CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS DIVINÓPOLIS

Lucas Martins Soares

BALANCEAMENTO DE CARGA EM PROCESSAMENTO DE FLUXO DE DADOS DISTRIBUÍDOS:

Uma Solução de Fila Única para Cenários de Contrapressão

Divinópolis
2023

LUCAS MARTINS SOARES

BALANCEAMENTO DE CARGA EM PROCESSAMENTO DE FLUXO DE DADOS DISTRIBUÍDOS:

Uma Solução de Fila Única para Cenários de Contrapressão

Trabalho de Conclusão de Curso apresentado no curso de Graduação em Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Me. Michel Pires da Silva

Coorientador: Prof. Dr. André Luiz Maravilha Silva

DIVINÓPOLIS

2023

LUCAS MARTINS SOARES

BALANCEAMENTO DE CARGA EM PROCESSAMENTO DE FLUXO DE DADOS DISTRIBUÍDOS:

Uma Solução de Fila Única para Cenários de Contrapressão

Trabalho de Conclusão de Curso apresentado no curso de Graduação em Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Aprovado em 11 de dezembro de 2023.

gov.br Documento assinado digitalmente
MICHEL PIRES DA SILVA
Data: 19/02/2024 17:22:45-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Michel Pires da Silva

CEFET-MG Campus Divinópolis

gov.br Documento assinado digitalmente
ANDRE LUIZ MARAVILHA SILVA
Data: 19/02/2024 19:35:33-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. André Luiz Maravilha Silva

CEFET-MG Campus Divinópolis

gov.br Documento assinado digitalmente
THABATTA MOREIRA ALVES DE ARAUJO
Data: 19/02/2024 19:09:34-0300
Verifique em https://validar.iti.gov.br

Prof. Me. Thabatta Moreira Alves de Araújo

CEFET-MG Campus Divinópolis

Dedico à minha família que me apoiou durante toda a minha trajetória.

AGRADECIMENTOS

Agradeço primeiramente aos professores do Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG) que me inspiraram durante o curso e me ajudaram a explorar o meu potencial.

Ao professor Ms. Michel Pires da Silva por me orientar e por me dar a liberadade de explorar novas ideias e soluções para o problema proposto.

Aos meus pais, Rildo Martins Ferreira e Sandra Vaz Soares Martins, e ao meu irmão, Vitor Martins Soares, por estarem sempre ao meu lado e permitirem que eu me dedicasse aos estudos.

“O ontem é história, o amanhã é um mistério, mas o hoje é uma dádiva. É por isso que se chama presente.”

Mestre Oogway

RESUMO

Este trabalho tem como objetivo melhorar a eficiencia do balanceamento de carga em sistemas de processamento de fluxo de dados em cenários de contrapressão. Para isso foi realizado um estudo sobre o mecanismo de balanceamento de carga do *Apache Flink* e, a partir disso, foi proposta uma nova abordagem de balanceamento de carga. No *Flink* o usuário precisa definir manualmente o paralelismo para cada estágio do *dataflow* e em decorrência disso, menos possibilidades de otimização podem ser aplicadas. Além disso, para manter essa restrição em cenários de sobrecarga local, o *Flink* implementa um mecanismo para distribuir a fila de mensagens atrasadas conhecido como contrapressão. Este mecanismo consiste em mensagens que o nó sobrecarregado dispara para os antecessores notificando-os para diminuir a sua taxa de produção de mensagens, o que impede o sistema de utilizar 100% de sua capacidade computacional. Como solução, é apresentada uma nova abordagem de balanceamento de carga que consiste em delegar a função de balanceamento para a fila de eventos do *Tokio*, que, por sua vez, envia as operações para serem processadas em um pool de threads. Durante os experimentos verificou-se uma redução de aproximadamente 50% na latência média para rajadas de até 1.000 mensagens e um aumento de aproximadamente 80% na taxa de transferência para o mesmo cenário. Porém, o sistema não apresentou um bom desempenho para picos com 10.000 mensagens, mais que dobrando a latência e reduzindo a taxa de transferência em uma média de 15%. Com isso, conclui-se que a abordagem proposta é promissora, mas ainda carece de novas pesquisas.

Palavras-chave: Balanceamento de Carga; Processamento de Fluxo de Dados; Apache Flink; Sobrecarga local; Contrapressão.

ABSTRACT

This work aims to improve the efficiency of load balancing in data stream processing systems in backpressure scenarios. For this, a study on the load balancing mechanism of *Apache Flink* was carried out, and from this, a new load balancing approach was proposed. In *Flink*, the user needs to manually define the parallelism for each stage of the *dataflow* and as a result, fewer optimization possibilities can be applied. Furthermore, to maintain this restriction in local overload scenarios, *Flink* implements a mechanism for distributing the queue of delayed messages known as backpressure. This mechanism consists of messages that the overloaded node fires to its predecessors notifying them to decrease their message production rate, which prevents the system from using 100% of its computational capacity. As a solution, a new load balancing approach is presented, which consists of delegating the balancing function to the *Tokio* event queue, which, in turn, sends operations to be processed in a thread pool. During the experiments, a reduction of approximately 50% in average latency for bursts of up to 1,000 messages and an increase of approximately 80% in throughput for the same scenario were observed. However, the system did not perform well for spikes with 10,000 messages, more than doubling the latency and reducing the throughput by an average of 15%. With this, it is concluded that the proposed approach is promising, but still requires further research.

Keywords: Load Balancing; Data Stream Processing; Apache Flink; Local Overload; Backpressure.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Interface de Programação de Aplicativos, do inglês *Application Programming Interface*
CEFET-MG	Centro Federal de Educação Tecnológica de Minas Gerais
CPU	Unidade Central de Processamento, do inglês *Central Processing Unit*
DAG	Grafo Acíclico Dirigido, do inglês *Directed Acyclic Graph*
GC	Coletor de Lixo, do inglês *Garbage Collector*
VM	Máquina Virtual, do inglês *Virtual Machine*
GFS	Sistema de Arquivos Google, do inglês *Google Distributed File System*
HDFS	Sistema de Arquivos Distribuídos Hadoop, do inglês *Hadoop Distributed File System*
IoT	Internet das Coisas, do inglês *Internet of Things*
JSON	Notação de Objeto JavaScript, do inglês *JavaScript Object Notation*
NLP	Processamento de Linguagem Natural, do inglês *Natural Language Processing*
RAM	Memória de Acesso Aleatório, do inglês *Random-access memory*
RDD	Conjunto de Dados Distribuídos Resilientes, do inglês *Resilient Distributed Dataset*
RSP	Partição de Amostra Aleatória, do inglês *Random Sample Partition*
SO	Sistema Operacional
TI	Tecnologia da Informação
XML	Linguagem de Marcação Extensível, do inglês *eXtensible Markup Language*
RPC	Chamada de Procedimento Remoto, do inglês *Remote Procedure Call*

SUMÁRIO

1 INTRODUÇÃO

No cenário tecnológico atual, os dados de usuários e de sensores e dispositivos relacionados com Internet das Coisas, do inglês *Internet of Things* (IoT), são vistos como um ativo de alto valor agregado (Günther *et al.*, 2017). A partir deles, as empresas conseguem extrair informações valiosas sobre seus clientes, podendo, assim, atuar de forma mais eficiente e precisa para aumentar a sua lucratividade. Entretanto, devido ao enorme volume de informação gerada continuamente, surgem novos desafios para lidar com esses dados. Esses desafios e características estão relacionados com o conceito de *Big Data* (Oussous *et al.*, 2018).

Nesse contexto, um único dispositivo é incapaz de lidar com essa quantidade de dados individualmente, fazendo-se necessária a utilização de *clusters*, grupos de computadores sincronizados e coordenados para a realização de uma tarefa. No entanto, coordenar e manter aplicações distribuídas aumenta muito a complexidade de desenvolvimento da aplicação (Coulouris *et al.*, 2011). Por esse motivo, surgiram estruturas (*frameworks*) com o objetivo de padronizar e abstrair a lógica de baixo nível necessária para se construir essas aplicações.

Estes *frameworks* podem ser classificados de acordo com sua responsabilidade dentro dessa infraestrutura de *Big Data*: processamento de análise on-line, análise, armazenamento de dados, aquisição e transporte (Sun *et al.*, 2023). Entre as soluções de análise, destacam-se: o *MapReduce* (Dean; Ghemawat, 2008), o *Spark* (Zaharia *et al.*, 2012) e o *Flink* (Carbone *et al.*, 2015).

As soluções desenvolvidas por cada um dos *frameworks* podem ser agrupadas em processamento de lote ou processamento de fluxo. O processamento em lote é mais utilizado quando os dados já estão em posse no momento do processamento, enquanto o processamento em fluxo é ideal para ser utilizado quando os dados chegam em tempo real.

O *MapReduce*, desenvolvido pela Google em 2004 (Dean; Ghemawat, 2008), foi precursor do processamento em lote. Essa aplicação processou o conteúdo de inumeras páginas *Web* e melhorou a qualidade do serviço de recomendação de links por meio da utilização do algoritmo *PageRank*. No entanto, esse mecanismo utiliza o sistema de arquivo distribuído como entrada e saída de dados, o que atrasa o tempo de processamento.

Para resolver esse problema, em 2014, surgiu o *Apache Spark* (Zaharia *et al.*, 2012), que utiliza uma arquitetura de armazenamento em Memória de Acesso Aleatório, do inglês *Random-access memory* (RAM), chamada Conjunto de Dados Distribuídos Resilientes, do inglês *Resilient Distributed Dataset* (RDD), reduzindo significativamente o tempo de processamento dos dados.

Contudo, tanto o *MapReduce* quanto o *Spark* foram desenvolvidos para processar dados em lote e finitos (Javed; Lu; Panda, 2017). Nessa perspectiva, a fim de otimizar o processamento de dados em fluxo contínuo, surgiu o *Apache Flink*, que processa os dados à medida que chegam, com baixa latência, e armazena o seu estado internamente em RAM, tornando o sistema ideal para processamento em tempo real. Esse *framework* foi projetado com a filosofia de que muitas classes de aplicativos de processamento de dados, incluindo processamento de dados históricos e algoritmos iterativos (i.e. aprendizado de máquina, análise de gráficos), podem ser representados como um *dataflow* (Carbone *et al.*, 2015). Este *dataflow* é representado na forma de um Grafo Acíclico Dirigido, do inglês *Directed Acyclic Graph* (DAG), em que cada nó representa uma tarefa, as arestas conectam os nós origem aos nós de destino e cada nó contém um nível de paralelismo associado.

Embora essa abordagem se mostre eficaz na maioria dos casos, ela possui limitações. No cenário ideal, os dados chegam com uma taxa constante menor ou igual à capacidade máxima de processamento do *cluster*. No entanto, devido à natureza imprevisível do fluxo dos dados, pode ocorrer uma sobrecarga, global ou local, na capacidade de processamento da rede. Quando isso acontece, os nós sobrecarregados começam a disparar eventos de contrapressão (i.e. *backpressure*) para os nós anteriores, notificando-os para reduzir a sua taxa de envio de pacotes. Dessa forma, quando um nó recebe essa notificação, ele começa a armazenar os pacotes que deveriam ter sido enviados, aumentando o consumo de RAM e de Disco (Celebi, 2022). Nesse contexto, se o *dataflow* estiver perfeitamente balanceado e a quantidade recebida de pacotes estiver além da capacidade de escoamento do sistema, o *Flink* utilizará 100% do poder de processamento disponível para mitigar esse gargalo o mais rápido possível (i.e. sobrecarga global). Entretanto, muitas vezes os gargalos começam a acontecer antes que o sistema chegue em sua capacidade máxima de processamento (i.e. sobrecarga local), o que pode acontecer devido a uma configuração não ótima da arquitetura do sistema —

dado que o sistema deve cumprir como paralelismo estático definido pelo usuário — ou devido a fatores externos como Coletor de Lixo, do inglês *Garbage Collector* (GC) (Celebi, 2022).

