

# Jellyfish: Networking Data Centers Randomly

Nelson G. Prates Junior

September 30, 2018

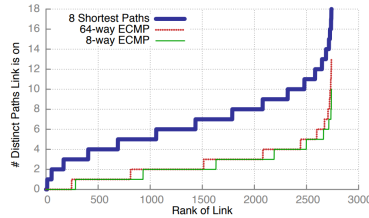
As topologias para *data centers* são projetadas especificamente para que os conjuntos de máquinas lidem com grandes demandas de tráfego, exigindo altos índices de largura de banda. Topologias especiais, como o *Fat-Tree* proposto por [Al-Fares et al., 2008], lidam com esse tráfego melhor que o modelo hierárquico padrão. O *Fat-Tree* é capaz de lidar com o mesmo número de *hosts* por aproximadamente 16% do custo de uma topologia hierárquica [Al-Fares et al., 2008]. No entanto, a estaticidade da topologia *Fat-Tree* gera limitações quando nos referimos a inclusão de novos servidores. Além do *Fat-Tree* outros modelos de topologias propostos na literatura, como [Bhuyan and Agrawal, 1984] e [Popa et al., 2010] apresentam a mesma limitação.

O crescimento da base de usuários ou a implantação de aplicativos que aumentam a demanda de largura de banda motivam a inclusão de mais servidores nas arquiteturas de rede. Esse fato gera uma demanda de topologias que habilitem a capacidade de inclusão novos servidores. O trabalho de [Singla et al., 2012] apresenta o *Jellyfish*, uma topologia para redes de servidores de alta capacidade baseada em um grafo aleatório. Essa topologia aumenta a capacidade de incluir até 25% mais servidores quando comparada com a topologia *Fat-Tree* utilizando os mesmos equipamentos. A topologia *Jellyfish* apresenta um projeto onde cada *host* é conectado a um único *switch* e este *switch* é conectado a todos os outros *switches* da rede usando um grafo de aproximação aleatório.

[Singla et al., 2012] realiza uma série de experimentos para comprovar a eficiência da topologia. Outras topologias também são avaliadas utilizando os mesmos equipamentos. Através disso o autor compara os resultados conseguindo provar a eficiência do *Jellyfish* diante das outras topologias. Nos testes que avaliam as propriedades das topologias foram avaliadas a eficiência, flexibilidade e resiliência a falhas. As principais métricas avaliadas para provar a eficiência foram largura de banda, taxa de transferência e custo em quantidade de portas. Para flexibilidade o autor relata a capacidade de expansão da topologia e avalia o *Jellyfish* através de métricas como custo e bisseção de largura de banda. A avaliação sobre a resiliência a falhas, foram utilizados métricas de taxa de transferência sob falhas de link aleatórias. O *Jellyfish* se prova melhor que outras topologias em todos os quesitos. No entanto, o autor seleciona uma topologia por propriedade para realizar a comparação com o *Jellyfish*, aparentando determinada inconsistência nas avaliações.

Neste trabalho, exploramos as afirmações feitas por [Singla et al., 2012] sobre

as melhorias do *Jellyfish* em comparação com a topologia *Fat-Tree*. O principal objetivo é reproduzir os resultados apresentados pela Figura 1. Para a realização da simulação, realizei pesquisas através da Internet, onde encontrei um repositório <sup>1</sup> as quais inspirei o desenvolvimento dos meus testes. Para a reprodução foi utilizada uma maquina virtual com dois processadores, 4GB de memória RAM com o sistema operacional Ubuntu Server 16.04. As atividades desenvolvidas para a reprodução dos testes envolveram a definição das ferramentas, a construção da topologia e a realização dos testes.



(a) Quantidade de caminhos distintos por link no *Jellyfish* traçados por diferentes protocolos de roteamento

Congestion control	Fat-tree (686 svrs) ECMP	Jellyfish (780 svrs) ECMP	Jellyfish (780 svrs) 8-shortest paths
TCP 1 flow	48.0%	57.9%	48.3%
TCP 8 flows	92.2%	73.9%	92.3%
MPTCP 8 subflows	93.6%	76.4%	95.1%

(b) Resultados da simulação de transferência pacotes para diferentes protocolos de roteamento e congestionamento para *Jellyfish* (780 servidores) e uma árvore de gordura de mesmo equipamento (686 servidores).

Figure 1: Resultados originais do [Singla et al., 2012]

## Definição das ferramentas e Construção da Topologia

Para reproduzir o gráfico referente a Figura 1(a), utilizei um script Python que modela o grafo aleatório regular (do inglês, *Random Regular Graph* - RRG) e as bases de cálculo conforme cada protocolo de roteamento. Para a modelagem do grafo o *script* utiliza a biblioteca *Python Networkx* que oferece uma função que através dos parâmetros de grau e quantidades de nós retorna um RRG. O *script* também ajusta as bordas do grafo conforme especificado pelo *Jellyfish*, realiza os cálculos e a plotagem dos gráficos. Os gráficos foram plotados através da biblioteca *Python Matplotlib*.

