Universidade Federal de Pernambuco
Centro de Informática
Graduação em Sistemas de Informação

Análise Comparativa de Controladores *Fuzzy* para Adaptação do RabbitMQ.

Trabalho de Graduação

Aluno(a): Valter José da Silva Junior (vjsj@cin.ufpe.br)

Orientador(a): Nelson Souto Rosa (nsr@cin.ufpe.br)

Área: Sistemas Distribuídos

Valter José da Silva Junior

Análise Comparativa de Controladores *Fuzzy* para Adaptação do RabbitMQ.

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação, da Universidade Federal de Pernambuco, como requisito parcial para a Obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Nelson Souto Rosa

Recife, 27 de Junho de 2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva Junior, Valter José da.
Análise comparativa de controladores Fuzzy para adaptação do RabbitMQ / Valter José da Silva Junior. - Recife, 2024.
38p, tab.

Orientador(a): Nelson Souto Rosa
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Sistemas de Informação - Bacharelado, 2024.
Inclui referências.

1. Sistemas Distribuídos. I. Rosa, Nelson Souto. (Orientação). II. Título.

000 CDD (22.ed.)

Trabalho de conclusão de curso submetido por **Valter José da Silva Junior** ao programa de graduação em **Sistemas de Informação** do **Centro de Informática** da **Universidade Federal de Pernambuco**, sob o título **Análise Comparativa de Controladores *Fuzzy* para Adaptação do RabbitMQ**, orientado pelo Prof. **Nelson Souto Rosa**, defendido e aprovado em 7 de Agosto de 2024, com banca examinadora formada pelos seguintes professores:

Orientador(a)
Prof. Nelson Souto Rosa
Universidade Federal de Pernambuco

Examinador(a)
Prof. Vinícius Cardoso Garcia
Universidade Federal de Pernambuco

Agradecimentos

Primeiramente, quero agradecer a Deus, que por meio do Seu filho Jesus Cristo, me concedeu Sua vida, nunca me desamparou, e por Sua ajuda e amor, eu pude chegar até aqui. Agradeço aos meus pais, Valter e Rozeane, que não mediram esforços dentro de suas possibilidades para que eu alcançasse meus objetivos. Ao meu irmão, Victor, que sempre acreditou que eu poderia conseguir. Ao meu irmão que recebi por adoção, João, que tanto me ajudou durante a graduação e me suportou em momentos de grande ansiedade, e à sua esposa, Natália, por apoiá-lo nessa jornada. À minha prima, Carolina, que sempre foi um suporte para mim quando precisei. Aos meus amigos e irmãos na fé que oraram por mim e sempre acreditaram nos planos de Deus para minha vida. Aos amigos que fiz na faculdade, que me auxiliaram e foram cooperadores nessa formação. Ao professor Nelson, pelo seu tempo e por toda a ajuda no processo de escrita deste trabalho. Obrigado.

Resumo

Os *brokers* têm um papel fundamental na comunicação de sistemas distribuídos, pois controlam o fluxo de mensagens entre produtores e consumidores, evitando a sobrecarga destes últimos. Este trabalho tem como objetivo realizar uma avaliação comparativa entre controladores *fuzzy* para a adaptação do *broker* RabbitMQ. A avaliação busca definir métricas específicas do *broker* e, com base nessas métricas, realizar uma análise comparativa para determinar qual controlador *fuzzy* realiza o melhor controle de fluxo. Para comparar os controladores, foi utilizado o NRMSE (*Normalized Root Mean-Square Error*) e implementada uma aplicação *publisher-subscriber*.

Abstract

Brokers play a fundamental role in the communication of distributed systems as they control the flow of messages between producers and consumers, preventing the latter from becoming overloaded. This work aims to conduct a comparative evaluation of fuzzy controllers for adapting the RabbitMQ broker. The evaluation seeks to define specific broker metrics and, based on these metrics, perform a comparative analysis to determine which fuzzy controller achieves the best flow control. To compare the controllers, the NRMSE (Normalized Root Mean-Square Error) was used, and a publisher-subscriber application was implemented.

Sumário

1 Introdução **8**
1.1 Motivação 8
1.2 Problema 8
1.3 Soluções Atuais 9
1.4 Objetivos 9
1.5 Estrutura do documento 10

2 Conceitos Básicos **11**
2.1 Sistemas Distribuídos 11
2.2 RabbitMQ 12
2.3 Controladores Fuzzy 14

3 Implementação **17**
3.1 Publish - Subscribe 17
3.2 Broker 20
3.3 Membership Functions 21
3.3.1 Triangular 21
3.3.2 Gaussian 22
3.3.3 Bell 24

4 Avaliação dos Experimentos **26**
4.1 Objetivo 26
4.2 Experimentos e Ferramentas 26
4.3 Resultados e Discussão 27

5 Conclusão e Trabalhos Futuros **36**
5.1 Conclusão 36
5.2 Limitações 36
5.3 Trabalhos Futuros 36

Referências **38**

1 Introdução

Este capítulo apresenta a motivação, a definição do problema, os objetivos e a metodologia adotada neste trabalho. O objetivo é apresentar os elementos que fundamentam esta pesquisa, bem como as estratégias metodológicas adotadas para a investigação e análise.

1.1 Motivação

Em um mundo cada vez mais conectado, a necessidade de sistemas robustos e eficientes que garantam a comunicação contínua entre diferentes plataformas e tecnologias é mais crucial do que nunca. A escalabilidade horizontal de sistemas, com alta interoperabilidade, é frequentemente adotada quando é essencial uma comunicação eficaz entre sistemas independentes. Isso garante que o impacto percebido pelo usuário final seja minimizado na eventualidade de um desses sistemas estar indisponível. Essa abordagem é comumente utilizada para resolver questões de escalabilidade em sistemas robustos. Devido à diversidade tecnológica, à presença de sistemas legados e ao alto custo de integração, a comunicação direta entre sistemas pode ser impraticável.

Nesse contexto, os *brokers*, como o RabbitMQ (M. Roy, 2018), emergem como soluções viáveis para gerenciar a comunicação entre sistemas. O RabbitMQ, um *broker* amplamente utilizado, permite a execução de tarefas assíncronas e se destaca pela consistência na entrega de mensagens e alta disponibilidade. Sua arquitetura é composta por produtores, consumidores, filas e trocas de informações, com capacidade de conectar múltiplas filas. Essa estrutura reduz a dependência de um único sistema e aumenta a disponibilidade.

A aplicação de controladores *fuzzy* (Zimmermann, 2001) no ajuste de sistemas de mensagens, como o RabbitMQ, introduz uma abordagem significativamente diferente das técnicas tradicionais, que se baseiam em controle estático ou adaptativo não-*fuzzy*. Historicamente, a gestão do fluxo de mensagens em brokers, como o RabbitMQ, dependia de algoritmos fixos ou adaptativos que não consideravam a incerteza ou a imprecisão, inerentes aos ambientes operacionais dinâmicos. Em contraste, os controladores *fuzzy* oferecem uma modelagem mais flexível e responsiva das variáveis de controle, como o prefetch count, adaptando-se melhor às variações nas taxas de produção e consumo de mensagens.

1.2 Problema

No contexto dos sistemas distribuídos que utilizam RabbitMQ, o controle do fluxo de mensagens enfrenta desafios significativos que podem afetar diretamente o "desempenho" e a estabilidade do sistema.

Um dos principais desafios é o congestionamento de mensagens, caracterizado pelo recebimento de mensagens pelos consumidores em uma quantidade superior à sua capacidade de processamento em tempo hábil. Esse cenário resulta no acúmulo de uma fila de mensagens não processadas, causando atrasos na entrega e, nos casos mais severos, perda de mensagens.

Esse problema é mais evidente em sistemas com alta carga de mensagens ou em que os consumidores possuem capacidades de processamento distintas. A utilização do *prefetch count* (Videla, A., & Williams, J. J. W, 2012) é uma estratégia eficaz para mitigar esse

problema, pois limita o número de mensagens enviadas a cada consumidor até que este esteja pronto para processar novas mensagens, evitando assim o congestionamento.

Outro desafio no controle de fluxo de mensagens é o bloqueio de filas, também conhecido como *starvation*. Em cenários com mensagens de diferentes prioridades, um sistema sem um controle de fluxo eficiente pode processar prioritariamente as mensagens de baixa prioridade, impedindo o acesso às mensagens de alta prioridade, que são essenciais para a operacionalidade do sistema. Este problema pode ser exacerbado em sistemas distribuídos, nos quais os consumidores podem estar geograficamente dispersos ou operar em ambientes heterogêneos. Uma estratégia para mitigar esse problema envolve o uso de algoritmos de controle de fluxo que consideram tanto a carga de trabalho atual quanto as prioridades das mensagens, implementando *priority queues* em conjunto com limites de *prefetch*.

Por fim, a variação de carga representa um desafio significativo no controle de fluxo de mensagens em sistemas distribuídos. Esses sistemas comumente experimentam picos de demanda ou variações na carga de trabalho, e sem um controle de fluxo adequado, podem falhar em responder apropriadamente a essas mudanças. A incapacidade de ajustar dinamicamente o número de mensagens entregues a cada consumidor conforme sua capacidade momentânea de processamento pode levar à sobrecarga, afetando negativamente o "desempenho". Portanto, mecanismos de controle de fluxo, como o ajuste dinâmico do *prefetch count*, são vitais para adaptar-se a mudanças na carga de trabalho, garantindo a eficiência e a confiabilidade do sistema.

1.3 Soluções Atuais

Diversas soluções já são utilizadas para realizar o controle do fluxo de mensagens em sistemas distribuídos que utilizam o RabbitMQ. O *RabbitMQ Management Plugin* [1] é um plugin amplamente utilizado que permite ajustes finos baseados em métricas de "desempenho" em tempo real. Este plugin é crucial para monitorar e gerenciar "nós" e seus *clusters*, oferecendo ferramentas essenciais para o controle eficaz do fluxo de mensagens.

Outra solução é o gerenciamento de cluster com conscientização de recursos, que consideram a utilização dos recursos de cada máquina no *cluster* [2] assegurando que o sistema opere dentro das capacidades de infraestrutura disponíveis e evitando sobrecargas desnecessárias [3]. Além disso, a combinação de *Prometheus* e *Grafana* [4] é frequentemente utilizada para monitoramento avançado, fornecendo um sistema robusto para a visualização de métricas que pode prever e ajustar o comportamento dos consumidores antes que problemas de congestionamento se manifestem.

1.4 Objetivos

O objetivo principal deste trabalho é implementar e avaliar controladores *fuzzy* capazes de regular o fluxo de mensagens entre produtores e consumidores implementados no RabbitMQ.

Para alcançar este objetivo principal, os seguintes objetivos secundários foram definidos: implementar controladores *fuzzy* com diferentes configurações, implementar uma aplicação produtor consumidor no RabbitMQ que possa ser usada na avaliação comparativa entre

[1] https://www.rabbitmq.com/docs/management
[2] https://www.ibm.com/docs/en /was-nd/8.5.5?topic=servers- introduction-clusters
[3] https://kubernetes.io/docs/concepts/cluster-administration/cluster-autoscaling/
[4] https://grafana.com/docs/grafana /latest/getting-started/get-started-grafana-prometheus/

os controladores; e realizar uma avaliação comparativa entre os controladores e vários cenários.