1.1 Objetivos

Por esse motivo, o objetivo desse trabalho consiste em desenvolver um *framework* de processamento de stream de dados que implemente uma nova abordagem de balanceamento de carga a fim de reduzir o impacto da sobrecarga de uma tarefa e otimizar a utilização da Unidade Central de Processamento, do inglês *Central Processing Unit* (CPU).

Esse objetivo principal será buscado por meio dos seguintes objetivos específicos:

a) Identificar lacunas na arquitetura do funcionamento do *Flink*;
b) Propor uma nova arquitetura de balanceamento de carga que neutralize o impacto de um nó sobrecarregado;
c) Desenvolver essa solução;
d) Criar um cenário de teste;
e) Avaliar o desempenho da aplicação neste cenário de teste.

1.2 Estrutura do Texto

Esse texto foi estruturado em 4 capítulos além da Capítulo 1. No Capítulo 2 são apresentados os conceitos teóricos necessários para o entendimento do trablaho. No Capítulo 3 as técnicas e devisões de projeto utilizadas são apresentadas. No Capítulo 4 são apresentados os resultados obtidos e, por fim, no Capítulo 5 são apresentadas as conclusões e trabalhos futuros.

2 FUNDAMENTOS TEÓRICOS

Para construir um *framework* de processamento de *streaming* de dados, é necessário entender as suas características (Seção 2.1), alguns conceitos de desempenho, escalabilidade e seus principais desafios (Seção 2.2). Além disso, é importante conhecer o funcionamento do *frameworks* de processamento de dados mais utilizados atualmente (Seção 2.3).

2.1 Big Data

Aplicações *Big Data* são comumente associadas a 5 características fundamentais, também conhecidas como 5Vs: volume, variedade, velocidade, valor e veracidade (Anuradha *et al.*, 2015).

O Volume se refere à quantidade massiva de dados gerados continuamente. Segundo Duarte (2023), em 2023 serão gerados 120 zettabytes de dados, um aumento de 23.71% em relação a 2022 (97 zettabytes). Esse crescimento é impulsionado pelo aumento do tempo de uso da internet, pelo crescimento dos dispositivos de IoT, pela digitalização de processos e pelo aumento da quantidade de dados gerados por esses dispositivos. Isso apresenta desafios significativos em termos de armazenamento e processamento, exigindo infraestruturas de Tecnologia da Informação (TI) robustas e tecnologias de processamento de dados poderosas, como *Hadoop MapReduce*, Sistema de Arquivos Distribuídos Hadoop, do inglês *Hadoop Distributed File System* (HDFS) (Shvachko *et al.*, 2010), e *Spark*.

A Variedade está relacionada aos diferentes tipos de dados gerados e coletados. Estes podem ser categorizados em três tipos principais: estruturados, semi-estruturados e não estruturados (Giudice *et al.*, 2019). Os dados estruturados possuem formato e modelo predefinidos, sendo facilmente armazenados em bancos de dados tradicionais. Os semi-estruturados contêm *tags* ou outros marcadores para separar elementos semânticos e impor hierarquias de registros e campos, como Linguagem de Marcação Extensível, do inglês *eXtensible Markup Language* (XML), Notação de Objeto JavaScript, do inglês *JavaScript Object Notation* (JSON), e e-mails. Já os dados não estruturados, como dados de texto, mídias sociais, vídeos, áudios, imagens e documentos de texto, não

têm uma estrutura predefinida, o que requer novas abordagens para armazenar e processar esses dados, levando ao surgimento de ferramentas como NoSQL e Processamento de Linguagem Natural, do inglês *Natural Language Processing* (NLP).

A Velocidade representa a taxa na qual os dados são gerados, processados e analisados. Em muitos casos, a utilidade dos dados diminui com o tempo. Por exemplo, uma empresa de comércio eletrônico que deseja recomendar produtos aos usuários com base em seu comportamento de navegação precisa processar esses dados quase em tempo real. Nessa perspectiva, a tomada de decisões nesse modelo tem um impacto direto na viabilidade de um negócio, como na identificação de transações fraudulentas em um sistema bancário (Cabrera *et al.*, 2020). Portanto, um sistema de *Big Data* deve ser capaz de se adaptar a variações abruptas no fluxo de dados para garantir a estabilidade do sistema, o que pode ser alcançado através de sistemas de mensageria e de processamento de *stream* em tempo real.

A Veracidade se relaciona à qualidade, precisão e confiabilidade dos dados coletados. Com a quantidade massiva de informações sendo geradas a partir de várias fontes, garantir sua veracidade pode ser um desafio significativo. Dados imprecisos, incompletos ou enganosos podem levar a análises errôneas e a tomadas de decisões equivocadas. Além disso, seus métodos de coleta devem ser consistentes para evitar a invalidação devido a mudanças na forma de coleta (Thudumu *et al.*, 2020).

O Valor diz respeito à capacidade de obter *insights* a partir dos dados brutos. Faroukhi *et al.* (2020) define a cadeia de valor de *Big Data* em 4 etapas: geração, coleta, análise e intercâmbio. Inicialmente, diversas informações são geradas em vários dispositivos. Em seguida, esses dados são enviados para um servidor centralizado que recebe, trata e armazena essas informações. Posteriormente, os *insights* são obtidos através de técnicas de análise e visualização de dados, que são utilizadas para fundamentar decisões operacionais e estratégicas. Por fim, os dados tratados e os *insights* são disponibilizados interna ou externamente, permitindo a extração de valor a partir dos dados.

Dadas estas características, é necessário conhecer alguns conceitos relacionados a desempenho e escalabilidade.

2.2 Desempenho e Escalabilidade

Nessa seção é abordado alguns conceitos essenciais para otimizar o desempenho computacional e aumentar sua escalabilidade, abordando a programação multitarefa (Subseção 2.2.1), escolha de linguagens de programação (Subseção 2.2.2) e sistemas distribuídos (Subseção 2.2.3).

2.2.1 Programação Multitarefa

Para se extrair o máximo do desempenho computacional do *hardware*, dois conceitos se tornam fundamentais: concorrência e paralelismo (Grossman; Anderson, 2012).

A computação paralela é muito útil quando se deseja utilizar todos os núcleos de uma CPU. Dessa forma, várias tarefas que consumiriam muito poder de processamento podem ser executadas paralelamente, ou seja, ao mesmo tempo. Para que isso seja possível, a linguagem de programação deve permitir que o programador requisite novas *threads* ao Sistema Operacional (SO) e este ficará responsável por escalonar o tempo de CPU para cada uma das *threads* (Tanenbaum; Bos, 2014).

Além disso, é possível executar várias tarefas simultaneamente utilizando apenas um núcleo da CPU por meio da concorrência. Existem duas formas de concorrência: concorrência preemptiva e a concorrência cooperativa. Na concorrência preemptiva, um escalonador é responsável por decidir quanto tempo de CPU um processo tem dentro do mesmo núcleo. Na concorrência cooperativa, por sua vez, uma tarefa deve renunciar o tempo de CPU quando chegar em um ponto definido no código. Isso é útil para se definir operações com alto tempo de espera, como leituras de disco ou requisições à internet (Hoare, 1978).

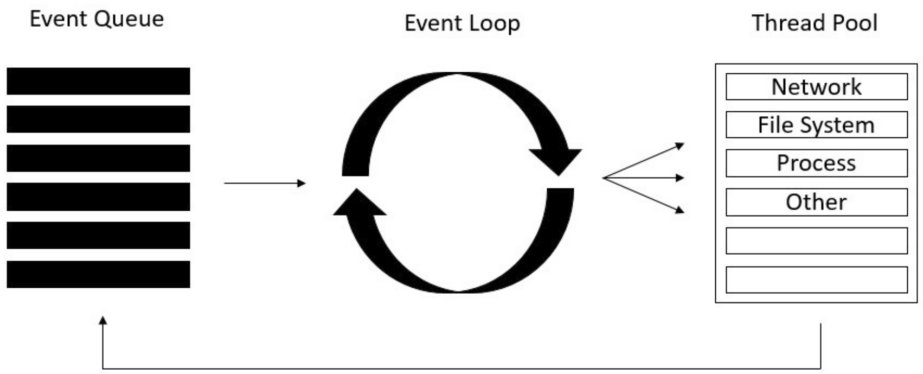
Com isso, para se obter o desempenho máximo de uma CPU, é necessário fazer um uso inteligente desses dois conceitos ao mesmo tempo. Ambos se complementam para se extrair o máximo de recursos disponíveis no *hardware*.

Entretanto, essa forma de programação revela uma nova classe de problemas, como *Data Race* e *Racing Condition*. O *Data Race* acontece quando duas tarefas, tentam acessar o mesmo endereço de memória sem sincronização, quando ao menos um desses

acessos é feito para escrita. Dessa forma, não há uma garantia na consistência do dado lido. Já *Racing Condition* acontece quando há uma sincronização no momento da leitura e escrita de um endereço de memória, porém, essa tarefa como um todo não é sincronizada, levando a problemas relacionados à ordem das operações (Netzer; Miller, 1992).

Para lidar com esses problemas, existe uma abstração chamada modelo de atores. Uma aplicação construída com o modelo de atores representa suas entidades de forma sequencial, como classes no paradigma orientado a objetos, em que um ator deve sempre ser executado em uma única *thread*, mas uma *thread* pode executar um ou mais atores. Dessa forma, o *event loop* garante que as mensagens sejam processadas na ordem de chegada de forma concorrente e paralela (Hewitt; Bishop; Steiger, 1973). O *event loop*, representado na Figura 1, é um algoritmo que executa no *runtime* da aplicação, responsável por coordenar as múltiplas tarefas concorrentes. Esse algoritmo funciona enfileirando todas as mensagens ou eventos a serem executados quando o processador estiver disponível (Network, 2023).

Figura 1 – *Event Loop*



Fonte: Novikov (2023).

Outro problema relacionado ao paralelismo é a criação excessiva de *threads*. A operação de se criar uma *thread* é cara, portanto, deve-se manter controlado o número de *threads* criadas. Para isso, existe o *thread pool*, que funciona definindo um número de *threads* e coordenando a execução das tarefas nessas *threads* pré-alocadas (Ling; Mullen; Lin, 2000).

2.2.2 Linguagens de Programação

Além da programação multitarefa, para obter o desempenho máximo da aplicação, também precisa-se levar em consideração algumas característica na hora de escolher a linguagem de programação, são elas: a forma de tradução, a plataforma alvo, o gerenciamento de memória e sistema de tipos.

Quanto à forma de tradução, uma linguagem pode ser classificada como compilada ou interpretada. Nas linguagens compiladas, antes de ser executado, todo código-fonte é transformado na linguagem de destino ao mesmo tempo. Dessa forma, o compilador consegue realizar várias otimizações para aumentar o desempenho do binário final. Nas linguagens interpretadas, apenas uma linha de código é traduzida por vez. Essa abordagem permite que o desenvolvedor verifique o resultado das suas modificações mais rapidamente, sem que todo o projeto seja recompilado. Porém, isso limita as otimizações que podem ser feitas no código gerado, já que o interpretador não possui todo o contexto necessário para realizar as otimizações (Sebesta, 2015).

Quanto à plataforma alvo, estas podem ser: uma máquina nativa ou uma Máquina Virtual, do inglês *Virtual Machine* (VM). A linguagem de programação *C* é compilada para uma máquina nativa, já o *Java* é compilado para uma VM. Uma VM permite que o desenvolverdor não trate o SO alvo durante o desenvolvimento, diferentemente da linguagem *C* que, embora seja multiplataforma, necessita que macros sejam definidas para customizar o binário final de uma plataforma específica. Todavia, essa facilidade vem com um custo, já que existe uma camada extra de abstração entre o código e o SO (THE. . . , 2023), o que adiciona um *overhead* no tempo de execução da aplicação. Além disso, a VM precisa ser instalada no SO alvo, o que pode ser um problema em sistemas embarcados ou em sistemas que não possuem acesso à Internet.

