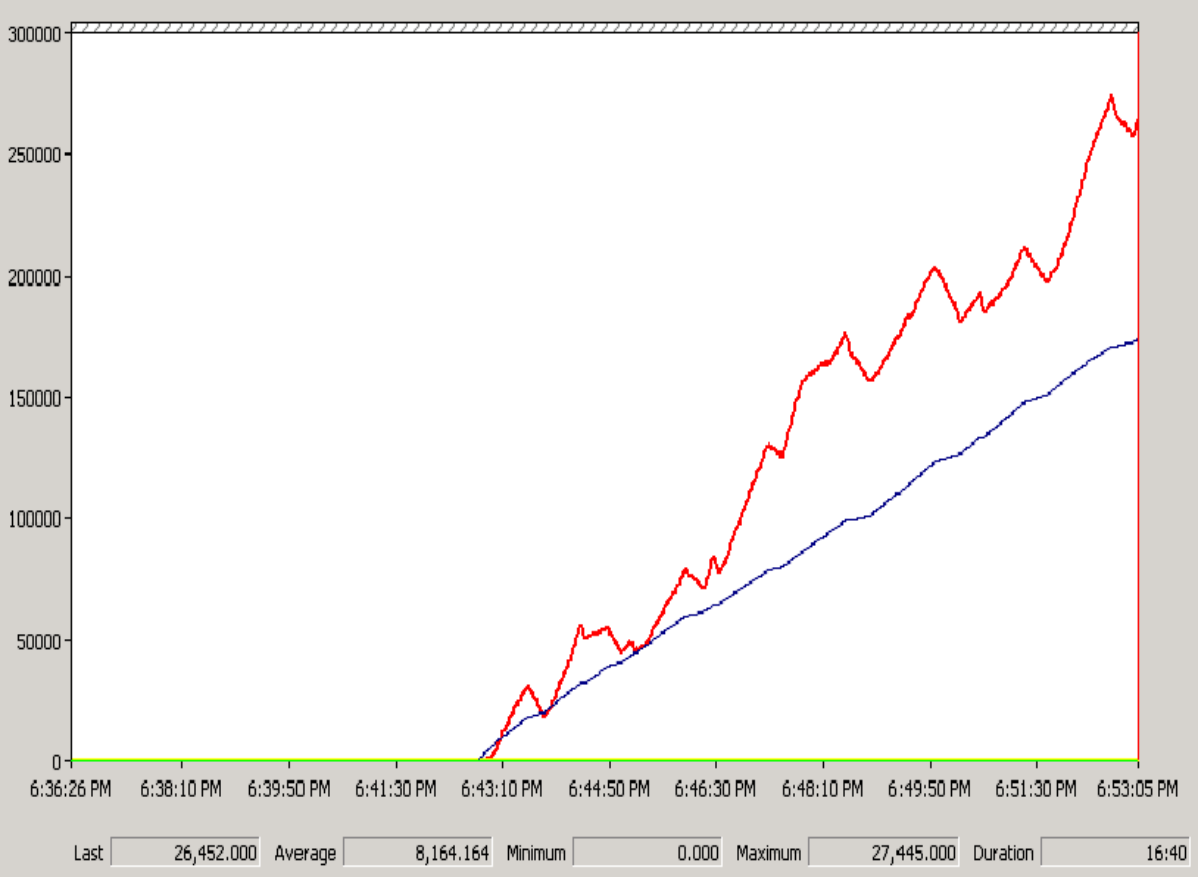


Figura 4.8: Tamanho das Filas de Menssagens para Requisições de 15 mensagens por Cronograma de Conexão.

Além disso, o sistema não conseguiu realizar um *polling* completo em todas as contas de cada sistema em menos de 80 segundos, como mostra a Tabela 4.5. Porém, como a CPU

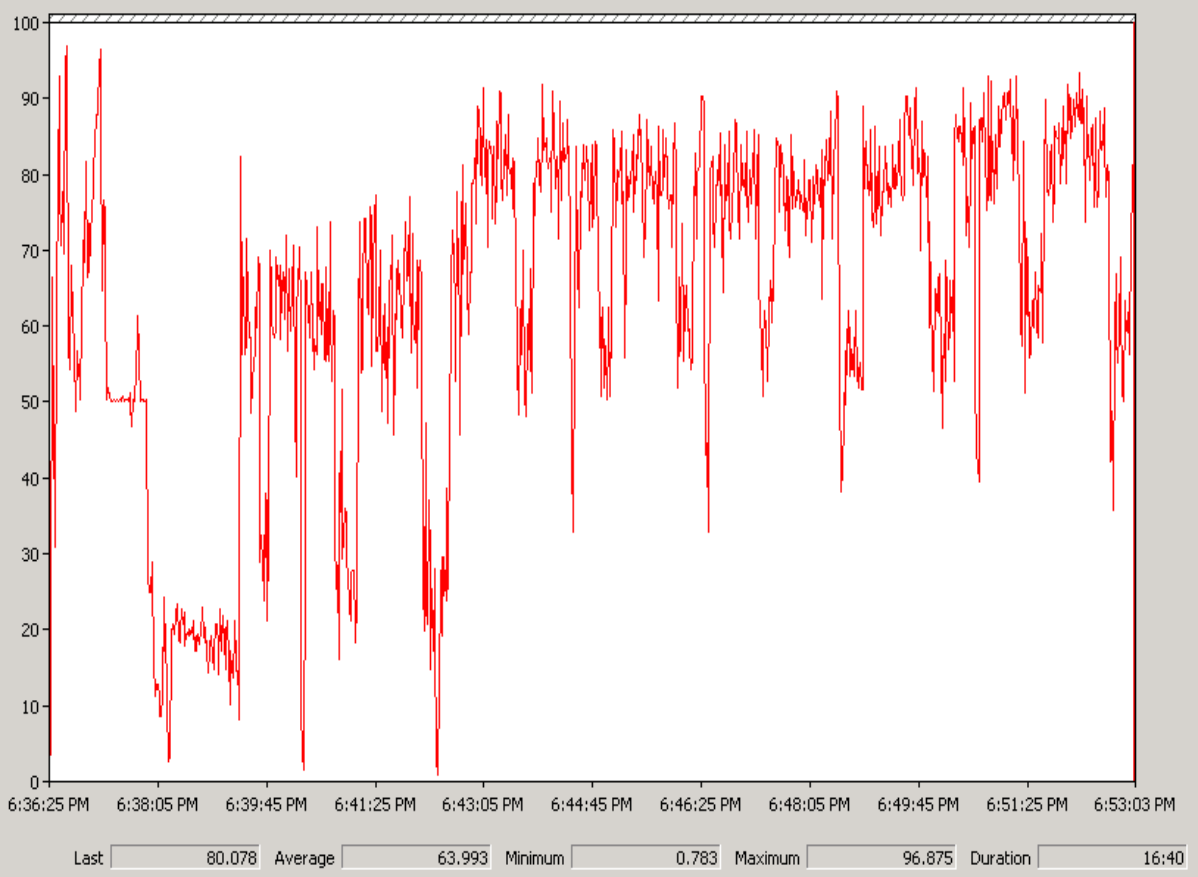


Figura 4.9: Consumo de CPU para Requisições de 15 mensagens por Cronograma de Conexão.

Tabela 4.5: Estatística de Conexão de Contas por Sistema Satelital.

Início	Fim	Tempo (seg.)	Contas Conectadas	Sistema
18:48:39	18:53:24	274.765234	338	1
18:47:19	18:56:10	500.880625	478	2
18:39:19	18:57:13	1073.881075	849	3

ficou com média de uso de 63%, conforme mostrado na Figura 4.9, existe margem para a escalabilidade necessária a ser usada com o emprego de algoritmos com suporte à elasticidade, de forma a aumentar o número de *worker threads* de tratamento de mensagens e evitar o enfileiramento de mensagens.

4.5 Considerações Finais

Neste capítulo foi construído um simulador do servidor de comunicação mostrado na seção 3.1, que será utilizado para testar os algoritmos de tolerância a falhas e elasticidade,

que foram implementados no Capítulo 3, de forma a verificar o comportamento das filas de mensagens e CPU diante dos cenários mostrados neste capítulo.

Capítulo 5

Resultados das Simulações e Adequação ao Ambiente Real da Arquitetura

Este capítulo tem como objetivo descrever e analisar os resultados obtidos a partir dos testes realizados com o *middleware* proposto neste trabalho.

A primeira seção descreve o ambiente de execução e de coleta de dados utilizado para os testes. A segunda e a terceira seções, apresentam, respectivamente, os testes de tolerância a falhas e de elasticidade. Ao final, na quarta seção, são descritas as adequações necessárias para adaptar e migrar o *middleware* proposto para o ambiente real do *cluster* estudado.

5.1 Ambiente de Execução

Para os testes do *middleware* desenvolvido foram utilizados quatro servidores BLADE BL460c G1, com dois processadores *Intel Xeon Quad Core/Eight Threads* modelo E5540 com *clock*, de 2.53 GHz. Três destes servidores, com 32 *gigabytes* de memória, foram utilizados para os testes do *middleware* desenvolvido, e do simulador do servidor de comunicação SERVSIM. A quarta máquina, com 16 *gigabytes* de memória, foi utilizada como servidor de banco de dados SQL *Server*.

Inicialmente, para os testes de melhoria da inicialização do sistema, três destas máquinas foram utilizados para virtualizar 11 servidores com quatro núcleos e quatro *gibabytes* de memória cada uma. Após estes testes, duas máquinas físicas, com oito núcleos e 16 *gigabytes* de memória, foram utilizadas para os testes, uma delas para o teste do *middleware* desenvolvido, e outra para uso com o simulador do servidor de comunicação SERVSIM. Todas as máquinas que foram utilizadas excutavam o sistema operacional *Microsoft Windows* 2008 R2 64 *bits*.

Neste ambiente, para os testes de elasticidade, o *middleware* desenvolvido foi executado tanto em um simulador para os testes de aumento e diminuição de *worker threads*, quanto no ambiente real da arquitetura da aplicação distribuída, após as adequações que foram realizadas. Para os testes de tolerância a falhas foi utilizada diretamente a aplicação distribuída se comunicando com o simulador SERVSIM para a troca de mensagens. Os binários do simulador do servidor de comunicação e a aplicação distribuída foram gerados com versão de *release* (otimizada) para a plataforma 64 *bits*. Já os binários do simulador do *middleware* foram compilados para a versão de *debug*, devido a necessidade de depuração e ajustes durante a fase dos experimentos. Todos os códigos foram programados na linguagem C/C++ com a ferramenta *Microsoft Visual Studio 2012*.

Para a coleta dos dados foi utilizado o aplicativo *Peformance Monitor* [22], que é uma ferramenta do *Windows* usada para coletar informações de componentes do sistema operacional em tempo real, e por meio de dados gravados em *logs* para análise posterior. O *Performance Monitor* foi utilizado para coletar os dados das filas de mensagens, consumo de CPU e número de *threads* dos processos. Além disso, o *middleware* foi modificado para gravar em *logs*, a cada segundo, informações sobre a vazão, número de *threads* e tamanho da fila de mensagens.

5.2 Tolerância a Falhas

Para as simulações dos testes de tolerância a falhas foi utilizada a estrutura do *cluster* responsável pelo funcionamento da base de dados BD2 , conforme mostrado na Tabela 3.1, pelo fato de ser a que possui o maior número de instâncias lógicas (um total de 11 instâncias), e o maior número de contas de comunicação, ao todo 4.141.

As subseções a seguir descrevem os testes que visam simular a diminuição do tempo de carga e a diminuição do número de servidores, por meio do agrupamento de instâncias lógicas em um servidor físico. Estes testes objetivam verificar o comportamento de recuperação rápida do sistema em caso de falha ou reinicialização total.

5.2.1 Diminuição do Tempo de Inicialização

O objetivo deste teste é verificar a diminuição do tempo de carga total do sistema, para que, em casos de falhas ou atualizações do sistema, ocorra uma reinicialização rápida.

Dessa forma, foram feitos experimentos de inicialização total do sistema com a versão atual (a versão em produção), e a versão modificada com as otimizações de melhoria de carga, as quais foram especificadas na Seção 3.2. Para isso, foram utilizados 11 servidores virtualizados, cada um com quatro núcleos e 4 *gigabytes* de memória. Assim, cada servidor foi responsável pelo tratamento de uma instância lógica do banco de dados BD2, mostrado

na Tabela 3.1. Todas as inicializações foram realizadas com carga em paralelo de todos os servidores.

Os tempos de inicialização com a versão atual de produção, são mostrados na Tabela 5.1. É possível observar que os servidores da versão de produção responsáveis pelo funcionamento da base de dados BD2 demoram mais de 15 minutos para realizar a carga inicial do sistema distribuído. De acordo com a Figura 5.1, é possível observar também que o servidor de banco de dados opera com folga de consumo de CPU.

Tabela 5.1: Tempo de Inicialização da Versão Atual em Produção.

Servidor	Tempo do Sistema Atual
*instance*1	4'10"
*instance*2	12'05"
*instance*3	8'59"
*instance*4	15'06"
*instance*5	12'11"
*instance*6	15'37"
*instance*7	15'43"
*instance*8	12'51"
*instance*9	4'56"
*instance*10	4'32"
*instance*11	16'02"
Total	**16'02"**

Tabela 5.2: Tempo de Inicialização com Otimizações.

Servidor	Tempo do Sistema Otimizado
*instance*1	27,82"
*instance*2	11,94"
*instance*3	7,22"
*instance*4	26,32"
*instance*5	15,11"
*instance*6	7,59"
*instance*7	9,91"
*instance*8	9,05"
*instance*9	5,48"
*instance*10	5,10"
*instance*11	14,25"
Total	**27,82"**

Os tempos de inicialização do sistema com a versão otimizada são mostrados na Tabela 5.2 e na Figura 5.2. É possível observar que os servidores realizam a carga em pouco mais de 27 segundos. Assim, embora nota-se na Figura 5.2 que o uso de CPU é maior, com as otimizações feitas o tempo para reiniciar todos os serviços é de 27 segundos.

Após a implementação das otimizações do tempo de inicialização, é possível verificar as melhorias alcançadas através da diminuição do tempo de carga dos servidores. O banco de dados atendeu a expectativa de escalabilidade, por meio do uso mais eficiente da CPU, em que ocorrem picos de consumo máximo dos oito núcleos, durante o período de tempo em que ocorre a carga da versão otimizada.

Sendo assim, como a versão otimizada possui tempos de carga menores do que um cronograma de conexão de 80 segundos, o *cluster* é tolerante a falhas em caso de reinicialização total por causa de falhas, ou atualizações por conta de manutenções corretivas ou evolutivas, tornando esse impacto, para os usuários do sistema, mínimo.

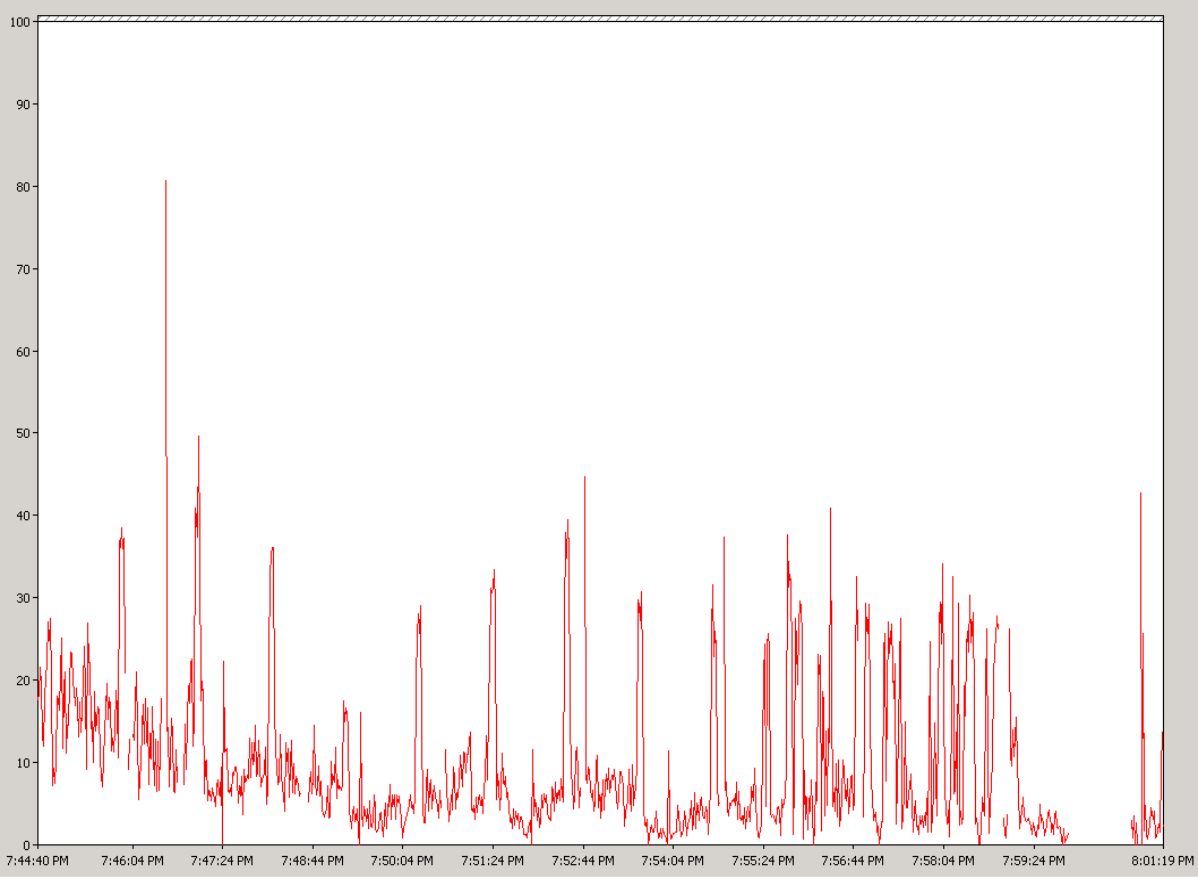


Figura 5.1: Consumo de CPU do Servidor de Banco de Dados (8 núcleos) durante a Inicialização do Sistema Distribuído sem Otimizações.

5.2.2 Agrupamento de Instâncias Lógicas em um Servidor Físico

O objetivo deste teste é verificar o comportamento do sistema diante da diminuição do número de servidores, por meio do agrupamento de instâncias lógicas de banco de dados em um servidor físico. Dessa forma, com a diminuição do tempo de inicialização do sistema, mostrada na subseção anterior, a diminuição do número de servidores se torna viável e, visa, além do uso mais eficiente dos recursos, a possibilidade da transformação de parte dos servidores, que não serão mais necessários, em *failover*, que é a mudança para um sistema ou componente de *hardware* redundante em caso de falha ou término anormal do sistema ativo anteriormente.

Para este teste foi utilizado um servidor físico, e gradualmente foi incrementado o número de instâncias lógicas tratadas, por meio de alteração na configuração do banco de dados BD2 da Tabela 3.1. Dessa forma, foi necessário aumentar a memória do servidor de 4 para 8 *gigabytes*, tornando possível realizar a carga dos serviços para a memória das informações utilizadas pelos 19 processos do sistema distribuído. Os resultados são mostrados na Tabela 5.3.

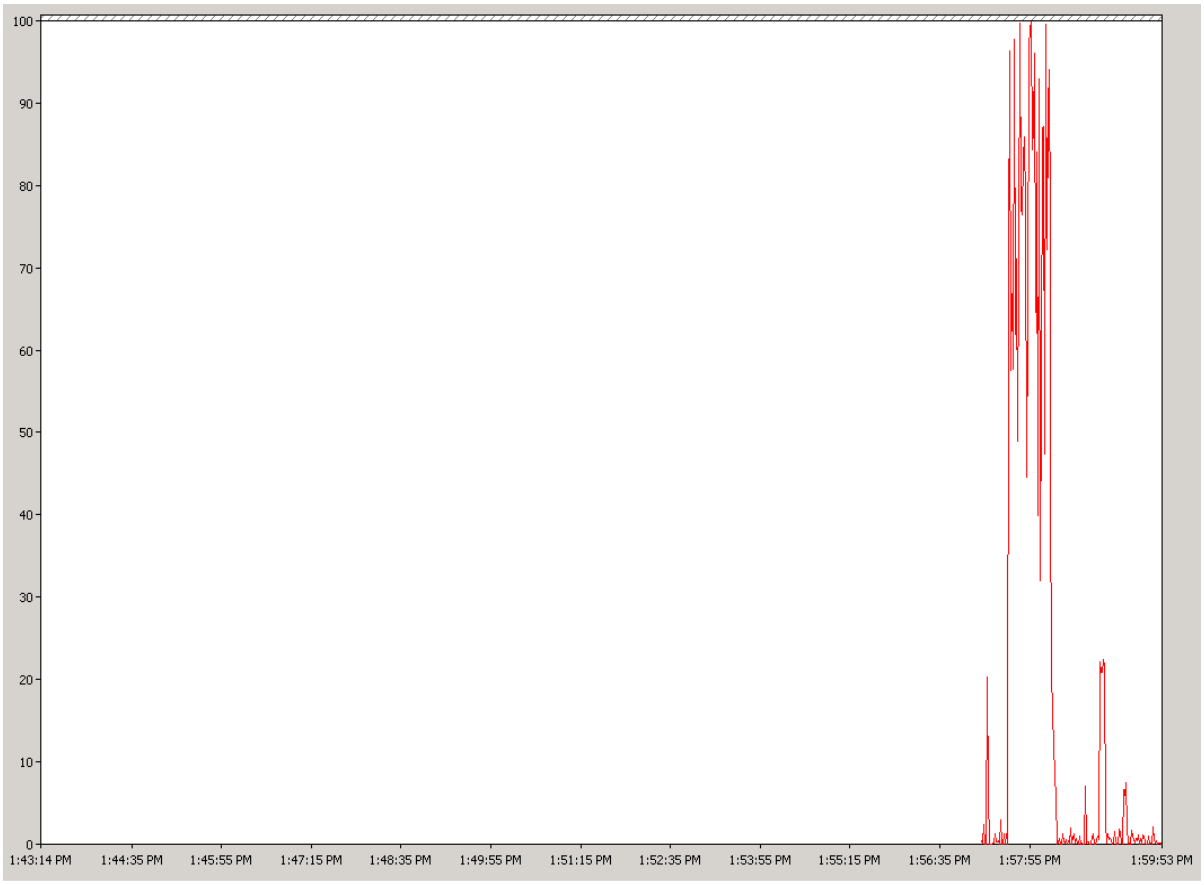


Figura 5.2: Consumo de CPU do Servidor de Banco de Dados (8 núcleos) durante a Inicialização do Sistema Distribuído com Otimizações.