1.5 Estrutura do documento

Os capítulos seguintes estão organizados da seguinte forma: Capítulo 2 introduz os conceitos básicos necessários ao entendimento do trabalho e apresenta os usos dos conceitos relacionados ao tema; Capítulo 3 apresenta os conceitos e detalhes da implementação da solução utilizada para o controle do RabbitMQ; Capítulo 4 apresenta a avaliação de "desempenho" de cada controlador *fuzzy* no ajuste do RabbitMQ; e no Capítulo 5 são apresentadas as conclusões, limitações e sugestões de trabalhos futuros.

2 Conceitos Básicos

Este capítulo apresenta os conceitos fundamentais necessários à compreensão deste trabalho. Serão discutidos os principais temas: sistemas distribuídos, RabbitMQ e controladores *fuzzy*.

2.1 Sistemas Distribuídos

As redes de computadores estão presentes em toda parte, sendo a internet uma das principais redes acessíveis a muitos, facilitando uma comunicação rápida e síncrona entre dispositivos, independentemente de sua localização. Dispositivos de hardware podem se comunicar através de redes diversas, incluindo redes móveis, corporativas, industriais, acadêmicas, residenciais e até veiculares. Essa comunicação interdispositivos estabelece um ambiente conhecido como sistemas distribuídos.

Os sistemas distribuídos consistem em um conjunto de computadores que se comunicam e colaboram para alcançar um objetivo comum, operando como um sistema unificado. Esses sistemas são definidos pela capacidade de seus componentes, localizados em diferentes computadores conectados em rede, de comunicarem-se e coordenarem suas ações por meio de mensagens. Características críticas como concorrência de componentes, ausência de um relógio global e falhas independentes de componentes são fundamentais e devem ser gerenciadas efetivamente (Coulouris; Dollimore; Kindberg; Blair, 2013).

Entre as características de um sistema distribuído, a concorrência é um dos principais fatores de sua existência. Ela se refere à capacidade de múltiplos processos ou agentes executarem simultaneamente em diferentes componentes do sistema, interagindo potencialmente entre si. Os processos que interligam diferentes dispositivos dentro de um sistema operam em paralelo, assegurando a simultaneidade em diversos nós da rede. Esta interação ocorre através da comunicação entre processos que utilizam protocolos específicos de comunicação, como o serviço de mensageria RabbitMQ. Adicionalmente, a escalabilidade é um fator crucial da concorrência, pois visa lidar com um grande volume de operações e usuários simultâneos sem degradar significativamente o "desempenho". Sistemas concorrentes são projetados para serem tolerantes a falhas, garantindo que a independência dos dispositivos ou processos concorrentes permita que o sistema continue operacional mesmo na ocorrência de falhas. Um dos grandes desafios dos sistemas concorrentes é a necessidade de sincronização e coordenação entre processos para garantir a ordem de execução e prevenir condições de corrida e problemas de integridade de dados.

Outra característica importante é a ausência de um relógio global, que influencia significativamente o design e a implementação de um sistema distribuído. O desafio reside em manter um relógio de tempo consistente e acessível a todos os componentes do sistema, independentemente da localização física do componente. Em sistemas distribuídos, onde múltiplos computadores operam simultaneamente e podem estar geograficamente dispersos, manter o relógio global sincronizado é particularmente desafiador devido a questões como latência de comunicação. Este desafio é agravado pela comunicação entre nós do sistema que pode sofrer atrasos variáveis, dificultando a garantia de que as mensagens sejam recebidas e processadas em uma ordem que reflete precisamente o momento em que foram enviadas. Além disso, é crucial considerar que cada dispositivo em um sistema distribuído possui seu próprio relógio interno que pode divergir ou derivar em relação ao tempo devido a variações na temperatura e no hardware. O desafio é alcançar uma sincronização perfeita entre os relógios, o que requer um mecanismo que possa compensar tanto a latência da rede quanto a deriva do relógio, o que é complexo e pode ser impraticável em grande escala.

A falha de componentes também é uma característica crucial de um sistema distribuído, pois lida com o desafio de manter um sistema disponível mesmo quando um componente falha. Dada a natureza complexa e interconectada dos sistemas distribuídos, onde múltiplas unidades operacionais colaboram para executar tarefas, qualquer interrupção em um único componente pode ter repercussões em todo o sistema. Lidar com essas falhas é um desafio para um sistema distribuído, em torná-las transparentes ao usuário que não percebe que o recurso falhou e que o sistema se recupera subsequentemente dessa falha. Esse mascaramento é um dos problemas mais difíceis, sendo o principal motivo a incapacidade de distinguir entre um recurso inativo e um recurso lento de um sistema (S. Tanenbaum; Van Steen, 2007).

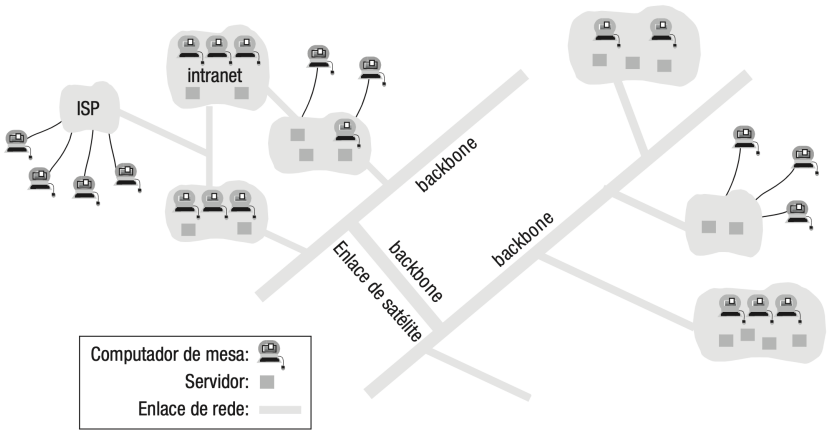

Figura 1: Modelo de sistemas distribuídos. Fonte: Coulouris; Dollimore; Kindberg; Blair, Sistemas Distruídos - Conceitos e Projetos, página 9, 2013.

2.2 RabbitMQ

O RabbitMQ é um *broker* de mensagens *open source* que atua como um intermediário no processamento e transmissão de mensagens entre sistemas distribuídos, aplicações e serviços. Implementa o protocolo AMQP (*Advanced Message Queuing Protocol*) e suporta outros protocolos, como MQTT (*Message Queuing Telemetry Transport*) e STOMP (*Streaming Text Oriented Messaging Protocol*), através de plugins. Analogamente, o RabbitMQ funciona como um carteiro, sendo responsável por receber e entregar mensagens aos seus destinatários. Ele utiliza termos específicos, como 'Produtores', que são os emissores das mensagens, e 'Filas', onde as mensagens são coletadas e organizadas. Os 'Consumidores' são as aplicações que recebem as mensagens. Neste contexto, os produtores podem enviar mensagens para uma única fila e vários consumidores podem acessar dados de uma mesma fila.

Desenvolvido em 1980 pelo *Ericsson Computer Science Laboratory* na linguagem de programação funcional Erlang, o RabbitMQ foi criado para resolver problemas em ambientes críticos de mensagens em larga escala. Uma das suas características principais é a tolerância a falhas, permitindo que componentes individuais falhem sem comprometer o sistema como um todo. A filosofia *Let It Crash* adotada permite que processos falhem e sejam reiniciados conforme necessário, dependendo dos supervisores de falhas para detectar e reiniciar processos. Outro aspecto é a capacidade de execução em *Soft Real-Time*, crucial em sistemas que requerem respostas quase instantâneas onde atrasos ocasionais são aceitáveis. O RabbitMQ também é conhecido por sua alta disponibilidade, frequentemente descrito como tendo uma confiabilidade de 'cinco noves', indicando operações confiáveis

99,999% do tempo, o que se traduz em cerca de 5,26 minutos de inatividade por ano, essencial para sistemas onde interrupções são altamente disruptivas (M. Roy, 2018).

Aspectos importantes do RabbitMQ incluem configurações do *broker* que asseguram a troca eficaz de mensagens entre produtores e consumidores. Algumas dessas configurações são o *Round-Robin Dispatching*, que distribui mensagens de forma sequencial e igualitária entre os *Workers*; o *Message Acknowledgment*, que envolve a confirmação de que uma mensagem foi recebida e processada com sucesso; a *Message Durability*, que garante que mensagens e filas sejam marcadas como duráveis para prevenir perdas em caso de falhas; e o *Fair Dispatch* , que ajusta o Prefetch Count para distribuir o trabalho equitativamente entre os *Workers*, considerando o tempo de processamento de cada tarefa. O RabbitMQ é uma ferramenta robusta com configurações que garantem a entrega eficiente e a velocidade das mensagens aos consumidores, destacando sua importância na operacionalidade, escalabilidade e eficiência de sistemas, como ilustrado na **Figura 2**.

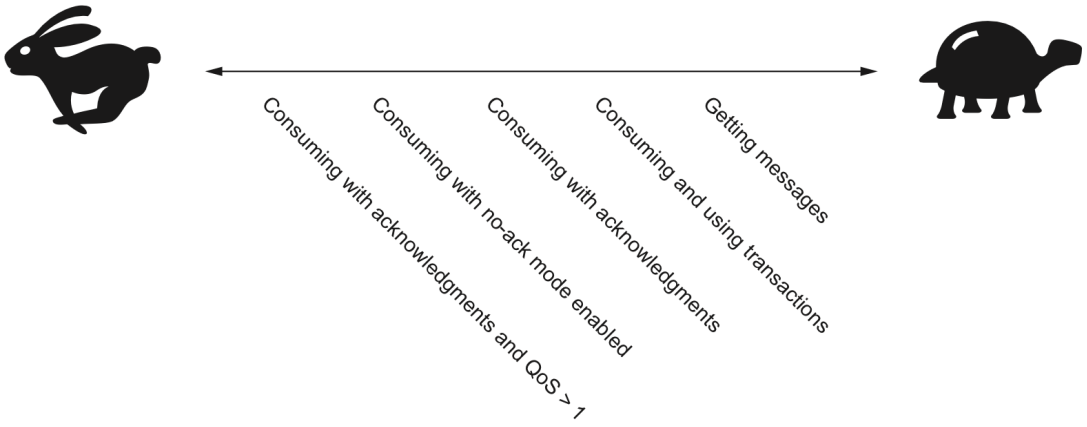

Figura 2: Ajuste dos consumidores para desempenho em escala. Fonte: M. Roy, RabbitMQ in depth, página 84, 2018.

No RabbitMQ, como demonstrado na **Figura 2**, é possível aprimorar a qualidade e a velocidade de entrega das mensagens aos consumidores, evitando sobrecargas do sistema ou ociosidade que poderiam tornar o sistema ineficiente e prejudicar processos. Isso é viabilizado através das configurações de QoS (*Quality Of Service Settings*), permitindo que os consumidores configurem um número predeterminado de mensagens que desejam receber. Com o QoS configurado, o RabbitMQ torna-se mais eficiente ao especificar quantas mensagens serão predefinidas para o consumidor (M. Roy, 2018). Essa eficiência é alcançada pela configuração do *Prefetch Count*. Para evitar impactos negativos na taxa de transferência de mensagens, é crucial avaliar o "desempenho" dessa configuração em aplicações consumidoras de alta velocidade e considerar que os benefícios das configurações específicas podem variar conforme a composição da mensagem, comportamentos dos consumidores e outros fatores, como o sistema operacional e a linguagem utilizada.