Outro fator que impacta no desempenho final da aplicação é a forma de gerenciamento de memória. Sempre quando uma memória é alocada no monte, do inglês *heap*, ela viverá para sempre, a menos que seja desalocada explicitamente, diferentemente de memórias alocadas na pilha, do inglês *stack*, que são desalocadas quando se encerra o escopo do bloco. Essas características das variáveis dinâmicas do monte fazem com que elas sejam desalocadas de uma forma desacoplada do bloco. Para isso, existem duas formas de realizar esta desalocação: o gerenciamento manual e o

gerenciamento automático de memória (Sebesta, 2015).

No gerenciamento manual, o desenvolvedor é responsável por definir explicitamente o momento de alocar e desalocar uma variável. Porém, apesar desta ser a abordagem mais performática, ela aumenta muito a complexidade do desenvolvimento da aplicação e torna os códigos suscetíveis a vazamento de memória (Orlovich; Rugina, 2006).

Para resolver esse problema, existem duas abordagens de gerenciamento automático de memória: contagem de referência e GC. Na contagem de referência, sempre que uma variável é duplicada o seu conteúdo é mantido o mesmo, mas a sua referência é duplicada e um contador é incrementado, e quando se encerra o escopo do bloco o contador é decrementado, desalocando-a quando chega em zero. Dessa forma, é possível fazer um gerenciamento automático de memória com um impacto mínimo no tempo de execução da aplicação. Porém, essa abordagem tem um problema, dado que em determinadas situações, a desalocação de uma variável pode depender da desalocação de outra, levando, assim, à dependências cíclicas, impedindo-a de ser desalocada (Sebesta, 2015).

Uma solução para resolver esse problema é o GC, que verifica referenências cíclicas em tempo de execução. Contudo, todo o código deve ser pausado para que o mecanismo conte as referências e verifique suas dependências, o que adiciona uma sobrecarga em tempo de execução consideravelmente maior do que a contagem de referências (Hertz; Berger, 2005).

Por fim, outro fator que pode adicionar um tempo de processamento extra na aplicação final é o sistema de tipos, que pode ser estático ou dinâmico (Pierce, 2002). Em linguagens de programação de tipagem estática, cabe ao desenvolvedor definir os tipos dos cabeçalhos de funções e de todas as variáveis se não houver inferência de tipos. Nessa forma de tipagem, nenhuma instrução extra é adicionada para verificar o tipo em tempo de execução e toda inferência de tipo é feita em tempo de compilação.

Já em linguagem de tipagem dinâmica, a responsabilidade de descobrir os tipos das variáveis é do *runtime* da linguagem. Isso possibilita que uma menor quantidade de código seja escrita e que o tipo da variável mude com o passar do tempo. Essa característica acelera o tempo de desenvolvimento de aplicações simples, mas também dificulta a legibilidade do código por terceiros no futuro e adiciona um *overhead* no tempo de execução da aplicação, já que a linguagem não consegue descobrir esses tipos em

tempo de compilação, Além disso, essa abordagem torna o código mais suscetível a erros de mudança de tipos que não foram previstos (Tucker, 2004).

2.2.3 Sistemas Distribuídos

Devido às características das aplicações *Big Data*, é necessário combinar o poder de processamento de vários computadores conectados (*clusters*), utilizando-se técnicas de sistemas distribuídos. Contudo, coordenar, comunicar e sincronizar esse tipo de aplicação aumenta significativamente a complexidade do seu desenvolvimento, dado que essa configuração possui vários pontos de falha independentes que devem ser capazes de se recuperar automaticamente (Coulouris *et al.*, 2011).

Um forma de realizar a comunicação entre processos é a Chamada de Procedimento Remoto, do inglês *Remote Procedure Call* (RPC). Nesse modelo, o cliente faz uma chamada de um procedimento remoto para o servidor, que o executa e retorna o resultado. Essa abordagem é muito útil, já que abstrai a lógica de comunicaçãos dos *sockets* envolvidos (Birrell; Nelson, 1984).

2.3 Estruturas de Processamento Distribuído

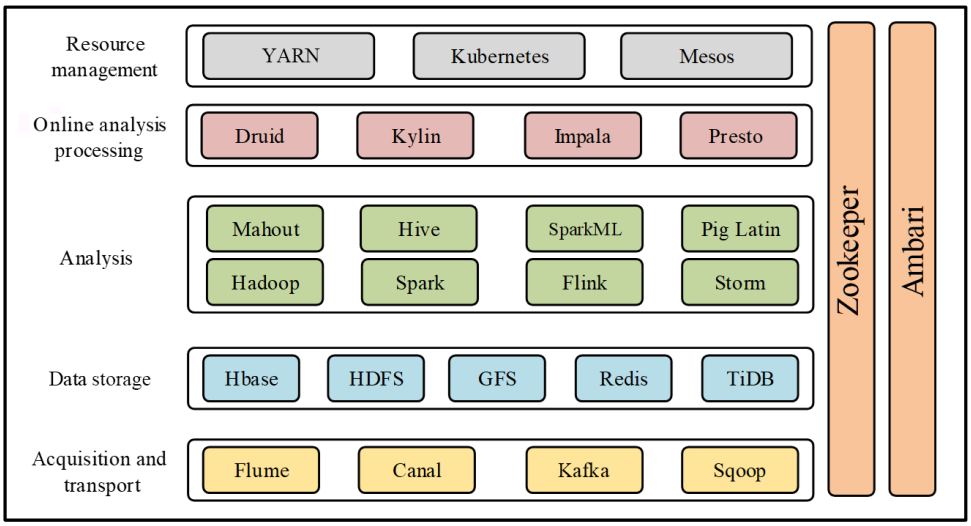
A Google observou que muitas aplicações de *Big Data* possuíam requisitos semelhantes, o que permitiu a construção de abstrações que poderiam ser aplicadas em diversos cenários, além de ocultar a complexidade repensável por estabilizar a aplicação (Dean; Ghemawat, 2008).

Ao longo do tempo, essas abstrações evoluíram significativamente tornando-se o padrão de mercado. No entanto, o volume e a velocidade dos dados aumentam mais rapidamente do que a capacidade de processamento, sendo necessário um constante aprimoramento das técnicas de processamento distribuído para que seja possível otimizar a eficiência na utilização dos recursos físicos.

Em um cenário real, vários *frameworks* são utilizados de forma complementar, podendo ser agrupados de acordo com o seu papel dentro de uma infraestrutura de *Big Data*. A Figura 2 representa uma visão geral das principais ferramentas disponíveis no mercado e suas classificações: gerenciamento de recursos, processamento de análise on-

line, análise, armazenamento de dados e aquisição e transporte.

Figura 2 – Pilha de Tecnologias utilizadas em *Big Data*



Fonte: Sun *et al.* (2023).

Os *frameworks* de análise podem ser classificados de acordo com 5 critérios: armazenamento primário (disco ou RAM), granularidade (tamanho da tarefa a ser processada), armazenamento do estado, forma de execução (lote ou fluxo) e o volume (finito ou infinito).

O armazenamento primário, seja em disco ou RAM, é um critério fundamental na classificação dos *frameworks* de processamento distribuído. O armazenamento em disco, sendo não volátil e de baixo custo, é adequado para tarefas que manipulam grandes volumes de dados, quando tempos de leitura e escrita maiores são aceitáveis. Por outro lado, a RAM, apesar de ser volátil, oferece acesso mais rápido aos dados, sendo ideal para tarefas que exigem processamento em tempo real, mas é limitada pelo valor elevado em relação ao disco rígido(Tanenbaum; Steen, 2006).

A granularidade em sistemas distribuídos refere-se ao tamanho das tarefas que são distribuídas entre os diferentes nós de um sistema, podendo ser classificada em duas categorias principais: granularidade fina e granularidade grossa. Na granularidade fina, as tarefas são divididas em partes muito pequenas, o que permite uma distribuição mais uniforme das tarefas, podendo levar a um melhor equilíbrio de carga. Contudo, a granularidade fina também pode levar a um aumento na comunicação entre os nós, o que pode resultar em uma sobrecarga de comunicação. Por outro lado, na granularidade grossa as tarefas são divididas em partes maiores, o que pode reduzir a sobrecarga

de comunicação, visto que há menos tarefas sendo passadas entre os nós. No entanto, alcançar um equilíbrio de carga eficaz pode ser mais difícil, pois algumas tarefas podem ser significativamente maiores do que outras. Nesse sentido, a escolha da granularidade em um sistema distribuído é um compromisso entre o equilíbrio de carga e a sobrecarga de comunicação: uma granularidade muito fina pode levar a uma sobrecarga de comunicação, enquanto uma granularidade muito grossa pode resultar em um equilíbrio de carga ineficaz (Kwiatkowski, 2002).

Essas estruturas também podem ser agrupadas pela forma de processamento, sendo estas: lote e fluxo. No processamento em lote, os *frameworks* coletam um conjunto de tarefas e as operam todas de uma vez, já no processamento em fluxo as tarefas são processadas à medida que chegam (Stonebraker; Çetintemel; Zdonik, 2005). Nessa perspectiva, a execução em lote pode ser mais eficiente para tarefas grandes que não requerem resultados em tempo real, enquanto a execução em fluxo pode ser mais adequada para tarefas que requerem resultados em tempo real ou para tarefas que chegam continuamente (Das *et al.*, 2014).

Por fim, alguns *frameworks* são projetados para lidar com volumes finitos de dados, enquanto outros são projetados para lidar com volumes infinitos de dados. O volume de informações a ser analisada pode influenciar significativamente a seleção do *framework*, pois diferentes sistemas podem apresentar habilidades distintas e eficácias ao manipular dados limitados ou ilimitados (Lipák; Macak; Rossi, 2019).

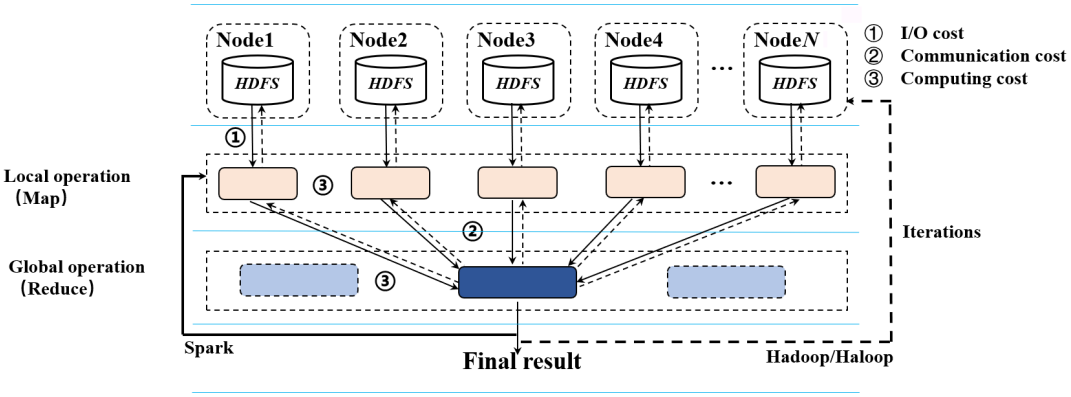
2.3.1 Hadoop MapReduce/HDFS

O termo *MapReduce* se originou de duas funções muito utilizadas em linguagens de paradigma funcional. A função *Map* é responsável por aplicar uma função a um grupo de dados e a função *Reduce* utiliza uma função para transformar um grupo de dados em um único valor, sem armazenar o estado interno das operações. Com essa abstração, foi possível simplificar a construção de um grande número de algoritmos distribuídos, já que a operação *Map* poderia ser executada em paralelo de forma independente, evitando, assim, problemas de concorrência. Por esse motivo, o *MapReduce* foi, e ainda é, amplamente utilizado em diversos cenários. Entretanto, essa tecnologia não está aberta ao publico, sendo exclusiva da Google (Dean; Ghemawat, 2008). Por esse motivo surgiu

uma implementação de código aberto chamada *Hadoop MapReduce*.