Tabela 5.3: Tempo de Inicialização com o Agrupamento de Instâncias Lógicas em um Servidor Físico.

Instâncias Lógicas	Tempo de Carga (seg.)	*Mbytes*
1	6,022"	533
2	12,8745"	1.002
3	18,0264"	1.496
4	27,6133"	1.998
5	30,107"	2.436
6	38,2855"	3.397
7	42,6664"	4.004
8	48,4815"	4.219
9	56,405"	4.891
10	68,1319"	5.620
11	73,0103"	6.330

Assim sendo, é possível observar que existe uma linearidade no tempo de carga e no

uso de memória, tornando possível prever o comportamento do sistema a medida em que mais instâncias lógicas são agrupadas em um servidor físico. Além de ser possível realizar a carga de 11 instâncias lógicas em um servidor, a nova versão ainda se mostrou mais rápida, com o tempo de 73,01", do que a versão antiga, que inicializa em 16'02"e demandava 10 servidores físicos a mais, conforme mostrado na Tabela 5.1. Isso possibilita um uso mais eficiente dos recursos disponíveis, bem como a diminuição do custo com os servidores necessários para a carga do sistema distribuído.

Porém, pode ser observado na Figura 5.4 que quando ocorre o aumento do número de instâncias lógicas em um servidor físico, ocorre também o aumento do número de mensagens nas filas MSMQ, de tal forma que, mesmo após três cronogramas de conexão, não foi possível tratar todos os pacotes pendentes, pois a vazão de entrada de mensagens nas filas é maior do que o consumo, gerando enfileiramentos.

Também é possível observar na Figura 5.3 que o consumo médio de CPU ficou em 69%, o que torna viável o emprego de algoritmos com suporte à elasticidade, de forma a aumentar o número de *worker threads* para aumentar a vazão de consumo de mensagens e conter o crescimento das filas que estão gerando acúmulos entre cronogramas de conexão. Este tema será detalhado na próxima seção.

5.3 Elasticidade

Os testes de elasticidade visam verificar o aumento e a diminuição de *worker threads*, diante do comportamento do enfileiramento de mensagens e do uso de CPU.

Para os testes de elasticidade foi utilizada uma máquina virtual, com quatro núcleos e 4 *gigabytes* de memória. Também, foi criado um simulador, onde foram codificadas duas *threads*: uma produtora que escreve pacotes na *message queue*, e outra consumidora que lê da *message queue* e dispacha a mensagem para o *singleton ElasticityManager*, classe que contém as implementações do *middleware* desenvolvido.

As subseções a seguir descrevem os testes que visam verificar a sobrecarga da estrutura de dados responsável pela separação das filas e o consumo das mensagens das subfilas, 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 do desvio padrão da vazão média de entrada por conta, a ativação dos algoritmos de elasticidade após o crescimento da fila para além do limite do ponto de entrada.

5.3.1 Sobrecarga da Estrutura de Dados

Estes testes visam verificar a sobrecarga que a estrutura de dados, desenvolvida no *middleware*, apresenta ao realizar o tratamento de uma mensagem, para medir o impacto

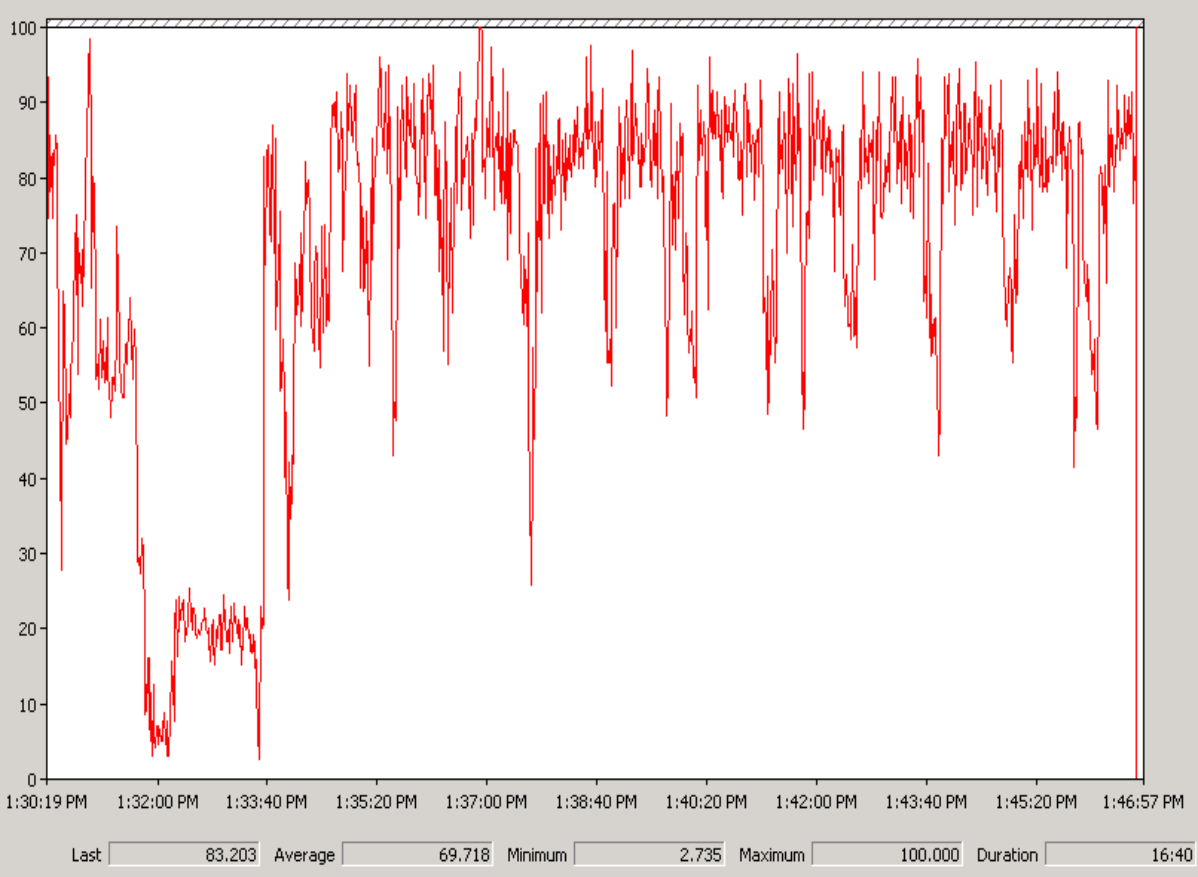


Figura 5.3: Consumo de CPU com 11 Instâncias Lógicas Agrupadas em um Servidor Físico.

da separação e o consumo das subfilas de mensagens, sem considerar o funcionamento dos algoritmos de criação e de destruição das *threads*. Assim, foram realizados experimentos com variações nos tempos de espera entre a geração e o consumo de pacotes nas duas *threads* criadas no simulador.

Primeiramente, foi colocado um tempo de espera de 50 milisegundos entre as mensagens, tanto na *thread* de geração de pacotes quanto na *thread* de consumo. Os testes mostraram que em cinco minutos ocorreu o enfileiramento de 92 mensagens, como mostrado na Figura 5.5. Com isso é perceptível que existe um atraso de 0.3 mensagens por segundo, durante o tratamento pela estrutura de dados responsável pela separação das filas e consumo das mensagens, uma vez que não existe diferença nos intervalos de tempo de espera nas *threads* de geração e de consumo de mensagens.

Em seguida, o tempo de dormir da *thread* produtora foi diminuido para 10 milisegundos, de forma a aumentar a taxa na *thread* de geração de mensagens. O acúmulo de mensagens foi maior que no caso anterior, chegando ao pico de 23.651 mensagens em cinco minutos, como mostrado na Figura 5.6. Apesar do aumento da frequência na geração de mensagens ter sido de cinco vezes, o aumento do acúmulo foi de 257 vezes, demonstrando

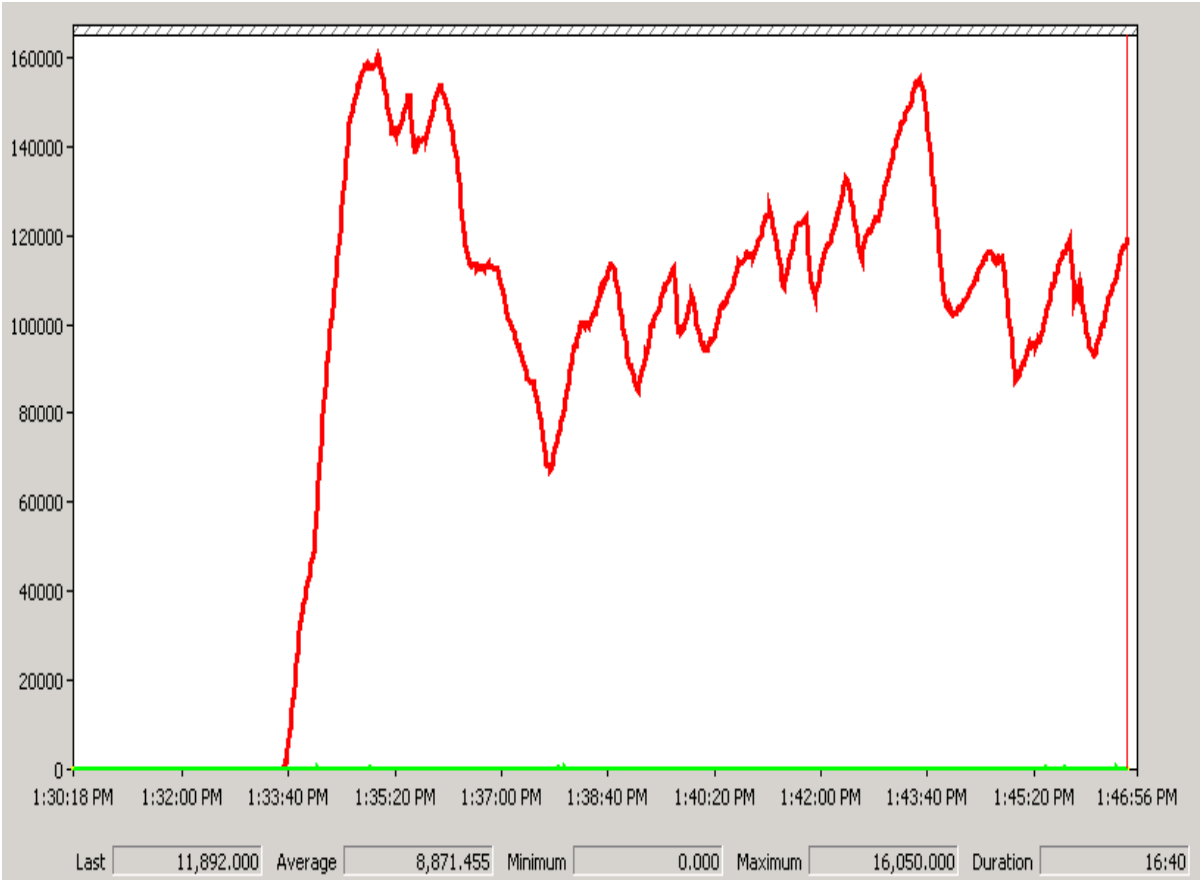


Figura 5.4: Fila de Mensagens com 11 Instâncias Lógicas Agrupadas em um Servidor Físico.

um comportamento não linear em relação ao aumento do acúmulo.

Assim, após constatar essa situação, foram realizadas algumas modificações como tentativa de melhorar o desempenho do consumo de mensagens. Entre elas estão a inicialização da lista de *worker buffers* previamente para evitar as serializações de criação dinâmicas das estruturas responsáveis pelo tratamento, e consumo de mensagens de contas novas. Assim, quando uma mensagem de uma nova conta chega, as únicas serializações passam a ocorrer somente durante as trocas de listas de leitura e de escrita para o tratamento e o consumo do pacote. Dessa forma, foram realizados novos testes com essas modificações, porém ocorreu uma diminuição no desempenho, com o acúmulo de 125 mensagens em cinco minutos, conforme mostrado na Figura 5.7. O gráfico de acúmulo passou de linear para logarítmico com um tempo de atraso de 0.41 mensagens por segundo.

Devido a esses problemas de desempenho, foram feitas diversas otimizações na estrutura de consumo e sinalização das *worker threads*. O consumo de mensagens passou a ter uma lista de sinalização por conta, para que a *worker thread*, ao ser acordada pelo semáforo, consuma diretamente o pacote mais antigo, sem precisar procurar por uma conta que ainda não foi tratada na rodada atual. Após essas mudanças, a fila de mensagens teve

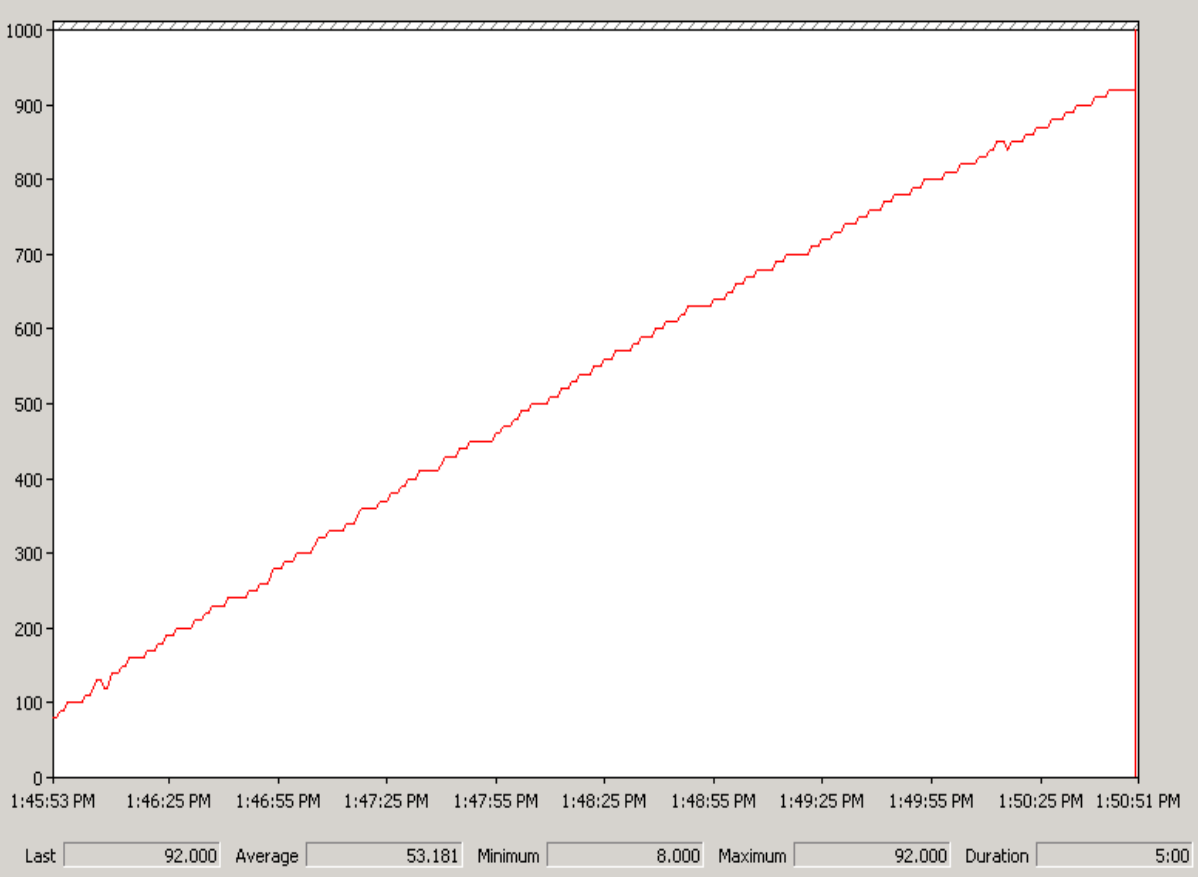


Figura 5.5: Produtor e Consumidor com 50ms.

pico de três mensagens, e média de 1,244 mensagens durante cinco minutos, como mostrado na Figura 5.8. Isso demostra que a sobrecarga da estrutura, após as otimizações, é praticamente inexistente.

A próxima seção realiza os testes de criação e remoção de *threads* para atender a vazão de mensagens identificada.

5.3.2 Criação e Remoção de *Threads* Baseadas na Vazão Média de Entrada e de Saída de Mensagens

Estes testes tem o objetivo de verificar o funcionamento da criação e da remoção de *worker threads*, dinamicamente, em função da medição da vazão média de entrada e de saída de mensagens.

Inicialmente, foram feitos testes com rajadas de geração de pacotes a cada 10 milisegundos e consumo a cada 50 milisegundos, ou seja, com a geração de mensagens a uma taxa cinco vezes maior do que o consumo, de forma a testar o aumento dinâmico do número de *threads*. Sendo assim, os experimentos mostraram que ocorreu o aumento

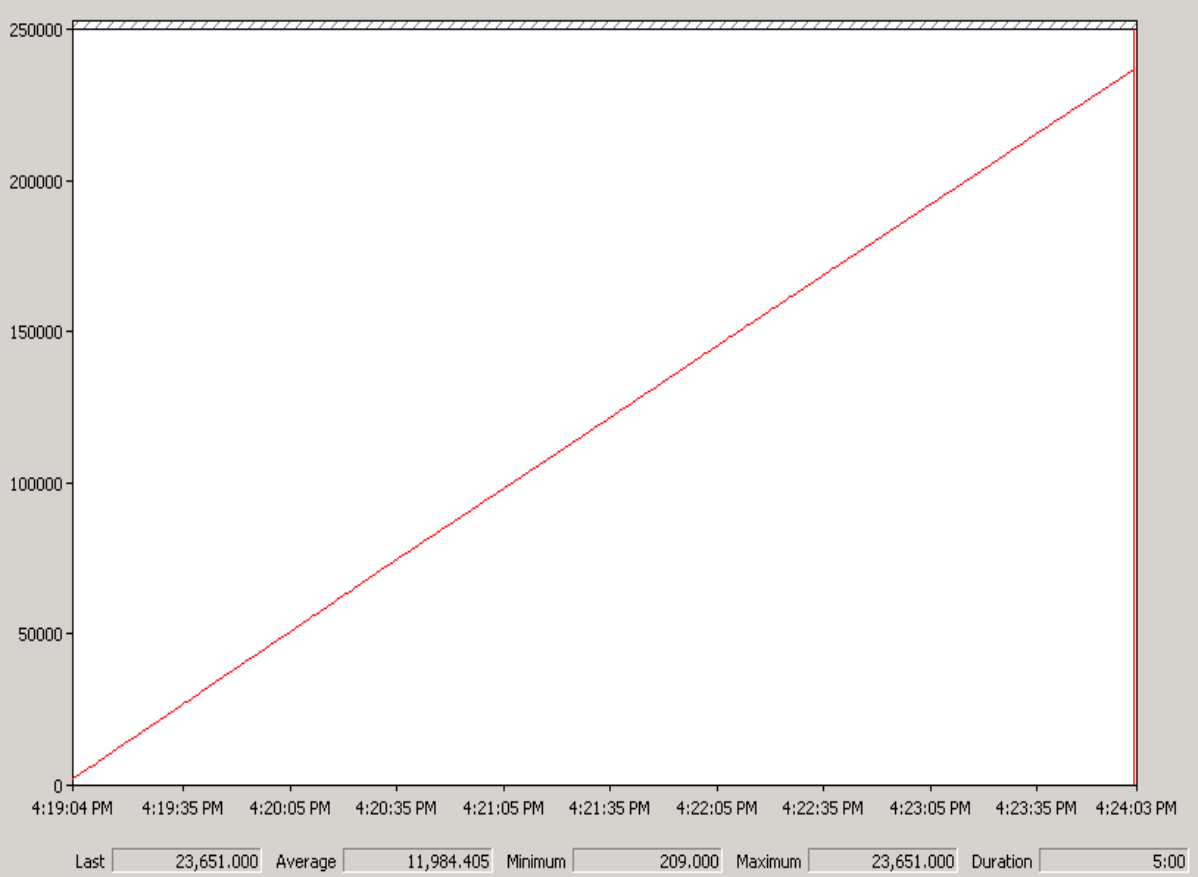


Figura 5.6: Produtor a 10ms e Consumidor a 50ms.

do número de *worker threads* para incrementar a vazão de consumo, como mostram as Figuras 5.9 e 5.10.