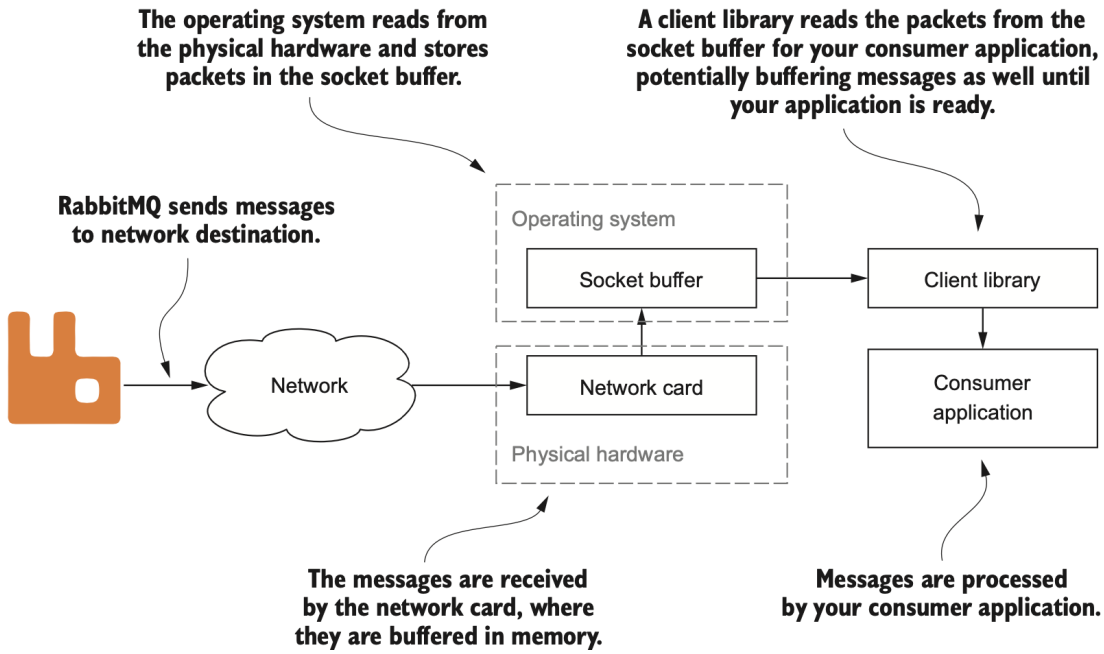

Figura 3: Representação do funcionamento do envio de mensagens aos consumidores. Fonte: M. Roy, RabbitMQ in depth, página 85, 2018.

2.3 Controladores Fuzzy

Os controladores *fuzzy*, ou sistemas de controle difuso, são baseados na lógica *fuzzy*. Esta metodologia de controle se fundamenta na lógica difusa, uma extensão da lógica booleana que gerencia conceitos de verdade parcial, permitindo que a verdade varie entre 0 e 1. Contrariamente à lógica booleana clássica, que assume valores binários de verdade (0 ou 1), a lógica difusa é particularmente útil em sistemas complexos onde as relações entre entradas e saídas não são precisas ou quando essas relações são difíceis de modelar usando métodos convencionais de controle.

Uma das aplicações mais notáveis da lógica *fuzzy* é o controlador de trens utilizado no metrô de Sendai, no Japão. Este sistema superou tanto operadores humanos quanto controladores automatizados convencionais. Os controladores convencionais iniciam ou param os trens reagindo a marcadores de posição que indicam a distância até uma estação. Devido à programação rígida, os trajetos podem ser desconfortáveis; por exemplo, o controlador automatizado aplicaria a mesma intensidade de frenagem a um trem que estivesse a 100 metros de uma estação, independentemente de estar subindo ou descendo uma ladeira. Este problema foi resolvido na metade dos anos 1980, quando engenheiros da Hitachi utilizaram regras *fuzzy* para controlar a aceleração, desaceleração e frenagem dos trens de forma mais suave do que um operador humano habilidoso poderia fazer. As regras, baseadas em conjuntos *fuzzy*, consideravam uma ampla gama de variáveis do "desempenho" contínuo do trem, como a frequência e a magnitude das mudanças de velocidade e a proximidade da velocidade real em relação à velocidade máxima. Em testes simulados, o controlador *fuzzy* superou uma versão automatizada em conforto dos passageiros, redução dos tempos de viagem e até mesmo alcançou uma redução de dez por cento no consumo de energia do trem (Kosko; Isaka, 1993).

As discussões sobre a teoria dos conjuntos difusos, empregada em controladores que utilizam a lógica *fuzzy*, têm sido conduzidas ao longo de mais de 50 anos, com a primeira publicação feita por Zadeh em 1965 e Goguen entre 1967 e 1969. Ao longo de mais de quatro décadas, os conceitos, conforme definidos por Zimmermann (2001), incluem:

- Como uma teoria formal que, ao amadurecer, tornou-se mais sofisticada e especificada, e foi ampliada por ideias e conceitos originais, bem como pela "incorporação"de áreas clássicas da matemática como álgebra, teoria dos grafos, topologia, etc., através da generalização (fuzzificação) das mesmas;

- Como uma tecnologia *fuzzy* orientada à aplicação, ou seja, como uma ferramenta para modelagem, resolução de problemas e mineração de dados que se mostrou superior a métodos existentes em muitos casos e como um complemento atraente para abordagens clássicas em outros casos;

As características principais de um conjunto difuso, além de modelar incertezas, incluem a implementação de modelos de relaxamento, compactação, raciocínio preservador de significado e determinação eficiente de soluções aproximadas.

O modelo de relaxamento é utilizado pelas teorias *fuzzy* para flexibilizar ou generalizar métodos clássicos de caráter dicotômico para um caráter gradual. Conforme mencionado por Zimmermann (2001), alguns métodos incluem a programação matemática *fuzzy*, agrupamento *fuzzy*, redes de *Petri Fuzzy* e análise multicritério *fuzzy*. Todos esses métodos compartilham uma lógica multivalorada que permite raciocínio aproximado, facilitando o manejo de conceitos vagamente definidos e situações onde as informações são imprecisas ou incertas.

Devido à capacidade limitada da memória de curto prazo humana e de sistemas, os conjuntos *fuzzy* também empregam um modelo de compactação. Esse modelo reduz a complexidade dos dados a um grau aceitável, geralmente utilizando variáveis linguísticas, permitindo que grandes volumes de dados sejam apresentados a observadores humanos de forma compreensível.

O modelo de Raciocínio Preservador de Significado é outra característica dos conjuntos difusos. Utiliza-se do raciocínio aproximado, onde significados são atribuídos, diferentemente de uma lógica verdadeira ou falsa, a palavras e frases por meio de variáveis linguísticas. Esse processo é realizado através de inferências que são capazes de processar expressões linguísticas significativas, substituindo símbolos por funções de pertinência de conjuntos *fuzzy*, que podem ser retransmitidas em palavras ou frases através de aproximação linguística.

Como um modelo de conjuntos difusos, a Determinação Eficiente de Soluções Aproximadas busca determinar soluções aproximadas para problemas reais de maneira eficiente ou acessível. Apesar dos desafios para alcançar esse objetivo, sistemas baseados em regras *fuzzy* têm se mostrado muito mais eficientes do que sistemas que utilizam equações diferenciais para quantificar incertezas em sistemas complexos, segundo Zimmermann (2001).

Os controladores *fuzzy* possuem um conceito simples de implementação, consistindo em três fases: entrada de dados, processamento de dados e saída de dados. Na etapa de entrada, os dados são coletados através de sensores, valores configurados como entrada em um sistema ou interruptores, sendo mapeados para as funções de pertinência e valores verdade apropriados. A etapa de processamento baseia-se em uma coleção de regras lógicas no formato "Se-Então", onde a parte "Se"constitui o antecedente e a parte "Então"o consequente, responsável por ativar cada regra apropriada e combinar os resultados obtidos. Por fim, a fase de saída converte os resultados combinados em um valor específico de controle.

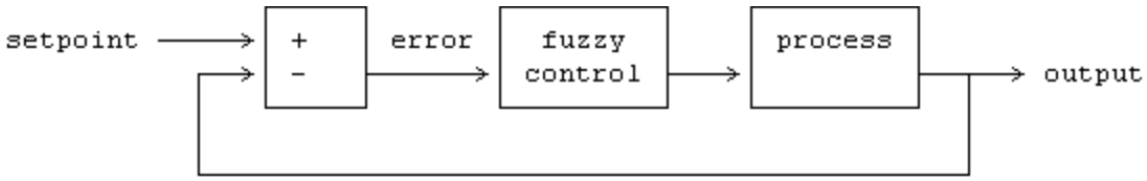

Figura 4: Arquitetura de implementação de um controlador *fuzzy*.

3 Implementação

Este capítulo aborda os conceitos e as implementações discutidos no **capítulo 2**, que foram utilizados para os experimentos.

3.1 Publish - Subscribe

O *Publish - Subscribe*, ou modelo de produtores e consumidores, é um padrão de troca de mensagens onde os remetentes das mensagens, denominados produtores, não enviam mensagens diretamente aos receptores específicos, ou seja, aos consumidores. Como descrito na **Seção 2.1**, este modelo permite que sistemas distribuídos comuniquem-se classificando mensagens em categorias sem conhecer quais consumidores (ou sistemas) poderão recebê-las. Os consumidores manifestam interesse em uma ou mais categorias e recebem apenas as mensagens pertinentes, sem saber quais produtores as enviaram. O modelo produtor-consumidor apresenta diversos benefícios, entre eles:

- Separa subsistemas que ainda precisam se comunicar. Os subsistemas podem ser gerenciados independentemente e as mensagens podem ser gerenciadas corretamente mesmo se um ou mais destinatários estiverem offline;
- Aumenta a escalabilidade e melhora a capacidade de resposta do remetente. O remetente pode enviar rapidamente uma única mensagem para o canal de entrada, e voltar às responsabilidades de processamento principais. A infraestrutura de mensagens é responsável por garantir que as mensagens sejam entregues aos assinantes interessados;
- Aumento da confiabilidade. Mensagens assíncronas ajudam na execução tranquila dos aplicativos sob cargas maiores e lidam com falhas intermitentes com mais eficiência;
- Permite o processamento agendado ou adiado. Os assinantes podem esperar para ver as mensagens em um horário de pouco movimento, ou as mensagens podem ser roteadas ou processadas de acordo com um agendamento específico;
- Ela permite uma integração mais simples entre sistemas com plataformas diferentes, linguagens de programação ou protocolos de comunicação, bem como entre sistemas locais e aplicativos em execução na nuvem.

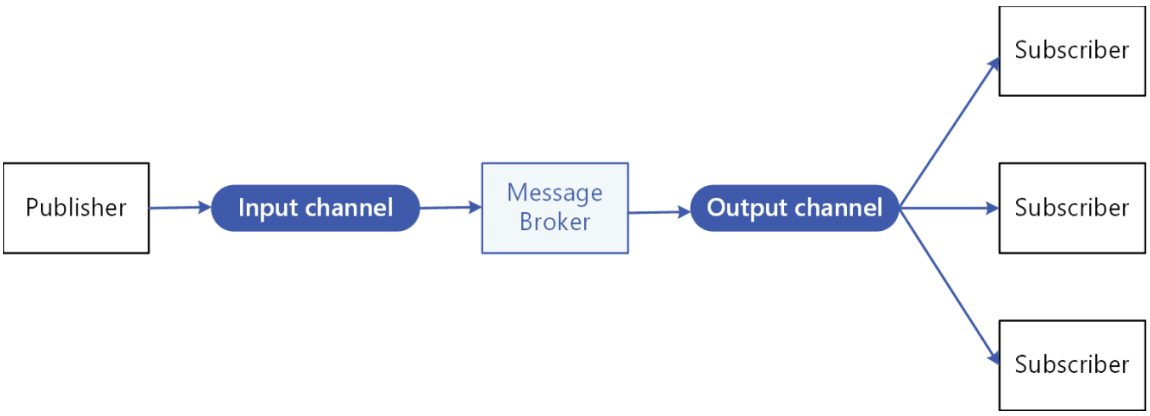

Figura 5: Diagrama lógico dos componentes do modelo Pulicador-Consumidor.