O *MapReduce*, além de ser uma estrutura de processamento, é também o modelo de representação de aplicações distribuídas utilizado pelo *Spark* — tanto o *MapReduce* quanto o *Spark* podem ser representados pela Figura 3. Existem outros modelos, como a Partição de Amostra Aleatória, do inglês *Random Sample Partition* (RSP) (Salloum; Huang; He, 2019), mas eles não são o foco deste trabalho.

Figura 3 – Modelo de funcionamento *MapReduce*



Fonte: Sun *et al.* (2023).

O *MapReduce* foi desenvolvido em uma época em que a memória RAM ainda era muito cara, por isso, ele utiliza o disco rígido como canal de entrada e saída de lotes de dados (granularidade grossa). Esse dispositivo era acessado através de um sistema de arquivos distribuídos chamado Sistema de Arquivos Google, do inglês *Google Distributed File System* (GFS) (Ghemawat; Gobioff; Leung, 2003). No GFS, ou sua implementação livre HDFS (Shvachko *et al.*, 2010), cada arquivo é gravado de forma particionada, distribuída e tolerante a falhas. Ou seja, cada arquivo é dividido em pedaços e esses pedaços são armazenados em vários computadores diferentes com redundância. O particionamento distribuído aumenta a velocidade de leitura e escrita total do sistema, já que partes diferentes do arquivo eram lidas em paralelo e a redundância permite que os pedaços de um arquivo perdido por alguma falha possam ser recuperados de outros nós do *cluster*, porém esse cenário só funciona quando todos os dados já estão armazenados no disco.

2.3.2 Spark

Em maio de 2014, surgiu o *Spark*, construído na linguagem *Scala*. Este *framework*, diferentemente do *MapReduce*, carregava todos os dados do disco para a RAM em estruturas de armazenamento distribuído chamadas RDD (Zaharia *et al.*, 2012). Com essa mudança, o *Spark* reduziu o tempo de execução em 5 vezes na média (Shi *et al.*, 2015), o que, juntamente com a redução nos custos da memória RAM, fez com que o *Spark* se tornasse o *framework* de processamento distribuído mais utilizado na atualidade.

O *Spark*, assim como o *MapReduce*, funciona com o mecanismo de processamento de dados em lote (granularidade grossa) e sem armazenar o estado internamente. Isso significa que, antes de iniciar o processamento, o *framework* requer que todos os dados sejam carregados previamente, por isso ele também é classificado como *frameworks* de processamento de dados finitos. Embora seja possível executar o *Spark* em micro lotes utilizando o *Spark Stream*, isso adiciona uma latência extra ao sistema, tornando-o não ideal para sistemas de tempo real.

2.3.3 Flink

Para resolver a latência em sistemas de processamento em tempo real, foi projetado o *Flink*, um *framework* que processa os dados individualmente à medida que são recebidos pelo sistema (granularidade fina). Seu funcionamento é baseado em um DAG chamado *dataflow*. Para construir este DAG, o *Flink* disponibiliza quatro APIs: SQL e *TableAPI* (Alto Nível), *DataStream*/*DataSetAPI* (Nível Intermediário) e *Stateful Stream Processing* (Baixo Nível). Este trabalho se concentrará nas Interface de Programação de Aplicativos, do inglês *Application Programming Interface* (API) de nível intermediário, pois estas facilitam o entendimento do funcionamento interno do *Flink*.

A Tabela 1 apresenta uma visão geral dos *frameworks* de processamento distribuído discutidos neste trabalho. Nela, é possível observar que o *MapReduce* e o *Spark* possuem características semelhantes, enquanto o *Flink* se destaca por possuir uma granularidade mais fina, armazenar o estado interno e processar dados infinitos.

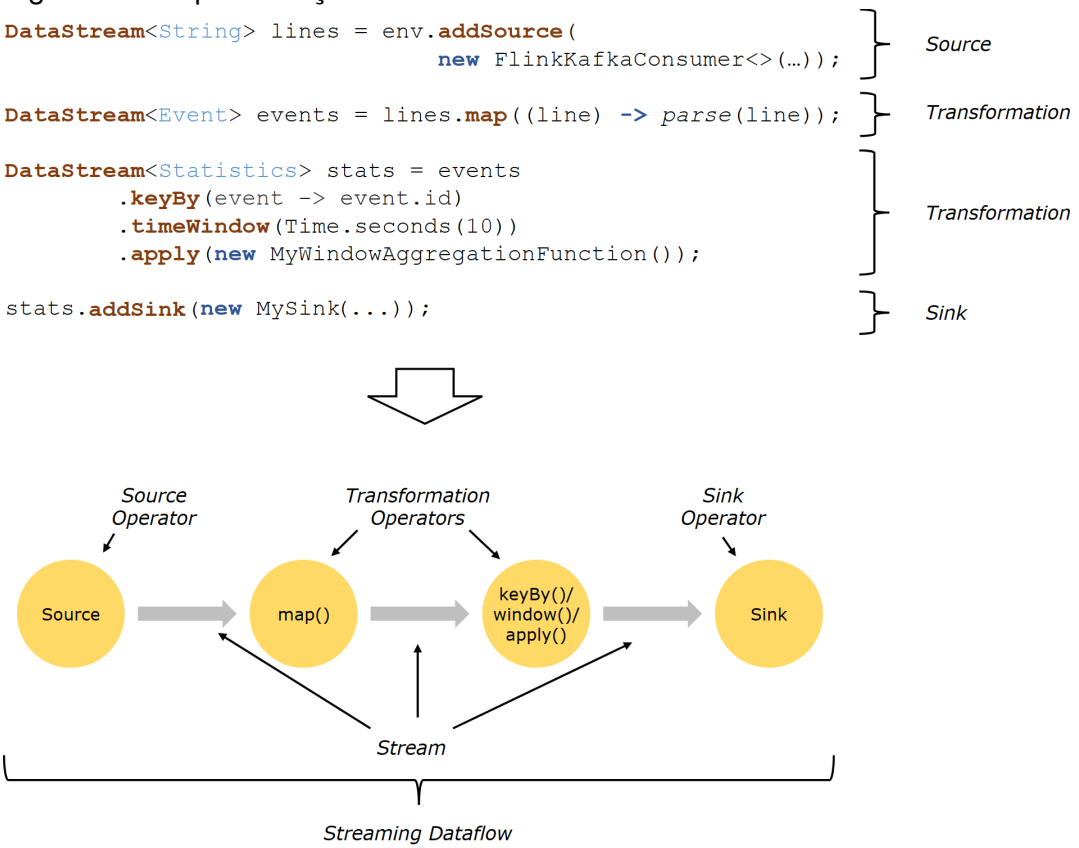
Tabela 1 – Comparação dos Frameworks

	MapReduce	**Spark**	**Flink**
Armazenamento	Disco	RAM	RAM
Granularidade	Grossa	Grossa	Fina
Estado	Sem	Sem	Com
Processamento	Lote	Micro lotes	Stream
Volume	Finito	Finito	Infinito
Linguagem.	Java	Scala	Java

Fonte: Ververica (2023)

A Figura 4 apresenta um código *Java* que utiliza a API *DataStream* do *Flink* para criar um *dataflow*. Nela se observa que a primeira linha de código cria um objeto *DataStream* – uma sequência de dados que pode ser processada em paralelo pelo *Flink* – chamado *lines*, que possui uma *String* interna que foi recebida de um tópico *Kafka* através da classe *FlinkKafkaConsumer*. A segunda linha aplica a função *parse* em cada elemento de *lines*, que retorna um novo *DataStream* do tipo *Event*. Na terceira linha, o método *keyBy* é usado para agrupar os eventos por *id*, o método *timeWindow* é usado para definir uma janela de tempo de 10 segundos e o método *apply* é usado para aplicar uma função de agregação a cada janela de tempo — essa função de agregação é definida pela classe *MyWindowAggregationFunction*. O resultado será um *DataStream* do tipo *Statistics*. Por fim, a quarta linha adiciona um *sink* local para onde os dados são enviados após o processamento – ao *pipeline* por meio da classe *MySink*.

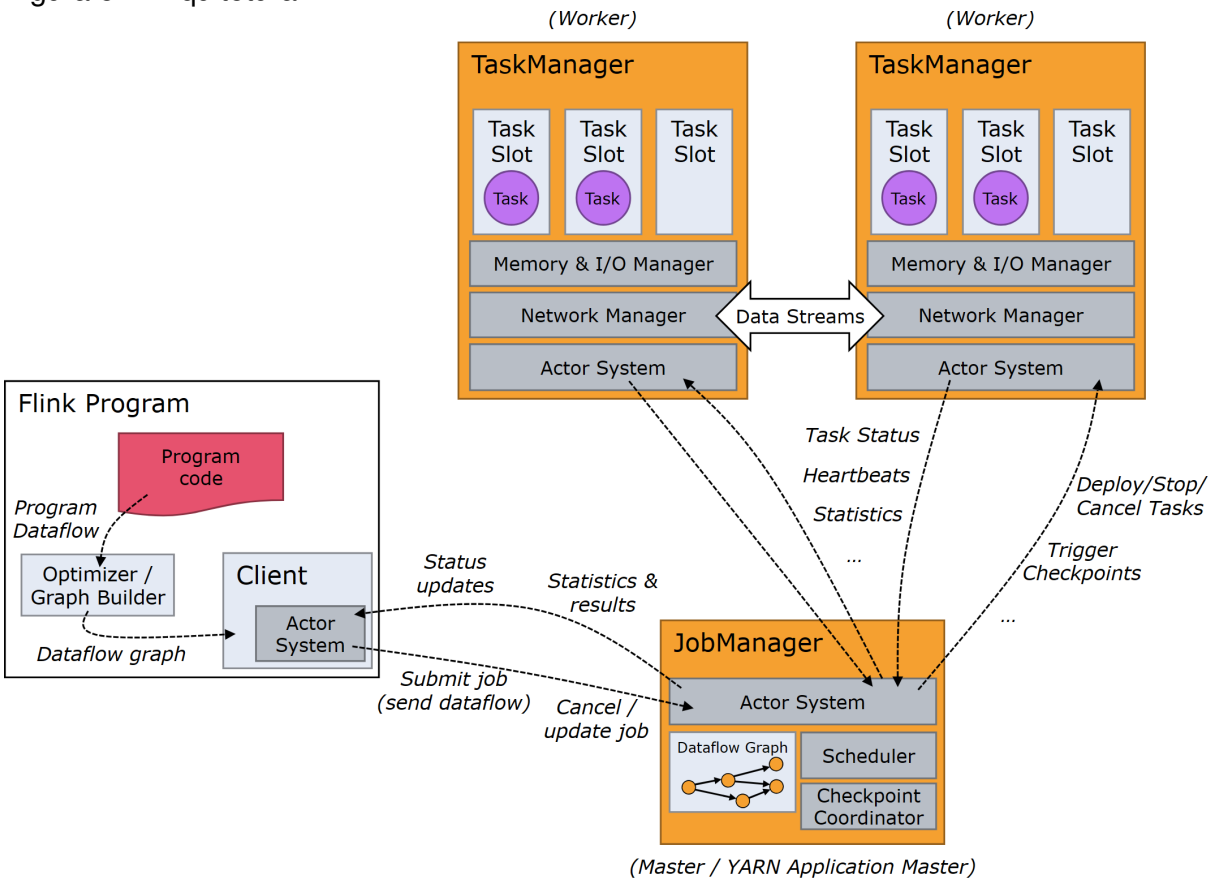
Figura 4 – Representação do funcionamento do Flink



Fonte: OVERVIEW... (2023).