Tabela 5.4: Tempos do Algoritmo de Elasticidade.

Momento	Início	Duração	Entrada	Saída	*Threads*	Enfileiramento
Início	18:20:06	14	98	19	1	0
Scale Up	18:20:20	24	99	148.81	7	1280
Scale Down	18:20:44	42	99	95.44	6	175

No cenário em questão, o valor do ponto de entrada foi detectado com 1.280 mensagens na fila, aos 14 segundos após o início da rajada de mensagens, como mostrado na Tabela 5.4. Aos 18 segundos após escalar para cima, o algoritmo decide baixar uma *thread* de forma que a vazão média de saída fique compatível com a vazão média de entrada. Apesar de se mostrar satisfatório quanto à criação e remoção de *worker threads*, em função da vazão, o número de mensagens ficou em um nível alto, pois a estabilização do algoritmo ocorreu com 175 mensagens de entrada e saída, ou seja, um valor não próximo de zero.

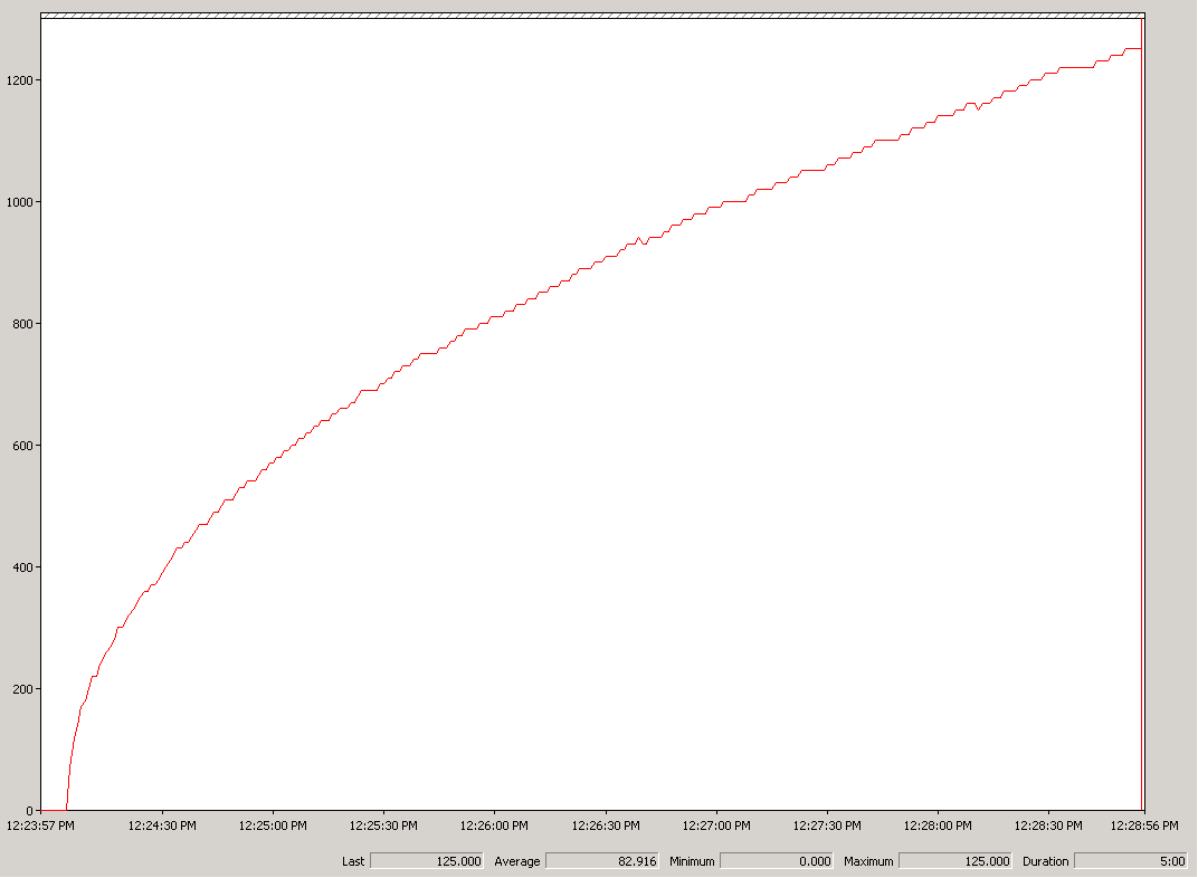


Figura 5.7: Produtor e Consumidor a 50ms e Inicialização dos *Worker Buffers*.

Dessa forma, na próxima subseção serão vistas as otimizações necessárias no algoritmo de elasticidade para que o número de mensagens se estabilize em um valor mais próximo de zero.

Otimizações

Diante do exposto, para melhorar o desempenho e diminuir o número de mensagens no momento da estabilização da vazão de entrada e de saída, foi realizada a sinalização com *timestamp* associado a conta para cada uma das mensagens, na ordem em que chegam. Assim, a *worker thread* ao ser sinalizada obtém o pacote diretamente, sem a necessidade de buscas por contas que não foram tratadas em uma rodada justa. Após estas otimizações foram obtidos os resultados mostrados nos gráficos de fila de mensagens, CPU e *threads*, mostrados nas Figuras 5.11 e 5.12.

De acordo com a Tabela 5.5, a vazão de saída após o *Scale Down* é de 99 mensagens por segundo e, portanto, se mantém mais próxima da vazão de entrada do que a versão sem otimizações, que era de 95.44 mensagens por segundo, de acordo com a Tabela 5.4.

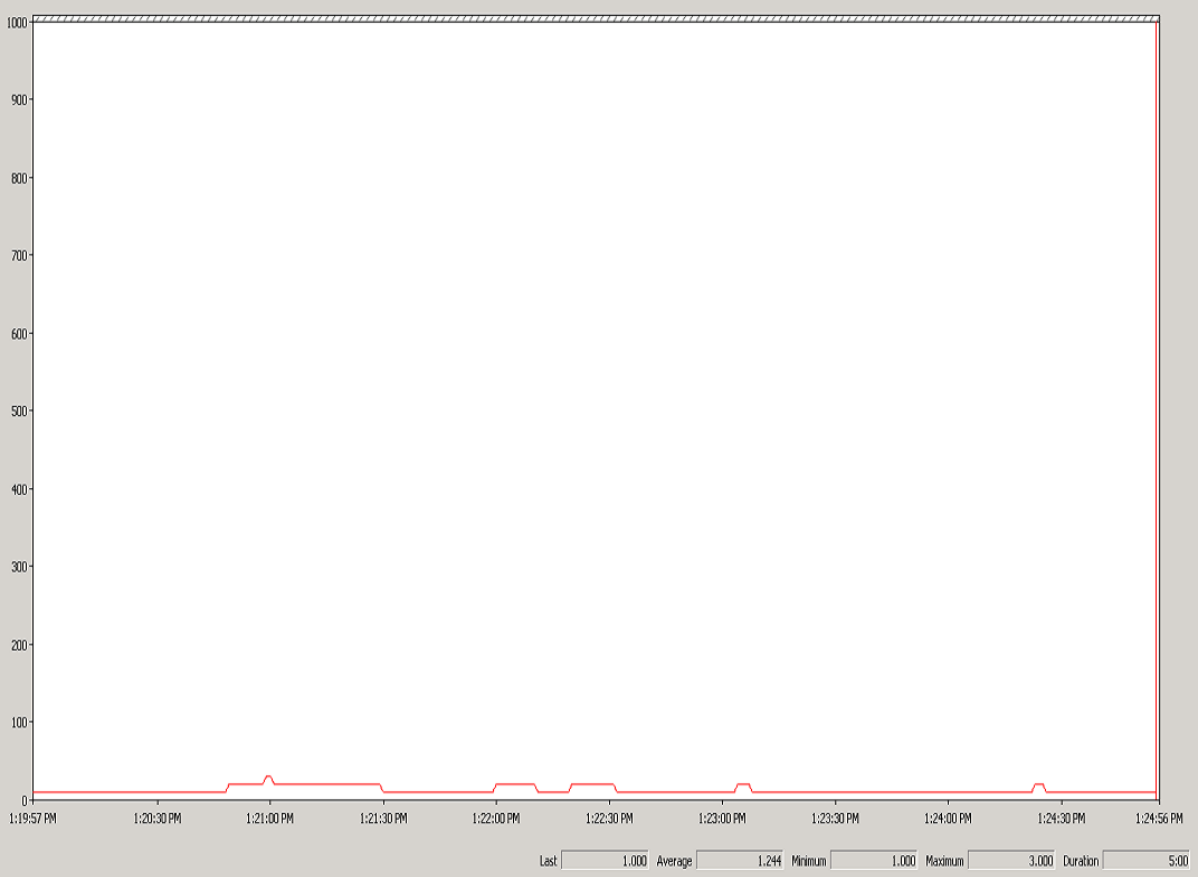


Figura 5.8: Produtor e Consumidor a 50ms e com Otimizações de Sinalização e Consumo.

Tabela 5.5: Novos Tempos do Algoritmo de Elasticidade.

Momento	Início	Duração	Entrada	Saída	*Threads*	Enfileiramento
Início	13:20:57	14	99	19	1	0
Scale Up	13:21:11	9	99	197	12	1130
Scale Down	13:21:20	57	99	99	7	33

Além disso, a fila se estabiliza com 33 mensagens em média, ou seja, mais próxima de zero do que a versão anterior que era de 175.

Por outro lado, é perceptível na Figura 5.12 que durante os momentos de *Scale Up* e *Scale Down*, ocorrem picos de uso de CPU. Isto se deve à necessidade de procurar e mover entre *threads* as novas estruturas de dados que contém os *timestamps* associados às mensagens. Porém, mesmo com esta necessidade de processamento adicional, é possível observar que, de acordo com a Tabela 5.5, a duração do *Scale Up* diminuiu de 24 para 9 segundos, ou seja, ocorreu um ganho de desempenho de 62.5%.

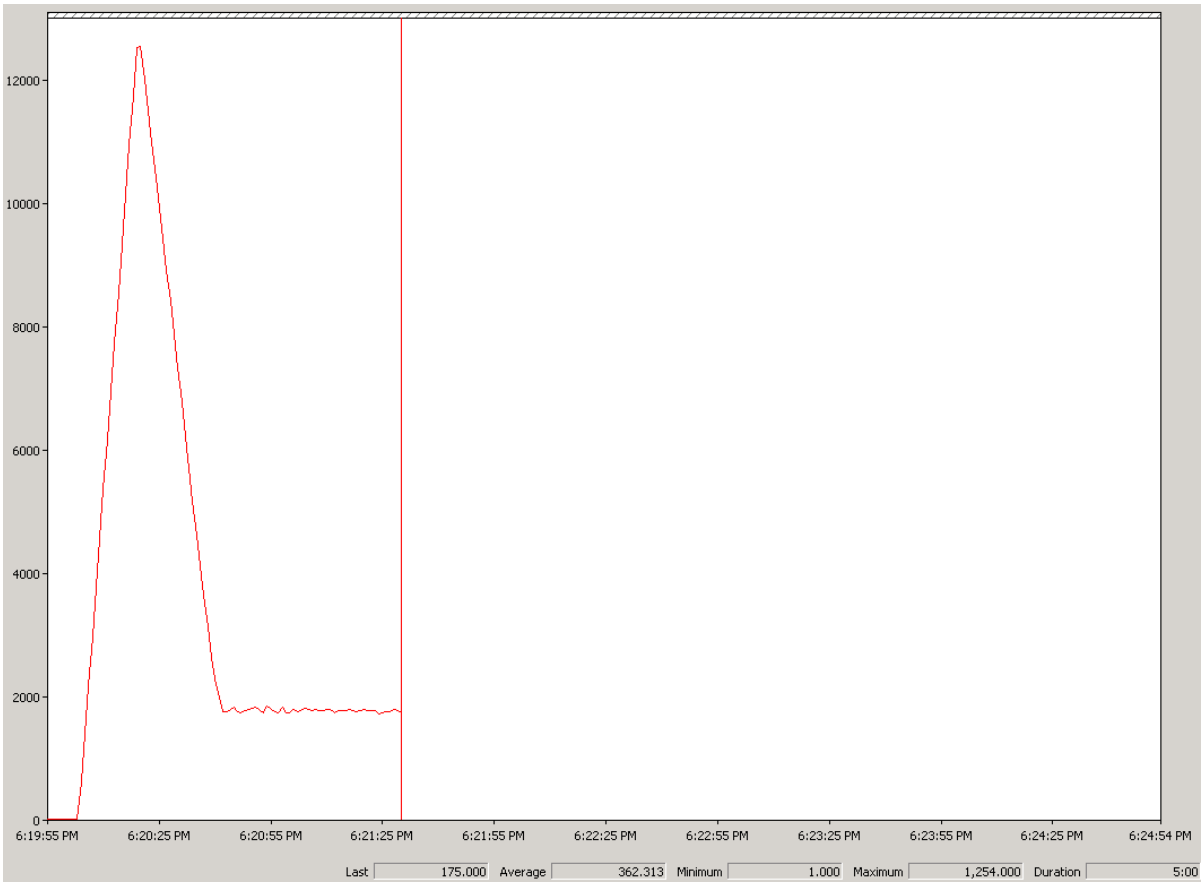


Figura 5.9: Comportamento da Fila de Mensagens com o Aumento das *Threads*.

5.3.3 Criação de *Threads* Limitada pelo Consumo Médio de CPU

O objetivo destes testes é validar se os algoritmo de elasticidade evitam 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*[1] 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 de mensagens. Os resultados de uso de CPU, *threads* e tamanho das filas de mensagens são mostrados nas Figuras 5.13 e 5.14.

Como pode ser observado na Figura 5.13, 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 5.11, pois não são criadas todas as *worker threads* necessárias para que a diminuição das mensagens nas filas ocorra no tempo requerido.

[1] Consumo de CPU esperando que um recurso seja liberado.

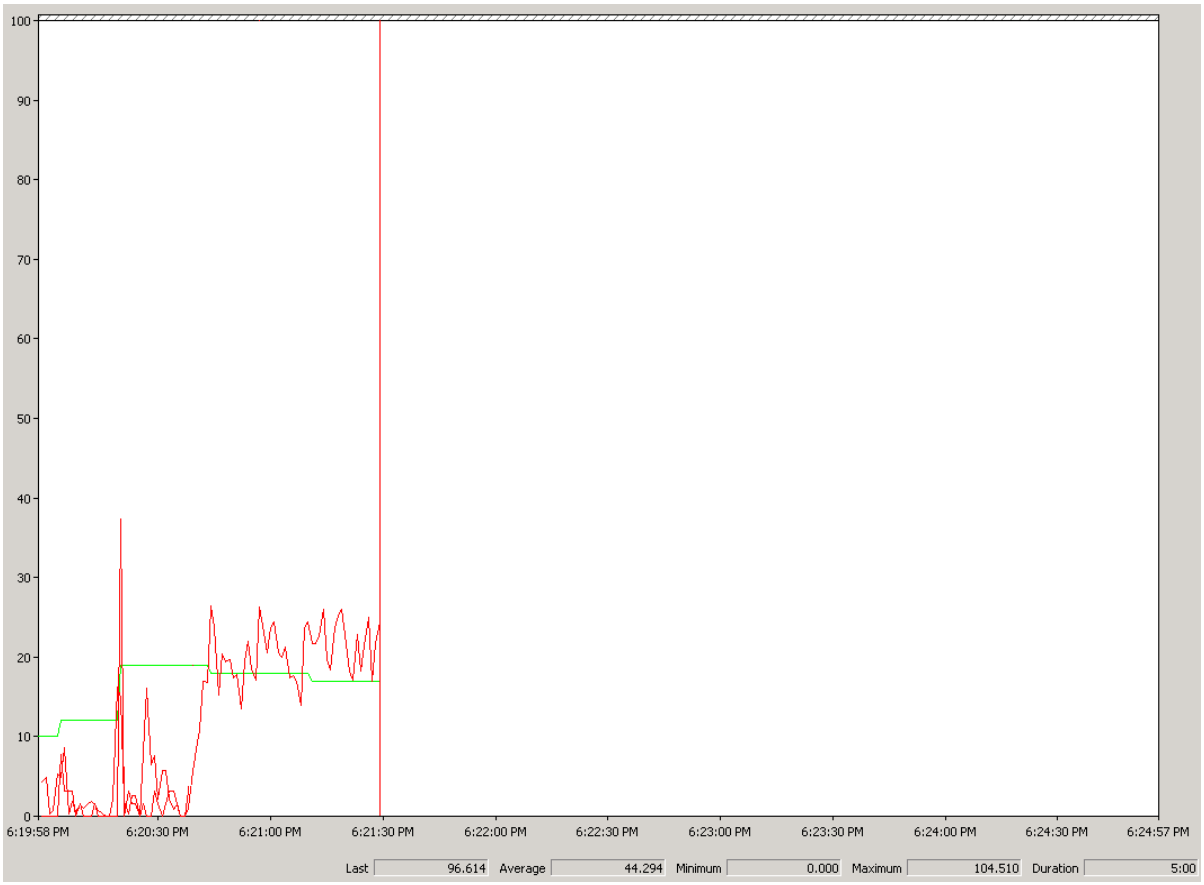


Figura 5.10: Comportamento da CPU e *Threads* com a Ativação da Elasticidade.

Tabela 5.6: Comportamento das *Threads* no Algoritmo de Elasticidade com Limites de CPU.

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

Tabela 5.7: Comportamento da CPU no Algoritmo de Elasticidade com Limites de CPU.

Momento	Média por *Worker Thread*	Total Necessária	Total Residual do Servidor
Scale Up	11.3088%	90.47%	73.89%
ScaleDown	N/A	N/A	N/A

De acordo com a Tabela 5.6, o *middleware* deveria ter criado oito *worker threads*, baseado na vazão média de entrada. Porém, como mostrado na Tabela 5.7, o consumo médio de CPU por *worker thread* foi de 11.30%, e ultrapassaria o limite de CPU residual

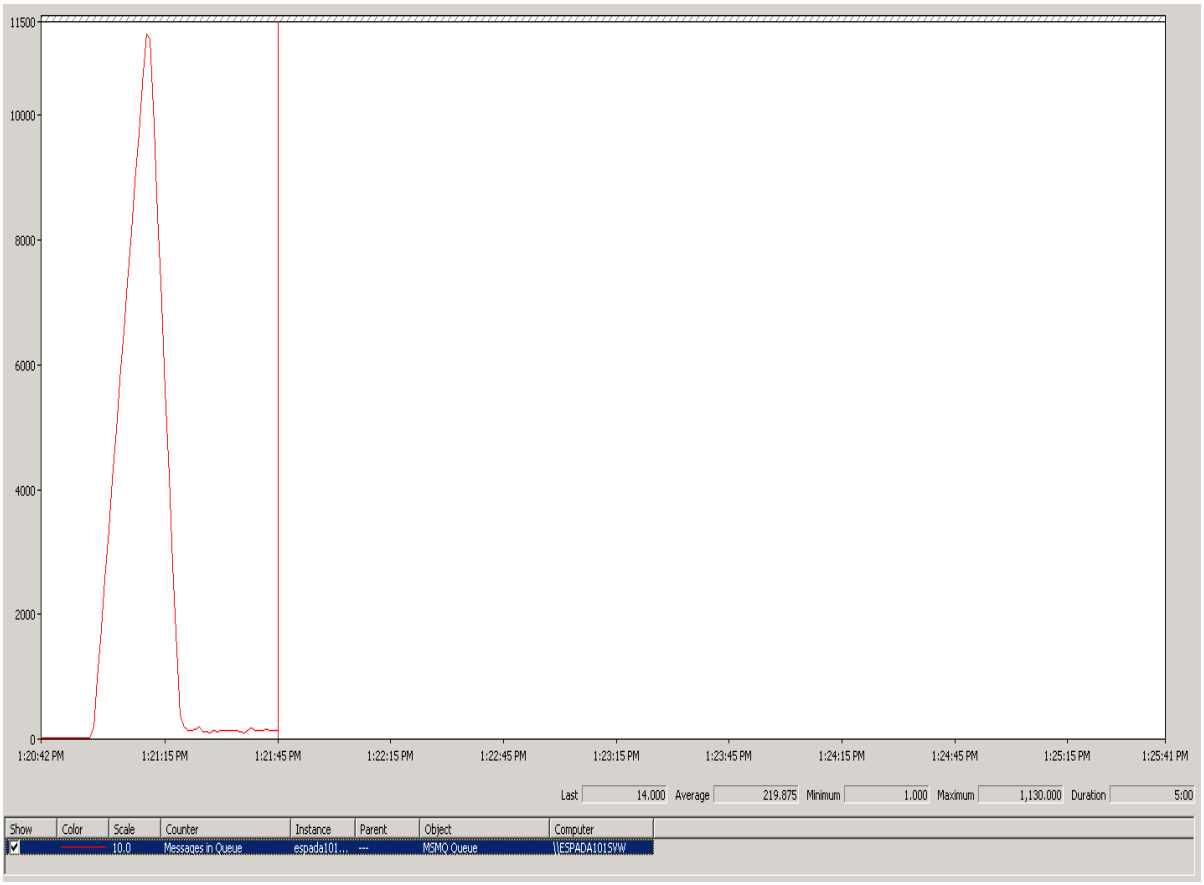


Figura 5.11: Novo Comportamento da Fila de Mensagens com as Otimizações de Sinalização para Consumo.