Como parte da implementação para os experimentos, foram adotadas as seguintes características na criação do Produtor:

- Uso da linguagem de programação Go;
- O *broker* utilizado foi o RabbitMQ, com o canal de comunicação aberto na variável ch;
- A fila foi declarada através do método *QueueDeclare*, chamado no canal aberto;
- As mensagens foram enviadas utilizando o método *Publish*, que empregou a biblioteca AMQP e o método *Publishing* para configurar o formato de envio das mensagens;
- Foram publicados um bilhão de mensagens.

O código que implementa o *publish* é mostrado a seguir:

```
1  q, err := ch.QueueDeclare(
2      "task_queue", // name
3      true,         // durable
4      false,        // delete when unused
5      false,        // exclusive
6      false,        // no-wait
7      nil,          // arguments
8  )
9
10 failOnError(err, "Failed to declare a queue")
11
12 for i := 1; i <= 1000000000; i++ {
13
14     body := fmt.Sprintf("The queue - Message %d", i)
15
16     err = ch.Publish(
17         "",     // exchange
18         q.Name, // routing key
19         false,  // mandatory
20         false,  // immediate
21         amqp.Publishing{
22             DeliveryMode: amqp.Persistent,
23             ContentType:  "text/plain",
24             Body:         []byte(body),
25         }
26     )
27
28     failOnError(err, "Failed to publish a message")
29
30     log.Printf(" [x] Sent %s", body)
31
32 }
33
34 failOnError(err, "Failed to publish a message")
35
36 log.Printf(" [x] Sent %s", body)
```

Neste código, a Linha 1 inicia a declaração da fila chamada *task queue*, configurada como durável, não exclusiva e sem espera automática após o uso, utilizando o método *QueueDeclare*. Após a declaração da fila, a Linha 10 verifica e trata erros que podem ocorrer durante a declaração, garantindo que qualquer falha seja adequadamente sinalizada com a mensagem *Failed to declare a queue*.

O loop iniciado na Linha 12 itera um bilhão de vezes, onde cada iteração, a partir da Linha 14, prepara e envia uma mensagem. As mensagens, formatadas para incluir o número da iteração, são configuradas para persistência e tipo de conteúdo *text/plain* através do método *Publish* chamado na Linha 16. Este método é usado para publicar cada mensagem na fila anteriormente declarada, usando a chave de roteamento correspondente ao nome da fila. A persistência das mensagens é garantida pelo parâmetro *DeliveryMode*.

Erros durante a publicação das mensagens são tratados nas Linhas 28 e 34, onde falhas são reportadas com a mensagem *Failed to publish a message*. A conclusão do envio de cada mensagem é registrada pelo método *log.Printf*, conforme indicado nas Linhas 30 e 36, reiterando a mensagem enviada.

Para os experimentos, configurou-se um Consumidor com características específicas detalhadas a seguir:

- Utilização da linguagem de programação Go;
- O *broker* RabbitMQ foi utilizado, com o canal de comunicação estabelecido na variável ch;
- Declaração da fila para consumo das mensagens por meio do método *QueueDeclare*, executado no canal aberto;
- Consumo das mensagens efetuado através do método *Consume*, com o *Auto-Ack* configurado como falso, o que indica que a confirmação de recebimento das mensagens pelo RabbitMQ é manual;
- Configuração inicial do *Prefetch Count* como 1;
- Estabelecimento de um contador de tempo para mensurar a média de mensagens consumidas por segundo a cada 15 segundos.

O código que implementa o *subscribe* é mostrado a seguir:

```
1  if prefetchValue == 1 {
2      err = ch.Qos(prefetchValue, 0, true)
3      failOnError(err, "Failed to set QoS")
4  }
5
6  q, err := ch.QueueDeclare(
7      "task_queue", // name
8      true,         // durable
9      false,        // delete when unused
10     false,        // exclusive
11     false,        // no-wait
12     nil,          // arguments
13 )
14
15 failOnError(err, "Failed to declare a queue")
16
```

```
17 msg, err := ch.Consume(
18     q.Name, // queue
19     "",     // consumer
20     false,  // auto-ack
21     false,  // exclusive
22     false,  // no-local
23     false,  // no-wait
24     nil,    // args
25 )
26
27 failOnError(err, "Failed to register a consumer")
28
29 ticker := time.NewTicker(15 * time.Second)
30 defer ticker.Stop()
```

Neste código, a configuração inicial do *Quality of Service* (QoS) ocorre entre as Linhas 1 e 4. Quando *prefetchValue* é igual a 1, o sistema ajusta o QoS para este valor na Linha 2 com a função ch.Qos, e verifica erros na Linha 3, garantindo que a configuração seja aplicada corretamente.

A declaração de uma fila chamada *task queue* é realizada nas Linhas 6 a 13. Esta fila é configurada como durável, não exclusiva e sem espera imediata, com a função *QueueDeclare*. Erros durante a declaração da fila são tratados na Linha 15, assegurando que a fila seja declarada sem problemas.

A configuração do consumo de mensagens ocorre nas Linhas 17 a 25, utilizando a função *ch.Consume*. Esta função especifica a fila para consumo, configurando o sistema para não acusar automaticamente o recebimento das mensagens e garantindo que o consumo não é exclusivo ou local. Erros durante o registro do consumidor são tratados na Linha 27.

Finalmente, um temporizador é estabelecido na Linha 29 com a função *time.NewTicker*, que dispara a cada 15 segundos. A Linha 30 utiliza defer *ticker.Stop()* para assegurar que o temporizador seja interrompido ao final do processo, prevenindo execuções indesejadas após o término das operações do consumidor.

3.2 Broker

Um *broker* de mensagens atua como mediador entre diferentes componentes do sistema, facilitando a troca de mensagens de forma eficaz e confiável. Essa arquitetura permite que os componentes se comuniquem indiretamente, sem a necessidade de conhecer os detalhes específicos uns dos outros, o que promove desacoplamento e escalabilidade. Para os experimentos, implementou-se o *broker* RabbitMQ, discutido na **Seção 2.2**, com as seguintes características:

- Utilização da linguagem de programação GO;
- A biblioteca AMQP e o método Dial foram utilizados para conectar com o *broker*, tanto no produtor quanto no consumidor;
- Utilização da *RabbitMQ Latest Image* para executar o *broker* no *Docker*;

O código que implementa o *broker* é mostrado a seguir:

```
1 conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
2 failOnError(err, "Failed to connect to RabbitMQ")
3 defer conn.Close()
4
5 ch, err := conn.Channel()
6 failOnError(err, "Failed to open a channel")
7 defer ch.Close()
```

Neste código, a Linha 1 estabelece uma conexão com o *broker* RabbitMQ utilizando a biblioteca AMQP, especificando as credenciais de usuário e a URL do servidor. A Linha 2 verifica se houve erro na tentativa de conexão e reporta um erro caso a conexão falhe. A Linha 3 garante que a conexão seja fechada quando a execução do código terminar, evitando vazamentos de recursos.

Em seguida, a Linha 5 cria um canal de comunicação sobre a conexão estabelecida, essencial para a operação com o *broker*. A Linha 6 verifica se houve erro na abertura do canal, reportando um erro caso falhe. Por fim, a Linha 7 assegura que o canal seja fechado ao final da execução, similar ao procedimento com a conexão.

3.3 Membership Functions

As *Membership Functions*, ou funções de pertinência, são utilizadas para definir o grau de pertencimento de um elemento a um conjunto *fuzzy*. Uma função de pertinência atribui a cada elemento de um universo de discurso um valor real entre 0 e 1, representando o grau de verdade da afirmação "o elemento x pertence ao conjunto A". Valores próximos de 1 indicam alta compatibilidade ou pertinência do elemento ao conjunto, enquanto valores próximos de 0 indicam baixa pertinência, conforme discutido na **Seção 2.3**.

Para os experimentos, foram implementadas três funções de pertinência: *Triangular*, *Gaussian* e *Bell*. A escolha das funções de pertinência *gaussianas*, *triangulares* e *bell* foi motivada por suas características específicas de suavidade e capacidade de capturar nuances nas variações das taxas de mensagens. A função *gaussiana* foi selecionada devido à sua propriedade de suavidade e forma simétrica, ideal para modelar fenômenos onde as mudanças ocorrem de maneira gradual e centrada. A função *triangular* foram utilizadas por sua simplicidade e eficiência computacional, proporcionando uma boa aproximação em cenários com transições mais abruptas. Já a função *bell*, com seus contornos suaves e ajustáveis, foram aplicadas para casos que exigiam uma modelagem mais flexível das transições de estado.

3.3.1 Triangular

O código que implementa a *Triangular Membership Function* e a fuzzificação das entradas é mostrado a seguir:

```
1 func triangularMF(x float64, a float64, b float64, c float64) float64 {
2     return math.Max(0, math.Min((x-a)/(b-a), (c-x)/(c-b)))
3 }
4
5 func fuzzyficationMsgSecInput(msgSec float64) map[string]float64 {
6     fuzzy := make(map[string]float64)
7
8     fuzzy[VeryLargeNegative] = triangularMF(msgSec, -10000, -10000, -8000)
```

```
9       fuzzy[LargeNegative] = triangularMF(msgSec, -9000, -8000, -6000)
10      fuzzy[MediumNegative] = triangularMF(msgSec, -7000, -6000, -4000)
11      fuzzy[SmallNegative] = triangularMF(msgSec, -5000, -4000, -2000)
12      fuzzy[VerySmallNegative] = triangularMF(msgSec, -3000, -2000, 0)
13      fuzzy[Zero] = triangularMF(msgSec, -1000, 0, 1000)
14      fuzzy[VerySmallPositive] = triangularMF(msgSec, 0, 2000, 3000)
15      fuzzy[SmallPositive] = triangularMF(msgSec, 2000, 4000, 5000)
16      fuzzy[MediumPositive] = triangularMF(msgSec, 4000, 6000, 7000)
17      fuzzy[LargePositive] = triangularMF(msgSec, 6000, 8000, 9000)
18      fuzzy[VeryLargePositive] = triangularMF(msgSec, 8000, 10000, 10000)
19
20      return fuzzy
21  }
22
23  func fuzzyficationOutput(n float64) map[string]float64 {
24      r := map[string]float64{}
25
26      r[VeryLargeDecrease] = triangularMF(n, -8, -7, -6)
27      r[LargeDecrease] = triangularMF(n, -7, -6, -5)
28      r[MediumDecrease] = triangularMF(n, -6, -5, -4)
29      r[SmallDecrease] = triangularMF(n, -5, -4, -3)
30      r[VerySmallDecrease] = triangularMF(n, -4, -3, -2)
31      r[Maintain] = triangularMF(n, -1, 0, 1)
32      r[VerySmallIncrease] = triangularMF(n, 2, 3, 4)
33      r[SmallIncrease] = triangularMF(n, 3, 4, 5)
34      r[MediumIncrease] = triangularMF(n, 4, 5, 6)
35      r[LargeIncrease] = triangularMF(n, 5, 6, 7)
36      r[VeryLargeIncrease] = triangularMF(n, 6, 7, 8)
37
38      return r
39  }
```

Neste código, a Linha 1 define a função *triangularMF*, que calcula o grau de pertinência usando parâmetros de forma triangular. A **Linha 5** inicia a função *fuzzyficationMsgSecInput*, que cria um *array* para categorizar a taxa de mensagens por segundo (msgSec) em diferentes níveis de pertinência. As Linhas 8 a 18 preenchem este *array* utilizando a função *triangularMF* para determinar o grau de pertinência em categorias variando de muito negativo a muito positivo, cada uma definida por diferentes parâmetros de triangularidade específicos para a aplicação. A Linha 20 retorna o *array* completo de categorias com seus respectivos graus de pertinência.

