2SYN: Congestion-Aware Multihoming

Kfir Toledo[1,2], Isaac Keslassy[1]
[1] *Technion* [2] *IBM Research Haifa*

***Abstract*—When sending flows to arbitrary destinations, current multihoming routers adopt simple congestion-oblivious mechanisms. Therefore, they cannot avoid congested paths.**

In this paper, we introduce 2SYN, the first congestion-aware multihoming algorithm that works for any destination. We explain how it dynamically selects a preferred path for new connections, even given previously-unseen destinations. We further demonstrate that it can be easily implemented in Linux. Finally, in a real-world experiment with either LTE or a wired link, we show how 2SYN dynamically adapts to the quality of the connection and outperforms alternative approaches. Thus, 2SYN helps companies better manage their networks by leveraging their multihoming capabilities.

I. Introduction

To obtain a reliable Internet connection, companies are increasingly abandoning their expensive MPLS (Multiprotocol Label Switching) access, and using multihoming instead [1]. By correctly managing their multihomed network, companies can leverage several low-cost commercially-available internet-access links, such as LTE, DSL, cable, or fiber optics, to build a high-performance and high-availability WAN transport [2]. As Fig. 1 illustrates, companies can use multihoming for any arbitrary website destination D, including public SaaS (Software as a Service) websites like Office365 or Workday.

To decide which of the outgoing WAN links should be used, current multihoming routers adopt *congestion-oblivious* algorithms: Either (1) a static failover algorithm, which uses a preferred WAN link until it is disconnected, then uses a second preferred link, etc.; or (2) a static load-balancing algorithm that load-balances flows across the links [3]. This load-balancing could be random or round-robin, uniform or weighted, but it is always congestion-oblivious. Thus, a large proportion of the connections are always at risk of suffering in a clogged or low-speed link, even when another high-speed link is uncongested.

In this paper, we focus on TCP flows, as they constitute the vast majority of corporate-oriented connections. When the destination D belongs to the corporate network, the company can implement congestion-aware algorithms, e.g., (1) replace TCP by multi-path protocols like MPTCP (Multipath TCP) [2], [4], [5], or (2) use SD-WAN (Software-Defined Wide Area Network) to establish tunnels to D through each outgoing link, monitor the tunnels, and then choose the best link for current network conditions [6], [7]. However, we want our algorithm to work for *any* arbitrary destination D, e.g., any SaaS website. Nothing guarantees that D is configured to accept MPTCP, or that D belongs to the corporate network. Therefore, these restricted algorithms do not apply to the general case. *The goal of this paper is to provide a general congestion-aware flow load-balancing algorithm given any arbitrary destination.*

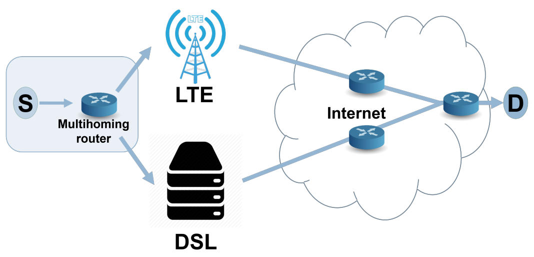


Fig. 1. *Multihoming overview*. 2SYN runs in the multihoming router.

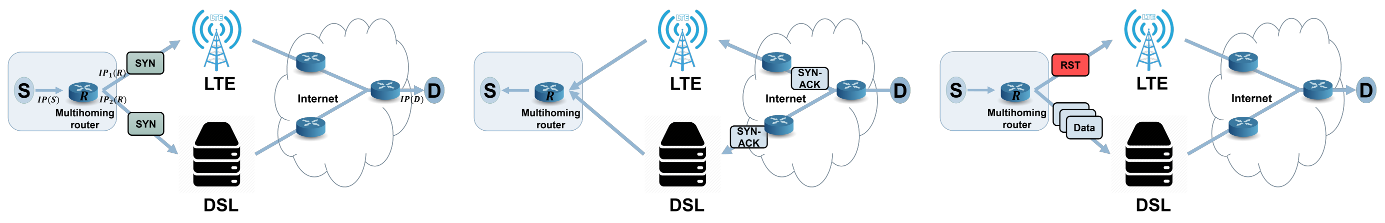
Related work. As mentioned above, there are many solutions for MPTCP [2], [4], [5] and SD-WAN [6], [7]. In particular, RobE [5] sends SYNs over two parallel paths for an accelerated MPTCP handshake mechanism, although it does not later discard one of the SYN-ACKs. However, all these solutions assume that we have control over the destination D. Other works like CPR [8] and the IETF architecture for transport services [9] deal with congestion-oblivious failover routing.