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*.

5.3.4 Criação de *Threads* Limitada pelo Desvio Padrão da Vazão Média de Saída por Conta

Estes testes visam verificar o funcionamento da criação de *worker threads* limitada pelo número de contas que se distanciam para um valor duas vezes acima do desvio padrão da vazão média de saída por contas. Sendo assim, estes testes objetivam validar se durante uma rajada de pacotes, as mensagens são originadas de poucas contas, pois se forem criadas as *worker threads* baseado somente na vazão média de saída, serão desperdiçados recursos com a criação de *worker threads* que não irão conter o crescimento da

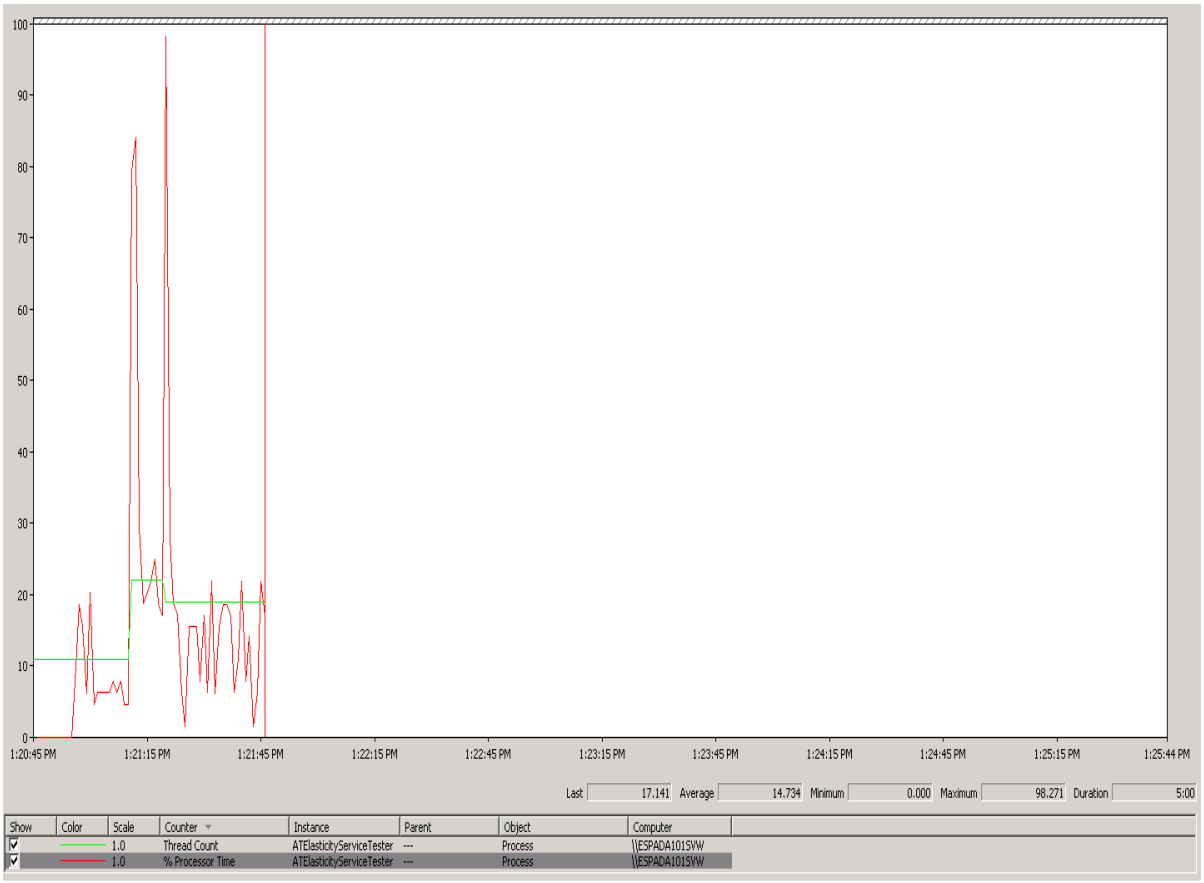


Figura 5.12: Novo Comportamento da CPU e *Threads* com as Otimizações de Sinalização para Consumo.

fila. Dessa forma, este procedimento visa testar a detecção de contas que tenham rajadas de mensagens isoladas e, verificar se ocorreu a diminuição do uso de recursos por meio de um número menor de *worker threads* criadas.

Neste cenário foram realizados testes de rajadas consecutivas para três contas diferentes, e utilizado o tempo de *sleep* de 10 milisegundos na *thread* de geração e de 50 milisegundos na *thread* de consumo. Foram obtidos os resultados mostrados nos gráficos de fila de mensagens, CPU e *threads* das Figuras 5.15 e 5.16.

De acordo com a Figura 5.16 são criadas apenas três *threads*. Além disso, as filas de mensagens não conseguem ser contidas, como pode ser observado na Figura 5.15. Este comportamento ocorre, pois não é possível criar mais de uma *thread* por conta, devido a restrição de integridade sequencial dos dados provenientes de uma mesma conta. Assim, apesar de ocorrer o aumento do número de *threads*, o número de pacotes nas filas não diminui, pois não há possibilidade de separar os dados de uma mesma conta para paralelizar o processamento. Dessa forma, foi possível economizar recursos, criando menos *worker threads* do que as que seriam detectadas pelo algoritmo associado à vazão

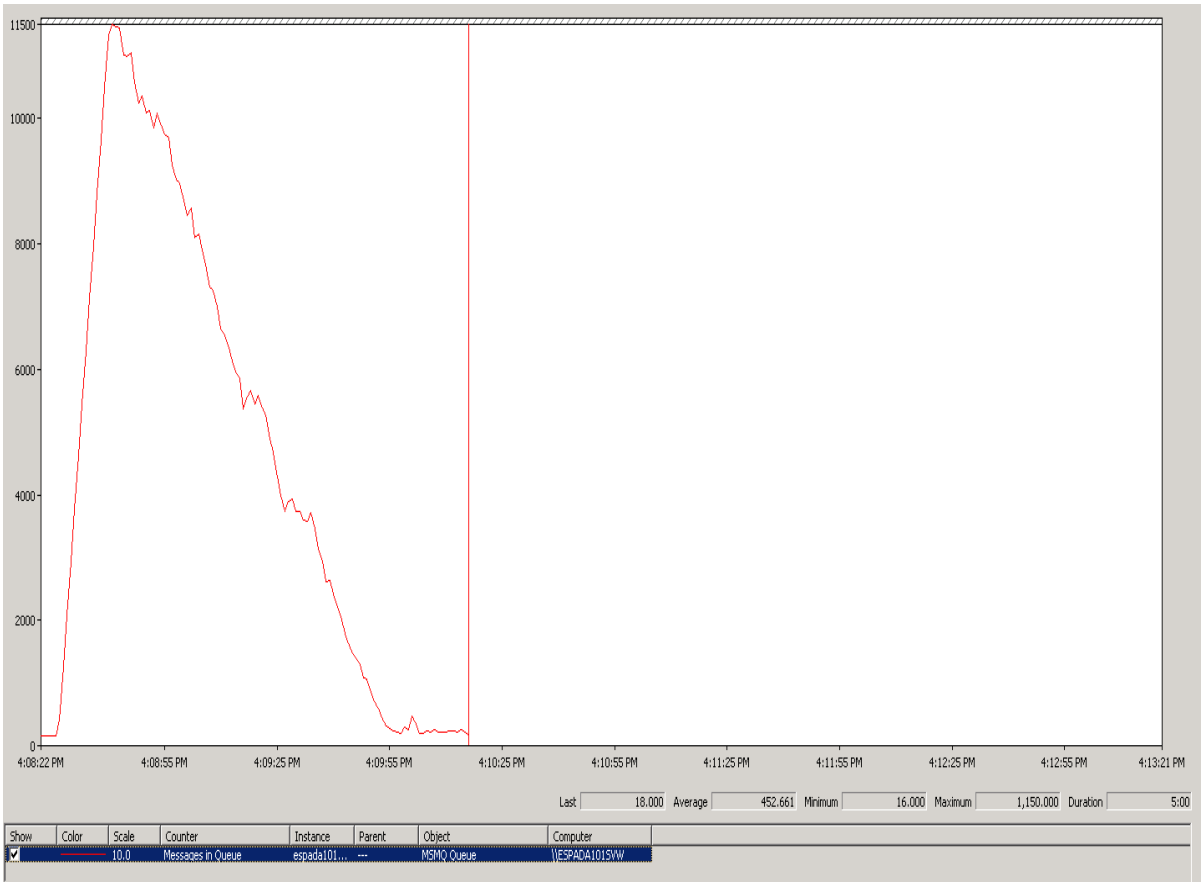


Figura 5.13: Comportamento da Fila de Mensagens com Limites de CPU.

média de saída.

5.3.5 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 ponto de entrada.

Os resultados obtidos são mostrados nas Figuras 5.17 e 5.18. A reta paralela ao eixo x na Figura 5.17, e a inexistência de *threads* no mesmo intervalo de tempo na Figura 5.18, correspondem ao tempo em que o processo ficou inoperante. Assim, alguns segundos após o reinício do processo, a inclinação do consumo de mensagens aumenta, indicando o incremento do número de *worker threads* para consumir as mensagens antes do término do cronograma de conexão. Da mesma forma, também é possível observar na Figura 5.18, o aumento do número de *threads* após o reinício rápido do processo. Este comportamento

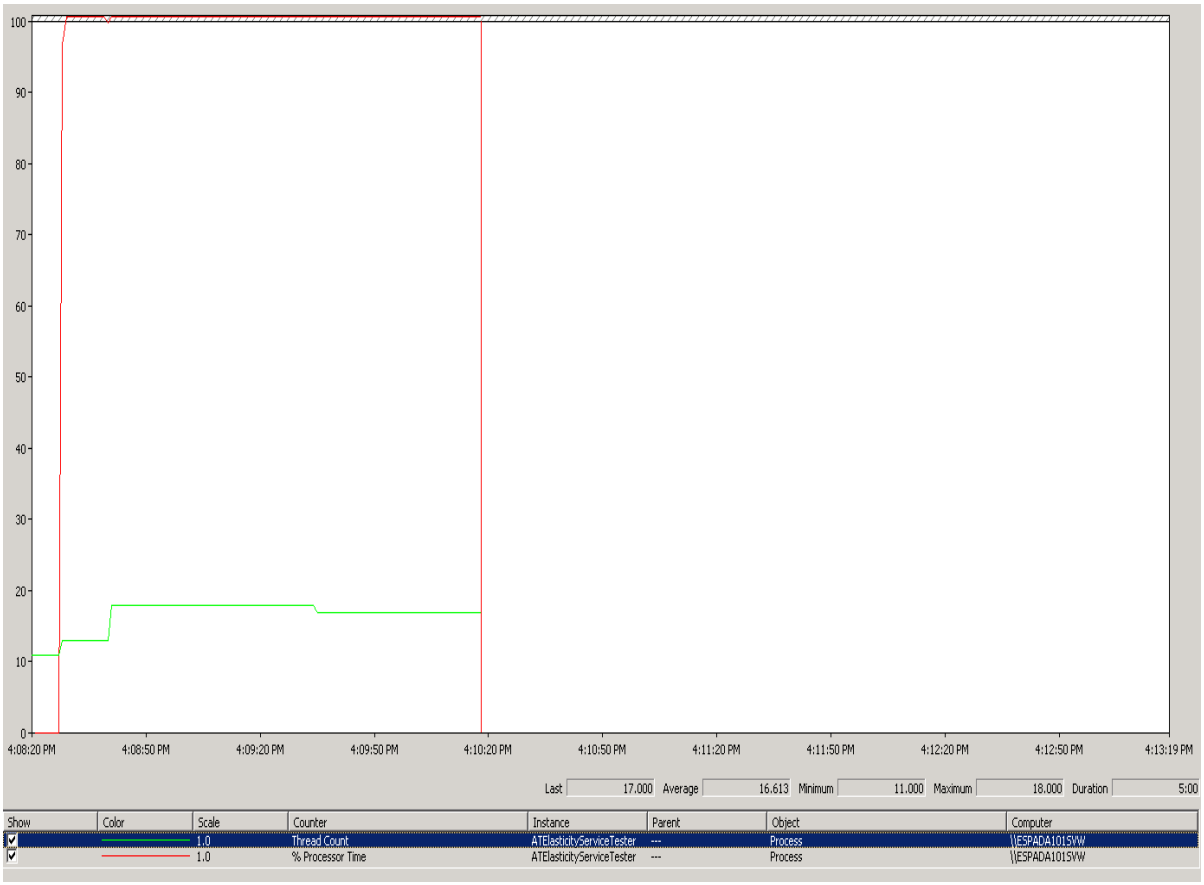


Figura 5.14: Comportamento da CPU e *Threads* com Limites de CPU.

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.

5.4 Adequação do *Middleware* ao Ambiente Real da Arquitetura

Esta seção descreve a migração da camada do *middleware* do ambiente simulado para a arquitetura real de funcionamento do sistema distribuído.

Primeiramente, foi construída uma biblioteca para conter as classes desenvolvidas para o funcionamento do *middleware*. Em seguida, esta biblioteca foi ligada ao projeto da biblioteca de IPC da aplicação distribuída, que é responsável pelo tratamento dos pacotes que são lidos das filas de mensagens. Porém, o acoplamento entre o *middleware* desenvolvido e o IPC é alto, pois a camada de elasticidade precisa acessar as filas diretamente, para que possa ler, tratar, mover e retirar mensagens. Sendo assim, as classes desenvolvidas no *middleware*, foram acopladas diretamente na biblioteca da camada de IPC da aplicação distribuída. As próximas seções descrevem as modificações que foram necessárias durante

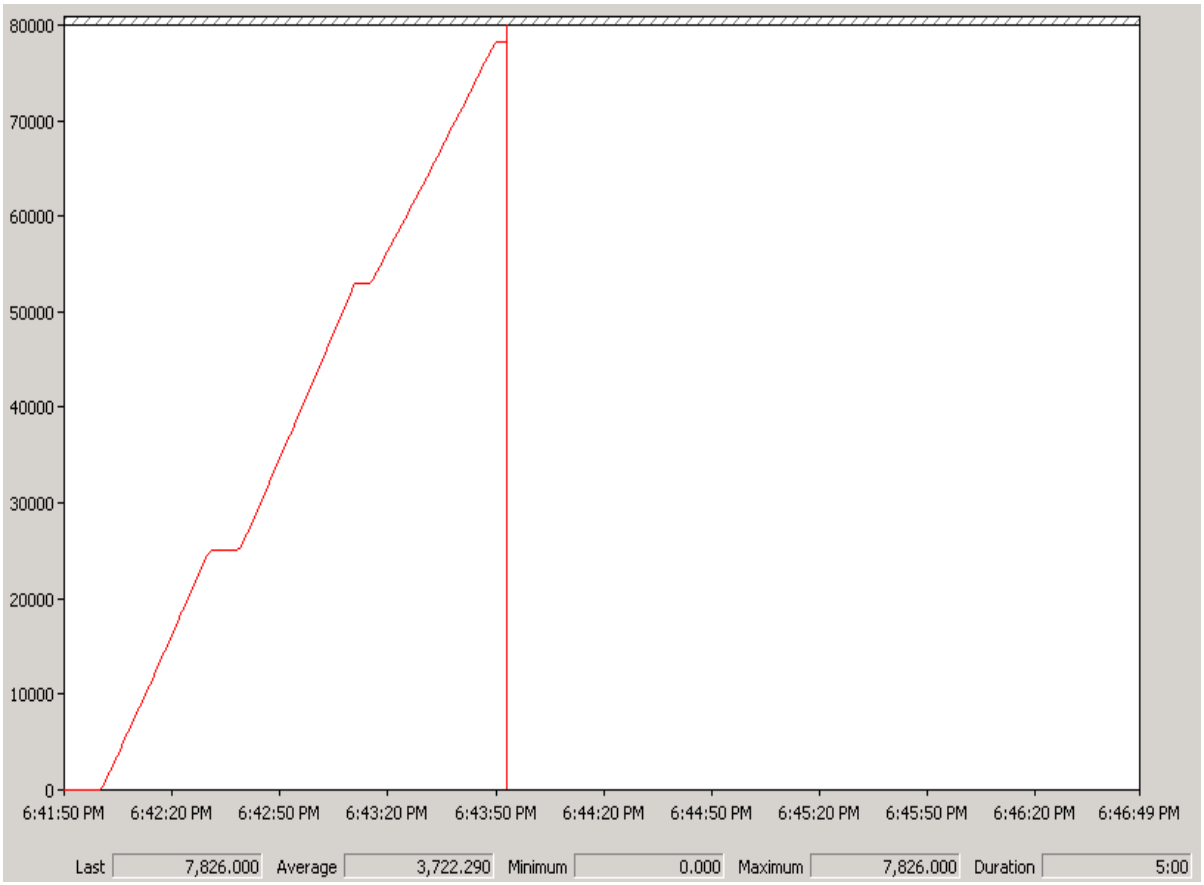


Figura 5.15: Comportamento da Fila de Mensagens com Rajadas de Três Contas.

a adequação do *middleware* na camada de IPC, e o teste do *middleware* adequado no ambiente real da arquitetura.

5.4.1 Modificação do Protocolo de IPC

O *middleware* trabalha com a categorização das mensagens em grupos, de acordo com a conta, para que possa paralelizar o tratamento dos pacotes por meio de subfilas. Sendo assim, foi necessário modificar o protocolo de IPC para que o *header* desse protocolo tivesse o campo que será responsável por agrupar pacotes que estão associados à mesma conta. Dessa forma, ao repassar o pacote para o *middleware*, ocorre a associação correta com as estruturas relacionados ao processamento com consumo justo e *round robin*.

Dessa forma, foi criado o campo *index* no *header* de todos os pacotes do protocolo de IPC, com o tamanho de 4 *bytes*, contendo o código da conta ao qual o pacote está associado. Assim, os processos ao lerem uma mensagem a partir da fila, realizam a deserialização rápida do campo *index*, sem a necessidade de realizar o *parsing* completo do *payload* do pacote. Após descoberto o campo *index*, o pacote é repassado para o *middleware*, que então ativa os algoritmos necessários para o tratamento com elasticidade.