Após a sua definição, o DAG é dividido e cada estágio é multiplicado pelo paralelismo definido e por fim enviado para os processos responsáveis pelo funcionamento do *Flink*: o *Client*, o *JobManager* e o *TaskManager*. No *Client*, o código *Java* é de fato representado na forma de DAG, otimizado e enviado ao *JobManager*. Este, por sua vez, é responsável por fragmentá-lo em várias tarefas independentes, baseada no grau de paralelismo definido para cada nó, e distribuir as tarefas entre os trabalhadores (*TaskManagers*), conforme mostrado na Figura 5.

Figura 5 – Arquitetura Flink



Fonte: FLINK. . . (2023).

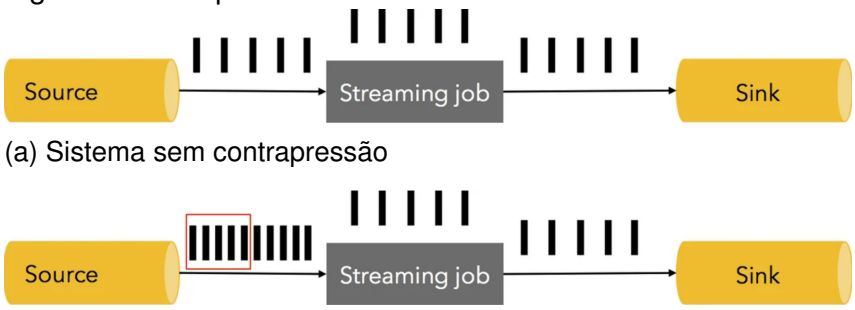
2.3.3.1 Contrapressão

Quando analisamos a arquitetura de funcionamento do *Flink*, dois problemas ficam evidentes: o problema do paralelismo estático definido pelo usuário e o problema da contrapressão. Quando o usuário define estaticamente o paralelismo de cada estágio, isso reduz a flexibilidade de manipulação dinâmica do paralelismo, dado que o *Flink* precisa cumprir com essas restrições a todo custo. Isso também adiciona a necessidade de mecanismos para estabilização do fluxo de dados em cenários de sobrecarga local (Hanif; Yoon; Lee, 2020).

No *Flink*, essa estabilização é alcançada por meio de múltiplas filas e mensagens de contrapressão, do inglês *backpressure*, representado na Figura 6. De forma mais detalhada, cada estágio do *dataflow* possui uma fila de mensagens a serem processadas e quando um estágio é sobrecarregado, ele começa a emitir mensagens para os estágios anteriores, solicitando-os para diminuirem a sua velocidade de processamento. Dessa

forma, o armazenamento das mensagens fica distribuído entre as filas de todos os estágios, não apenas no estágio sobrecarregado (Matteussi *et al.*, 2022).

Figura 6 – Backpressure



(a) Sistema sem contrapressão

(b) Sistema com contrapressão

Fonte: Celebi (2022).

No entanto, nesse sistema, é impossível que 100% da capacidade computacional seja utilizada para resolver a sobrecarga local, dado que os eventos de contrapressão são um aviso para se limitar a capacidade de processamento. Além disso, ao se definir um *dataflow*, deve-se especificar o nível de paralelismo de cada estágio do grafo. Dessa forma, nota-se que o *Flink* tem menos possibilidades de otimização no uso de recursos.

3 METODOLOGIA

Neste capítulo, serão apresentados os métodos utilizados para o desenvolvimento do *framework* proposto, sendo estes: a escolha das ferramentas (Seção 3.1) e a arquitetura proposta (Seção 3.3). Além disso, será apresentada uma expicação detalhada de como foi realizado o processo de coleta e avaliação de desempenho (Seção 3.4).

3.1 Escolha das ferramentas

A primeira etapa em todo o processo de desenvolvimento é sempre a seleção das ferramentas. E, dada a natureza desse trabalho, a linguagem de programação deve ser: performática, produtiva e multitarefa.

Nessa perspectiva, a linguagem de programação ideal deve cumprir com todas as escolhas de projeto que aumentaria o desempenho do resultado final. Para isso, ela deve ser compilada para a máquina nativa, tipada estaticamente, e não deve possuir um GC.

Além disso, para viabilizar o desenvolvimento da aplicação em tempo hábil, a linguagem deve possuir as seguinte características que aumentam a produtividade do desenvolvedor: *language server*, tipagem estática, segurança nula, bibliotecas maduras e uma comunidade ampla. O *language server* em conjunto com a tipagem estática e a segurança nula permite capturar mais erros e de forma mais fácil durante desenvolvimento. Um ecossistema de bibliotecas maduras fornecem abstrações que permitem realizar tarefas complexas com menos esforço. E, por fim, uma comunidade ampla para colaborar com a solução de problemas que possam surgir.

Além disso, essa linguagem deve fornecer as ferramentas necessárias para trabalhar com multitarefa, do inglês *multitasking*, envolvendo concorrência e paralelismo. Ademais, é importante que ela permita ao desenvolvedor utilizar modelos de atores para reduzir a complexidade envolvida nesse tipo de aplicação.

Durante o processo de escolha da linguagem de programação, foram levadas em considerações três linguagens: *Erlang*, *Kotlin* com o *framework Akka* e *Rust* com o *framework Actix*. A linguagem *C++* foi descartada por não garantir a segurança nula e pela falta ausência de um gerenciador de dependências, o que aumenta a complexidade de se manter um projeto.

A linguagem Erlang possui na sua base um modelo de concorrência de atores distribuídos, que faz parte da biblioteca padrão. Além disso, ela garante a segurança desses atores por meio de um modelo de supervisores que monitora os atores em tempo de execução (Armstrong, 2003). Contudo, por mais que seja uma linguagem madura e robusta para ser executada em cenários de produção, Erlang possui tipagem dinâmica, é compilada para uma VM, a Beam, e possui GC. Ademais, possui baixa popularidade, o que faz com que o seu ecossistema e a sua comunidade sejam pequenos.

Uma alternativa é o *Kotlin* (KOTLIN. . . , 2024), que, além de ser madura, possuir tipagem estática e garantir a segurança nula, permite a utilização do modelo de atores por meio do *framework Akka* (AKKA. . . , 2024), que implementa também a segurança por supervisores. No entanto, ela também compila para a VM do *Java* e também possui um GC para gerenciamento de memória.

A terceira opção analisada foi *Rust* (RUST. . . , 2024), que apresentou uma solução alternativa ao gerenciamento de memória, com GC ou manual. Essa alternativa utiliza os conceitos de *Ownership, Borrowing e Lifetimes* para informar ao compilador o momento certo de fazer a liberação da memória automáticamente. Dessa forma, o desenvolvedor não precisa se preocupar com o gerenciamento manual de memória, mas também não se faz necessário a utilização de um *garbage collector* em tempo de execução. Porém, essa abordagem possui uma limitação, já que todas as funções e estruturas devem ser construídas com essas diretrizes para que o compilador tenha as informações necessárias para fazer a desalocação. No entando, algumas bibliotecas não foram projetadas com essas diretrizes, fazendo-se necessária a utilização de contagem de referência para evitar cópias desnecessárias de valor. Além disso, *Rust* é muito segura dado o seu sistema de tipos robusto e aos tipos opcionais.

No primeiro momento, quando analisadas as ferramentas de programação concorrente e paralela em *Rust*, percebe-se que a linguagem, por padrão, fornece apenas diretrizes de acesso às *threads* do SO, sem nenhum nível de abstração. Porém, ela permite que o usuário adicione um tempo de execução assíncrono como dependência, liberando, assim, novas diretrizes na linguagem para programação concorrente. Atualmente, o *runtime* assíncrono mais popular é o *Tokio* (TOKIO. . . , 2024).

Assim como no ecossistema *Java*, a linguagem *Rust* também possui um *framework* de atores de baixo nível chamado *Actix* (ACTIX. . . , 2024). Esse modelo de atores permite

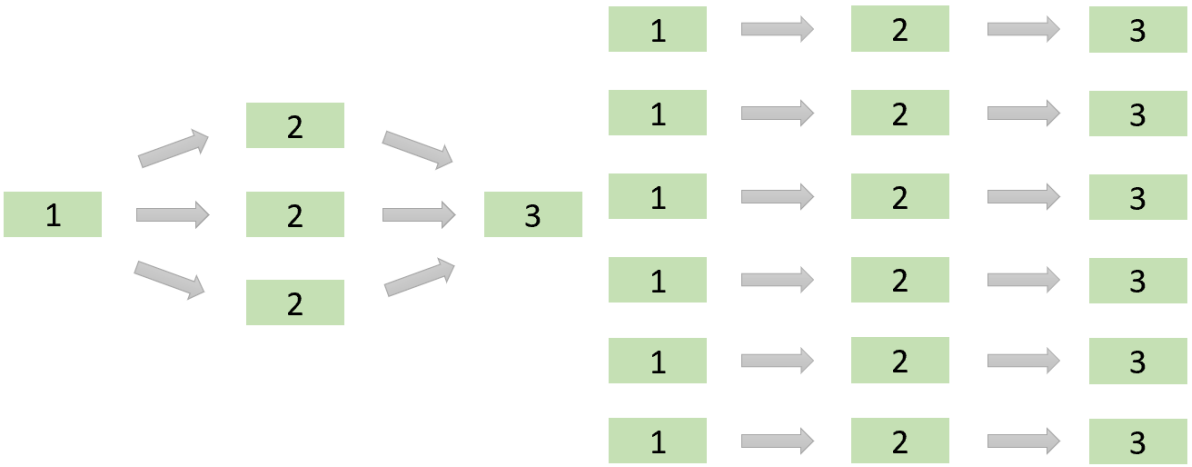
que o desenvolvedor escolha em qual árbitro o ator será executado. Um árbitro consistem em um *event loop* que processará as mensagens de todos os atores que se inscreverem nele, permitindo, assim, que o desenvolvedor crie um árbitro por *thread*.

Com base nas informações analisadas, *Rust* cumpriu com a maior quantidade de requisitos, tornando-se a escolha mais promissora.

3.2 Dataflow

Para aumentar a flexibilidade do balanceamento de carga, primeiro é necessário representar o *dataflow* de forma diferente. No *Apache Flink* o fluxo de dados é representado em uma única estrutura com o paralelismo definido estaticamente para cada operador (7a), o que restringe a capacidade de balanceamento de carga. Portanto, para resolver esse problema, foi proposto uma arquitetura de multiplos *dataflows* (7b), em que cada *dataflow* funcionaria de forma independente e assíncrona, permitindo que o *event loop* coordenasse a execução de todas as mensagens recebidas automaticamente.

Figura 7 – Dataflow.
(a) Flink. (b) Arquitetura Proposta.



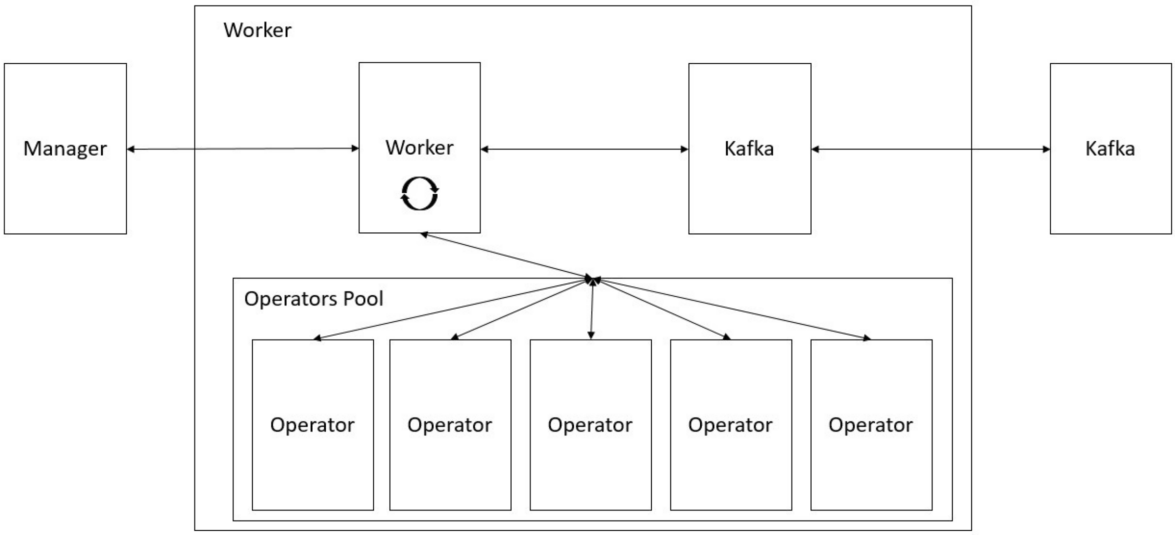
Fonte: Elaborado pelo autor, 2023.