A Linha 23 começa a função *fuzzyficationOutput*, que prepara um *array* similar para a saída do controlador *fuzzy* baseado em uma entrada n. As Linhas 26 a 36 atribuem graus de pertinência para várias categorias de aumento ou diminuição da saída, utilizando a mesma função *triangularMF* ajustada para cada categoria específica. A Linha 38 retorna este *array*, finalizando a definição de como a saída do controlador é ajustada em resposta a diferentes condições.

3.3.2 Gaussian

O código que implementa a *Gaussian Membership Function* e a fuzzificação das entradas é mostrado a seguir:

```
1   func gaussianMembership(x, mu, sigma float64) float64 {
2       if sigma == 0 {
3           if x == mu {
4               return 1.0
5           }
```

```
6           return 0.0
7       }
8       return math.Exp(-math.Pow(x-mu, 2) / (2 * math.Pow(sigma, 2)))
9   }
10
11  func calculateSigma(leftPeak, rightPeak float64) float64 {
12      return math.Abs(rightPeak-leftPeak) / 2
13  }
14
15  func fuzzyficationMsgSecInput(msgSec float64) map[string]float64 {
16      fuzzy := make(map[string]float64)
17
18      peaks := []float64{-10000, -5000, -2500, -1250, -625, 0,
19      625, 1250, 2500, 5000, 10000,}
20
21      sigmas := make([]float64, len(peaks))
22      for i := range peaks {
23          if i == len(peaks)-1 {
24              sigmas[i] = calculateSigma(peaks[i-1], peaks[i])
25          } else {
26              sigmas[i] = calculateSigma(peaks[i], peaks[i+1])
27          }
28      }
29
30      fuzzy[VeryLargeNegative] = gaussianMembership(msgSec, peaks[0], sigmas[0])
31      fuzzy[LargeNegative] = gaussianMembership(msgSec, peaks[1], sigmas[1])
32      fuzzy[MediumNegative] = gaussianMembership(msgSec, peaks[2], sigmas[2])
33      fuzzy[SmallNegative] = gaussianMembership(msgSec, peaks[3], sigmas[3])
34      fuzzy[VerySmallNegative] = gaussianMembership(msgSec, peaks[4], sigmas[4])
35      fuzzy[Zero] = gaussianMembership(msgSec, peaks[5], sigmas[5])
36      fuzzy[VerySmallPositive] = gaussianMembership(msgSec, peaks[6], sigmas[6])
37      fuzzy[SmallPositive] = gaussianMembership(msgSec, peaks[7], sigmas[7])
38      fuzzy[MediumPositive] = gaussianMembership(msgSec, peaks[8], sigmas[8])
39      fuzzy[LargePositive] = gaussianMembership(msgSec, peaks[9], sigmas[9])
40      fuzzy[VeryLargePositive] = gaussianMembership(msgSec, peaks[10], sigmas[10])
41
42      return fuzzy
43  }
44
45  func fuzzyficationOutput(n float64) map[string]float64 {
46
47      r := map[string]float64{}
48
49      peaks := []float64{-8, -6, -4, -2, -1, 0, 1, 2, 4, 6, 8}
50
51      sigmas := make([]float64, len(peaks)-1)
52
53      for i := range sigmas {
54          sigmas[i] = calculateSigma(peaks[i], peaks[i+1])
55
56      }
57
58      r[VeryLargeDecrease] = gaussianMembership(n, -8, sigmas[0])
59      r[LargeDecrease] = gaussianMembership(n, -6, sigmas[1])
60      r[MediumDecrease] = gaussianMembership(n, -4, sigmas[2])
61      r[SmallDecrease] = gaussianMembership(n, -2, sigmas[3])
62      r[VerySmallDecrease] = gaussianMembership(n, -1, sigmas[4])
63      r[Maintain] = gaussianMembership(n, 0, sigmas[5])
64      r[VerySmallIncrease] = gaussianMembership(n, 1, sigmas[6])
65      r[SmallIncrease] = gaussianMembership(n, 2, sigmas[7])
66      r[MediumIncrease] = gaussianMembership(n, 4, sigmas[8])
67      r[LargeIncrease] = gaussianMembership(n, 6, sigmas[9])
68      r[VeryLargeIncrease] = gaussianMembership(n, 8, sigmas[9])
```

```
69
70      return r
71  }
```

Neste código, a Linha 1 introduz a função *gaussianMembership*, que calcula o grau de pertinência de um valor x para um conjunto *fuzzy* com base em uma distribuição gaussiana, determinada pela média (mu) e desvio padrão (sigma). As Linhas 2 a 8 lidam com o cálculo desta função, considerando condições especiais para evitar divisão por zero e fornecendo a fórmula para a curva gaussiana normal.

A Linha 11 define a função *calculateSigma*, que calcula o valor de *sigma* como metade da diferença absoluta entre dois picos, fornecendo uma medida de dispersão para a função gaussiana. A Linha 15 inicia a função *fuzzyficationMsgSecInput*, que cria um *array* para armazenar graus de pertinência para diferentes categorias baseadas na taxa de mensagens por segundo (msgSec). As Linhas 18 a 28 preparam os dados necessários para este cálculo, definindo picos e calculando *sigmas* correspondentes para cada categoria.

As Linhas 30 a 40 aplicam a função *gaussianMembership* para determinar o grau de pertinência de msgSec para várias categorias, de *VeryLargeNegative* a *VeryLargePositive*, utilizando os picos e *sigmas* pré-calculados. A Linha 42 retorna o *array* completo, que associa cada categoria com seu grau de pertinência correspondente.

A função *fuzzyficationOutput*, iniciada na Linha 45, opera de maneira similar para calcular e mapear graus de pertinência para as saídas de um controlador *fuzzy* com base em um valor n. As Linhas 49 a 55 preparam os picos e *sigmas* para a saída, e as Linhas 58 a 68 aplicam a função *gaussianMembership* para categorizar a saída em níveis desde *VeryLargeDecrease* até *VeryLargeIncrease*. A Linha 70 retorna o *array* resultante.

3.3.3 Bell

O código que implementa a *Bell Membership Function* e a fuzzificação das entradas é mostrado a seguir:

```
1  func GeneralizedBellMembership(x, a, b, c float64) float64 {
2      return 1.0 / (1.0 + math.Pow(math.Abs((x-c)/a), 2*b))
3  }
4
5  func fuzzyficationMsgSecInput(msgSec float64) map[string]float64 {
6      fuzzy := make(map[string]float64)
7
8      centers := map[string]struct{ a, b, c float64 }{
9          VeryLargeNegative: {3000, 2, -10000},
10         LargeNegative:     {2000, 2, -5000},
11         MediumNegative:    {1000, 2, -2500},
12         SmallNegative:     {500, 2, -1250},
13         VerySmallNegative: {250, 2, -625},
14         Zero:              {125, 2, 0},
15         VerySmallPositive: {250, 2, 625},
16         SmallPositive:     {500, 2, 1250},
17         MediumPositive:    {1000, 2, 2500},
18         LargePositive:     {2000, 2, 5000},
19         VeryLargePositive: {4000, 2, 10000},
20     }
21
22     for label, params := range centers {
23         fuzzy[label] = GeneralizedBellMembership(msgSec,
24         params.a,  params.b, params.c)
25     }
26     return fuzzy
```

```
27 }
28
29 func fuzzyficationOutput(x float64) map[string]float64 {
30     result := make(map[string]float64)
31
32     cValues := map[string]struct{ a, b, c float64 }{
33         VeryLargeDecrease: {4.0, 2.0, -8},
34         LargeDecrease:     {3.5, 2.0, -6},
35         MediumDecrease:    {3.0, 2.0, -4},
36         SmallDecrease:     {2.5, 2.0, -2},
37         VerySmallDecrease: {2.0, 2.0, -1},
38         Maintain:          {1.5, 2.0, 0},
39         VerySmallIncrease: {2.0, 2.0, 1},
40         SmallIncrease:     {2.5, 2.0, 2},
41         MediumIncrease:    {3.0, 2.0, 4},
42         LargeIncrease:     {3.5, 2.0, 6},
43         VeryLargeIncrease: {4.0, 2.0, 8},
44     }
45
46     for label, params := range cValues {
47         result[label] = GeneralizedBellMembership(x, params.a,
48         params.b, params.c)
49     }
50
51     return result
52 }
```

Neste código, a Linha 1 define a função *GeneralizedBellMembership*, que calcula o grau de pertinência usando uma função de pertinência em forma de sino generalizada. Essa função recebe quatro parâmetros (x, a, b, c) e retorna o valor de pertinência calculado. A Linha 5 inicia a função *fuzzyficationMsgSecInput*, que cria um *array* chamado *fuzzy* para armazenar graus de pertinência para diferentes categorias com base na taxa de mensagens por segundo (msgSec). As Linhas 8 a 20 definem os centros e parâmetros de cada categoria, variando de *VeryLargeNegative* a *VeryLargePositive*.

As Linhas 22 a 25 aplicam a função de pertinência a cada categoria usando os parâmetros definidos, atribuindo o resultado ao *array fuzzy*. A Linha 26 retorna o *array* completo com todos os graus de pertinência calculados para msgSec.

A função *fuzzyficationOutput*, iniciada na Linha 29, opera de forma similar à anterior, mas é aplicada para calcular e mapear graus de pertinência baseados em um valor x que representa a saída do sistema. As Linhas 32 a 44 preparam um novo *array* chamado *result* e definem os parâmetros para categorias que variam de *VeryLargeDecrease* a *VeryLargeIncrease*.

As Linhas 46 a 49 calculam o grau de pertinência para cada uma dessas categorias de saída utilizando a função *GeneralizedBellMembership*. Finalmente, a Linha 51 retorna o *array result*, que associa cada categoria com seu respectivo grau de pertinência, completando assim a definição do processo de fuzzyficação para a saída do controlador.

4 Avaliação dos Experimentos

Este capítulo aborda a avaliação de "desempenho" comparativa das *Membership Functions* utilizadas para ajustar o RabbitMQ. O capítulo está estruturado em três partes principais: Objetivo, que descreve o propósito da avaliação; Experimentos e Ferramentas, detalhando o ambiente e os experimentos realizados; e Resultados, apresentando as discussões sobre os achados.