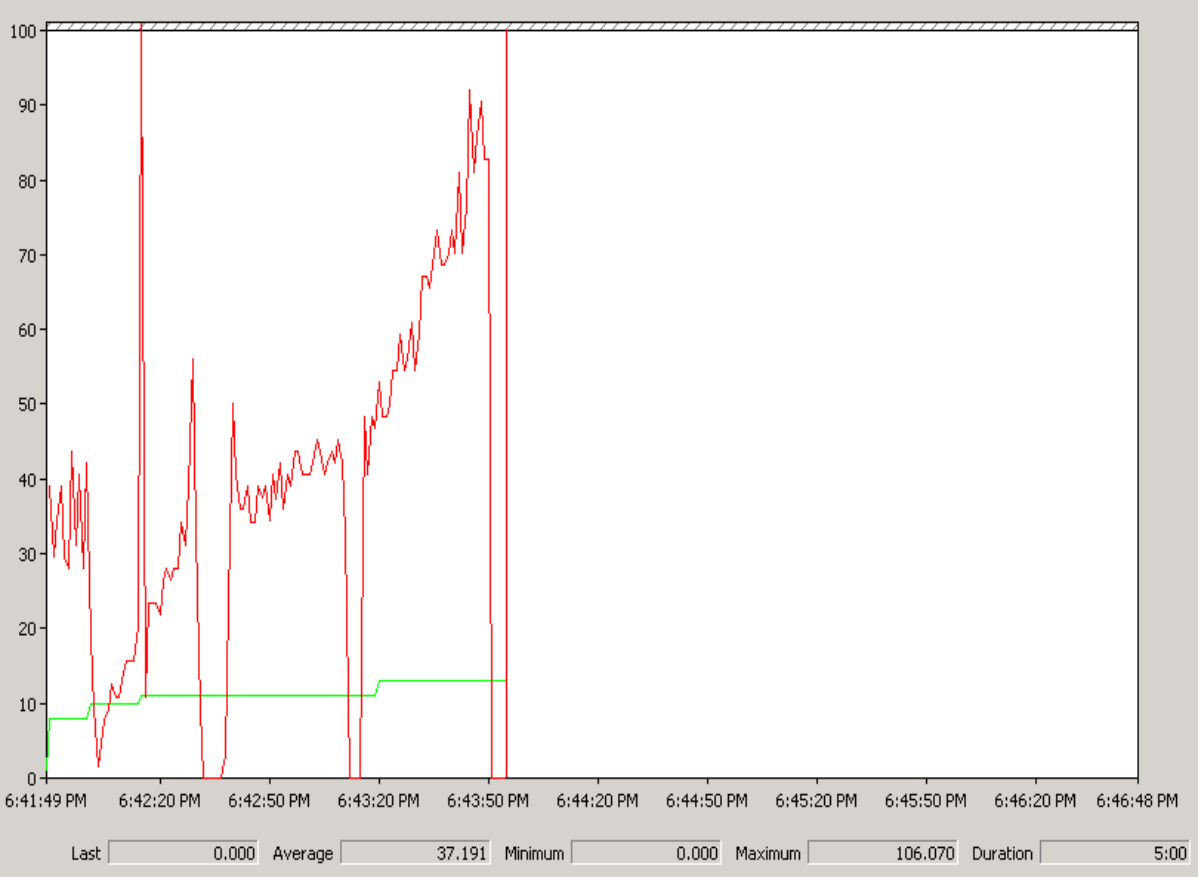


Figura 5.16: Comportamento da CPU e *Threads* com Rajadas de Três Contas.

5.4.2 *Thread* de Monitoramento do Uso de CPU

Ao realizar a migração do *middleware* para a biblioteca de IPC, foi constatado que a *thread* responsável pela medição do consumo de CPU estava com um uso médio de 9% de CPU em 5 minutos, como mostrado na Figura 5.19. Este consumo é alto, se for considerado que cada um dos 19 processos possui esta mesma *thread* de monitoramento e, inviabiliza o uso do *middleware*. Portanto, foi necessário realizar uma otimização na forma de obter o uso de CPU.

Assim sendo, foi criada a classe *CpuUsage* pra medir o uso de CPU da *thread* atual sem *polling*, por meio das chamadas de funções *GetThreadTimes* e *GetProcessTimes* para obter a média de uso durante as rajadas de pacotes. Estas funções são forncecidas pela API *Win32*, e obtém a quantidade de tempo em unidades de 100 nanosegundos do tempo de uso do processador pelas *threads* e processos. Dessa forma, foi necessário também obter o *timestamp*, em nanosegundos das chamadas de medição no início e no fim para calcular a porcentagem de tempo utilizada no intervalo. Após estas otimizações, o uso de CPU da *thread* de monitoramento ficou próximo de zero, tornando o seu uso viável.

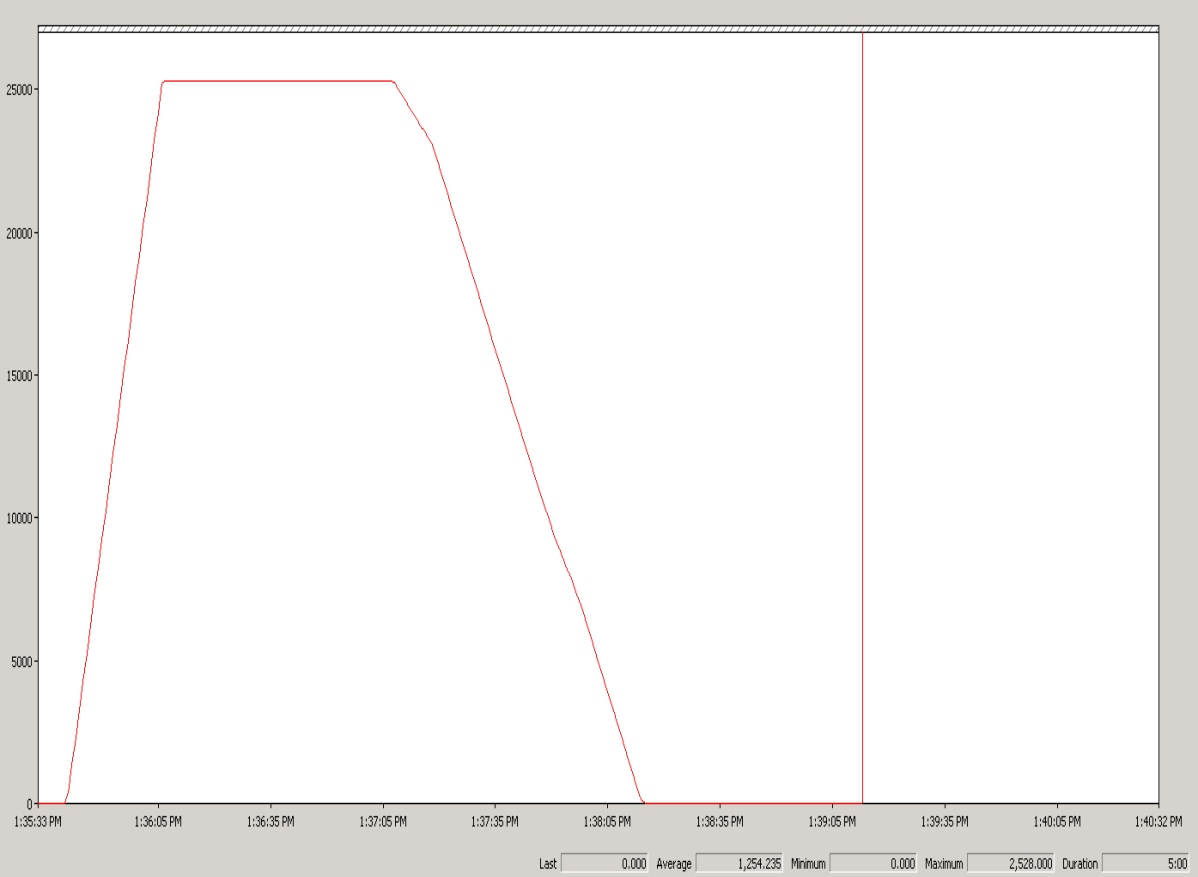


Figura 5.17: Comportamento da Fila de Mensagens após o Reinício Rápido.

5.4.3 Funcionamento do *Middleware* na Arquitetura Real

Após as otimizações e adequações, o *middleware* desenvolvido foi executado no ambiente real da arquiteutura. Para isto, foi utilizado um servidor físico com 8 núcleos e 16 *gigabytes* de memória, que foi configurado com o agrupamento de 11 instâncias lógicas do banco de dados BD2 da Tabela 5.2. Também foi utilizado o servidor de comunicação SERVSIM, para que ocorresse a geração de rajadas de mensagens. Este ambiente tem o objetivo de verificar se, diante do aumento e da diminuição de mensagens nas filas, ocorre a adaptação dinâmica do número de *worker threads* na arquitetura real de funcionamento.

Assim, após a realização dos testes, foi verificado que o serviço de gerenciamento de mensagens teve um acúmulo de mensagens em sua fila, conforme mostrado na Figura 5.20. Dessa forma, o *middleware* ativou a criação de mais *worker threads* após 15 segundos de rajada de mensagens, por meio da verificação do tamanho da fila (1.226), que estava acima do ponto de entrada (1.182), como mostrado na Tabela 5.8.

Além disso, como a média de uso de CPU por *worker thread* foi de 0,104032%, e a CPU residual do servidor estava em 99,5709%, o *middleware* não detectou a necessidade de limitar a criação das *worker threads* necessárias para conter o crescimento de mensagens

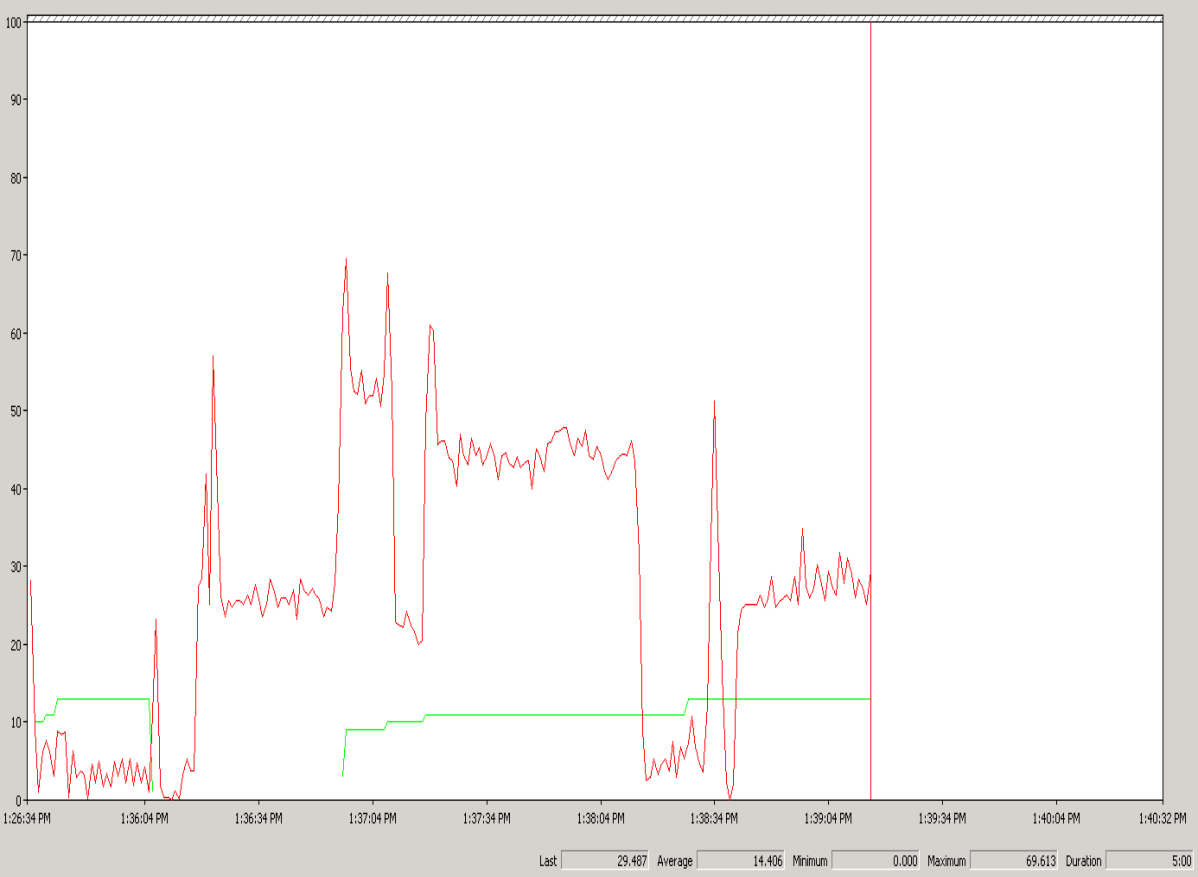


Figura 5.18: Comportamento da CPU após o Reinício Rápido.

e zerar a fila, conforme mostrado na Tabela 5.9. Assim sendo, o *middleware* criou sete *threads*, correspondente ao somatório do valor de *avgIO*, que são as *threads* de contenção, mais o número de *threads* para zerar as filas (1,04214), mais um fator de correção de uma *thread*, para evitar o fenômeno da elasticidade para cima e para baixo, conforme mostrado na Tabela 5.10. O *middleware* também não detectou casos de contas com rajadas isoladas, não ativando o limite de criação de *threads* pelo desvio padrão da vazão média de entrada por conta.

Dessa forma, aos 60 segundos depois do início da rajada de mensagens, a vazão média de entrada foi de 96,09 mensagens por segundo, e a vazão média de saída foi de 114 mensagens por segundo, como mostrado na Tabela 5.11. Assim, a vazão média por *thread* foi de 16,28 mensagens por segundo, uma vez que foram criadas sete *threads*. Dessa forma, o *middleware* não detectou a necessidade de remover *threads*. Isto ocorreu porque dividindo-se a vazão média de saída pela vazão média por *thread*, obtém-se o valor 5,90 *threads*. Porém, como o *middleware* utiliza um fator de correção para cima, este valor foi arredondado para sete, mantendo-se o número de *threads* que foram originalmente criadas, ou seja, novamente para evitar o fenômeno de elasticidade para cima e para baixo.

Tabela 5.8: Medições do *Middleware* Antes do *Scale Up*.

Time	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*	*Entry Point*
0	48,00	14,00	3,43	50	1.120
1	82,50	16,00	6,50	133	1.264
2	87,67	17,00	5,16	212	1.326
3	82,00	17,25	3,61	259	1.328
4	82,40	17,60	4,42	324	1.337
5	87,67	17,83	6,00	420	1.336
6	90,71	18,00	5,74	509	1.332
7	88,63	18,13	3,89	576	1.323
8	91,11	18,22	5,84	656	1.311
9	93,50	18,30	6,05	752	1.299
10	91,64	18,27	4,06	807	1.278
11	90,83	18,17	4,82	888	1.253
12	93,15	18,23	6,37	975	1.239
13	93,86	18,21	5,72	1060	1.220
14	94,07	18,20	5,39	1138	1.201
15	**94,07**	**18,20**	**5,19467**	**1.226**	**1.182**

Tabela 5.9: Comportamento da CPU antes do *Scale Up*.

Vazão Média por *Worker Thread*	CPU Necessária	Residual de CPU
0,104032%	0,728223%	99,5709%

Tabela 5.10: *Scale Up*.

Start	*Reaction*	*ZeroThreads*	*ContentionThreads*	*TotalThreads*	*ZeroTime*	*StdDev*
14:44:59	14:45:14	1.04214	5.19467	7	65	0

Tabela 5.11: Antes do *Scale Down*.

Time	*Input*	*Output*	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*
15	99,00	118,00	99,00	118,00	0,84	1211
16	93,00	128,00	96,00	123,00	0,78	1176
17	75,00	116,00	89,00	120,67	0,74	1137
18	121,00	123,00	97,00	121,25	0,80	1132
19	98,00	131,00	97,20	123,20	0,79	1100
20	84,00	119,00	95,00	122,50	0,77	1065
21	84,00	120,00	93,43	122,14	0,76	1045
22	100,00	124,00	94,25	122,38	0,77	1005
23	118,00	124,00	96,89	122,56	0,79	1001
24	87,00	121,00	95,90	122,40	0,78	970
25	94,00	112,00	95,73	121,45	0,79	946
26	96,00	126,00	95,75	121,83	0,79	928
27	118,00	130,00	97,46	122,46	0,80	905
28	76,00	116,00	95,93	122,00	0,79	875
29	93,00	112,00	95,73	121,33	0,79	846
30	111,00	116,00	96,69	121,00	0,80	841
31	97,00	127,00	96,71	121,35	0,80	817
32	78,00	119,00	95,67	121,22	0,79	769
33	105,00	120,00	96,16	121,16	0,79	755
34	109,00	120,00	96,80	121,10	0,80	744
35	96,00	131,00	96,76	121,57	0,80	709
36	80,00	102,00	96,00	120,68	0,80	691
37	113,00	121,00	96,74	120,70	0,80	679
38	102,00	117,00	96,96	120,54	0,80	664
39	86,00	123,00	96,52	120,64	0,80	626
40	88,00	123,00	96,19	120,73	0,80	594
41	101,00	121,00	96,37	120,74	0,80	570
42	99,00	129,00	96,46	121,04	0,80	543
43	89,00	121,00	96,21	121,03	0,80	510
44	93,00	116,00	96,10	120,87	0,80	487
45	104,00	127,00	96,35	121,06	0,80	464
46	94,00	125,00	96,28	121,19	0,79	434
47	83,00	122,00	95,88	121,21	0,79	393
48	101,00	113,00	96,03	120,97	0,79	382
49	100,00	125,00	96,14	121,09	0,79	357
50	87,00	129,00	95,89	121,31	0,79	317
51	97,00	133,00	95,92	121,62	0,79	280
52	112,00	132,00	96,34	121,89	0,79	259
53	94,00	133,00	96,28	122,18	0,79	225
54	88,00	126,00	96,08	122,28	0,79	181
55	104,00	125,00	96,27	122,34	0,79	161
56	95,00	125,00	96,24	122,40	0,79	129
57	90,00	123,00	96,09	122,42	0,79	97
58	99,00	115,00	96,16	122,25	0,79	81
59	93,00	119,00	96,09	122,18	0,79	56
60	**96,00**	**114,00**	**96,09**	**122,00**	**0,79**	**38**

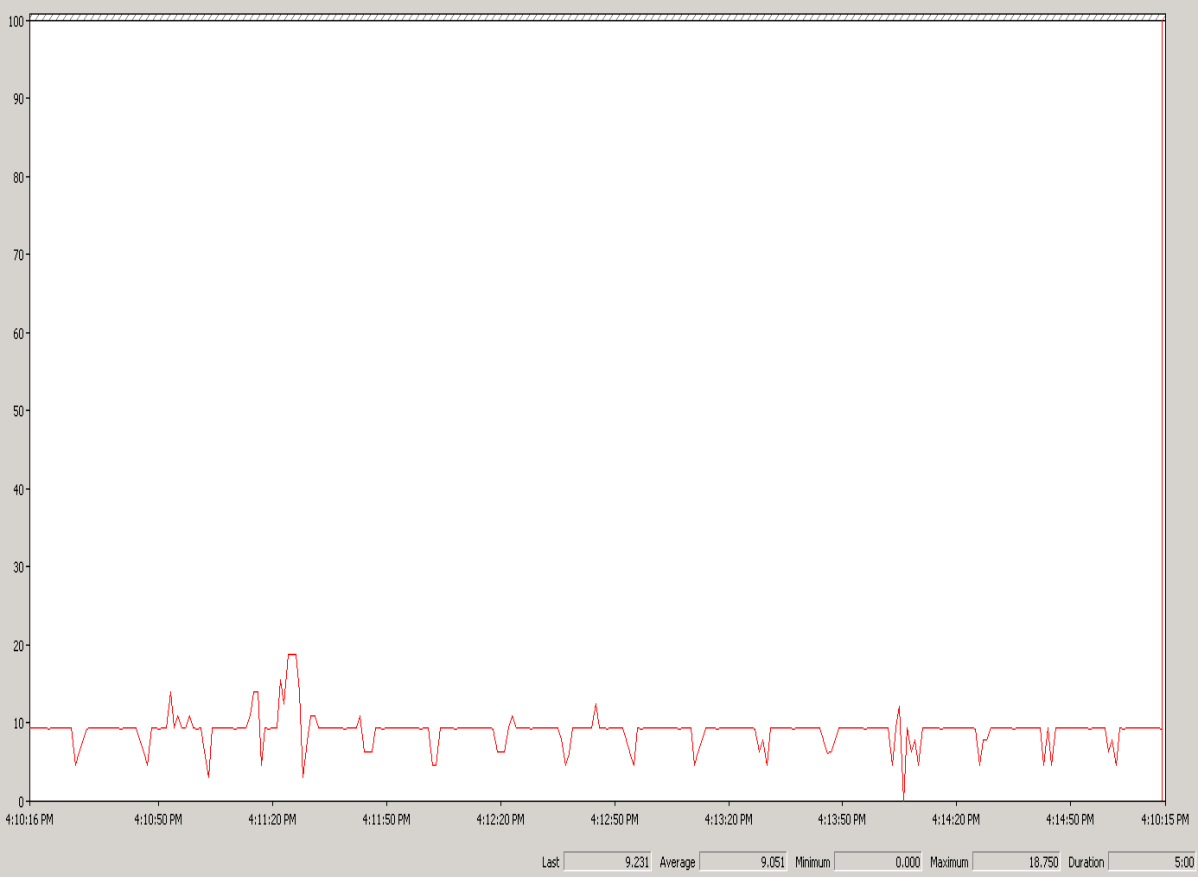


Figura 5.19: Comportamento da CPU da *Thread* de Medição de Consumo de CPU.

Dessa forma, após as otimizações e as adequações, o *middleware* proposto se mostrou satisfatório para a ativação de elasticidade na aplicação distribuída. Para os demais serviços, mostrados na Figura 5.21, o *middleware* não detectou a necessidade de aumento de *worker threads*, pois o número de mensagens acumuladas não alcançou o ponto de entrada.

