Investigando o Impacto dos Algoritmos de Controle de Congestionamento na execução do Apache Spark

Enzo B. Boscatto[1], **Anderson H. da S. Marcondes**[2], **Guilherme P. Koslovski**[2]

[1]Universidade do Estado de Santa Catarina - UDESC

[2]Programa de Pós-Graduação em Computação Aplicada - PPGCAP

Resumo. *O Apache Spark é um framework de processamento distribuído que supera limitações de modelos como o Hadoop MapReduce, através de otimizações como a execução em memória e suporte a operações iterativas. Sua capacidade de processar dados, transmiti-los e aplicar algoritmos de machine learning o torna essencial para as demandas computacionais atuais. No entanto, ao lidar com grandes volumes de dados, o Apache Spark enfrenta desafios oriundos do congestionamento de rede em momentos importantes, como na redistribuição de dados entre nós, que pode saturar a largura de banda existente. O presente estudo percorre diferentes cenários, replicando tráfego de rede utilizando da ferramenta iPerf, para comparar a eficácia dos protocolos de controle de congestionamento TCP, em específico: Cubic, Reno e DCTCP, na execução de uma aplicação Spark com características determinantes para comparação.*

1. Introdução

O Apache Spark surgiu como uma ferramenta unificada para análise de dados em larga escala em uma variedade de cargas de trabalho [Salloum et al. 2016], sendo uma proposta de código aberto, utilizado em aplicações iterativas e em tempo real, como *streaming* e *machine learning*. Sua arquitetura baseada em *Resilient Distributed Datasets* (RDDs) admite a distribuição paralela de tarefas entre aglomerados, no entanto, operações como o *shuffle* - estágio que executa a redistribuição dos dados entre nós - geram tráfego intenso na rede, tornando o controle de congestionamento essencial para evitar gargalos e manter a eficiência do sistema.

Em *datacenters* modernos, o congestionamento ocorre quando a demanda por recursos excede a capacidade disponível, causando atrasos e perda de pacotes. Especificamente em *clusters* Spark, isso afeta diretamente operações dependentes de comunicação entre nós. Protocolos de transporte como TCP Reno [Jacobson 1988], Cubic [Ha et al. 2008] e DCTCP [Alizadeh et al. 2010] empregam algoritmos distintos para gerenciar congestionamentos: enquanto a janela de transmissão do Cubic é uma função cúbica, o DCTCP utiliza notificações explícitas (ECN) para priorizar baixa latência. Este estudo tem como objetivo comparar o desempenho de aplicações gerenciadas e executadas com Apache Spark quando executadas em *datacenters* com diferentes configuração TCP para controle de congestionamento. Enquanto Cubic representa a configuração padrão atualmente para as distribuições Linux, DCTCP representa uma versão específica e otimizada do Reno, utilizada em infraestruturas contemporâneas. Em suma, o foco é analisar como cada algoritmo de controle de congestionamento influencia no tempo de execução de trabalhos no Apache Spark, sob diferentes condições de rede.

2. Desenvolvimento

Os testes foram conduzidos a partir de um *cluster* Spark (versão 3.5.4) com cinco nós (1 master e 4 trabalhadores), cada um utilizando 1 núcleo de CPU e 1 GB de memória. O cenário computacional foi hospedado unicamente em um computador equipado com um processador AMD Ryzen 7 5700u e 12GBs de memória RAM, utilizando o sistema operacional Ubuntu (versão 24.04). Para emular condições de rede controladas, uma regra foi aplicada limitando a largura de banda a 1Gbps, juntamente de um atraso de 10ms na interface de rede utilizada para comunicação entre os membros do aglomerado. Quatro cenários foram avaliados: o primeiro sem tráfego concorrente, já nos três subsequentes, foi imposto tráfego de 100% da rede, 90% e 50% respectivamente, durante todo o trabalho, simulado através da ferramenta iPerf (versão 2.1.9) utilizando o protocolo UDP (para gerar um tráfego concorrente agressivo). Sua utilização nos testes é vantajosa por não possuir controle de congestionamento e gerar um tráfego previsível, sem ajustar-se à rede, aproveitando sempre de sua totalidade.