Contributions. We introduce the *2SYN* algorithm. 2SYN leverages the TCP 3-way handshake protocol to determine the path with the best initial RTT. To do so, during the connection establishment to D, the router sends a SYN through each of the relevant router links. It then remembers the router link through which it obtains the first SYN-ACK, and keeps using it to route future packets while telling D to disregard the other paths. We further implement 2SYN in Linux and discuss its implementation tradeoffs. We also introduce alternative approaches based on ML (machine-learning) algorithms and explain why they appear inadequate.

We evaluate the performance of 2SYN both in a lab testbed and in cross-continental transmissions. We show how it adapts to differing path propagation or queueing delays. We also show its resilience to a sudden bandwidth drop. Then, using long-haul transmissions and comparing LTE and DSL links, we confirm that 2SYN tends to pick the best link, and adapts at connection time to artificial bandwidth drops or heavy congestion. We further compare it to alternative static, random, and ML-based approaches. We demonstrate how 2SYN can outperform them and significantly decrease the average FCT (Flow Completion Time) in a dynamic environment.

In summary, the main paper contributions are: (1) The 2SYN multihoming algorithm that dynamically avoids congested paths when establishing the connection to any destination D. (2) The 2SYN implementation in Linux. (3) The real-world experiments where 2SYN outperforms the alternative static, random, and ML-based approaches.

The open-source code of 2SYN is available online [10].



(a) 1st step: duplicate SYN (b) 2nd step: wait for first SYN-ACK (c) 3rd step: pick path and cancel other paths

Fig. 2. Key steps of the 2SYN algorithm.

II. The 2SYN algorithm

A. 2SYN algorithm

We introduce *2SYN*, a congestion-aware load-balancing algorithm for a multihoming router R. The router R is connected to k different WAN links (e.g., DSL or LTE), so each flow from S to D can take k different possible paths. We define a flow using its 5-tuple, and only consider TCP flows.

Key idea. 2SYN tries to choose the path with the shortest initial RTT by leveraging the TCP 3-way handshake protocol. When opening a TCP connection, it sends a SYN on each path, and only keeps the connection on the path through which it receives the first SYN-ACK, while canceling the other paths.

Initialization. We assume that 2SYN manages an AF (Active Flow) table, which lists the current active flows that pass through the multihoming router. AF is initially empty. Then, as illustrated in Fig. 2, 2SYN performs the following steps:

1. Duplicate SYN. As shown in Fig. 2(a), for every new TCP flow, i.e., for every new SYN that reaches the router R from some source S within the corporate branch, 2SYN duplicates this SYN to the k paths. As in the case without 2SYN, since there are several router links with distinct ISPs, the 2SYN router performs NAT (Network Address Translation) to replace the source IP address of each SYN by the IP address of the corresponding router link. For example, in Fig. 2(a), the source IP address of the top SYN is $IP_1(R)$ instead of $IP(S)$.

2. Wait for the first SYN-ACK. Later, the destination D receives the k SYNs. Since each SYN has a different source IP, D treats them as k independent new-flow requests. Therefore, as Fig. 2(b) illustrates, it sends a SYN-ACK for each of these flows along their respective paths.

3. Pick a path and cancel the other paths. When the first SYN-ACK reaches the router, 2SYN chooses the corresponding link as the preferred link for this flow. (1) As shown in Fig. 2(c), 2SYN sends RST messages to D via the other $k-1$ links, causing D to cancel them. (2) 2SYN updates the routing table to route the remaining packets through the chosen link, and adds the flow to the AF table.

4. Flow completion. 2SYN then waits for the flow to complete in its selected path (starting with a FIN or RST flag from S or D). When the flow completes (or a timeout expires), 2SYN removes it from the AF table and deletes its rule from the routing table.