5.5 Considerações Finais

Como pôde ser visto ao longo deste capítulo, o suporte à tolerância a falhas, por meio da melhoria do tempo de carga inicial e da diminuição do número de servidores necessários para tratar a aplicação distribuída, e à elasticidade, por meio da criação e da remoção de *worker threads* conforme mais ou menos vazão seja requerida para consumir as filas de mensagens, se mostraram aceitáveis para o uso no *cluster* da aplicação distribuída. As demais situações, vistas em ambiente simulado, são de difícil ocorrência em ambiente real, porém, caso ocorram, também mostraram um comportamento satisfatório, como

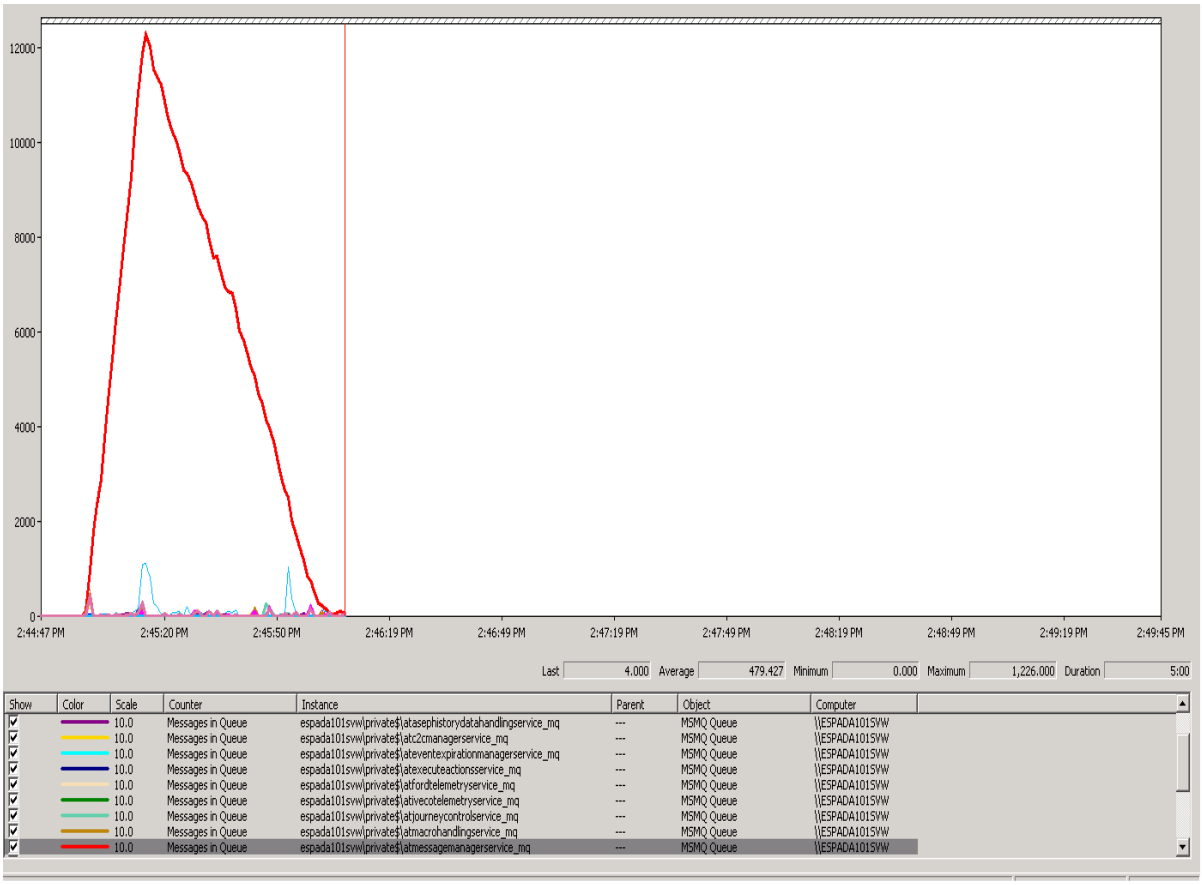


Figura 5.20: Enfileiramento e Contenção da Filas do Serviço de Gerenciamento de Mensagens.

mostrado nas seções anteriores. Assim sendo, serão apresentadas as conclusões pertinentes ao trabalho realizado no próximo capítulo.

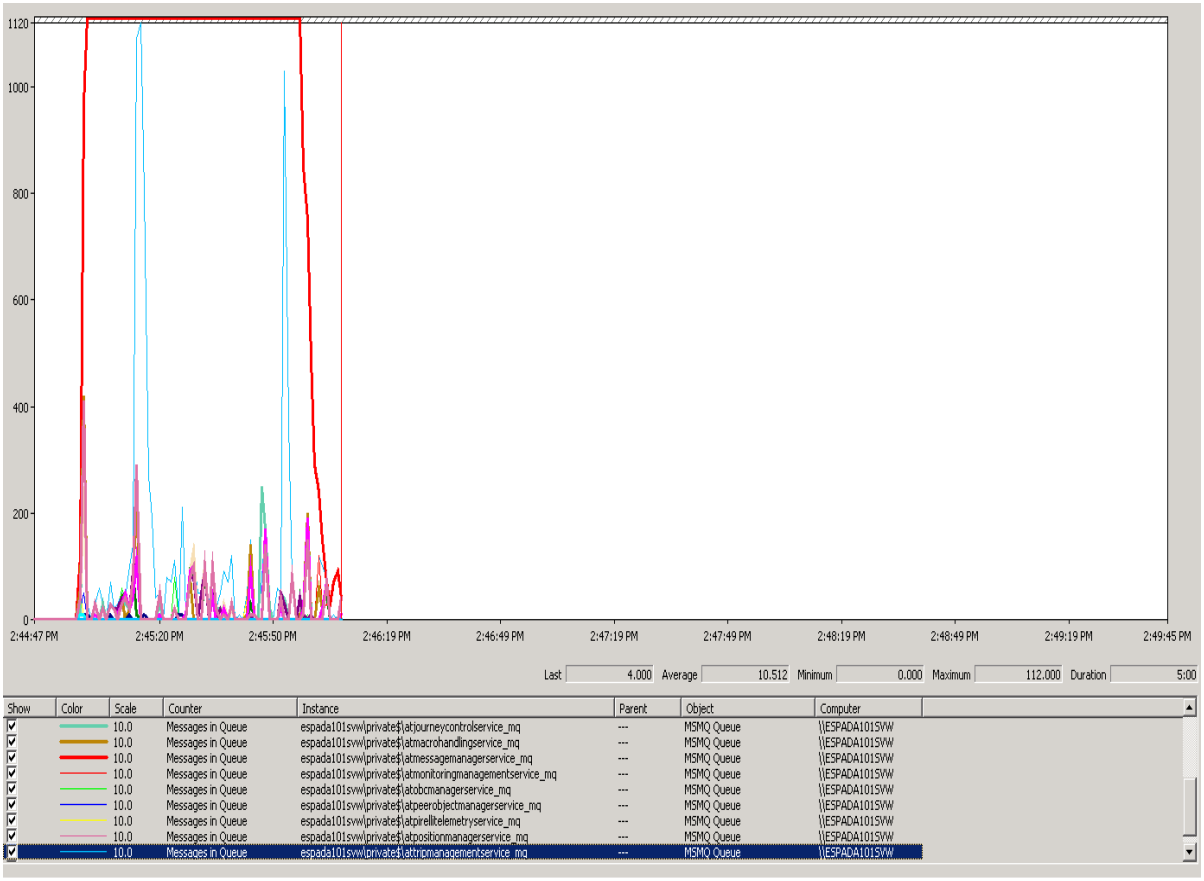


Figura 5.21: Comportamento das Filas de Mensagens após a Adequação do *Middleware*.

Capítulo 6

Conclusão

Este trabalho apresentou conceitos relacionados à sistemas distribuídos, à plataforma *cluster* e à camada de software *middleware*, que é responsável pela abstração de programação das características da arquitetura de computação distribuída utilizada.

Por meio desta pesquisa foi possível obter uma análise dinâmica do comportamento da vazão das filas, e a descoberta de fórmulas para encontrar os pontos críticos de criação e remoção de *worker threads* para a contenção do acúmulo de mensagens. Assim, estas fórmulas puderam ser aplicadas ao *middleware* proposto para suportar as características de elasticidade e tolerância a falhas.

Com o desenvolvimento do suporte à elasticidade e à tolerância a falhas, mostrou-se possível economizar recursos de TI por meio da diminuição do número de servidores necessários para processar as filas de mensagens, uma vez que os recursos de CPU e memória são utilizados mais eficientemente a medida em que são necessários. Também foi demonstrado que mesmo em caso de falhas de processos, o sistema é tolerante e continua a funcionar com alta disponibilidade, sem a necessidade de intervenção manual.

O custo cada vez mais alto de equipamenos de infraestrutura mostra que é essencial utilizar os recursos disponíveis de maneira otimizada e eficiente. Dessa forma, o dimensionamento dinâmico mais próximo da quantidade de processamento necessária para prover serviços, se mostra cada vez mais importante, pois significa economia de custos sem perda da qualidade dos serviços prestados.

Assim, por meio do *middleware* desenvolvido nesta pesquisa foi possível diminuir o número de servidores necessários para tratar as demandas de processamento das filas de mensagens, sem comprometer a qualidade de serviço imposta pelos clientes do sistema distribuído. O tempo de reinicialização do sistema por conta de quedas por falhas e manutenções corretivas e/ou evolutivas para atualizações também foi reduzido. O suporte a tolerância a falhas no *middleware* tornou o sistema resistente a queda ou falha de

processos durante o tratamento de mensagens, possibilitando alcançar alta disponibilidade no *cluster*.

Dessa forma, este trabalho mostrou ser possível alcançar um considerável aumento na qualidade dos serviços prestados por meio da diminuição do tempo de carga inicial do *cluster*, de 16 minutos para 27 segundos, ou seja, com um ganho de 3.457,94% em relação a versão em produção, e, também, uma economia de recursos de infraestrutura de 580%, por meio da diminuição do número de servidores do *cluster*, de 58 para 10, ou seja, 1 servidor para cada base de dados mostrada na Tabela 3.1. Consequentemente, isso reduz os custos com o *hardware* necessário para executar as aplicações distribuídas, e o equipamento redundante pode ser utilizado para outras aplicações, bem como para o uso em *failover* do próprio *cluster*. A redução drástica da diminuição do tempo de carga ocorreu, principalmente, devido à otimização das camadas de persistência, em que foram evitadas diversas cópias de objetos durante as consultas realizadas no banco de dados. Dessa forma, os resultados dessas consultas são carregados mais eficientemente em memória. Já a diminuição dos servidores, ocorreu como uma consequência da diminuição do tempo de inicialização. Entretanto, como ocorreram enfileiramentos devido a esse agrupamento, só foi possível a diminuição efetiva de servidores devido ao emprego de algoritmos, em nível de *middleware*, que exploraram a elasticidade por meio do aumento de *threads* para alcançar o paralelismo de consumo e conter as rajadas de mensagens.

Todavia, no *cluster* em estudo, existe a necessidade do controle de priorização para o aumento de *worker threads*, de forma que os processos relacionados às funcionalidades mais importantes, mostrados na Tabela 3.3, tenha uma alta disponibilidade funcional, para que não ocorra o enfileiramento de mensagens críticas da aplicação distribuída.

Dessa forma, se um processo menos prioritário, ou seja, que não pertença à classe dos processos mais importantes, solicitar o aumento de *worker threads*, deverá existir algum mecanismo de priorização para os processos críticos, e que se encontram em um estado próximo de requisição do aumento de *worker threads*. Assim sendo, como trabalhos futuros, existe a necessidade de estudos sobre quais mecanismos utilizar para a realizar esse controle de priorização dos processos críticos do *cluster*.

Uma outra limitação encontrada na arquitetura atual do *cluster*, está associada ao fato de não ser possível associar um servidor a mais de uma base de dados. Essa característica pode tornar possível uma diminuição ainda maior do número de servidores necessários para executar a aplicação distribuída, sendo mais um item a ser analisado em trabalhos futuros.

Além disso, uma limitação atual do *middleware* desenvolvido é o fato de não ser possível paralelizar o tratamento de mensagens associadas à mesma conta de comunicação, devido a restrição de integridade sequencial. Também, como trabalhos futuros, estão os

estudos de mecanismos e técnicas para paralelizar o tratamento de mensagens associadas à mesma conta, de forma que tenham seu processamento adiantado até um ponto em que não gerem conflitos de integridade sequencial.

Entretanto, o aumento do número de threads tem uma relação direta com o consumo de energia e com a emissão de calor dos processadores. Dessa forma, devido ao grande volume de servidores necessários para executar a aplicação distribuída vista neste trabalho, os custos com o consumo de energia e ar condicionado são muito significantes neste ambiente. Assim, também como trabalhos futuros, estão os estudos dos impactos na economia de energia causados pela diminuição do número de servidores e da consequente adaptação dinâmica de *threads* para a contenção das filas de mensagens.

Referências

[1] Heithem Abbes, Christophe Cerin, Mohamed Jemni, and Walid.Saad. Fault tolerance based on the publish-subscribe paradigm for the bonjourgrid middleware. *11th IEEE/ACM International Conference on Grid Computing*, pages 57–64, 2010. 10, 21

[2] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. pages 204–212, 2012. 11

[3] Tekin Bicer, Wei Jiang, and Gagan Agrawal. Supporting fault tolerance in a data-intensive computing middleware. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010. 10, 21, 42

[4] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999. 1, 5, 6

[5] Tracey Hughes Cameron Hughes. *Parallel and Distributed Programming Using C++*. Editora Addison Wesley, first edition, 2003. 7

[6] Marcela Castro, Dolores Rexachs, and Emilio Luque. Transparent fault tolerance middleware at user level. *2012 International Conference on High Performance Computing and Simulation (HPCS)*, pages 566–572, 2012. 9, 10, 20, 21, 42

[7] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. Editora Pearson Prentice Hall, fourth edition, 2009. 2, 7, 9, 10

[8] Douglas Downing and Jeffrey Clark. *Estatica Aplicada*. Editora Saraiva, third edition, 2011. 34

[9] Wenjing Fang, Beihong Jin, Biao Zhang, Yuwei Yang, and Ziyuan Qin. Design and evaluation of a pub/sub service in the cloud. *International Conference on Cloud and Service Computing*, pages 32–39, 2011. 11, 12, 14, 21, 25

[10] Michael J. Flynn. Some computer organization and their efectivenes. *IEEE Transactions on Computers*, pages 948–960, 1972. 1

[11] John Shapley Gray. *Interprocess Communications in Linux: The Nooks Crannies*. Editora Pearson Prentice Hall, first edition, 2003. 8

[12] Yike Guo, Rui Hanm, Benjamin Satzger, and Hong-Linh Truong. Programming directives for elastic computing. *Internet Computing and IEEE*, pages 72–77, 2012. 14

[13] Chen He, Derek Weitzel, David Swanson, and Ying Lu. Hog: Distributed hadoop mapreduce on the grid. *SC Companion: High Performance Computing and Networking Storage and Analysis*, pages 1276–1283, 2012. 10, 21, 42

[14] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 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, 2013. USENIX. 11

[15] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. Elastic scalable cloud computing using application-level migration. *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 91–98, 2012. 13, 14, 29, 31

[16] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing Principles and Algorithms and Systems*. Editora Cambridge University, first edition, 2008. 1, 7, 8

[17] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 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, 2012. 13, 14, 25

[18] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. *IEEE International Parallel & Distributed Processing Symposium*, pages 1254–1265, 2011. 12, 13, 14, 25

[19] Ricky K.K. Ma, King Tin Lam, Cho-Li Wang, and Chenggang Zhang. A stack-on-demand execution model for elastic computing. *39th International Conference on Parallel Processing*, pages 208–217, 2010. 11, 13, 14, 24

[20] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forums*, pages 1085–1094, 2012. 24

[21] Rolando Martins, Priya Narasimhan, Lu Lopes, and Fernando Silva. Lightweight fault-tolerance for peer-to-peer middleware. *29th IEEE International Symposium on Reliable Distributed Systems*, pages 313–317, 2010. 10, 20, 21

[22] Microsoft. Windows Performance Monitor. `http://technet.microsoft.com/en-us/library/cc749249.aspx`, 2014. Accessado em 13-Julho-2014. 57

[23] Julien Perez, Ccile Germain-Renaud, Balzs Kgl, and Charles Loomis. Responsive elastic computing. *ACM/IEEE Conference on International Conference on Autonomic Computing*, pages 55–64, 2009. 5, 11

[24] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschman. *Pattern-Oriented Software Architeture*. Editora John Wiley & Sons, first edition, 2000. 12, 24, 37, 38

[25] Jonathan M. Smith. A survey of software fault tolerance techniques. pages 165–170, 1986. 9, 20

[26] Akiyoshi Sugiki and Kazuhiko Kato. An extensible cloud platform inspired by operating systems. *Fourth IEEE International Conference on Utility and Cloud Computing*, pages 306–311, 2011. 13, 14

[27] Sumant Tambe, Akshay Dabholkar, and Aniruddha Gokhale. Fault-tolerance for component-based systems - an automated middleware specialization approach. *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 47–54, 2009. 10

[28] Andrew S. TANENBAUM. *Modern Operating Systems*. Editora Pearson Prentice Hall, third edition, 2007. 1

[29] Andrew S. TANENBAUM and Maarten Van Steen. *Distributed systems: principles and paradigms*. Editora Pearson Prentice Hall, second edition, 2008. 5, 6, 7, 8, 17

[30] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Editora Pearson Prentice Hall, fifth edition, 2010. 16

[31] Andrew S. TANENBAUM and Albert S. Woodhull. *Operating Systems Design and Implementation*. Editora Pearson Prentice Hall, third edition, 2007. 13, 26

[32] Nam-Luc Tran, Sabri Skhiri, and Esteban Zimnyi. Eqs: an elastic and scalable message queue for the cloud. *Third IEEE International Conference on Cloud Computing Technology and Science*, pages 391–398, 2011. 12, 13, 14, 25

[33] Jun Wang, Yu Deng Jian-Wen Chen, and Di Zheng. 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, 2009. 9, 10, 20, 21, 42

[34] Anthony Williams. *C++ Concurrency in Action - Pratical Multithreading*. Editora Manning, first edition, 2011. 23

[35] Wenbing Zhao, P. M. Melliar-Smith, and L. E. Moser. Fault tolerance middleware for cloud computing. *3rd International Conference on Cloud Computing*, pages 67–74, 2010. 10, 21

Anexo A

Artigo Científico.

Foi escrito um artigo científico com base na pesquisa realizada neste trabalho, e submetido na *26th International Symposium on Computer Architecture and High Performance Computing* (SBAC - PAD 2014), no dia 26 de junho de 2014. O artigo encontra-se anexado. Segue a confirmação, por *email*, do recebimento da submissão realizada.

```
submission of SBAC - PAD 2014 paper 49

SBAC - PAD 2014 <sbacpad2014@easychair.org>

26 de jun

Dear authors,

We acknowledge the receipt of your
submission to SBAC - PAD 2014.

  Number: 49
  Authors: Eduardo Teixeira and Aletéia Araújo
  Title: Dynamic Analysis of Message Queues Throughput Behavior
  to Increase Consumption Parallelism

The paper was submitted by Eduardo Teixeira <edu.henr@gmail.com>.

You can access the new version of your paper if you log in to
the SBAC - PAD 2014 submission Web page.
```

Dynamic Analysis of Message Queues Throughput Behavior to Increase Consumption Parallelism

Eduardo H. Teixeira
Department of Computer Science - CIC
University of Brasilia (UnB)
Brasília, DF - Brazil
Email: edu.henr@gmail.com

Aletéia P. F. Araújo
Department of Computer Science - CIC
University of Brasilia (UnB)
Brasília, DF - Brazil
Email: aleteia@cic.unb.br

***Abstract*—A major challenge to achieve elasticity in distributed computing is to avoid the phenomenon of Threshold Detection (TD) up and down. This article presents the dynamic analysis of message queue flows, and how to increase the parallelism of consumption based on their throughput behavior. Subsequently, it presents the implementation of the IOD (Increase On Demand) middleware with support for the increase and decrease of worker threads to restrain growth of message queues using the technique of heuristic-based limits for a given time, and grouping message subqueues according to a criterion of classification. Finally, we present the results of running the middleware under conditions of constant burst, up threads, limited by CPU consumption and standard deviation of the input flow, as well as support the elasticity and fault-tolerance.**

Keywords*—Distributed Computing, Parallel Computing, Cluster, Middleware, Elasticity, Message Queues.

I. Introduction

Elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible [1]. This dynamic resource provisioning is defined as elastic computing [2].

Elastic computing has enormous advantages for application providers, including cost savings, preventing over- and under-provisioning of IT (Information Technology) resources by monitoring demand and acquisition of resources strictly required by applications to achieve a high level of quality [3]. In other words, elasticity aims at matching the amount of resources allocated to a service with the amount of resources it actually requires.

The decision problem associated to elastic computing extends the scheduling framework to adjusting the number of computing resources available for maximizing the utilization of the resources [2]. To this end, an architecture middleware is needed – one that is elastic enough to adapt to the requested queue growth, so that messages do not accumulate.