A reprodução da Figura 1(b) envolveram a definição de ferramentas para a emulação da estrutura de rede, geração de carga e coleta de dados. Como os testes avaliam a topologia com *hosts* sob o controle dos protocolos TCP e MPTCP. Primeiramente baixei o MPTC e compilei com o *kernel* do sistema operacional. Para emular a estrutura utilizei a ferramenta Mininet que disponibiliza uma API. Então, através disso, seguindo a lógica do grafo modelado e ajustado (como citado anteriormente), o *script* cria a topologia virtualizada. O repositório disponibiliza um controlador POX personalizado para controlar os equipamentos de rede seguindo as lógicas de roteamento dos protocolos ECMP

<sup>1</sup><https://github.com/ahalayini?tab=followers>

e *8-Shortest-Path*. Com a topologia montada, o *script* roda o *iperf* para gerar tráfego entre os *hosts* e coletar as informações dos links.

## Realização dos Testes

Os testes são fundamentados em dois *scripts* Python principais (*script 1* e *script 2*). O *script 1* implementa a modelagem do grafo, os cálculos e a plotagem do gráfico. O *script 2* recebe parâmetros de entrada para emular a topologia, gerar as cargas, coletar e avaliar os dados. Para executá-los eu implementei um terceiro *script* em *SHELL*, ele é responsável por controlar os dois *scripts* principais.

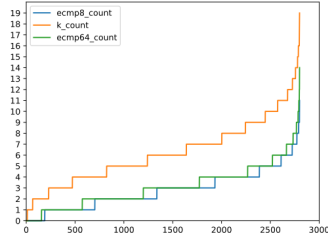
Os parâmetros necessários para o *script 2* são referentes às configurações da topologia, dos testes e da seleção dos protocolos de roteamento. O parâmetro referente à configuração da topologia é a quantidade de *hosts*, ele é representado pela letra "s". O parâmetro que define o protocolo de roteamento recebe valores entre 0 e 2, sendo 0 para o protocolo ECMP8, 1 para o protocolo ECMP64 e 2 para o protocolo *8Shortest Path*, ele é representado pela letra "r". Os parâmetros referentes aos testes são para a configuração da ferramenta *iperf*, são elas; a quantidade de fluxos simultâneos e o tempo de cada teste realizado. A quantidade de fluxos simultâneos é representado pela letra "f", podendo ser um valor entre 1 e 8. O parâmetro referente tempo dos testes define o tempo de cada transmissão entre os *hosts*, ele é representado pela letra "t". A definição dos protocolos de controle de congestionamento é realizada através do sistema operacional antes de cada rodada de teste, compreendendo a ativação ou não do MPTCP.

Então, para rodar os testes basta realizar os seguintes passos:

1. Assegure-se de que o python2 e o pip2 estão instalados.
2. Executar "pip2 install matplotlib"
3. Executar "pip2 install networkx"
4. Executar "git clone git://github.com/mininet/mininet"
5. Executar "mininet/util/install.sh -a"
6. Usando o comando "ls", você constatará diversas pastas e arquivos.
7. Executar "`https : //github.com/ngpjunior/INFO7015_Redes_de_Computadores_TP1`"
8. Executar "`cdINFO7015_jellyfish/pox/ext/`"
9. Executar: `sudo./start.sh`

## Resultados

Os resultados adquiridos são apresentados conforme a Imagem 2. Nota-se que os resultados da Imagem 2(a) é semelhante ao resultado original, comprovando a eficiência do protocolo *8ShortestPath* quando em operação sobre a topologia *Jellyfish*. No entanto, os testes referentes a Imagem 2(b) não foram tão bem sucedidos. Devido a falta de recursos computacionais e possíveis instabilidades entre as ferramentas de simulações, acredito as simulações foram prejudicadas. Não sendo possível executar os testes com 8 fluxos e consequentemente sem o MPTCP. Diante da disponibilidade que tinha no momento, realizei os testes com o número de servidores reduzidos. Então, testei a média de consumo de largura de banda entre os servidores organizados pelas topologias *Fat-Tree* e *Jellyfish* usando os protocolos ECMP e *k-shortest-path*.



(a) Quantidade de caminhos distintos por link no *Jellyfish* traçados por diferentes protocolos de roteamento

		Jellyfish						fatree	
		mptcp2			top1			top1	
		ecmp64	ecmp8	shortest	ecmp64	ecmp8	shortest	ecmp64	
		63,42	82,43	72,53	89,7	92,15	91,14		77,3
		70,37	83,67	64,33	93,94	92,65	93,57		65,54
		81,65	81,1	60,29	92,09	92,15	92,15		70,03
		75,6	83,79	73,53	92,34	91,86	92,94		94,01
	Média	72,76	83,2475	67,67	92,0175	92,2025	92,45		74,22

(b) Jellyfish e Fat-Tree por diferentes protocolos de roteamento

Figure 2: Resultados obtidos a partir das simulações

## References

- [Al-Fares et al., 2008] Al-Fares, M., Loukissas, A., and Vahdat, A. (2008). A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM.
- [Bhuyan and Agrawal, 1984] Bhuyan, L. N. and Agrawal, D. P. (1984). Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on computers*, (4):323–333.
- [Popa et al., 2010] Popa, L., Ratnasamy, S., Iannaccone, G., Krishnamurthy, A., and Stoica, I. (2010). A cost comparison of datacenter network architectures. In *Proceedings of the 6th International Conference*, page 16. ACM.
- [Singla et al., 2012] Singla, A., Hong, C.-Y., Popa, L., and Godfrey, P. B. (2012). Jellyfish: Networking data centers, randomly. In *NSDI*, volume 12, pages 1–6.