While we described the algorithm for the general case of k paths, we focus on $k = 2$ in the remainder for simplicity.

B. 2SYN implementation in Linux

A recent report unpacked 89% of the measured home routers and found that all of them were based on Linux [11]. Given this Linux ubiquity, we develop a proof-of-concept implementation of 2SYN in Linux [10]. For any packet that reaches the router, 2SYN uses iptables to check whether it is a SYN packet for a new TCP flow from S to D. Specifically, given any packet with a SYN flag, it checks whether its flow satisfies the following four conditions: (1) It does not already have a specific rule in the Linux routing table, (2) it is a TCP flow, (3) it comes from a pre-defined router interface corresponding to the corporate branch, and (4) it is not destined to a pre-defined corporate gateway router through an existing tunnel. If all conditions are satisfied, 2SYN uses iptables to automatically duplicate the SYN packet to all outgoing router interfaces and update the corresponding source IP. Else, it is routed with the standard Linux route rules.

We use the Python Scapy API [12] to monitor traffic. To accelerate packet processing, Scapy API filters packets using BPF (Berkeley Packet Filter) [13] and only monitors packets with set SYN, SYN-ACK, FIN and RST flags. When a SYN arrives, 2SYN knows that it needs to wait for its first SYN-ACK. Then, when the first SYN-ACK arrives, it adds the flow to the AF table and the Linux routing table. Finally, with a FIN or RST, 2SYN removes the flow from both the AF table and the Linux routing table.

In addition, we implemented the option to redefine a flow as a (source IP, dest. IP) pair, neglecting ports and protocol. This allows sharing routing decisions among flows with the same (source IP, dest. IP) pair, reducing SYN duplication rates at the cost of a more complex table management.

C. Overhead

Router overhead. To minimize the computation overhead, our 2SYN implementation delegates to the Linux router both the path routing computation and the SYN duplication operation, both with negligible time. The only non-negligible computation overhead is in updating the Linux routing table, either to add a new flow or to remove a completed flow. Using our simple Python Scapy implementation, it takes some 4 ms on average. In addition, adding and removing entries in the AF table is done in an $O(1)$ negligible time by using the hash-table data structure. Using BPF, the 2SYN algorithm processes only packets with the SYN, SYN-ACK, FIN, and

RST flags, so even for high flow rates, the processing time is negligible compared to the update time of the Linux routing table. We checked with mpstat [14] the CPU usage overhead when processing a sudden burst of 100 parallel SYN arrivals. At peak time, the CPU overhead is 3%.

Traffic overhead. For each new TCP flow, 2SYN creates one additional SYN (i.e., two SYNs instead of one), one SYN-ACK, and one RST (for the non-selected path). This appears to be negligible in practice.

Destination overhead. For each new flow, we send an additional SYN to D, and cancel it with a RST after one RTT. Thus, we roughly increase the number of half-open connections at D by RTT/FCT_D, where FCT_D denotes the average FCT as seen by D. Assuming a median FCT_D of about 100 RTTs (the median flow lasts about 6 sec. [15]), the overhead is around 1%, which seems reasonable.

NAT overhead. In 2SYN, the NAT overhead is negligible: we translate 2 SYN/SYN-ACK messages per flow instead of 1, and proceed with other packets without change.

III. Alternative approach

We suggest an alternative lightweight ML-based approach, assimilating the problem to a Multi-Armed Bandits (MAB) problem. In the MAB setting, k slot machines offer random rewards, drawn from k unknown distributions. The goal is to maximize the cumulative reward by selecting machines wisely. Similarly, in our routing problem, the k paths represent slot machines, offering different FCTs. We aim to choose the path with the lowest FCT (highest reward), balancing exploitation (choosing paths with historically low FCTs) and exploration (sampling less-traveled paths). We consider three standard MAB algorithms: *ϵ–greedy*, *Upper Count Bound (UCB)*, and *Thompson sampling* (see Burtini et al. [16]). For instance, for some small $\epsilon > 0$, ϵ–greedy chooses the historically-optimal path with a probability of $1 - \epsilon$, and explores other paths uniformly at random otherwise. These MAB algorithms are lightweight, do not require dedicated hardware (e.g., GPUs), and do not need pre-training for each possible D.