To create efficient elastic environments, existing services must be extended with elastic computing functionality and resource provisioning policies that match resource deployments with demand [4]. Thus, it is possible, for example, to reduce the number of necessary servers to process message queues of a distributed system and, consequently, save computational resources for evolutionary and corrective maintenance. It is also possible, to determine the actual demand of processing required in a cluster, according to the load that can be measured as a function of the input flow into a message queue, and which may be limited by CPU consumption.

Thus, this paper presents the implementation of the IOD middleware to achieve elasticity for distributed system architectures based on message queues, such as high performance and high availability clusters. The IOD middleware dynamically analyzes the throughput behavior of message queues to determine the need to increase or decrease the number of threads according to the average input and output flows. IOD also analyzes the average CPU (Central Process Unit) consumption to determine the scalability limits of the servers to avoid saturation of the cluster.

The rest of this paper is organized as follows: Section II overviews some related works on elasticity. Section III describes the architecture, design and implementation of the IOD middleware. Section IV presents the evaluation and analysis of the IOD middleware with a discussion about the importance of the work done. Section V presents our conclusions and describes the next steps in this work.

II. Related Works

When sequence integrity in processing messages from the queues is not a functional requirement, the approaches cited in [5][6] are considered favorable in the effort to achieve scalability transparently and automatically. Thus, the messages in the queues can be scaled and redistributed according to the required flow without worrying about the order in which they are processed. However, this can mean a disadvantage when there is the need to maintain sequential consistency based on the order of delivery to consumers. In this case, the events raised by consumers of service queues must preserve the time restriction in which they were generated.

Thus, the asymmetries in the distribution of assignments made by the publishers to subscribers [5][7][8] are common in systems with multiple message queues [5][7] and multiple consumers. Identifying the best candidate matches for the consumption of queues is a need to reduce the latency and avoid saturation of overloaded consumers. So, it is possible to determine the best candidates for the consumption of new items identified on the greater average flow [5][7] and lower average CPU consumption [7].

In distributed systems with CPU bound characteristics, it's possible to explore the transparency of elasticity for the application [5][6][9] according to the required load demand measured by the CPU consumption [7][9][10]. Thus, if the number of messages remains close to the detection limits of elasticity, problems of rapid onset and termination of worker threads may occur. To maintain the flow at acceptable levels [5][7][9], using the approach with the heuristic-based limits technique for a given time, proves favorable when applied to calculating the number of worker threads that are started and terminated [9]. These heurisitics can be used to solve the problem of the TD of elasticity up or down. Thus, maintaining the threads created for more time to minimize the reset cost, decreasing the latency of the consumption of messages.

When occur a falling or locking of some process the operationality of a cluster can be easily recovered and held by another group of replicated services that access the same data [13]. However, architectures with fast failure recovery mechanisms [11][12][13][14] with elasticity support after restart, present an alternative approach without any additional communication cost and memory required for use with the replication service group accessing the same data [13].

III. *IOD Middleware*

In IOD middleware, several classes were developed in C++, which compiled a static library that can be linked to the executable that will with stand the elasticity characteristics. The following subsections explain each of the points required for the implementation of support for elasticity in the IOD middleware.

A. Data Structure Architecture

To be able to achieve consumption parallelism, is necessary, beforehand, to separate the messages from a queue in distinct groups, according to a criteria of classification. Thus, each queue will give rise to several subqueues each indexed by a key that identifies a group. This should be done to increase the number of threads that will be responsible for processing the messages, distributing the groups among the various threads initiated, thereby, increasing the consumption parallelism with increasing flow rate.

Besides the separation into groups and subqueues, some architectural characteristics are still needed in the data structure of message handling, to ensure the orderly, fair and balanced treatment of messages. First, incoming messages must first be addressed beforehand, with the First-come First-served algorithm (FCFS) [17]. It is also necessary to order the groups that have the highest number of messages , then the distribution of subqueues among the consumer threads using round robin algorithm, so that every suqueue is handled in a round, if they have messages . Furthermore, consumption in the same thread should be done using fair queueing (FQ) between subqueues processed – fairly, with a message from each group per round. The FQ is used to avoid starvation and heavy flows, ensuring justice in consumption of all subqueues flows separated by group. An algorithm was then implemented to consume messages, in the order of arrival, with fair and balanced treatment, utilizing a data structure with the architectural features below:

- ***First-come First-served***
  Messages are sorted in order of arrival with timestamp in milliseconds. Thus, the oldest messages are handled first;
- ***Round Robin***
  All subqueues must be treated in a round. If there are no messages in the group round, is processed a message from a group that has not been treated in this round;
- ***Fair Queueing***
  Fair treatment, where each group has only one message handled in rounds, independent of how many messages are queued in the same grouping;
- ***Subqueues***
  Messages separated by grouping criterion in different subqueues;
- ***Worker Threads***
  Creating worker threads to increase throughput and parallel consumption.

B. Analysis of Input and Output Average Flow

For the number of threads that will be needed to contain the growth and zero subqueues, it's necessary to calculate the average flow of inbound and outbound messages, and the average of the relationship between the input and output messages, from the time of queue growth detection until the creation of new threads (Entry Point).

Growth Detection is the queue's trend growth detection. It occurs when the relationship between the input and output messages becomes larger than one, generating queueing. It is detected by the Formula (1),

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

in which, *Input* is the number of input messages and *Output* is the number of output messages.

Scale Up occurs at the entry point, that is the moment where new threads are created to contain a burst of messages that generate queuing, and that could not be handled in a timely manner before the end of the maximum treatment time of messages. The entry point can be found when the number of messages in the queue is greater than the value given by the Formula (2),

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

in which, the *avgTOUT* variable is the average output flow, *schTIME* is the maximum time for treatment of messages in the queue, *hNOW* is the current time, *hSTART* is the start time since the value given by Formula (1) becomes true.

The Scale Down occurs in the exit point, which is the moment when threads are removed. It is arrived at by Formula (3),

$$qSIZE < \left(\frac{avgTIN}{2}\right) \quad (3)$$

in which, *qSIZE* is the current size of the queue and *avgTIN* is the input average flow since the entry point.

In addition to the threads containing gusts input (ContentionThreads), another set of threads (ZeroThreads) is required to consume the messages that have accumulated in the queue from the time of onset of the queue until the entry point, as shown in Figure 1 and Table II. Thus, it's possible to scale up again when the queue size reaches twice the size than the measurement of the latter Scale Up.

1) Scale Up*:* In Table I, *Time* corresponds to time in seconds of each measure; *avgTIN* is the input average flow; *avgTOUT* is the output average flow; *avgIO* is the the average ratio between the input and output messages; *QueueSize* is the number of messages in queue; *Entry Point* to value of Formula (2).

The value of *avgIO* is used to find the number of contention burst threads, when *QueueSize* is greater than *Entry Point*. The *avgTOUT* divided by the number of worker threads results in throughput by thread and determines the number of worker threads necessary to reset the messages queued.

Table I: Before Scale Up.

Time	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*	*Entry Point*
0	18.00	4.00	4.50	17	320
1	40.50	9.50	4.35	63	750
2	50.25	12.25	4.17	110	955
3	55.63	13.63	4.12	156	1049
4	58.31	14.31	4.09	202	1087
5	60.66	14.66	4.15	250	1098
6	61.83	14.83	4.17	298	1097
7	62.41	14.91	4.19	346	1088
8	62.71	14.96	4.19	394	1076
9	62.85	14.98	4.20	442	1063
10	63.43	14.99	4.23	490	1049
11	63.21	14.99	4.22	538	1034
12	63.11	15.00	4.21	586	1019
13	63.05	15.00	4.20	634	1004
14	62.03	14.50	4.28	679	956
15	62.51	14.75	4.24	727	957
16	62.76	14.87	4.22	775	951
17	62.88	14.94	4.21	823	940
18	62.94	14.97	4.21	871	927
19	**62.94**	**14.97**	**4.20252**	**919**	**912**

As can be seen in Table II, to avoid the problem of the detection threshold, is always set up one more thread than the number of ContentionThreads plus ZeroThreads, totaling 6 threads. The moment of Scale Up is presented in Figure 2.

Table II: Scale Up.

Start Time	*Reaction Time*	*ZeroThreads*	*ContentionThreads*	*Total Threads*
17:08:19	17:08:38	1.00542	4.20252	6

2) Scale Down*:* In the Table III, *Time* corresponds to time in seconds of each measure; *Input* is the number of input messages; *Output* is the number of output messages; *avgTIN* is the input average flow; *avgTOUT* is the output average flow; *avgIO* is the the average ratio between the input and output messages; *QueueSize* is the number of messages in queue.

To perform Scale Down it is necessary that the value given by the Formula (3) is true. This mechanism aims to avoid the problem of the detection threshold of elasticity down, which would cause the queues to have a tendency for new growth. The moment of Scale Down can be seen in the Exit Point in Figure 1.

According to Table III after the Scale Up, the average flow output is higher than the average input flow, causing the queue to be subdued, and it decreases to a value close to zero. After 49 seconds, is detected that the number of messages in the queue (26) is less than the value of *avgTIN* (63.00) divided by two (31.50), as in Formula (3). Thus, according to Table IV, is decided to remove 1 thread (Scale Down) based on the average flow output by thread, as we can see in Figure 2.

Table III: Before Scale Down.

Time	*Input*	*Output*	*avgTIN*	*avgTOUT*	*avgIO*	*QueueSize*
20	63	91	63.00	91.00	0.69	889
21	63	95	63.00	93.00	0.68	857
22	63	95	63.00	94.00	0.67	825
23	63	95	63.00	94.50	0.67	793
24	63	95	63.00	94.75	0.66	761
25	63	95	63.00	94.88	0.66	729
26	63	95	63.00	94.94	0.66	697
27	63	95	63.00	94.97	0.66	665
28	63	95	63.00	94.98	0.66	633
29	63	95	63.00	94.99	0.66	601
30	64	95	63.50	95.00	0.67	569
31	63	95	63.25	95.00	0.67	537
32	63	95	63.13	95.00	0.66	505
33	63	95	63.06	95.00	0.66	473
34	63	95	63.03	95.00	0.66	441
35	63	95	63.02	95.00	0.66	409
36	63	95	63.01	95.00	0.66	377
37	63	95	63.00	95.00	0.66	345
38	63	95	63.00	95.00	0.66	313
39	61	89	62.00	92.00	0.67	283
40	64	101	63.00	96.50	0.65	249
41	63	95	63.00	95.75	0.66	217
42	63	95	63.00	95.37	0.66	185
43	63	90	63.00	92.69	0.68	162
44	63	90	63.00	91.34	0.69	132
45	63	87	63.00	89.17	0.71	108
46	63	92	63.00	90.59	0.70	79
47	63	81	63.00	85.79	0.74	63
48	63	88	63.00	86.90	0.73	39
49	**63**	**74**	**63.00**	**80.45**	**0.79**	**26**

Table IV: Scale Down.

StartTime	*ReactionTime*	*avgTIN*	*avgTOUT*	*Throughput*	*RemoveThreads*
17:08:19	17:09:08	63.50	96	16.08	1

C. Analysis of CPU Consumption

Knowing the average CPU consumption by worker thread may be useful to determine limits for the Scale Up. Thus, to detect bursts of incoming messages can be initiated the process for measuring average CPU consumption, by worker thread, to determine the flow rate required to contain the growth and

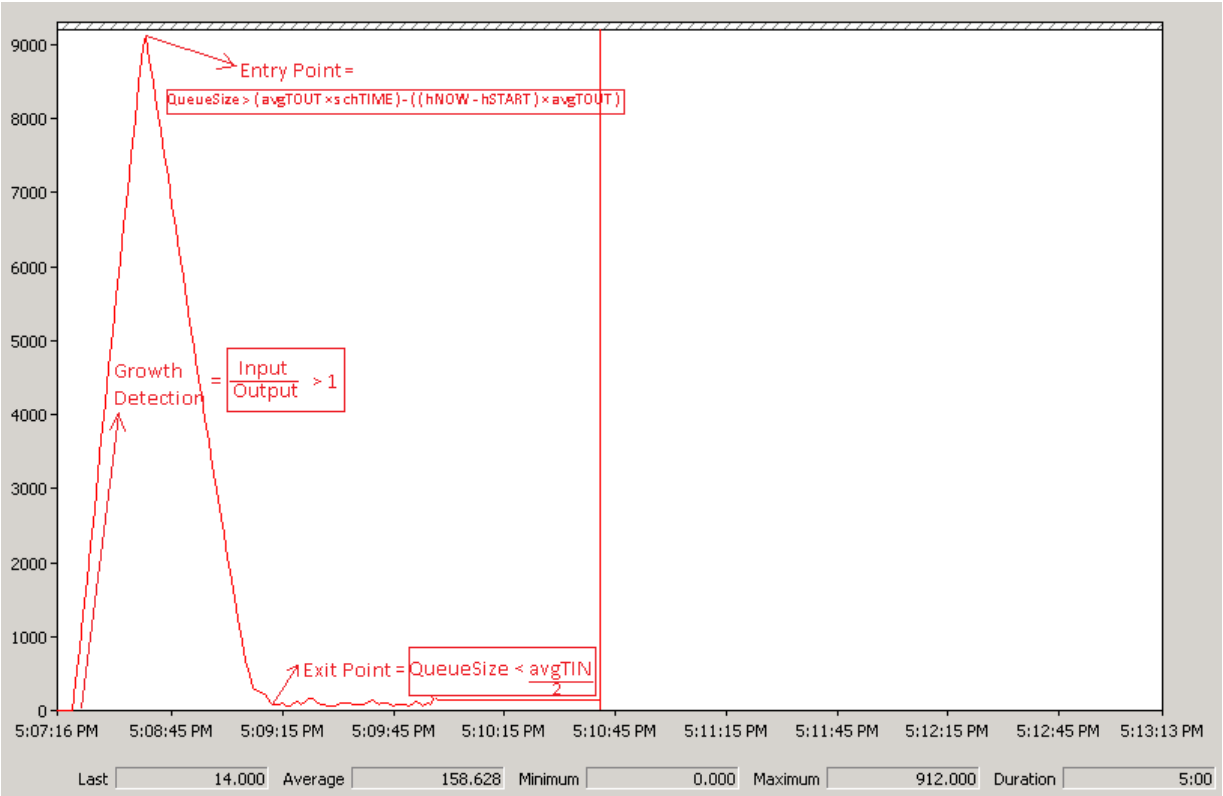


Figure 1: Analysis of Message Queue Average Flow Input and Output.

zero queues can be serviced by the server, without saturation of CPU usage.

Thus, through the average CPU consumption, it's possible to determine whether the number of worker threads, that the elasticity algorithm decided to create to meet the required flow, are enough. This aims to avoid a consumption that exceeds 100% of CPU usage, which will provoke the saturation of the server.

D. Analysis of Standard Deviation of Average Flow Input

The bursts of incoming messages can come from several groups of messages, as well as a few. To meet gusts of *n* groups, the algorithm decides to create several worker threads according to the average output flow, measured during the burst.

However, when the bursts come from isolated groups, creating multiple threads can mean a waste of resources and, depending on the consumption flow, may not be enough to contain the growth of queues and still generate recursive Scale Ups without actually solving the problem.

To solve this problem, in Scale Up performed after measurement of average flow of input and output during bursts process, is calculated the standard deviation of the mean flow entry per group is calculated. Thus, it is possible to determine how many groups are actively with values well above (3 times) the standard deviation. This aims to discover whether the number of creation threads based on throughput is higher than that determined by calculating the standard deviation. If so, only the number of worker threads are created with the number of groups whose average input flow is 3 times larger than the standard deviation of the mean flow entry per group.

Furthermore, groups that generate increased threads for this calculation are stored, so that if a new burst occurs for the same group, it will not generate recursive increase threads. If the queue continues to grow after Scale Up, it is an optimization problem of the algorithm treatment for each message.

E. Elasticity with Fault Tolerance

To support mechanisms for fast fault recovery [11][12][13][14], if a process falls and the number of messages in its queue is over than the entry point limit, elasticity mechanisms are necessary to provide Scale Up. Thus, when the processes are restarted, if the input bursts continue, the Scale Up occurs rapidly by detecting the entry point above the limit. However, another mechanism is required for providing elasticity in case the bursts are terminated.

In this situation, if the number of messages is above average output flow times, the maximum time to clear the queues, flow measurements are performed for 10% of this time. This is necessary to obtain measurements that will be used to calculate the number of threads to reset the queue at maximum time set, even without gusts of entry, such as the quick restart.

Figure 2: Analysis of Scale Up and Down Moments.

F. *Elasticity Algorithm*

Initially, just one thread is running for handling messages and is only allowed to Scale Up, never Scale Down. To avoid the problem of the detection threshold of elasticity upward or downward cited in [9], when the process scale the number of threads up, it goes into a state that does not allow increase or decrease of the threads until three measurements occur with the output flow greater than the input. After this period, the algorithm enters a state where it is allowed to scale up or down again.

When Scale Up is done, the queue size is stored. A new Scale Up is allowed only when the queue reaches a size 2 times higher than the previous measurement. The algorithm decides to Scale Down, in case of bursts, only when the queue size is less than the average input flow divided by two, as in Formula (3). This aims to avoid the problem of the elasticity detection threshold cited in [9].

To obtain the number of consumer threads, which will initiate the assignment of message queues, separated by groups the round robin algorithm shall be used, ordering the fuller queues in descending order and assigning them to the new consumer threads. In addition, only one message by each group is processed at a time, using the fair consumption. The use of these techniques is to avoid destabilizing consumers, overloading some threads with heavier queues, and also prevent startvation in the consumption of the heaviest queues at the expense of queues with less messages, in other words, a consumption fair of message queues.

For each message that arrives on the main queue of a consumer process, one subqueue that contains the packages associated with each group is created. Each consumer process has many threads: one reads from the message queue and writes in group subqueue; many worker threads that read from group subqueues, which then deserializes the message and performs packet processing. Thus, it was built a singleton[1] protected by a mutex, which contains a map group for the buffer package. Furthermore, all packages need to be changed so that the first four bytes contain the group number to which the message belongs. Thus, the thread responsible for reading from the queue quickly discovers which group belongs to the package and performs the movement of the current message to the subqueue group.

To insert the package, a semaphore associated with the worker thread, that handles the packets, is incremented to signal the need for treatment of another message. The worker threads should be waiting for both the semaphore signals and a mutex of group control structure. This is necessary because another thread responsible for the calculation of flows in and out can get the mutex to migrate the treatment groups between threads less overloaded, and increase or decrease the number of threads based on a threshold, which can be average CPU consumption by worker thread or the standard deviation of the mean flow entry per group.

When the worker thread is signaled by the semaphore

[1]Insure a class only has one instance, and provide a global point of access to it [15].

and gets the critical section mutex atomically, it performs the processing of messages by the round robin algorithm for groups to have a balanced and fair treatment. Then it performs the removal of the message subqueue, if the message has been handled without exceptions. It aims to perform a persistent treatment without missing the message. To move groups between threads, the thread management gets the mutex, performs down in the semaphore with the number of packages in the group to be migrated, assigns the group to the structure that associates groups to worker threads and, then, performs up with the number of packets in the semaphore of the target worker thread. Each *WorkerThread* can have several associated groups and performs the message consumption fair of each group from its list of treatment per round (FQ with round robin). So, the elasticity algorithm is responsible for the creation and destruction of new worker threads, as more or less flow is needed for the consumption of the messages' subqueues.

IV. EVALUATION

For simulations of the elasticity algorithms two threads were used: a producer, which writes packets in the message queue; and a consumer, which reads the message queue and dispatches the message to a singleton that contains the implementations of the algorithms described in the previous section.

The following subsections describe the tests that aim to verify: the overhead of the data structure responsible for the separation of queues and consumption of messages' subqueues; the dynamic increasing and decreasing of the worker threads according to the mean flow of input and output; the threads' increase with thresholds depending on CPU consumption and standard deviation of the mean flow entry per group; activation of algorithms elasticity after the queue growth beyond the limit of the entry point.

A. Overload Data Structure

To assess the overhead of the data structure and the algorithm of separation and message subqueue consumption, first, a waiting time of 50 milliseconds was placed between messages in both generating and consumer threads.

The tests showed that, in five minutes, queuing of 92 messages occurred, as shown in Figure 3a. Thus, it is apparent that there is an overhead of 30% (0.3 messages/sec delay) of the data structure responsible for the separation and use of message queues, since there is no difference in the time intervals in generation and consumption threads.

Then, sleeping time of the production thread was decreased to 10 milliseconds. The accumulation of messages was higher than in the previous case, reaching a peak of 23.651 messages in five minutes, as shown in Figure 3b. Despite the five-times growth in the generation of the frequency of messages, the increased accumulation was exponential (257 times). This was due to performance problems in the signaling message data structure.