Além disso, espera-se que a quantidade de mensagens trocadas entre os nós seja reduzida, já que cada nó irá consumir de apenas uma partição *Kafka*. Isso garantirá que todas as mensagens de um mesmo grupo de chaves sejam processadas pelo mesmo nó.

3.3 Arquitetura

A arquitetura deste *framework* é composta por dois tipos de processos, um *Manager* e um *Worker*. Este *Worker* é composto por três atores: o *Kafka*, o *Operator* e o *Worker*. O processo *Manager* possui uma única tarefa, que é fornecer a partição do tópico *Kafka* que será lido por um determinado *Worker*. O restante da lógica está concentrada no *Worker* (Figura 8).

Figura 8 – Arquitetura do *Framework*.



Fonte: Elaborado pelo autor, 2023.

O ator *Kafka*, primeiramente, recebe a partição a ser consumida do ator *Worker* que faz essa requisição ao processo *Manager* via RPC. A seguir ele começa a monitorar as mensagens recebidas no tópico *input* de forma assíncrona para que cada mensagem seja desserializada em uma estrutura e enviada numa mensagem interna para o ator *Worker*. Além disso, este ator também é responsável por receber as mensagens do *Worker* que já foram processadas e enviar de volta para o *Kafka* no tópico de *output*.

O ator *Operator* é responsável por executar as funções definidas no *dataflow* na mensagem que foi recebida de forma síncrona e retornar o resultado para o *Worker* quando o processamento encerrar.

O ator *Worker* é responsável por coordenar a execução das funções definidas no *dataflow* em todos os núcleos disponíveis. Para isso, ele define um *pool* de *Operators* e

monitora de forma assíncrona as mensagens recebidas pelo ator *Kafka*.

É importante ressaltar que cada mensagem enviada pelos atores é enfileirada no *event loop*. Dessa forma, sempre quando uma mensagem é recebida, ela não é processada imediatamente, mas entra numa fila de processamento para ser executada quando o *event loop* estiver livre.

Com isso, quando uma mensagem recebida pelo *worker* for processada, ela é enviada ao *pool* de *Operators* definido anteriormente e a resposta é aguardada. O fato de se esperar a resposta de retorno do resultado libera o *event loop* para processar a mensagem seguinte enquanto aguarda o retorno. Com essa lógica, é possível deixar que o *event loop* coordene o processamento de todas as mensagens recebidas de forma automática e otimizada simultaneamente (Figura 9).

Figura 9 – Lógica de Balanceamento.

```
// No cenário real, deve-se percorrer o grafo e não um vetor
for i in 0..operations_len {
    data = operators
        .send(msg: Operation {
            data,
            operation_index: i,
        })
        .await
        .expect(msg: "Failed to received the operation result");
}
kafka
    .expect(msg: "Kafka actor not initialized")
    .do_send(msg: Output(data));
```

Fonte: Elaborado pelo autor, 2023.

Com essa lógica, quando uma mensagem com as instruções de processamento for enviada ao pool de *Operators*, ela será executada em qualquer núcleo que estiver disponível, levando assim a uma otimização na utilização da CPU. Além disso, como todos os *Operators* são capazes de executar todas as funções definidas no *dataflow*, o sistema pode delegar a terefa de balanceamento de carga entre os nós para o particionamento do tópico *Kafka*. Dessa forma, o sistema garante que todas as mensagens de um mesmo grupo de chaves sejam processadas pelo mesmo nó.

Porém, essa abordagem possui uma desvantagem: por mais que, teoricamente, a utilização mais eficiente da CPU permita um *throughput* maior, o *event loop* irá coordenar a execução de todas as mensagens recebidas simultaneamente, levando a um aumento

na latência, ou seja, no tempo gasto para uma mensagem entrar no sistema e sair dele.

3.4 Avaliação de Desempenho

Após encerrar a etapa de desenvolvimento, inicia-se a etapa de análise de desempenho. Nessa seção será apresentado como foi planejado o experimento e como foram coletadas os indicadores de desempenho.

3.4.1 Dataflow

Em um primeiro momento é necessário definir o *dataflow* que será executado. Para causar uma configuração desbalanceada de forma controlada, foi escolhida a função *fibonacci* distribuída em 5 estágios sequenciais de operação *map*. Essa função foi escolhida com o objetivo de gerar uma sobrecarga controlada no processador, diferentemente dos exemplos de contrapressão encontrados no *GitHub* que utilizavam a função *sleep* para pausar a *thread* sem sobrecarregar a CPU (FLINK. . . , 2024). O valor passado como argumento para a função *fibonacci* de cada estágio é recebido pelo tópico de *input* e foram sempre os mesmos: 10 - 20 - 30 - 20 - 10.

3.4.2 Ambiente de Experimento

Finalizando a definição do *dataflow*, iniciou-se a construção do ambiente de experimento. Para isso, foram necessárias as cinco ferramentas: uma ferramenta para simulação do *cluster*, o *framework* de processamento de stream que será analisado, um sistema de mensageiria, um sistema de produção e consumo de mensagens.

Para a simulação do *cluster* será utilizado o *Docker Compose* que inicia os contêiners das outras ferramentas e realiza o processo de compilação do *framework* desenvolvido, dado que, diferentemente da *Apache Fink*, a aplicação construída nesse trabalho deve ser compilada e os binários colocados na imagem final do contêiner.

Como sistema de mensageria, foi escolhido o *Apache Kafka* (Kreps; Narkhede; Rao *et al.*, 2011) , dada a sua ampla utilização em aplicações de *Big Data*. Além disso, foi necessário a inclusão de um contêiner do *ZooKeeper* devido ao fato que o *Kafka* o tem

como dependência.

Para produzir e consumir mensagens do tópico *Kafka*, foi desenvolvido uma aplicação cliente utilizando a linguagem *Kotlin*. Essa aplicação é responsável por produzir JSON com o seguinte formato:

Figura 10 – *JSON* de Entrada.

```
{
    "id": 0,
    "stage1": 10,
    "stage2": 20,
    "stage3": 30,
    "stage4": 20,
    "stage5": 10,
    "timestamp": 0
}
```

Fonte: Elaborado pelo autor, 2023.

Neste JSON o campo *id* é um identificador único da mensagem, os campos *stage1* a *stage5* são os valores passados para a função *fibonacci* em cada estágio e o campo *timestamp* é o tempo em que a mensagem foi produzida. Além disso, o campo *id* é utilizado para identificar a mensagem no momento da sua chegada no tópico *output*.

3.4.3 Configuração do Hardware

A plataforma em que foi executada essa arquitetura possui a seguinte configuração:

a) Intel Core i7-4870HQ de 4 núcleos físicos e 4 núcleos Hyper Threading de 3.700 GHz
b) 16 GB de memória RAM

3.4.4 Versões dos Software

As versões dos softwares utilizados para os experimentos foram as seguintes:

a) Rustc 1.73.0
b) Kotlin JVM 1.6.0
c) Linux debian 6.5.8
d) Apache Flink 1.18.0
e) Apache Kafka 3.6.0
f) Apache ZooKeeper 3.9.1
g) Docker 24.0.7

3.4.5 Experimentos

Para avaliar o desempenho dessa arquitetura, foram planejados oito experimentos. Nesses experimentos, ambos os *frameworks* foram submetidos à mesma carga de trabalho (pulsos de 1.000 e de 10.000 mensagens conforme apresentado em Figura 10), variando alguma de suas características, para avaliar o seu comportamento. As características variadas foram: a quantidade de réplicas do conteiner (1 ou 2) e a quantidade de *threads* (4 ou 8). A Tabela 2 apresenta as variáveis utilizadas em cada experimento.

Tabela 2 – Variáveis dos Experimentos.

Experimento	**Réplicas**	**Threads**	**Mensagens**
1	1	4	1000
2	1	4	10000
3	1	8	1000
4	1	8	10000
5	2	4	1000
6	2	4	10000
7	2	8	1000
8	2	8	10000

Fonte: Elaborado pelo autor, 2023.

Em todos os casos, os indicadores de desempenho utilizados foram a taxa de transferência, do inglês *throughput*, e a latência, do inglês *latency*. A taxa de tranferência representa a vazão do sistema, ou seja, quantas mensagens são processadas e enviadas em um intervalo de tempo. A latência se refere ao tempo que uma mensagem leva para

entrar, ser processada e sair do sistema (Van Dongen; Van den Poel, 2020).

Após a execução de uma rodada teste no *Flink*, verificou-se uma alta sobrecarga no nó central, em que a função *fibonacci* era executada no número 30. Por esse motivo, o paralelismo estático definido foi 1 - 1 - 4 - 1 - 1, dado que a configuração 1 - 2 - 3 - 2 - 1 entregou um desempenho inferior. Essa etapa não foi necessária no *framework* desenvolvido nesse trabalho, já que o paralelismo era definido dinamicamente em tempo de execução baseado no *event loop* e no *pool de Operators*.

Devido as características das ferramentas utilizadas, alguns detalhes devem ser levados em consideração. A biblioteca *RSKafka* requer que o número de contêiners *workers* corresponda ao mesmo número de partições, portanto, 1 ou 2 contêiners. Além disso, o *Apache Flink* lida com a quantidade total de *threads* disponível de uma forma diferente — por meio de *task slots*, que está relacionada, também, ao paralelismo definido pelo usuário no *dataflow*. Na aplicação proposta, como um dos objetivos era automatizar o processo de paralelismo ótimo, a quantidade de núcleos disponíveis é obtida no momento de inicialização. Entretanto, é possível verificar tanto a quantidade de núcleos físicos quanto a quantidade de núcleos *hyper-threading* (Saini *et al.*, 2011), que no dispositivo de teste correspondia a 4 e 8 respectivamente, as mesmas quantidades de *tasks slots* utilizada no *Flink*.

4 RESULTADOS

Ao término dos experimentos, foram obtidos 3 formas de resultados: uma tabela com o resumo dos indicadores de desempenho coletados em todos os experimentos (Seção 4.1), imagens do consumo de CPU durante a execução dos frameworks (Seção 4.2) e gráficos de latência em função do identificador da mensagem (Seção 4.3).

4.1 Tabela de Desempenho

A Tabela 3 apresenta os valores de latência média e taxa de transferência de ambos os *frameworks*. Além disso, ela também mostra qual foi a variação da abordagem proposta em relação ao *Flink*.

Para os experimentos com 1000 mensagens (1,3,5), a solução desenvolvida apresentou um aumento médio na taxa de transferência de 80% em relação ao *Flink*, isso não foi observado apenas no experimento 7. Além disso, para esses experimentos, também houve uma redução média de 53% na latência média, exceto no experimento 7.

Para os experimentos com 10.000 mensagens (2,4,6,8), o *Flink* obteve uma taxa transferência de 16 a 2% maior. No entanto, a latência média quase dobrou em comparação com a do *Flink*. Acredita-se que isso se deve ao fato do *event loop* iniciar o processamento de todas todas as mensagens ao mesmo tempo.