4.1 Objetivo

O objetivo principal dos experimentos é avaliar o "desempenho" de três *Membership Functions* — *Triangular*, *Gaussian* e *Bell* — utilizando três, cinco, sete, nove e onze conjuntos *fuzzy*. O intuito é identificar qual combinação oferece os melhores resultados na recomendação do *Prefetch Count* do RabbitMQ. As avaliações foram realizadas com base no NRMSE (*Normalised Root Mean Square Error*) e na porcentagem de quantos valores estão abaixo ou acima de 20% do objetivo de mensagens previamente estabelecido. Para atingir esse objetivo, consideraram-se as seguintes especificações:

- Medição da quantidade de mensagens a cada 15 segundos, onde a média é calculada com base no número de mensagens recebidas pelo consumidor a cada segundo durante esse período;
- Avaliação de qual *Membership Function* e qual quantidade de conjuntos *fuzzy* apresentaram o melhor "desempenho" na recomendação do *Prefetch Count*, buscando aproximar ao máximo a quantidade de mensagens consumidas do objetivo estabelecido;
- Avaliação dos resultados com base no NRMSE, que mensura a magnitude das diferenças entre modelos preditivos e valores observados.

4.2 Experimentos e Ferramentas

Os experimentos foram executados em um *Laptop* MacBook Air com as seguintes configurações:

- Processador: M1 ;
- CPU: 8 núcleos (4 de desempenho e 4 de eficiência);
- GPU: 7 núcleos, GPU de 8 núcleos;
- Neural Engine de 16 núcleos;
- SO: macOS;
- Kernel: XNU;
- Memória: 8GB;
- Armazenamento: 256GB SSD;

Para a execução dos experimentos, o Docker *Desktop* estava ativo com a *Latest Image* do RabbitMQ em operação, acessível através da porta 5672:5672. Após a conclusão de cada experimento para uma *Membership Function* e um conjunto *fuzzy*, o *Container* e a *Image* do RabbitMQ eram removidos e reiniciados para os experimentos subsequentes.

Para assegurar a validade dos experimentos, foram implementadas várias estratégias para minimizar a influência de variáveis externas e configuracionais. A uniformidade do ambiente de teste foi mantida através da padronização dos hardware e das configurações do software, e os testes foram repetidos em diferentes períodos para garantir a consistência dos resultados. A independência das funções de pertinência em relação às configurações específicas do ambiente foi verificada através de análises de sensibilidade, que confirmaram que as variações nos resultados eram atribuíveis às mudanças intencionais nos parâmetros do controlador fuzzy, e não a fatores externos não controlados.

O procedimento detalhado dos experimentos está descrito no **Algoritmo 1**.

Algorithm 1 Experimento da execução das *Membership Functions*

Abrir o Docker *Desktop*
repeat ▷ Repetir 15 vezes
    Executar o comando no terminal pra criar imagem e executar o *container* do RabbitMQ no docker
    Executar o *Publish*,
    Executar o *Subscribe*
until

O **Algoritmo 2** representa o passo-a-passo de coleta da média de mensagens recebidas.

Algorithm 2 Experimento da execução das *Membership Functions*

Receber mensagens do *Publish*
repeat ▷ Repetir 300 vezes
    A cada 15 segundos medir a média de mensagens recebidas por segundo.
    Salvar a média de mensagens no arquivo txt.
until
repeat ▷ Repetir 10 vezes
    A cada 30 amostras mudar o valor do objetivo de mensagens que quer receber do *broker*.
until

4.3 Resultados e Discussão

Nesta seção serão discutidos os resultados das implementações mencionadas na **Seção 3** e dos experimentos definidos nas **Seções 4.1 e 4.2**. Serão apresentados o melhor e o pior controlador para objetivos fixos e variáveis. Adicionalmente, serão destacados os melhores controladores considerando os fatores dinâmicos do experimento, como a variação do objetivo de mensagens, o NRMSE e a quantidade de amostras que ficaram 20% abaixo ou acima do objetivo estabelecido.

A **Figura 6** mostra o pior controlador para um objetivo fixo.

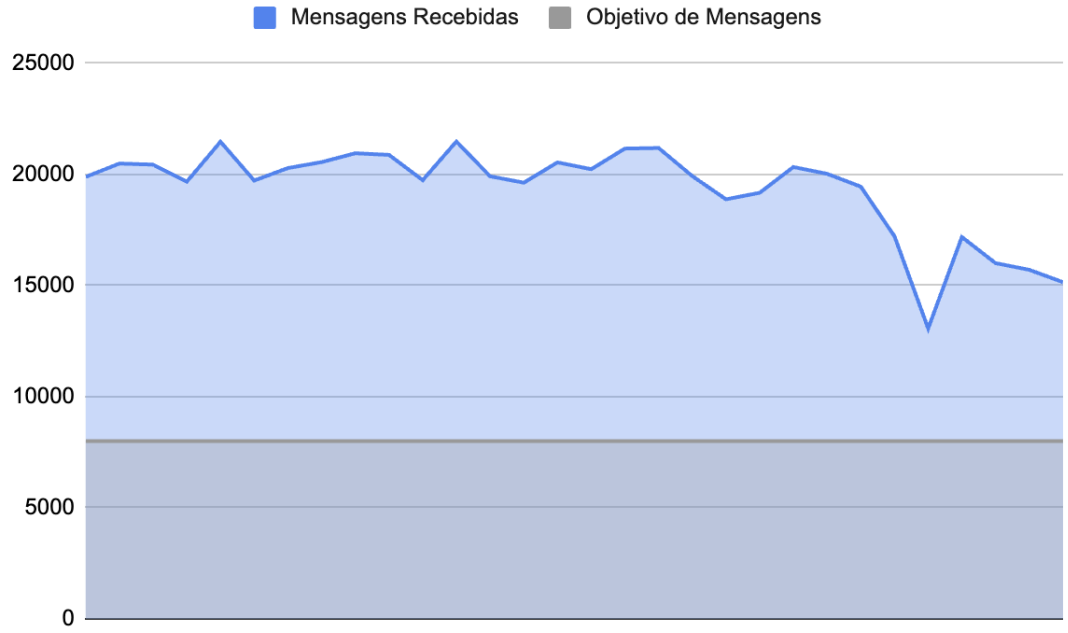

Figura 6: Resultado da Triangular Membership Fuction

O resultado da **Figura 6** refere-se a experimentos feitos com a *Triangular Membership Function* com o conjunto *Fuzzy* de sete regras e um objetivo de oito mil mensagens. Os detalhes dos resultados estão na **Tabela 1**.

$\sum_{i=1}^{n} i^2$	**Amostra**	**Max**	**Min**	**Diferença**	**RMSE**	**NRMSE**	**Objetivo**	**Goal Range**
3.983.602.083	30	21.482	13.067	8.415	11.523	1,3694	8.000	0,00%

Tabela 1: Detalhes dos resultados do experimento com Triangular Membership Function

A **Figura 7** mostra o "desempenho" geral do controlador que teve o pior "desempenho" para um objetivo fixo.

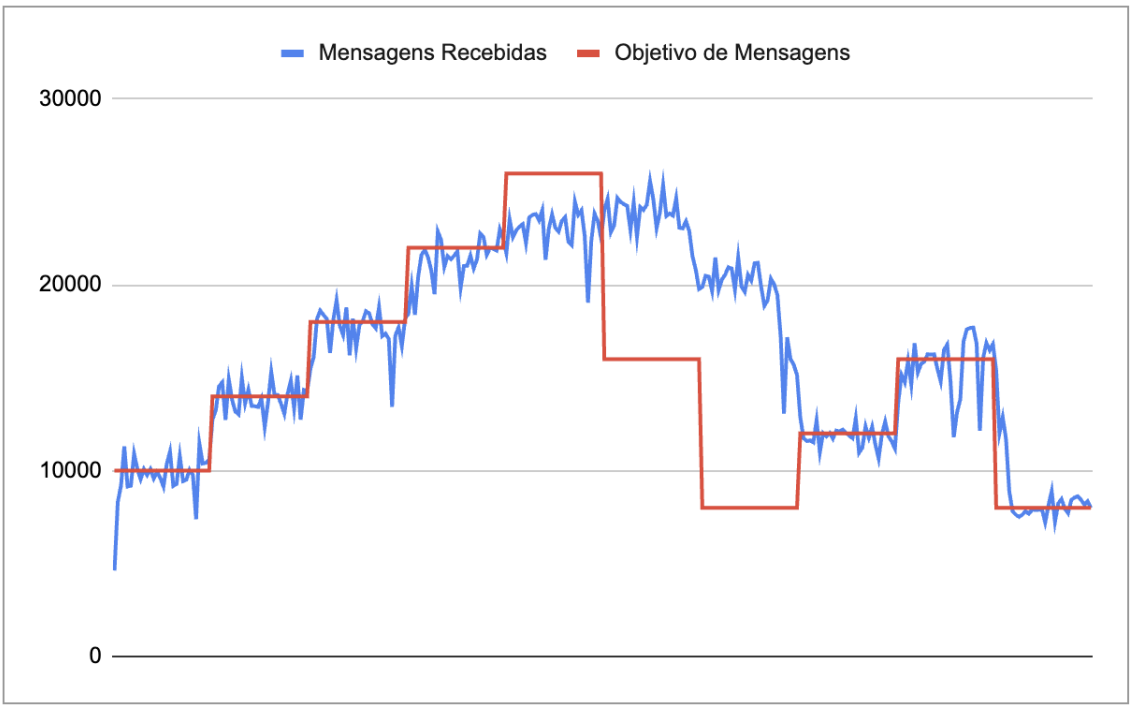

Figura 7: Gráfico com os resultado geral do controlador Triangular com pior resultado individual.

A **Tabela 2** mostra os detalhes do "desempenho" geral do controlador que teve o pior "desempenho" para um objetivo fixo.

$\sum_{i=1}^{n} i^2$	**Amostra**	**Max**	**Min**	**Diferença**	**RMSE**	**NRMSE**	**Objetivo**	**Goal Range**
50.887.598	30	11.461	4.626	6.835	1.302	0,1905	10.000	93,33%
19.042.485	30	15.327	12.348	2.979	797	0,2675	14.000	100,00%
47.331.172	30	19.186	13.431	5.755	1.256	0,2182	18.000	96,67%
53.875.883	30	22.991	18.388	4.603	1.340	0,2911	22.000	100,00%
302.995.471	30	24.475	19.043	5.432	3.178	0,5850	26.000	96,67%
1.778.386.046	30	25.597	19.775	5.822	7.699	1,3224	16.000	0,00%
3.983.602.083	30	21.482	13.067	8.415	11.523	1,3694	8.000	0,00%
9.228.778	30	12.869	10.653	2.216	555	0,2503	12.000	100,00%
72.010.512	30	17.701	11.799	5.902	1.549	0,2625	16.000	93,33%
113.258.600	30	15.383	7.212	8.171	1.943	0,2378	8.000	86,67%

Tabela 2: Detalhes dos resultados geral do controlador Triangular com pior resultado individual.

Média RMSE	Média NRMSE	Média Goal Range
3.114,29	0,4995	76,67%

Tabela 3: Média dos resultados da tabela 2.

A **Figura 8** mostra o "desempenho" do melhor controlador para um objetivo fixo.