A aplicação representando o usuário final foi desenvolvida em Python com a API PySpark. Ela cria um *dataset* e limita o número de partições utilizadas na distribuição dos dados, com ambos valores definidos pelo usuário. Executa operações em larga escala, como a ordenação, que redistribui as informações em ordem crescente, e agregação, que conta quantas vezes cada valor aparece no conjunto de dados, forçando *shuffles* massivos entre nós. Em suma, ela simula uma carga intensiva através da geração de um *dataset* sintético, permitindo avaliar o impacto dos protocolos TCP (Cubic, Reno e DCTCP) sob diferentes condições de rede. Os valores de partições utilizadas e linhas produzidas foram alterados para cada um dos três testes realizados, alterando o tamanho de cada partição em cada configuração, buscando a utilização completa da rede. Operações de ordenação (*orderBy*) e agregação (*groupBy*) foram utilizadas por dependerem intensamente de comunicação entre nós durante o *shuffle*, fator necessário para produzir o congestionamento. Além disso, o uso de *persist* e *unpersist* em ambas ações garante que os dados sejam armazenados em memória ou disco conforme necessário, otimizando o uso de recursos, promovendo maior fidelidade a cargas de trabalho típicas de processamento distribuído.

Na configuração do Spark a compressão durante o *shuffle* foi desabilitada, assegurando que o tráfego de rede seja o máximo possível e os efeitos do congestionamento se façam presentes. Também, a aplicação utiliza de operações que forçam a execução das transformações, como a ação *count*, que evita recomputações desnecessárias e força a materialização dos dados. Essa abordagem permite isolar o impacto dos protocolos TCP no desempenho do *cluster*, especialmente com tráfego UDP concorrente.

Os dados foram coletados a partir da interface de usuário da *web* do Apache Spark, utilizada para monitorar aplicações em execução, apresentando informações de cada worker em serviço, tempo tomado para realização de cada estágio, trabalhos ativos e finalizados, dados sobre o ambiente e diversas outras informações. A métrica utilizada para comparação dos resultados foi o tempo de execução de cada trabalho nos cenários propostos.

3. Discussão dos resultados

A Tabela 1 apresenta o resultado dos testes promovidos em três configurações diferentes, realizados em 4 cenários distintos. Nos testes empregando maior tráfego concorrente

Configuração	Cenário		Cubic	Reno	DCTCP
1000 partições 500M linhas	1	(0% UDP)	8,4 min	8,9 min	8,1 min
	2	(100% UDP)	9,5 min	9,4 min	8,8 min
	3	(90% UDP)	10,0 min	9,5 min	9,0 min
	4	(50% UDP)	9,6 min	9,0 min	8,3 min
500 partições 500M linhas	1	(0% UDP)	7,6 min	7,3 min	7,5 min
	2	(100% UDP)	8,2 min	8,2 min	8,0 min
	3	(90% UDP)	8,0 min	7,9 min	8,4 min
	4	(50% UDP)	8,0 min	7,8 min	7,9 min
400 partições 600M linhas	1	(0% UDP)	9,2 min	9,1 min	9,0 min
	2	(100% UDP)	9,8 min	9,9 min	9,5 min
	3	(90% UDP)	10,0 min	10,0 min	10,0 min
	4	(50% UDP)	9,5 min	9,4 min	9,6 min

Tabela 1. Resultados comparativos dos protocolos TCP em diferentes configurações e cenários

de rede (e.g., UDP 100%), o DCTCP demonstrou ser superior, reduzindo o tempo de execução em até 7,4% em relação ao Cubic, graças ao uso de Explicit Congestion Notification (ECN), que prioriza a baixa latência e consegue se adaptar com maior rapidez ao congestionamento. Em contrapartida, nos cenários moderados (e.g., UDP 50%), o Reno apresentou desempenho ligeiramente melhor, com redução modesta de 2% nos tempos de execução. Nessas situações, o algoritmo de controle de congestionamento do TCP Reno utiliza melhor a largura de banda disponível, por não adotar mecanismos preventivos para evitar congestionamento, ação que ocorre por exemplo, com o DCTCP.