Limitations. MAB algorithms suffer from several limitations. First, the analogy between slot machines and paths is misleading: Choosing a path reduces its future performance due to the increased congestion. This differs from the usual MAB setting. Second, as we illustrate in the experiments (Sec. IV), the path properties can vary with time in a non-stationary way, while MAB algorithms are not very reactive, which hurts future connections. Lastly, we would need to maintain a large database of flows with their past performances for each D.

IV. Experiments

A. Lab experiments

We first evaluate 2SYN in a lab testbed, then in real-world conditions. In the following lab testbed experiments, we start by studying the impact of propagation delay and queueing delay. Then, we study the resilience to bandwidth drops.

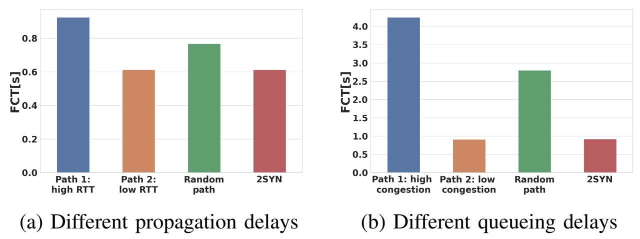


Fig. 3. *Paths with different delays.* Average FCT in a 1-MB file download using (a) different propagation delays, and (b) different queueing delays.

Setting. We use five Linux servers with Ubuntu 20.04. As in Fig. 1, they emulate the source S (iPerf3 client), the multihoming router, the two paths using two servers (path routers), and the destination D (iPerf3 server).

Tools. In all experiments, we use the iPerf3 (version 3.9) [17] network measurement tool to run the tests, download/upload a file and measure the average FCT. We also use the NetEm [18] network emulation tool to control the RTT and bandwidth of each path. The buffer size of each path router of bandwidth BW is set to $RTT \cdot BW$.

Algorithms. We compare four different algorithms. The first always chooses the first path, and the second chooses the second path. The third chooses the path randomly with equal probabilities. The last algorithm is 2SYN.

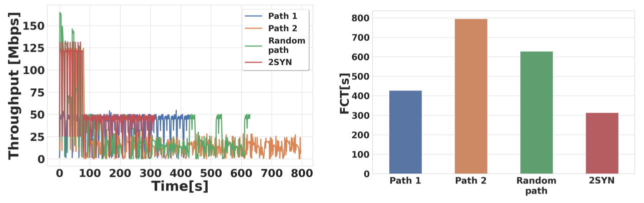
Impact of propagation delay. For each WAN path, we set the same bandwidth of 300 Mbps, but a different propagation delay: 120 ms for the first path and 80 ms for the second. Then, we measure the FCT to download a 1 MB file from D (a 100 MB file download yielded similar results). We repeat the experiment 20 times and measure the average download time. Fig. 3(a) illustrates the results. 2SYN always chooses the best path with the lowest propagation delay, providing FCTs that are comparable to those of path 2.

Impact of queueing delay. We now set both paths with the same bandwidth of 300 Mbps and RTT of 120 ms, but add background TCP traffic (five flows with a limit of 100 Mbps per flow) to path 1. We measure the average download time for a 1 MB file, using 20 experiments. Fig. 3(b) shows that 2SYN manages to choose the path with less congestion, again providing FCTs comparable to those of path 2.

Resilience to bandwidth drop. In Fig. 4, we download 100-MB files 20 times in a row. We reduce the bandwidth of path 2 from 300 Mbps to 30 Mbps after downloading 40% of the files, while that of path 1 remains constant at 100 Mbps. Fig. 4(a) shows how the throughput varies with time for all algorithms. The throughput oscillates due to TCP's slow start for each new flow. Significantly, 2SYN chooses the high-bandwidth path for the first connections, while after the bandwidth drop, it adopts the other path for new connections. Fig. 4(b) shows that it achieves a lower FCT than any constant-path algorithm.