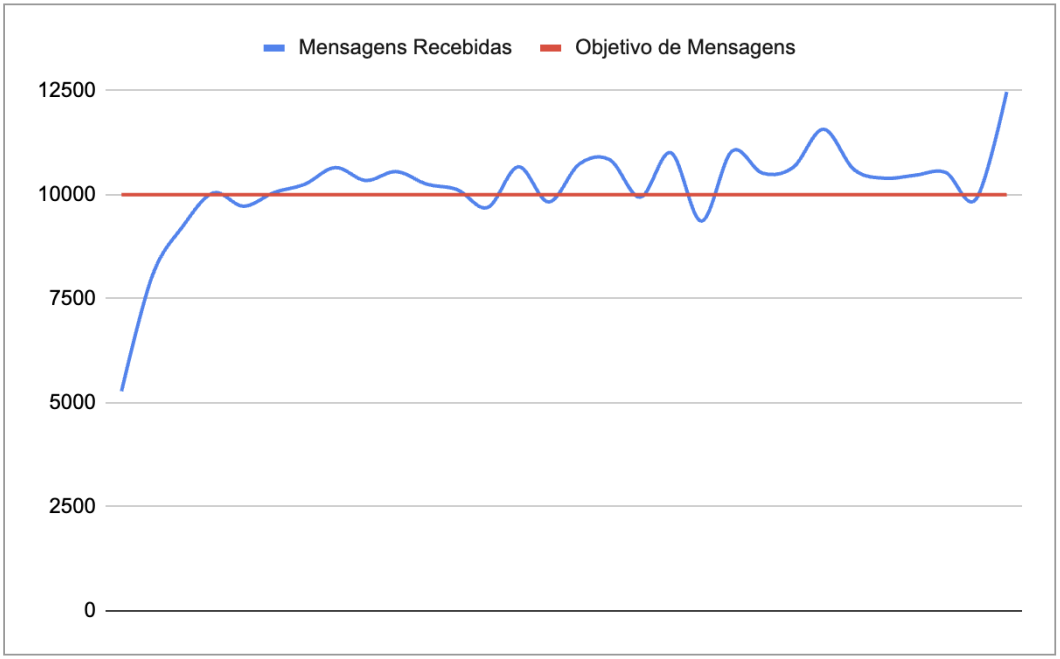

Figura 8: Gráfico do resultado do controlador *Gaussian* com melhor resultado individual.

O resultado do gráfico da **Figura 8** refere-se a experimentos feitos com a *Gaussian Membership Function* com o conjunto *Fuzzy* de onze regras e um objetivo de dez mil mensagens. Os detalhes dos resultados estão na **Tabela 4**.

$\sum_{i=1}^{n} i^2$	Amostra	Max	Min	Diferença	RMSE	NRMSE	Objetivo	Goal Range
42.509.296	30	12.474	5.279	7.195	1.190	0,1654	10.000	93,33%

Tabela 4: Detalhes do resultado do controlador Gaussian com melhor resultado individual.

A **Figura 9** mostra o "desempenho" geral do controlador que teve o melhor "desempenho" para um objetivo fixo.

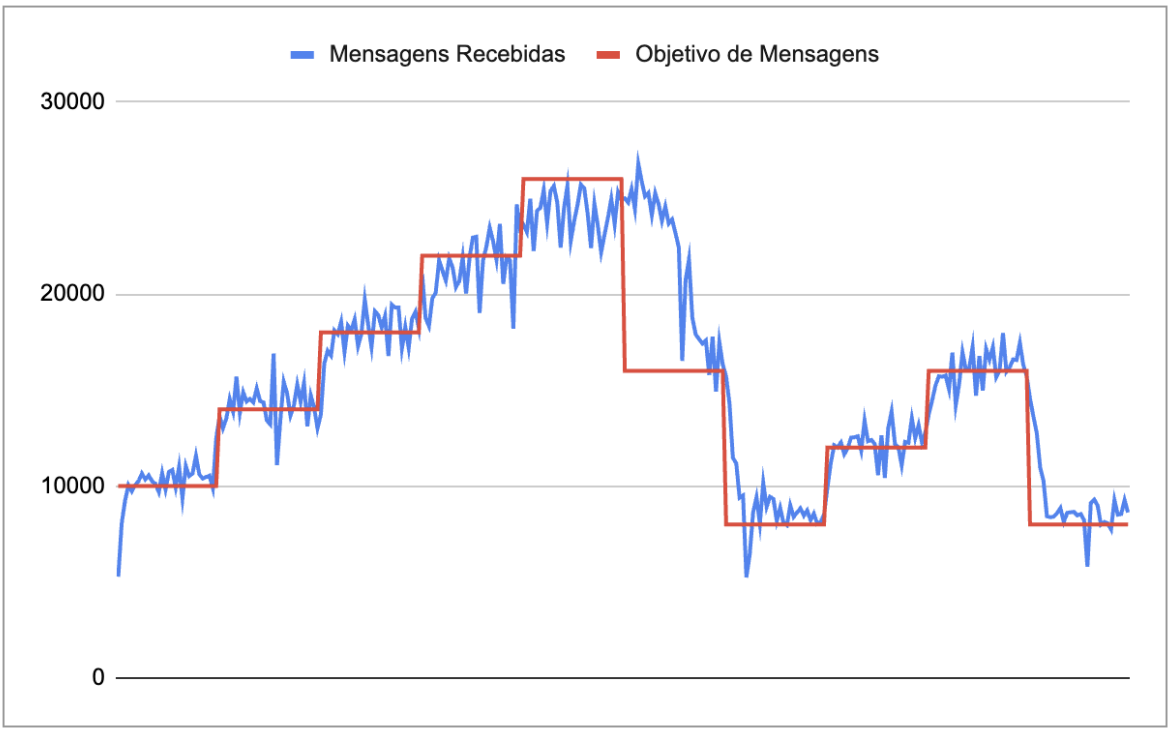

Figura 9: Gráfico com os resultado geral do controlador *Gaussian* com melhor resultado individual.

A **Tabela 5** mostra os detalhes do "desempenho" geral do controlador que teve o melhor "desempenho" para um objetivo fixo.

$\sum_{i=1}^{n} i^2$	**Amostra**	**Max**	**Min**	**Diferença**	**RMSE**	**NRMSE**	**Objetivo**	**Goal Range**
42.509.296	30	12.474	5.279	7.195	1.190	0,1654	10.000	93,33%
34.451.799	30	16.905	11.093	5.811	1.072	0,1844	14.000	93,33%
42.426.292	30	19.721	13.738	5.983	1.189	0,1988	18.000	96,67%
86.906.644	30	24.674	18.194	6.480	1.702	0,2626	22.000	100,00%
133.075.222	30	25.703	22.182	3.521	2.106	0,5982	26.000	100,00%
1.358.953.200	30	26.795	14.924	11.872	6.730	0,5669	16.000	36,67%
152.066.212	30	15.721	5.242	10.479	2.251	0,2149	8.000	80,00%
23.648.020	30	13.881	9.921	3.959	888	0,2242	12.000	100,00%
29.851.433	30	17.966	13.735	4.230	998	0,2358	16.000	100,00%
127.563.844	30	14.561	5.802	8.759	2.062	0,2354	8.000	80,00%

Tabela 5: Detalhes dos resultados geral do controlador *Gaussian* com melhor resultado individual.

Média RMSE	Média NRMSE	Média Goal Range
2.018,86	0,2887	88,00%

Tabela 6: Média dos resultados da tabela 5.

A **Figura 10** mostra o controlador que teve o pior resultado para o objetivo dinâmico.

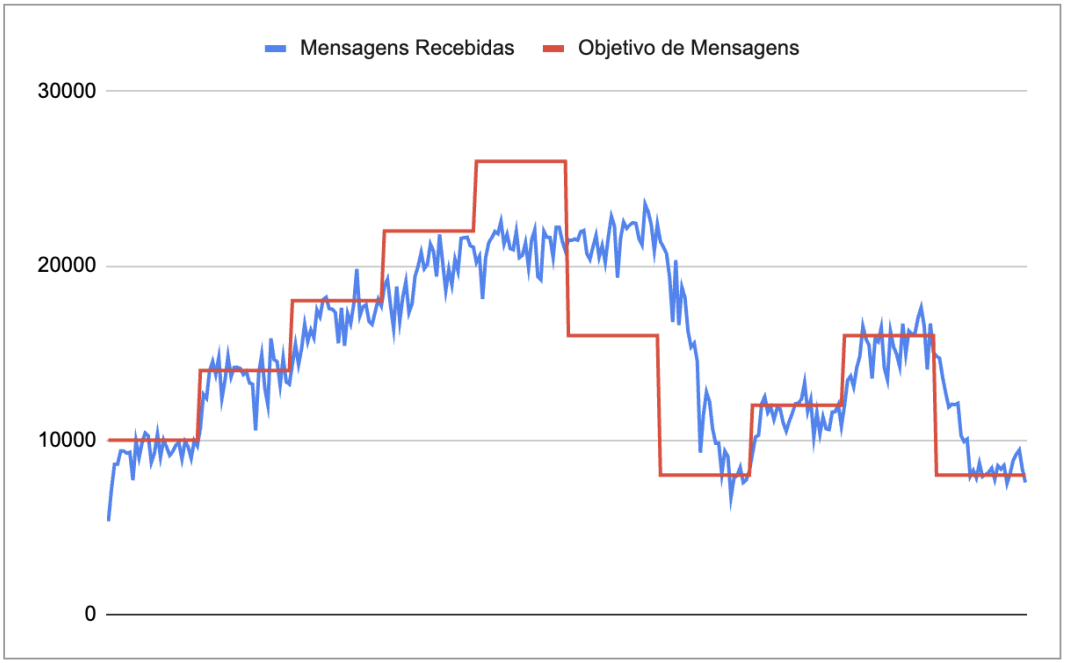

Figura 10: Gráfico dos resultados geral do controlador *Bell* com pior resultado dinâmico.

O resultado do gráfico da **Figura 10** refere-se a experimentos feitos com a *Bell Membership Function* com o conjunto *Fuzzy* de cinco regras e o objetivo de mensagens dinâmico. O detalhes dos resultados estão na **Tabela 7**.

$\sum_{i=1}^{n} i^2$	Amostra	Max	Min	Diferença	RMSE	NRMSE	Objetivo	Goal Range
48.844.553	30	10.410	5.347	5.063	1.276	0,2520	10.000	90,00%
45.074.444	30	15.842	10.568	5.274	1.226	0,2324	14.000	93,33%
79.981.036	30	19.826	14.345	5.481	1.633	0,2979	18.000	93,33%
235.135.889	30	21.800	16.405	5.395	2.800	0,5189	22.000	90,00%
754.823.546	30	22.543	18.104	4.439	5.016	1,1300	26.000	66,67%
992.670.078	30	23.536	19.321	4.214	5.752	1,3650	16.000	0,00%
1.455.988.866	30	21.391	6.734	14.658	6.967	0,4753	8.000	36,67%
33.320.769	30	13.305	9.230	4.074	1.054	0,2587	12.000	96,67%
73.980.380	30	17.607	11.973	5.633	1.570	0,2788	16.000	96,67%
229.484.587	30	14.821	7.519	7.302	2.766	0,3788	8.000	63,33%

Tabela 7: Detalhes dos resultados geral do controlador *Bell* com pior resultado para o objetivo dinâmico.

Média RMSE	Média NRMSE	Média Goal Range
3.005,91	0,5188	72,67%

Tabela 8: Média dos resultados da tabela 7.

A **Figura 11** mostra o controlador que teve o melhor resultado para o objetivo dinâmico.