Tabela 3 – Comparação dos Resultados

Experimento	**Taxa de Transferência** (msg/s)			**Latência Média** (ms)		
	Proposta	**Flink**	**Variação**(%)	**Proposta**	**Flink**	**Variação**(%)
1	301,66	166,58	81,09	2906	5197	-44,08
2	513,64	591,23	-13,12	16770	7718	117,28
3	302,02	168,35	79,40	2760	5311	-48,03
4	491,21	584,69	-15,99	17004	7534	125,70
5	307,98	172,24	78,81	2707	4918	-44,96
6	542,27	576,17	-5,88	14689	8030	82,93
7	160,75	163,72	-1,82	5582	5281	5,70
8	515,62	559,72	-7,88	15052	8454	78,05

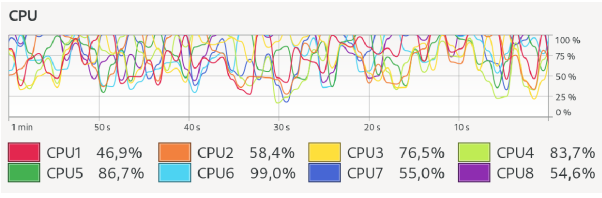
Fonte: Elaborado pelo autor, 2023.

4.2 Consumo de CPU

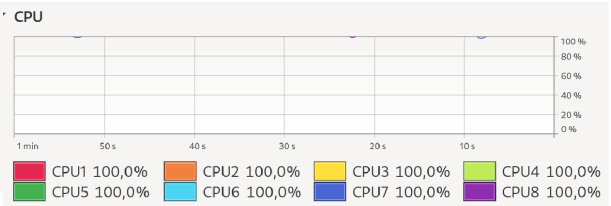
Ao comparar os gráficos de utilização de CPU do *Flink* (11a) e da aplicação desenvolvida (11b) durante o experimento 1, é possível perceber que o *Flink* não consegue se estabilizar em 100%, enquanto a solução proposta consegue. Isso é possível graças à retirada do paralelismo estático e consequentemente à utilização de uma só fila para controlar o balanceamento de carga, sem a necessidade de eventos de contrapressão.

Figura 11 – Consumo de CPU

(a) Flink em Sobrecarga.



(b) Arquitetura Proposta em Sobrecarga.



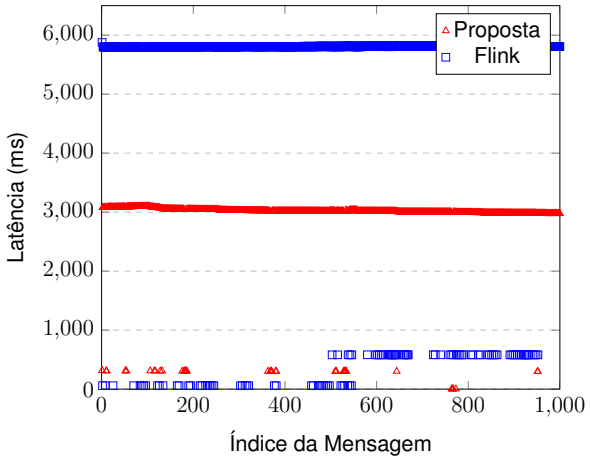
Fonte: Elaborado pelo autor, 2023.

4.3 Latência por Mensagem

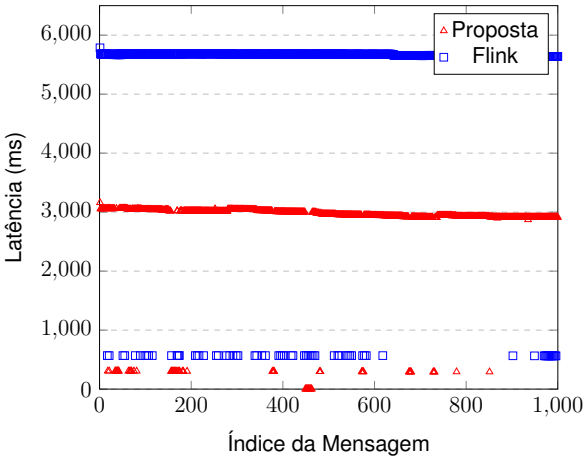
Para compreender como ambos os sistemas reagem em um cenário de sobrecarga deve-se analisar, além da latência média, a latência individual de cada mensagem. Para isso, abaixo são apresentados os gráficos de latência em função do índice da mensagem de todos os experimentos.

Figura 12 – Experimentos com 1000 mensagens.

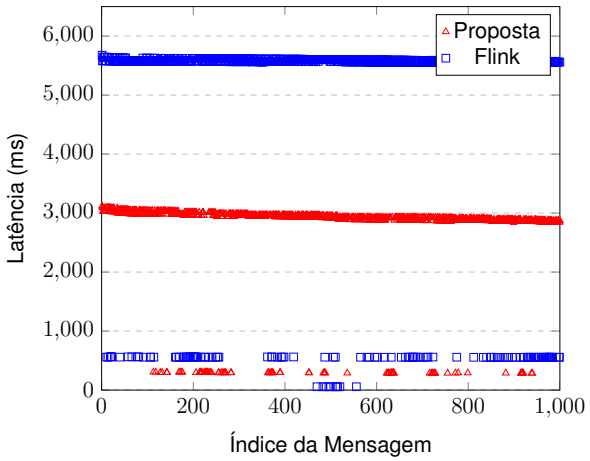
(a) Experimento 1: 1 Replica - 4 Threads



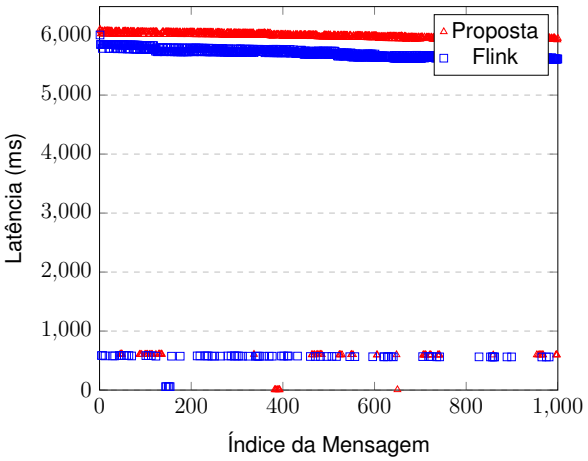
(b) Experimento 3: 1 Replica - 8 Threads



(c) Experimento 5: 2 Replica - 4 Threads



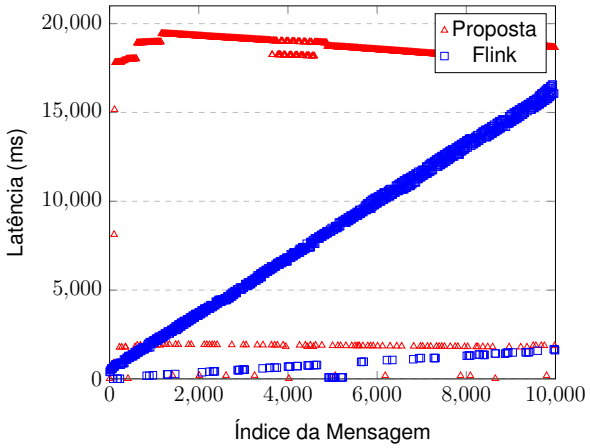
(d) Experimento 7: 2 Replica - 8 Threads



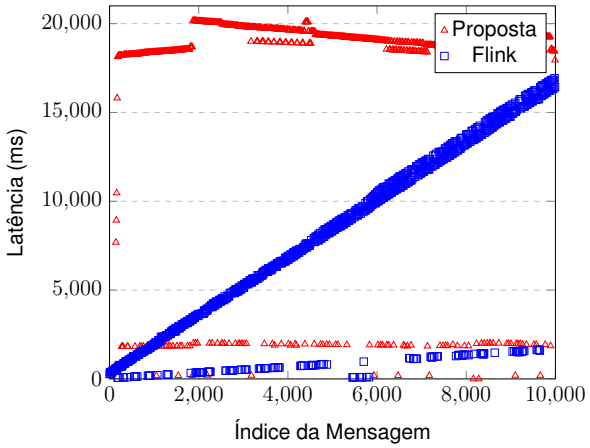
Fonte: Elaborado pelo autor, 2023.

Figura 13 – Experimentos com 10000 mensagens.

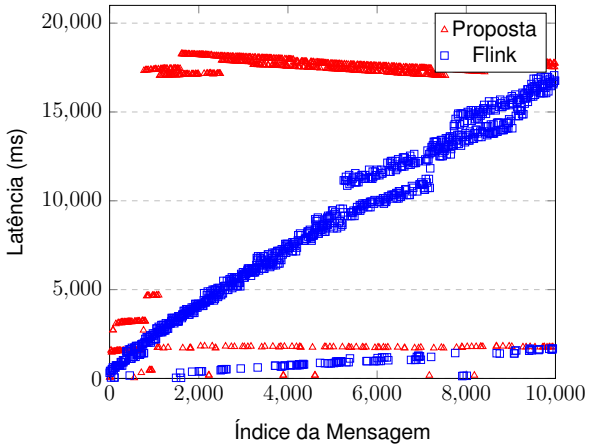
(a) Experimento 2: 1 Replica - 4 Threads



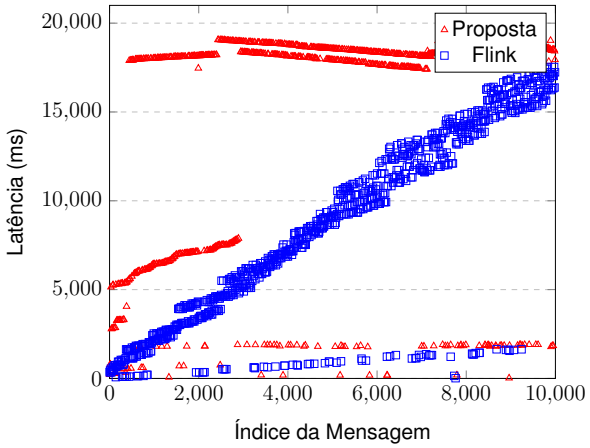
(b) Experimento 4: 1 Replica - 8 Threads



(c) Experimento 6: 2 Replica - 4 Threads



(d) Experimento 8: 2 Replica - 8 Threads



Fonte: Elaborado pelo autor, 2023.

Analisando os gráficos dos experimentos com 1000 mensagens (12a, 12b, 12c), com exceção da 12d, pode-se notar que, de fato, a latência da estrutura desenvolvida é em média 50% menor que a do *Flink*. Além disso, enquanto a latência do *Flink* cresce de forma linear para os experimentos com 10.000 mensagens, a da solução proposta se mantém praticamente constante no valor máximo para todos os casos (13a, 13b, 13c, 13d), o que reforça a conjectura de que o *event loop* inicia o processamento de todas as mensagens simultaneamente.

Além disso, é possível concluir que a variação da quantidade de threads e de replicas não possui impacto significativo na latência da maioria dos experimentos, com exceção do experimento 7, em que a latência da solução proposta foi ligeiramente maior que do *Flink*, possívelmente, por causa da troca de contexto excessiva.

5 CONCLUSÃO

Analisando os resultados dos experimentos, pode-se concluir que a solução apresentada nesse trabalho reduziu em aproximadamente 50% a latência média e aumentou a taxa de transferência em 80% para os casos 1,3 e 5. Entretanto, devido a natureza do *event loop* do *Tokio*, a latência dos experimentos de 10.000 mensagens dobrou. Além disso, percebe-se um uso mais eficiente dos recursos computacionais do *cluster* dado a sua natureza mais flexível, entretanto, sem uma melhora de desempenho para todos os casos na mesma proporção.

Tendo isso em vista, novas pesquisas se fazem necessárias para compreender os mecanismos de otimização utilizados no *Flink* e aplicá-los neste trabalho. Com isso, espera-se um aumento significativo do desempenho do sistema. Do mesmo modo, deve-se analisar a viabilidade de se customizar o *event loop* para não iniciar o processamento de todas as mensagens de uma só vez, com o objetivo de reduzir o impacto negativo na latência em momentos de pico.