Partições maiores resultam em transferências de dados mais volumosas durante o *shuffle*, aumentando a pressão sobre a rede e expondo diferenças na eficiência dos protocolos, a exemplo do terceiro cenário gerando apenas 400 partições para 600 milhões de linhas. Contudo, sob UDP 90%, todos os protocolos obtiveram o mesmo tempo de execução, indicando que a rede atingiu saturação, operando em seu limite de capacidade de banda devido a uma limitação física. Tal fato expõe a importância de ajustar dinamicamente o número de partições, equilibrando paralelismo e utilização da rede, visto que fluxos de dados reais podem sofrer com flutuações drásticas no seu volume de tráfego.

A superioridade do DCTCP nos testes realizados evidencia a relevância em escolher protocolos de controle de congestionamento alinhados às necessidades do ambiente operacional. Projetado para *datacenters*, o DCTCP atua em cenários de alta largura de banda, baixa latência e tráfego intenso, utilizando ECN para detectar congestionamentos precocemente e ajustar dinamicamente a taxa de transmissão, garantindo estabilidade mesmo sob carga extrema. Em contraste, o TCP Cubic - padrão do Linux -, é otimizado para redes de longa distância com latência variável, demandando uma abordagem conservadora. Sua função cúbica, baseada em perdas de pacotes para identificar congestionamentos, subutiliza redes de alta velocidade e reage tardiamente a picos de tráfego, gerando oscilações e gargalos em ambientes especializados, como os simulados. Assim, o estudo reforça que protocolos como o Cubic, são inadequados para infraestruturas críticas, onde eficiência e baixa latência são prioritárias.

4. Conclusão

O crescimento exponencial de dados superou os limites físicos de processamento das tecnologias atuais, tornando essenciais otimizações que manipulam essas informações, para evitar que ineficiência em operações críticas comprometam a escalabilidade e a produtividade dos sistemas. Neste contexto, torna-se fundamental avaliar a eficácia dos algoritmos de controle de congestionamento do TCP.

Os resultados do estudo destacaram superioridade no uso de algoritmos com mecanismos adaptativos de controle de congestionamento, como o DCTCP, entregando resultados superiores frente aos demais protocolos em cenários extremos. Essa vantagem, aliada ao ajuste dinâmico de partições, reforça que refinamentos em camadas de software são essenciais para extrair o máximo de infraestruturas existentes.

Como próximo passo, propõe-se investigar como técnicas de *machine learning* podem ser integradas ao Apache Spark, buscando reduzir o tempo de execução, aumentar a capacidade de processamento proporcional ao número de nós no *cluster* e viabilizar soluções antes impossíveis por limitações de *hardware*. Com isso, espera-se potencializar a redução dos tempos de execução dos trabalhos, oferecendo soluções mais eficientes para lidar com grandes volumes de dados.

Agradecimentos: Este trabalho foi desenvolvido no Laboratório de Processamento Paralelo e Distribuído (LabP2D) da Universidade do Estado de Santa Catarina (UDESC) e recebeu apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e da Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC).

Referências

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. (2010). Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74.

Foundation, T. A. S. (2024). Apache spark documentation. Disponível em: https://spark.apache.org/docs/3.5.4/. Acesso em: 15 jan. 2025.

Ha, S., Rhee, I., and Xu, L. (2008). Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74.

IBM (2021). What is apache spark? Disponível em: https://www.ibm.com/think/topics/apache-spark. Acesso em: 12 jan. 2025.

iPerf (2024). iperf - iperf3 and iperf2 user documentation. Disponível em: https://iperf.fr/iperf-doc.php. Acesso em: 15 jan. 2025.

Jacobson, V. (1988). Congestion avoidance and control. *ACM SIGCOMM computer communication review*, 18(4):314–329.

Salloum, S., Dautov, R., Chen, X., Peng, P. X., and Huang, J. Z. (2016). Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1:145–164. DOI: https://doi.org/10.1007/s41060-016-0027-9.