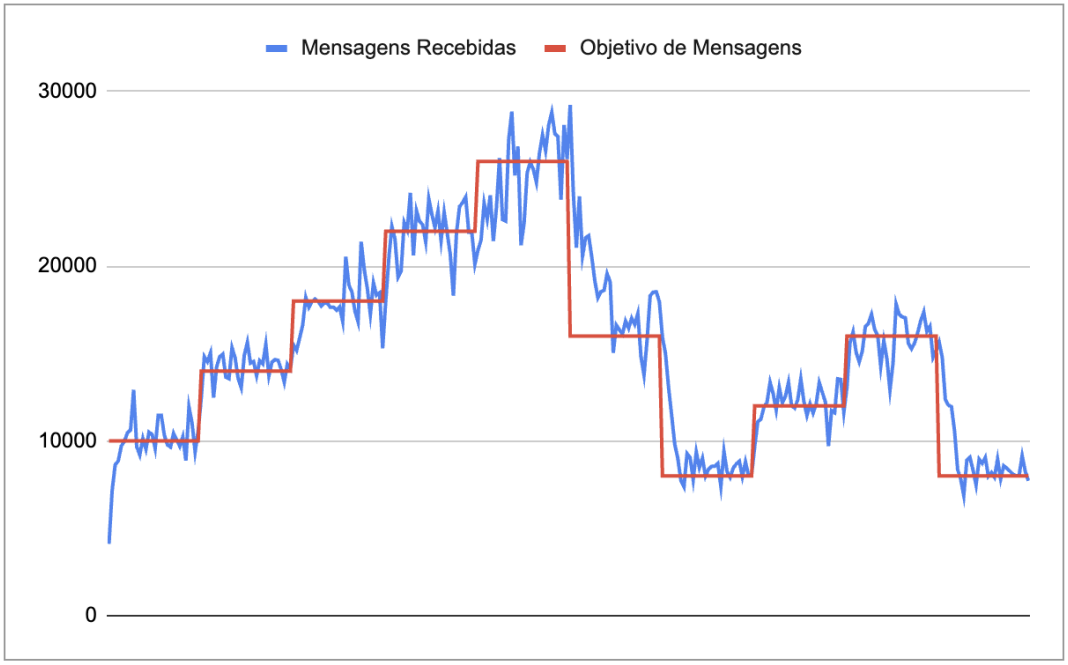

Figura 11: Gráfico dos resultados geral do controlador *Gaussian* com melhor resultado dinâmico.

O resultado do gráfico da **Figura 11** refere-se a experimentos feitos com a *Gaussian Membership Function* com o conjunto *Fuzzy* de nove regras e o objetivo de mensagens dinâmico. O detalhes dos resultados estão na **Tabela 9**.

$\sum_{i=1}^{n} i^2$	**Amostra**	**Max**	**Min**	**Diferença**	**RMSE**	**NRMSE**	**Objetivo**	**Goal Range**
68.610.298	30	12.926	4.110	8.816	1.512	0,1715	10.000	90,00%
20.809.814	30	15.644	12.480	3.164	833	0,2632	14.000	100,00%
56.926.876	30	21.403	15.174	6.229	1.378	0,2212	18.000	100,00%
74.547.528	30	24.199	18.011	6.187	1.576	0,2548	22.000	100,00%
194.699.431	30	28.849	20.928	7.921	2.548	0,3216	26.000	100,00%
522.905.706	30	29.233	13.769	15.464	4.175	0,2700	16.000	70,00%
165.331.939	30	15.933	7.383	8.550	2.348	0,2746	8.000	83,33%
28.426.138	30	13.555	9.670	3.884	973	0,2506	12.000	100,00%
45.105.431	30	17.832	12.913	4.919	1.226	0,2493	16.000	100,00%
171.792.133	30	15.649	6.879	8.770	2.393	0,2729	8.000	80,00%

Tabela 9: Detalhes dos resultados geral do controlador *Gaussian* com melhor resultado para o objetivo dinâmico.

Média RMSE	**Média NRMSE**	**Média Goal Range**
1.896,17	0,2550	92,33%

Tabela 10: Média dos resultados da tabela 9.

A **Tabela 11** mostra os detalhes do "desempenho" de todos os controladores experimentados.

Controlador	**RMSE**	**NRMSE**	**Goal Range**
C1	2.977,33	0,3578	77,33%
C2	2.261,54	0,2592	83,33%
C3	3.114,29	0,4995	76,67%
C4	2.731,48	0,3249	75,67%
C5	2.852,62	0,4074	75,33%
C6	2.081,59	0,3331	86,00%
C7	3.010,18	0,4375	71,00%
C8	2.251,33	0,2954	83,00%
C9	**1.896,17**	**0,2550**	**92,33%**
C10	2.018,86	0,2887	88,00%
C11	2.094,87	0,2995	86,33%
C12	**3.005,91**	**0,5188**	**72,67%**
C13	2.469,00	0,3583	81,67%
C14	3.145,61	0,3219	78,33%
C15	2.974,24	0,4312	79,00%

Tabela 11: Resultado dos quinze controladores do experimento.

Os dados da **Tabela 11** demonstram os resultados dos quinze controladores *fuzzy* experimentados. No trabalho com dados preditivos, utilizou-se o RMSE como método estatístico, frequentemente empregado para avaliar modelos. Este método é interpretado como a distância média dos resíduos até o zero (Kuhn; Johnson, 2013). A versão normalizada desse método, o NRMSE, foi empregada para normalizar o erro conforme a amplitude dos dados e comparar o "desempenho" dos modelos para cada controlador. Adicionalmente, para uma avaliação mais detalhada, definiu-se o *Goal Range*, que avalia a porcentagem de amostras que estão 20% acima ou abaixo do objetivo estabelecido, como tolerância às variações. Conforme demonstrado, o controlador C9 alcançou o melhor NRMSE, com um valor próximo de zero, sendo 0,2550, e um *Goal Range* de 92,33%, indicando níveis aceitáveis a moderados no experimento. Os resultados para objetivos fixos, apresentados na **Figura 8** e na **Tabela 4**, mostraram um NRMSE

de 0,1654 e um *Goal Range* de 93,33%, o que reflete um bom "desempenho" do modelo.

5 Conclusão e Trabalhos Futuros

Este capítulo apresenta a conclusão do presente trabalho, limitações observadas e sugestões trabalhos futuros.

5.1 Conclusão

Com o crescente avanço dos sistemas distribuídos e a necessidade de ajustes na comunicação entre eles para garantir disponibilidade, controle da ociosidade, latência e interoperabilidade, torna-se crucial a utilização de ferramentas como os *brokers*. Estas ferramentas permitem ajustes na comunicação com o objetivo de minimizar a intervenção humana. Neste contexto, o presente trabalho propôs o uso de controladores *fuzzy* para ajustar a quantidade de mensagens consumidas do RabbitMQ pelos consumidores, visando um melhor gerenciamento das mensagens e a prevenção da ociosidade.

Para este fim, foram implementadas três *Membership Functions* — *Triangular*, *Gaussian* e *Bell* — para ajustar o *Prefetch value* do RabbitMQ, baseando-se na taxa de erro calculada pela diferença entre o objetivo de mensagens estabelecido e a quantidade de mensagens efetivamente recebidas pelo consumidor. Para alcançar o objetivo, regras foram acrescentadas aos conjuntos *fuzzy*, iniciando com três regras e expandindo até onze para cada *Membership Function*. Na análise dos resultados, 46% dos controladores alcançaram um *Goal Range* acima de 80%, e 33,33% dos controladores apresentaram um NRMSE abaixo de 0,3, indicando resultados moderados. Quanto aos valores de NRMSE, 66,66% dos controladores demonstraram baixa eficácia na sugestão de ajustes; contudo, 100% dos controladores atingiram um Goal Range acima de 70%, representando um resultado moderado. O melhor resultado foi obtido para um objetivo fixo de dez mil mensagens, com NRMSE de 0,1654 e *Goal Range* de 93,33%, considerado bom. Para um objetivo dinâmico, o melhor NRMSE foi de 0,2550 com *Goal Range* de 92,33%, o que é considerado um resultado moderado. É interessante notar que as taxas de NRMSE apresentaram grande oscilação em mudanças bruscas de objetivo, o que sugere a necessidade de análises mais profundas para ajustes.

Em vista disso, o uso dos controladores *fuzzy* conseguiu atingir o objetivo desejado com a adição de mais regras aos conjuntos *fuzzy*, e os resultados do trabalho se mostraram satisfatórios.

5.2 Limitações

Os experimentos com os controladores foram realizados sem mensurar variáveis como a quantidade de usuários na mesma rede e se o *laptop* estava conectado à energia ou funcionando na bateria. Nos experimentos analisados, os testes foram conduzidos tanto com o *laptop* conectado à energia quanto usando a bateria, em redes distintas, fatores que podem ter influenciado os resultados obtidos. Outro fator limitante deste estudo incluem a dependência das configurações específicas do ambiente de teste e a necessidade de validação extensiva para generalização. As ameaças à validade dos resultados estão principalmente na possibilidade de sobreajuste dos controladores aos cenários de teste específicos.

5.3 Trabalhos Futuros

Para trabalhos futuros, algumas alterações poderão ser consideradas com o objetivo de obter um "desempenho" superior, como o aumento do número de amostras para avaliar se o tempo e a quantidade de amostras interferem nos resultados. Entre outras possíveis

mudanças, destaca-se a alteração do método de Defuzzificação, de *Centroide* , atualmente utilizado, para *Bissectriz* ou Máximo Mais à Esquerda e Máximo Mais à Direita. Outra possibilidade seria implementar um *Neuro-fuzzy controller*, que incorpora uma camada de Design de Rede Neural, utilizada para treinar a rede neural com os dados coletados das entradas e saídas dos controladores, visando aumentar a precisão no ajuste do RabbitMQ, bem como expandir os testes para múltiplos ambientes de produção com variáveis de carga e comportamento de rede distintas.

Referências

Kuhn, M., & Johnson, K. (2013). **Applied Predictive Modeling.** Springer.

Jain, R. (1991). Section 3.2. In Chapter 3, **Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurements, Simulation, and Modeling.** Wiley Computer Publishing.

IBM. (s.d.). **Clusters**. Recuperado de https://www.ibm.com/docs/en /was-nd/8.5.5?topic=servers-introduction-clusters

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). **Distributed Systems: Concepts and Design (5th ed.).** Pearson.

Tanenbaum, A. S., & Van Steen, M. (2007). **Distributed Systems: Principles and Paradigms (2nd ed.).** Prentice Hall.

Kosko, B., & Isaka, S. (1993). **Fuzzy Logic.** Scientific American, 269(1), 76-81.

Zimmermann, H.-J. (2001). **Fuzzy Set Theory—and Its Applications (4th ed.)**. Springer Science+Business Media.

Grafana Labs. (s.d.) **Grafana and Prometheus**. Recuperado de https://grafana.com/docs/grafana /latest/getting-started/get-started-grafana-prometheus/

Microsoft. (s.d.). **Publisher-subscriber pattern**. Recuperado de https://learn.microsoft.com/pt-br/azure/architecture/patterns/publisher-subscriber

Videla, A., & Williams, J. J. W. (2012). **RabbitMQ in Action: Distributed Messaging for Everyone.** Manning Publications.

Roy, G. M. (2018). **RabbitMQ in Depth.** Manning Publications.

RabbitMQ. (s.d.). **Tutorial Three: Publish/Subscribe in Go.** Recuperado de https://www.rabbitmq.com/tutorials/tutorial-three-go.html