Some modifications were performed, in an attempt to improve the performance of message consumption. Among them are the previously startup list buffers to avoid the dynamic creation locks of structures responsible for the treatment and message consumption of each group. Thus, when a new message arrives, the only serializations occur when there is an exchange of reading and writing lists. However, a worsening of 35.86% occurred. As shown in Figure 3c, the graph of accumulation changed from linear to logarithmic and, as the delay time increased to 0.41 messages per second, the accumulation of messages was higher (125). This was due to performance problems in the structure of the consumption of worker threads. Thus, various optimizations have been made on these points. The consumption of messages now count with a list signaling group for the worker thread, to be activated directly, to consume the oldest package, without needing to look for a group that has not been addressed in the current round. As shown in Figure 3d, after these changes the message queue peaked with three messages and an average of 1,244 messages, demonstrating that the overhead of the structure, after the optimizations, is negligible.

For the evaluation in the next subsections, it was used a maximum treatment time of 80 seconds was used, with the producer thread at 10 miliseconds of sleep between messages and the consumer thread at 50 miliseconds sleep. This aims to generate queueing and demonstrates the threads' creation, and removal behavior of the elasticity algorithm.

B. Creation and Removal of Threads Based on Input and Output Messages Average Flow

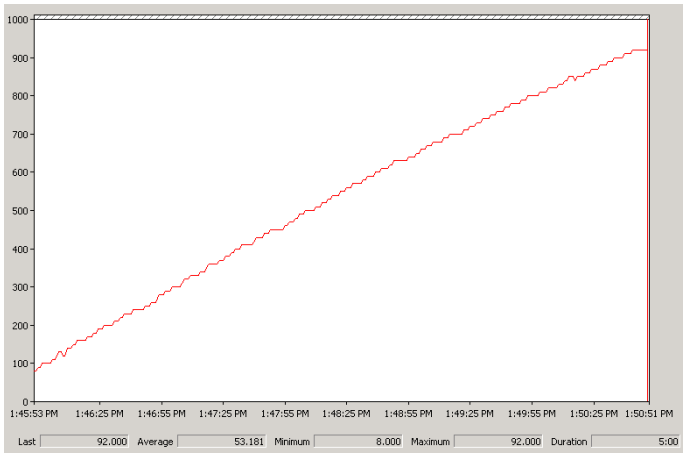
Tests were made with bursts of generation packets at every 10 milliseconds and, 50 milliseconds for each consumer, in order to test the dynamic increase and decrease of threads. For that, upon detecting an imbalanced relationship between incoming and outgoing packets, the detection algorithms for the increase in the number of threads are activated and, if this ratio remains greater than one until the entry point, the number of threads necessary to contain the growth and reset the message queue is calculated.

Thus, the experiments showed that increasing the number of threads to increase the flow of consumption, limit the growth of queues and reset the accumulated messages were satisfactory, as shown in Figures 4a and 4b. To find the optimal point where the threads should be increased, it was used the calculation where the number of messages in the queue can be reset was used, on a schedule that starts where the relation between the input and output messages becomes larger than one.

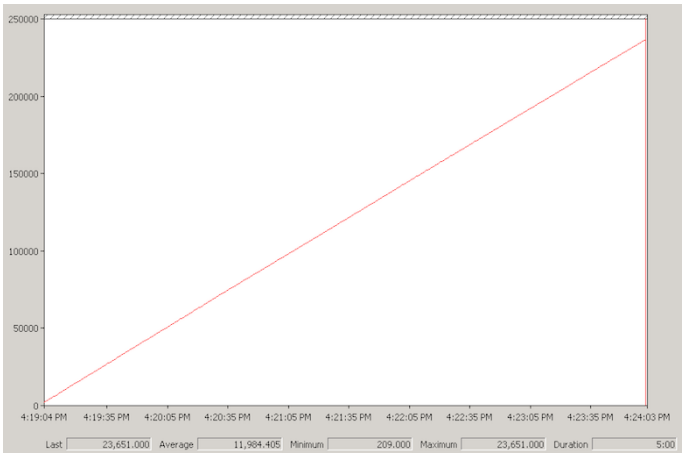
The algorithm uses the signalization with timestamp associated with each group for each of the messages, in the order they arrive. Thus, the worker thread to be signaled, get the package directly without the need to search for groups that were not treated in a fair round. The graphics with the behavior of message queue, CPU and threads are shown in Figures 4a and 4b.

According to Table V, the output flow, after the Scale Down, is 99 messages/second, and, therefore, remains close to the input flow. In addition, the queue maintains an average of 33 messages, in other words, closer to zero after stabilization of the algorithm.

Moreover, it is noticeable in Figure 4b, that during times of Scale Up and Scale Down, occurs spikes of CPU usage. This

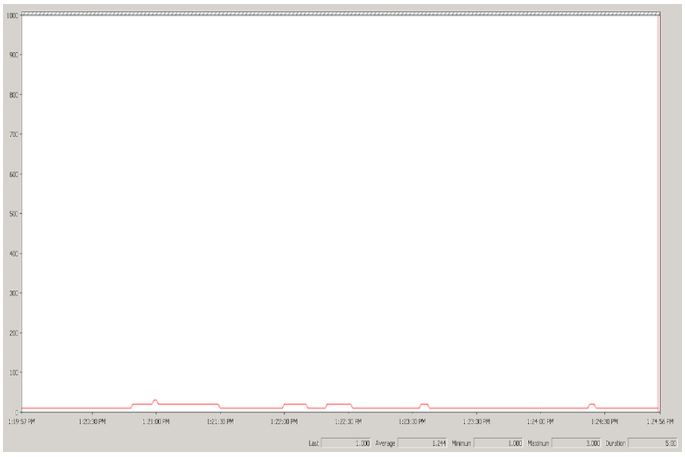


(a) Producer and Consumer at 50 ms Sleep.



(b) Producer with 10 ms e Consumer at 50 ms Sleep.

(c) Producer and Consumer at 50 ms Sleep and Attempting to Improve Performance.



(d) Producer and Consumer at 50 ms Sleep and Signaling Consuming Optimizations.

Figure 3: Overload of Data Structure in Relation to the Accumulation of Messages.

Table V: Elasticity Algorithm with Throughput Analysis.

Moment	Start	Duration	Input	Output	Threads	Queue
Start	13:20:57	14	99	19	1	0
Scale Up	13:21:11	9	99	197	12	1130
Scale Down	13:21:20	57	99	99	7	33

is due to the need to seek and move data structures containing the messages associated with timestamps between threads.

C. Creation of Threads Limited by Average CPU Consumption

To avoid saturating the CPU consumption of the servers, the average consumption of the CPU usage is computed during the burst of the messages in the queues, until the time of detection of entry point to contain the growth. Using the technique of busywait [2] and performing the same tests with the producer at 10 ms and consumer at 50 ms, was obtained the results of CPU usage and size of message queues were obtained as shown in Figures 4c and 4d.

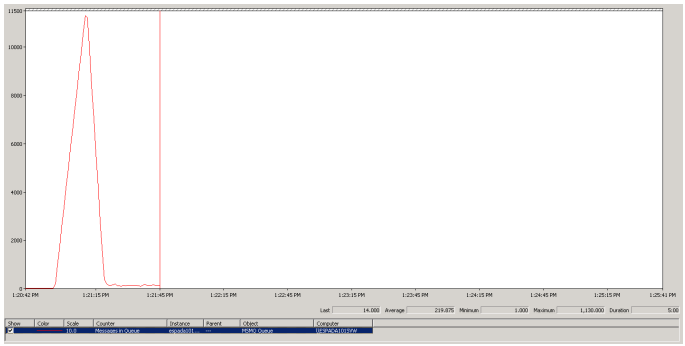
As can be seen in Figure 4c, reducing the number of messages occurs more slowly, because the number of threads is limited based on the average CPU usage of worker threads, using no more than 100% of the server. Furthermore, the decrease is non-linear as in the case where all threads needed for the containment of growth was created.

Table VI: Threads Behavior in Elasticity Algorithm with CPU Thresholds.

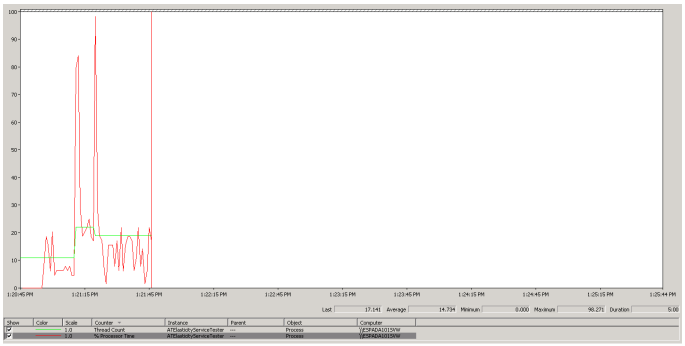
Moment	Needed Threads	Created Threads	Removed Threads
Scale Up	8	6	N/A
Scale Down	N/A	N/A	0

As is seen in Table VI and VII, eight threads should have been created, however, as the average CPU consumption by worker thread was 11.30%, the residual CPU (73.89%) limit was exceeded. Thus, the elasticity algorithm created six threads

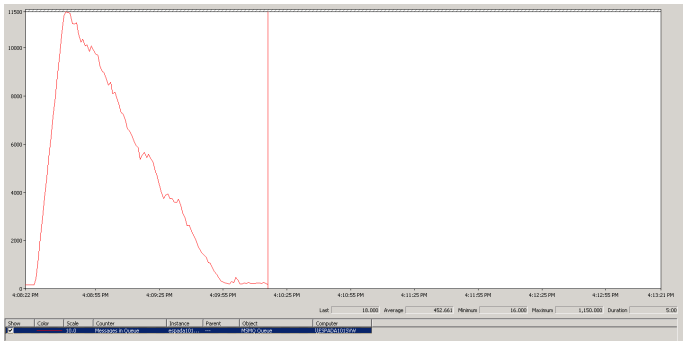
[2]CPU consumption occurs until a resource is released [17].



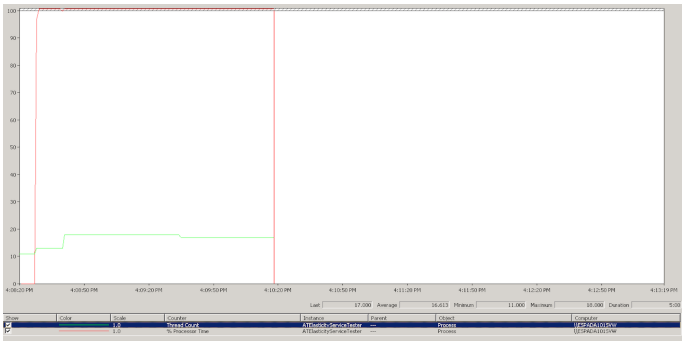
(a) Message Queue with Throughput Analysis.



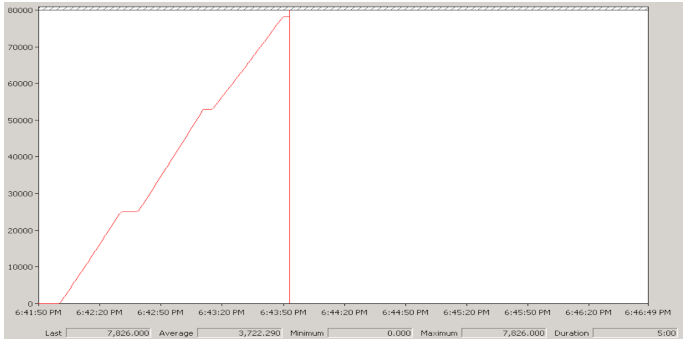
(b) CPU and Threads with Throughput Analysis.



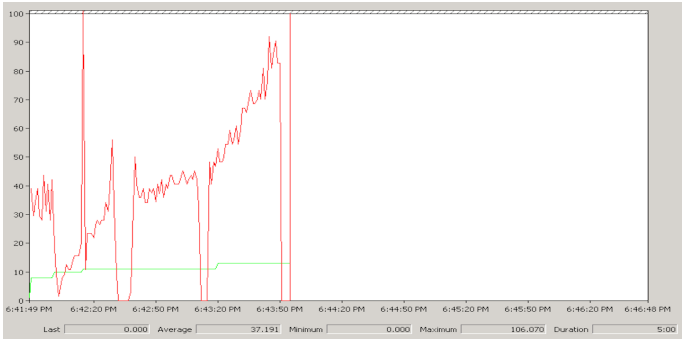
(c) Message Queue with CPU Threshold.



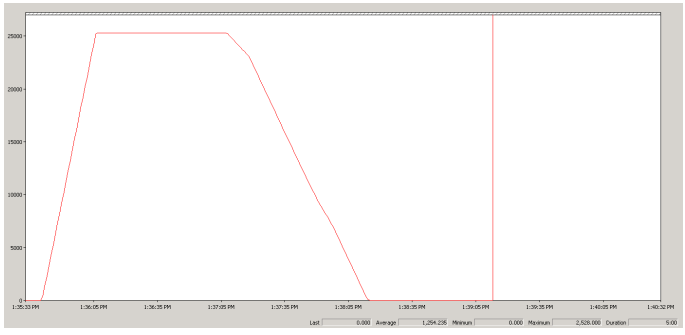
(d) CPU and Threads with CPU Threshold.



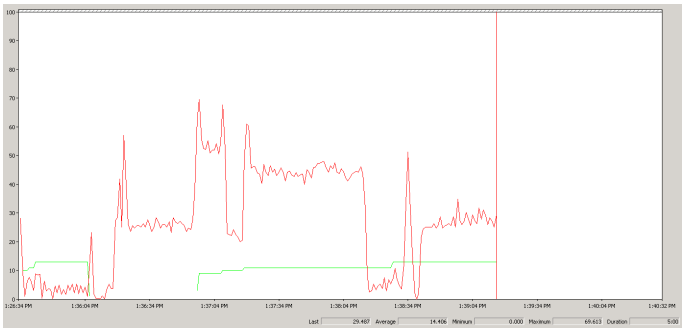
(e) Message Queue with Throughput Standard Deviation Threshold.



(f) CPU and Threads with Throughput Standard Deviation Threshold.



(g) Message Queue with Fast Fault Recovery.



(h) CPU and Threads with Fast Fault Recovery.

Figure 4: Behavior of Message Queue (left), CPU (right and vertical lines) and Threads (right and horizontal lines) with Elasticity Algorithms.

Table VII: CPU Behavior in Elasticity Algorithm with CPU Thresholds.

Moment	Mean by Thread	Total CPU Needed	Total CPU Residual
Scale Up	11.3088%	90.47%	73.89%
ScaleDown	N/A	N/A	N/A

(two less) to avoid saturation CPU to an estimated consumption over 100%. When scale down, as the average output flow (68 messages/second) was below average input flow (85.33 messages/second), the algorithm has not found the need to remove threads.

D. Creating Threads Limited by Standard Deviation of Average Flow Entry by Group

During a burst of incoming messages the average flow per input group is detected. Thus, a limit is imposed on the number of threads to be created, with the number of groups that are distant to a value three times the standard deviation of the mean flow input groups.

Thus, if the detection of the number of threads to be created exceeds the number of groups that have average throughput higher than the standard deviation, only the number of threads equal to the number of groups that distance themselves from more than 3 times of the standard deviation are created. This procedure aims to detect groups with isolated bursts of messages and, also reduce the use of resources because, even though most threads are created, this does not solve the problem of growing queues, since it is not possible to distribute the same data from a group to several threads, to avoid loss of sequential integrity. Performing consecutive bursts to test three groups, were obtained the graphics of message queue, CPU and threads were obtained as shown in Figures 4e and 4f.

As can be seen, the message queues can not be contained, because this becomes a problem of optimizing the processing of each message algorithm, removing elasticity. Despite increasing the number of threads, the number of packets in the queue does not decrease because there is no possibility of separating the data from the same group to parallelize processing, due to the processes that restricts sequential integrity in the logic of business rules. In the tests performed, the processing time of a message is 50 milliseconds and the inlet flow is five times higher (10 ms), causing queuing.

E. Enabling Elasticity After Queue Growth

Due to the problem of falling processes during the handling of messages, it may be necessary that the elasticity is started without the presence of bursts of packets and after quick restart of the process. Thus, the falling simulation process was carried out, as well as the restart with the queue containing messages with a number above the entry point.

As can be seen in Figure 4g, a few seconds after the restart of the process, the tilt of the consumption of messages increases, indicating the increase in the number of worker threads to consume the messages before the of end of the maximum set time. Likewise, it is also possible to observe in Figure 4h, the increase of the number of threads after the quick restart of the process.

F. Discussion

The use of message queues per process, presents a fault-tolerant method, so in case of a process fall, the impacts do not occur in other features [11][12]. Also, write, in different queues dealing with distinct features, parallels the treatment, the system scale and makes the processing faster.

Taking advantage of parallelism provides enormous benefits such as decreased cost of the infrastructure needed to handle message queues. Thus, it creates mechanisms that provide parallel consumption increase in message queues, which significantly improves the quality of services, since the same mass of data can be processed in less time and for fewer servers.

A scientific contribution achieved in this article was the creation of Formulas (2)(3) to determine the optimal point of entry and exit for parallel treatment of messages in queues, dynamically according to the inlet and outlet flows and, the growth detection given by Formula (1). These Formulas are based on the dynamic behavior of each server and the time limits imposed in the message processing and which is directly related to the QoS parameters defined with each client.

A current limitation of the IOD middleware is the fact that it can't deal with parallel treatment of associated messages to the same group, because of the restriction of sequential integrity.

V. CONCLUSIONS AND FUTURE WORKS

The increasingly high cost of infrastructure equipment shows that using and enjoying the available resources optimally is essential, and the automatic dynamic scaling of the amount of processing needed to provide services proves to be increasingly necessary. Thus, the proposed IOD middleware was shown to achieve the requirements of elasticity, to be adaptable to the conditions of the processing load by increasing or decreasing worker threads to meet the required demand. This article has shown how to optimize the use of IT resources through task parallelism by using multiple worker threads to process message queues. Future works include studies of pipelining for parallel execution of instructions that can be exploited, so that the messages associated with the same group has its processing advanced to the point that it does not conflict with sequential integrity.

REFERENCES

[1] HERBST, N. R.; KOUNEV, S.; REUSSNER, R. Elasticity in cloud computing: What it is, and what it is not. 10th International Conference on Autonomic Computing, p. 1085–1094, 2013.

[2] PEREZ, J. et al. Responsive elastic computing. ACM/IEEE Conference on International Conference on Autonomic Computing, p. 55–64, 2009.

[3] LEITNER, P. et al. Application-level performance monitoring of cloud services based on the complex event processing paradigm. 5th IEEE International Conference on Service-Oriented Computing and Applications, p. 1–8, 2012.

[4] MARSHALL, P.; TUFO, H.; KEAHEY, K. Provisioning policies for elastic computing environments. 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forums, p. 1085–1094, 2012.

[5] TRAN, N.-L.; SKHIRI, S.; ZIMáNYI, E. Eqs: an elastic and scalable message queue for the cloud. Third IEEE International Conference on Cloud Computing Technology and Science, p. 391–398, 2011.

[6] MA, R. K. et al. A stack-on-demand execution model for elastic computing. 39th International Conference on Parallel Processing, p. 208–217, 2010.

[7] LI, M. et al. A scalable and elastic publish/subscribe service. IEEE International Parallel & Distributed Processing Symposium, p. 1254–1265, 2011.

[8] FANG, W. et al. Design and evaluation of a pub/sub service in the cloud. International Conference on Cloud and Service Computing, p. 32–39, 2011.

[9] IMAI, S.; CHESTNA, T.; VARELA, C. A. Elastic scalable cloud computing using application-level migration. IEEE/ACM Fifth International Conference on Utility and Cloud Computing, p. 91–98, 2012.

[10] SUGIKI, A.; KATO, K. An extensible cloud platform inspired by operating systems. Fourth IEEE International Conference on Utility and Cloud Computing, p. 306–311, 2011.

[11] WANG, J.; CHEN, Y. D. J.-W.; ZHENG, D. Research of the middleware based fault tolerance for the complex distributed simulation applications. International Conference on Computational Intelligence and Software Engineering (CiSE), p. 1–4, 2009.

[12] CASTRO, M.; REXACHS, D.; LUQUE, E. Transparent fault tolerance middleware at user level. 2012 International Conference on High Performance Computing and Simulation (HPCS), p. 566–572, 2012.

[13] HE, C. et al. Hog: Distributed hadoop mapreduce on the grid. SC Companion: High Performance Computing and Networking Storage and Analysis, p. 1276–1283, 2012.

[14] BICER, T.; JIANG, W.; AGRAWAL, G. Supporting fault tolerance in a data- intensive computing middleware. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), p. 1–12, 2010.

[15] SCHMIDT, D. et al. Pattern-Oriented Software Architeture. First edition, John Wiley & Sons publisher, 2000.

[16] Subqueues. Available on http://msdn.microsoft.com/en-us/library/windows/desktop/ms711414(v=vs.85).aspx. Accessed in 13 Fev 2014.

[17] TANENBAUM, A. S.; WOODHULL, A. S. Operating Systems Design and Implementation. Third edition, Pearson Prentice Hall publisher, 2007.