Além disso, é necessário implementar outras operações de *stream* (*filter, reduce, join ...*) definidas no *Flink* (OPERATORS. . . , 2024). Isso é importante, pois a solução proposta nesse trabalho foi testada apenas com a operação de *map*, que sozinha não representam a complexidade de um *Dataflow* real.

Por fim, pode-se concluir que o resultado desse trabalho é promissor, mas carece de novas pesquisa, já que o seu desempenho foi superior ao do *Flink* em alguns cenários, sendo este uma aplicação com mais de 10 anos de desenvolvimento e que atualmente recebe em média 30 *commits* por semana. Ademais, essa aplicação remove os limites para se atingir o balanceamento de carga ótimo da aplicação, o que a torna economicamente interessante se obtiver um desempenho superior ao *Flink* nos outros casos analisados.

REFERÊNCIAS

ACTIX. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://actix.rs.

AKKA. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://akka.io.

ANURADHA, J *et al.* A brief introduction on Big Data 5Vs characteristics and Hadoop technology. **Procedia computer science**, Elsevier, v. 48, p. 319–324, 2015.

ARMSTRONG, Joe. **Making reliable distributed systems in the presence of software errors**. 2003. Tese (Doutorado) – Royal Institute of Technology.

BIRRELL, Andrew D; NELSON, Bruce Jay. Implementing remote procedure calls. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 2, n. 1, p. 39–59, 1984.

CABRERA, Victor E *et al.* Symposium review: Real-time continuous decision making using big data on dairy farms. **Journal of dairy science**, Elsevier, v. 103, n. 4, p. 3856–3866, 2020.

CARBONE, Paris *et al.* Apache flink: Stream and batch processing in a single engine. **The Bulletin of the Technical Committee on Data Engineering**, Institute of Electrical e Electronics Engineers (IEEE), v. 38, n. 4, 2015.

CELEBI, Ufuk. How Apache Flink™ handles backpressure. **Ververica GmbH**, Ververica GmbH, dez. 2022. Disponível em: https://www.ververica.com/blog/how-flink-handles-backpressure. Acesso em: 16 jun. 2023.

COULOURIS, George F *et al.* **Distributed Systems**. 5. ed. Upper Saddle River, NJ: Pearson, abr. 2011.

DAS, Tathagata *et al.* Adaptive stream processing using dynamic batch sizing. *In:* ACM SYMPOSIUM ON CLOUD COMPUTING. **Proceedings of the [...]** [*S. l.: s. n.*], 2014. p. 1–13.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. **Communications of the ACM**, ACM New York, NY, USA, v. 51, n. 1, p. 107–113, 2008.

DUARTE, Fabio. Amount of Data Created Daily (2023). **Exploding Topics**, Exploding Topics, abr. 2023. Disponível em: https://explodingtopics.com/blog/data-generated-per-day. Acesso em: 26 jun. 2023.

FAROUKHI, Abou Zakaria *et al.* Big data monetization throughout Big Data Value Chain: a comprehensive review. **Journal of Big Data**, SpringerOpen, v. 7, n. 1, p. 1–22, 2020.

FLINK Architecture. Jun. 2023. Disponível em: https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture. Acesso em: 18 jun. 2023.

FLINK Variance. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://github.com/owenrh/flink-variance/blob/master/src/main/scala/com/dataflow/flink/VarianceApp.scala.

GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. The Google file system. *In:* NINETEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES. **Proceedings of the [...]** [*S. l.: s. n.*], 2003. p. 29–43.

GIUDICE, Paolo Lo *et al.* An approach to extracting complex knowledge patterns among concepts belonging to structured, semi-structured and unstructured sources in a data lake. **Information Sciences**, Elsevier, v. 478, p. 606–626, 2019.

GROSSMAN, Dan; ANDERSON, Ruth E. Introducing parallelism and concurrency in the data structures course. *In:* 43RD ACM TECHNICAL SYMPOSIUM ON COMPUTER SCIENCE EDUCATION. **Proceedings of the [...]** [*S. l.: s. n.*], 2012. p. 505–510.

GÜNTHER, Wendy Arianne *et al.* Debating big data: A literature review on realizing value from big data. **The Journal of Strategic Information Systems**, Elsevier, v. 26, n. 3, p. 191–209, 2017.

HANIF, Muhammad; YOON, Hyeongdeok; LEE, Choonhwa. A Backpressure Mitigation Scheme in Distributed Stream Processing Engines. *In:* INTERNATIONAL CONFERENCE ON INFORMATION NETWORKING (ICOIN). **2020 IEEE [...]** Barcelona, Spain: [*s. n.*], 2020. IEEE, p. 713–716.

HERTZ, Matthew; BERGER, Emery D. Quantifying the performance of garbage collection vs. explicit memory management. *In:* ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS. **Proceedings of the 20th annual [...]** [*S. l.: s. n.*], 2005. p. 313–326.

HEWITT, Carl; BISHOP, Peter; STEIGER, Richard. A universal modular actor formalism for artificial intelligence. *In:* FORMALISMS FOR ARTIFICIAL INTELLIGENCE. **Session 8**

Advance Papers of [...] [*S. l.: s. n.*], 1973. v. 3. Stanford Research Institute Menlo Park, CA, p. 235.

HOARE, Charles Antony Richard. Communicating sequential processes. **Communications of the ACM**, ACM New York, NY, USA, v. 21, n. 8, p. 666–677, 1978.

JAVED, M. Haseeb; LU, Xiaoyi; PANDA, Dhabaleswar K. (DK). Characterization of Big Data Stream Processing Pipeline: A Case Study Using Flink and Kafka. *In:* IEEE/ACM INTERNATIONAL CONFERENCE ON BIG DATA COMPUTING, APPLICATIONS AND TECHNOLOGIES. **Proceedings of the Fourth [...]** Austin, Texas, USA: Association for Computing Machinery, 2017. p. 1–10.

KOTLIN Programming Language. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://kotlinlang.org.

KREPS, Jay; NARKHEDE, Neha; RAO, Jun *et al.* Kafka: A distributed messaging system for log processing. *In:* NETDB, 2011. **Proceedings of the [...]** [*S. l.: s. n.*], 2011. v. 11. Athens, Greece, p. 1–7.

KWIATKOWSKI, Jan. Evaluation of parallel programs by measurement of its granularity. *In:* 4TH INTERNATIONAL CONFERENCE. **Parallel Processing and Applied Mathematics: [...], PPAM 2001 Na Iczów, September 9–12, 2001 Revised Papers 4**. Poland: [*s. n.*], 2002. Springer, p. 145–153.

LING, Yibei; MULLEN, Tracy; LIN, Xiaola. Analysis of optimal thread pool size. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 34, n. 2, p. 42–55, 2000.

LIPÁK, Peter; MACAK, Martin; ROSSI, Bruno. Big data platform for smart grids power consumption anomaly detection. *In:* FEDERATED CONFERENCE ON COMPUTER SCIENCE AND INFORMATION SYSTEMS (FEDCSIS). **2019 [...]** [*S. l.: s. n.*], 2019. IEEE, p. 771–780.

MATTEUSSI, Kassiano J *et al.* Performance evaluation analysis of spark streaming backpressure for data-intensive pipelines. **Sensors**, MDPI, v. 22, n. 13, p. 4756, 2022.

NETWORK, Mozilla Developer. **The event loop**. Online; accessed 29. Nov. 2023. Set. 2023. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop.

NETZER, Robert HB; MILLER, Barton P. What are race conditions? Some issues and formalizations. **ACM Letters on Programming Languages and Systems (LOPLAS)**, ACM New York, NY, USA, v. 1, n. 1, p. 74–88, 1992.

NOVIKOV, Aleksei. NodeJS. Single Threaded Event Loop feature or limitation? **Medium**, Medium, fev. 2023. Disponível em: https://medium.com/@alexeynovikov_89393/nodejs-single-threaded-event-loop-feature-or-limitation-21fca4c2c7dc.

OPERATORS. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/overview.

ORLOVICH, Maksim; RUGINA, Radu. Memory leak analysis by contradiction. *In:* INTERNATIONAL STATIC ANALYSIS SYMPOSIUM. **[...]** [*S. l.: s. n.*], 2006. Springer, p. 405–424.

OUSSOUS, Ahmed *et al.* Big Data technologies: A survey. **Journal of King Saud University-Computer and Information Sciences**, Elsevier, v. 30, n. 4, p. 431–448, 2018.

OVERVIEW. Jun. 2023. Disponível em: https://nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/overview. Acesso em: 18 jun. 2023.

PIERCE, Benjamin C. **Types and Programming Languages**. London, England: MIT Press, jan. 2002. (The MIT Press).

RUST Programming Language. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://www.rust-lang.org.

SAINI, Subhash *et al.* The impact of hyper-threading on processor resource utilization in production applications. *In:* INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING. **2011 18th [...]** [*S. l.: s. n.*], 2011. IEEE, p. 1–10.

SALLOUM, Salman; HUANG, Joshua Zhexue; HE, Yulin. Random sample partition: a distributed data model for big data analysis. **IEEE Transactions on Industrial Informatics**, IEEE, v. 15, n. 11, p. 5846–5854, 2019.

SEBESTA, Robert W. **Concepts of programming languages**. 11. ed. Upper Saddle River, NJ: Pearson, fev. 2015.

SHI, Juwei *et al.* Clash of the titans: Mapreduce vs. spark for large scale data analytics. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 8, n. 13, p. 2110–2121, 2015.

SHVACHKO, Konstantin *et al.* The Hadoop Distributed File System. *In:* SYMPOSIUM ON MASS STORAGE SYSTEMS AND TECHNOLOGIES (MSST). **2010 IEEE 26th [...]** [*S. l.: s. n.*], 2010. p. 1–10.

STONEBRAKER, Michael; ÇETINTEMEL, Uur; ZDONIK, Stan. The 8 requirements of real-time stream processing. **ACM Sigmod Record**, ACM New York, NY, USA, v. 34, n. 4, p. 42–47, 2005.

SUN, Xudong *et al.* Survey of Distributed Computing Frameworks for Supporting Big Data Analysis. **Big Data Mining and Analytics**, TUP, v. 6, n. 2, p. 154–169, 2023.

TANENBAUM, Andrew S; BOS, Herbert. **Modern Operating Systems**. 4. ed. Upper Saddle River, NJ: Pearson, mar. 2014.

TANENBAUM, Andrew S; STEEN, Maarten van. **Distributed systems**. 2. ed. Upper Saddle River, NJ: Pearson, out. 2006.

THE JIT compiler. Online; accessed 29. Nov. 2023. Mar. 2023. Disponível em: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler.

THUDUMU, Srikanth *et al.* A comprehensive survey of anomaly detection techniques for high dimensional big data. **Journal of Big Data**, Springer, v. 7, p. 1–30, 2020.

TOKIO - An asynchronous Rust runtime. [Online; accessed 19. Feb. 2024]. Fev. 2024. Disponível em: https://tokio.rs.

TUCKER, Allen B (ed.). **Computer Science Handbook, Second Edition**. 2. ed. Philadelphia, PA: Chapman & Hall/CRC, jun. 2004.

VAN DONGEN, Giselle; VAN DEN POEL, Dirk. Evaluation of stream processing frameworks. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 31, n. 8, p. 1845–1858, 2020.

VERVERICA. **Streaming Concepts & Introduction to Flink**. [Online; accessed 15. Jul. 2023]. 2023. Disponível em: https://www.youtube.com/playlist?list=PLaDktj9CFcS9YAaJ4bKWMWpjptudLr782.

ZAHARIA, Matei *et al.* Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *In:* 9TH USENIX SYMPOSIUM ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION (NSDI 12). **Presented as part of the [...]** [*S. l.: s. n.*], 2012. p. 15–28.