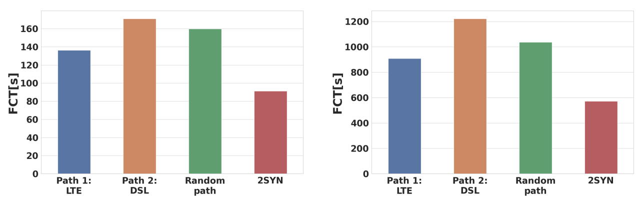
B. LTE vs. DSL experiments

Setting. We download or upload files to public SpeedTest servers in England [19], with our source S at the Technion,



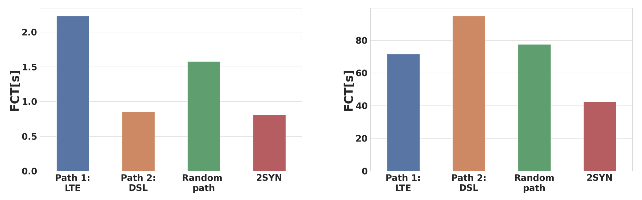
(a) Throughput over time (b) FCT

Fig. 4. *Resilience to bandwidth drop.* Path 1 has a constant bandwidth of 100 Mbps, while that of path 2 suddenly drops from 300 Mbps to 30 Mbps.



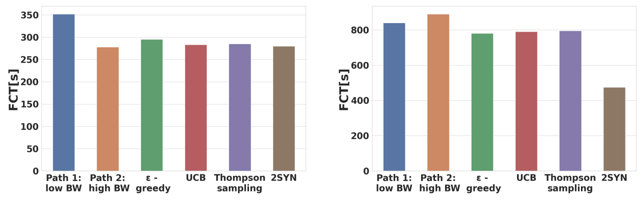
(a) Web search: download link (b) 30 MB file: upload link

Fig. 6. *LTE (path 1) vs. DSL (path 2) with sudden congestion*, due to other users on the DSL link.



(a) Links without issues (b) Links with bandwidth drop

Fig. 5. *LTE (path 1) vs. DSL (path 2).* Average FCT for the download of web-search files.



(a) Without bandwidth drop (b) With bandwidth drop

Fig. 7. *2SYN vs. MAB ML algorithms.* (a) With constant settings, MAB algorithms learn the best path and achieve results similar to 2SYN. (b) When bandwidth drops, 2SYN outperforms MAB algorithms since it adapts faster.

Israel. We connect our source router to (1) a standard cell phone with a 4G LTE connection; and (2) a wired internet link that is throttled to 100 Mbps for download and 10 Mbps for upload, based both on the median offered DSL package across all providers and on the median measured broadband bandwidth [20]. We run two workloads: (1) sending a *large file* of 30 MB for 20 times, or (2) sending *web-search* application traffic with a distribution of 62% for small files (< 100 KB), 18% for medium files (100 KB$-$1 MB), and 20% for large files (> 1 MB) [21]. We test three scenarios: one without issues; one with a bandwidth drop in the middle of transmission, modeling a sudden path router or ISP problem; and one with a sudden congestion increase caused by additional users in the middle of transmission. We get 2 transfer modes $\times$ 2 workloads $\times$ 3 scenarios, i.e., 12 experiments. We find that all results are similar, and show 4 typical ones.

Links without issues. Fig. 5(a) shows how in all cases, 2SYN keeps choosing the better DSL link as expected.

Links with bandwidth drop. Fig. 5(b) illustrates a case where we arbitrarily drop the DSL download bandwidth to 5 Mbps after 40% of the files are sent. 2SYN recognizes when the DSL bandwidth drops and switches to the LTE path for new flows. Its performance is better than sticking to any path.

Links with heavy congestion. Fig. 6 shows the impact of setting a significant congestion after 40% of the files are sent, by adding ten TCP flows to the DSL link to model congestion from other users. 2SYN quickly abandons the congested DSL link, thus yielding the best performance.

C. Why not use ML?

We compare 2SYN to the MAB ML algorithms (Sec. III) in the lab testbed. We set the bandwidth to 200 Mbps for the first path and 300 Mbps for the second. We send traffic on five (source, dest.) pairs (S_i, D_i), thus forming five learners.

Fixed settings. At first, for each pair, we send 10-MB files 50 times in a row. After a short time, the MAB algorithms correctly classify the path with the lowest bandwidth. Fig. 7(a) illustrates how they converge to near-optimal results.

Bandwidth drop. We now set the same initial bandwidth, but after 40% of the files are sent, we drop the bandwidth of path 2 from 300 Mbps to 100 Mbps. We send 20 files in a row, with a size of 200 MB per file. Fig. 7(b) shows that 2SYN clearly outperforms the MAB algorithms. The main reason is that the MAB algorithms base each decision on the path history, and therefore are less resilient to changes in the path conditions, while 2SYN adapts much more quickly. Real-time congestion-aware observations outperform history-based learning.

V. Conclusion

This paper has introduced 2SYN, a lightweight congestion-sensitive algorithm that can help companies efficiently manage their multi-homed networks. It has presented a Linux implementation of 2SYN. Then, using lab and real-world experiments, it has shown how 2SYN can adapt to various network conditions and outperform alternative algorithms.

Acknowledgment

Many thanks to Aran Bergman for his useful comments. This work was partly supported by the Israel Science Foundation (grant No. 1119/19), the Hasso Plattner Institute Research School, and the Louis and Miriam Benjamin Chair in Computer-Communication Networks.

References

[1] Brian Washburn. (2022) Why global enterprises move from MPLS to Internet. [Online]. Available: https://www.expereo.com/blog/6-reasons-why-global-enterprises-move-from-mpls-to-internet

[2] S. Deng *et al.*, "WiFi, LTE, or both? Measuring multi-homed wireless internet performance," in *ACM IMC*, 2014.

[3] DrayTek. (2025) Load-balancing. [Online]. Available: https://www.draytek.co.uk/information/solutions/load-balancing

[4] M. Baerts *et al.*, "Leveraging the 0-RTT convert protocol to improve wi-fi/cellular convergence," in *ACM/IRTF ANRW*, 2021.

[5] M. Amend *et al.*, "RobE: Robust connection establishment for multipath TCP," in *ACM/IRTF ANRW*, 2018.

[6] O. Michel and E. Keller, "SDN in wide-area networks: A survey," in *IEEE SDS*, 2017, pp. 37–42.

[7] I. Fajjari, N. Aitsaadi, and D. E. Kouicem, "A novel SDN scheme for QoS path allocation in wide area networks," in *IEEE Globecom*, 2017.

[8] R. Landa, L. Saino, L. Buytenhek, and J. T. Araújo, "Staying alive: Connection path reselection at the edge," in *Usenix NSDI*, 2021.

[9] A. Brunstrom, G. Fairhurst, and C. Perkins, "An architecture for transport services," *IETF, Internet Draft draft-ietf-taps-arch-13*, 2022.

[10] 2SYN. (2025) Project git. [Online]. Available: https://github.com/kfirtoledo/2SYN-Multihoming

[11] J. vom Dorp and R. Helmke, "Home router security report 2022," *Fraunhofer*, 2022.

[12] Scapy API, 2025. [Online]. Available: https://scapy.readthedocs.io/en/latest/index.html

[13] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX winter*, vol. 46, 1993.

[14] Mpstat, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/mpstat.1.html

[15] S. Bauer, B. Jaeger, F. Helfert, P. Barias, and G. Carle, "On the evolution of internet flow characteristics," in *ANRW*, 2021.

[16] G. Burtini *et al.*, "A survey of online experiment design with the stochastic multi-armed bandit," *arXiv:1510.00757 preprint*, 2015.

[17] iPerf3: Measuring network performance tool, 2025. [Online]. Available: https://iperf.fr/iperf-download.php

[18] NetEm, 2025. [Online]. Available: https://wiki.linuxfoundation.org/networking/netem

[19] SpeedTest Servers, 2025. [Online]. Available: https://as62240.net/speedtest

[20] SpeedTest Global Index: Median Country Speeds, 2025. [Online]. Available: https://www.speedtest.net/global-index

[21] J. Huang *et al.*, "QDAPS: Queueing delay aware packet spraying for load balancing in data center," in *IEEE ICNP*, 2018, pp. 66–76.