Reproducing “FlashRoute: Efficient Traceroute on a Massive Scale”

Ziqi Zhao, Zhengdong Wang, Liangtai Sun, Hao Yin
Shanghai Jiao Tong University
{bugenzhao,lnwzd2009,slt19990817,1163706928}@sjtu.edu.cn

1 INTRODUCTION

Traceroute is one of the basic tools for internet measurements. It is widely used to troubleshoot network issues and reveal the network topology. In its common applications, traceroute is used to explore the routing topologies for a large number of targets or even the entire internet [9, 10, 13, 14]. Doing so requires traceroute measurements on a massive scale, which means completing these measurements with as few probes and in as little time as possible.

Shortening the time for topology measurements is particularly critical, because the shorter the time to complete the measurement, the closer the result is to the snapshot and the easier it is to understand the dynamics of Internet routing changes at fine time granularity.

Huang et al.[8] proposes a new tool for massive tracerouting, FlashRoute, which further reduces the time required for tracerouting the entire /24 IPv4 address space, by reintroducing state into the probing process.

Its efficiency can be attributed to several factors. First, before the main scan, FlashRoute attempts to calculate the hop-distance to each destinations using a single probe, and then use these results to optimize the rest probing. Second, Flashroute encodes all information necessary into the header of probes. Third, FlashRoute adopts backward direction and stop probing methods when detecting the route convergence, which is originated in Doubletree [6].

In this paper, we reimplement FlashRoute as *flashroute.rs*, using a modern system programming language *Rust*, and achieve an even better performance in much fewer lines of code. Also, we use visualization technique to show the network topology of a local area network, which makes the result much more vivid.

To recap, the main contributions of our work are:

- We reimplement FlashRoute as *flashroute.rs* using *Rust*, and reduce the number of LoC by 50%.
- We make the probing process much more efficient, with performance improvement up to 40%.
- We adopt visualization method to show the network topology, which is much more vivid and easy to analyze the dynamic network state.

2 RELATED WORK

A number of solutions have been developed to improve the performance of traceroute since it was introduced. In particular, Doubletree [6] and its variants [5, 11] utilize the observation that the routes originated from a vantage point to the internet form a tree-like structure to reduce probing redundancy. Augustin et al. proposed Paris traceroute to reduce the interference caused by load-balancers in route exploration and also invented the multi-path detection algorithm (MDA) to explore alternative routes created by the load-balancers [1, 2]. CAIDA has an experiment to continuously measure the internet topology from distributed vantage points by Scamper [12], which utilizes both Doubletree and Paris traceroute techniques. Although these solutions improve traceroute in various ways, they are still based on the same probing technique as in the conventional traceroute: probes are issued sequentially to every hop for each route, issued probes are remembered and matched to the responses, and expiration of pending probes is managed separately for each probe to handle missing responses.

Yarrp [3] introduced a new probing technique inspired by ZMap [7], which greatly increases the probing parallelism and thereby the probing rate. Later, Yarrp6 [4] enhanced Yarrp by adding the "fill mode". Both Yarrp and Yarrp6 use a stateless probing design, allowing the probing to occur with negligible amount of state. However, this design also increases the volume needed for probes. As the state is removed, the scaner cannot utilize the feedback from the responses efficiently to adjust probing strategy. Therefore, both Yarrp and Yarrp6 explore topology largely in an exhaustive manner by sending probes to every possible hops for every destination.

FlashRoute [8] makes use of two types of optimizations of traceroute and avoids their drawbacks: the optimization with low probing parallelism but low measurement traffic, and the optimizations with high probing parallelism but high measurement traffic. As a consequence, FlashRoute can significantly reduce the time for a full IPv4 address space scan.

3 SYSTEM DESIGN

3.1 FlashRoute

While traceroute is a powerful networking tool, it is not practical to apply it on massive-scale networks, for example, the entire internet. There are several critical issues. First, traditional traceroute is too slow since it sends and receives packets in a synchronous way, where blocking occurs frequently due to a considerable number of unresponsive routers. Also, the traditional method of sending probes has great redundancy. We may force some routers to respond a huge number of times, which is believed too aggresive and may even be mistaken as DoS attack.

Flashroute combines several techniques for increasing the efficiency of the overall probing. Specifically, there are 3 key techniques we need to talk here.

Asynchronous Send & Receive. To discover the route towards some target, traceroute needs to send several probe packets with different TTLs. Traditional traceroute takes an one-by-one approach for TTLs, sending the next probe only if a response from the previous probe is successfully received, or if a timeout is detected. This is because, in general, we cannot decide which probe packet the ICMP response we receive corresponds to.

FlashRoute apply the **probe encoding** technique to cleverly encode all needed information to interpret the measurement, called *probing context*, into the probe packet header. According to the RFC standard, ICMP responses from the routers will return the information verbatim in their payloads, which enables us to match the responses with probes, and then allows us to dramatically speed up our work by sending and receiving asynchronously.

Doubletree. Traditional traceroute is not designed for working on massive-scale networks. If we simply apply it on all of the targets one by one, it is not difficult to see that a large number of intermediate nodes are repeatedly probed, due to the tree-like topology in most cases.

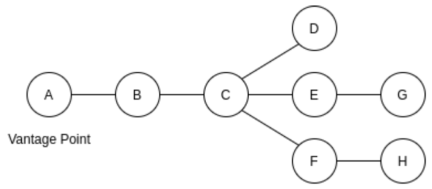

Figure 1: An ideal topology

FlashRoute chooses *Doubletree* manner as the probing strategy. Take the ideal topology in figure 1 for example. For every target, we may select a split point (or split TTL) first like 'C'. Then, do forward and backward probing from the split point simultaneously, while maintaining a *stop set* during the backward process. The next time for probing another target, if we encounter a node that has already been visited in the *stop set*, we assume that the following route is known and there's no need to do any redundant probings.

FlashRoute selects the split point as far away as possible and prefers backward probing, which reduces redundancy to a minimum. A problem is that some routes may be missed in real-world topology using such method. Considering that we are working on massive-scale networks, this has proved not to be really a problem.

Preprobing. Doubletree is a sensible approach, but requires us to find the proper split point for each target. If the split TTL is too large, even far beyond the real distance to reach the target, lots of probing will be meaningless and can be mistaken for a DoS attack. On the contrary if too small, we will miss the probes-saving opportunities that backward probing exploits.

FlashRoute uses an additional preprobing step to decide them. It presents an one-probe hop-distance measurement approach to quickly estimate the distances to targets, called *triggered TTL*s, which can then be used for their split TTLs. Besides, FlashRoute applies a method called *proximity span* to predict the distances to unresponsive routers. Experiments show that the results estimated in the preprobing phase reach a high accuracy, compared with the real measured distances.

3.2 *flashroute.rs*

The original FlashRoute is written in C++14 with Boost and Abseil libraries, which works well but may still be not modern, concise, or safe enough. We re-implement FlashRoute in Rust, a modern system programming language, and name it as *flashroute.rs*.

Why Rust? The reasons we choose Rust are…

- As a modern system programming language, Rust is blazingly fast and memory-efficient.
- Rust's unique ownership model guarantees memory-safety and thread-safety for concurrent applications.
- Rust has a great community which provides a lot of high-performance 3rd-party libraries for us to minimize the line of codes.

Redesign in the view of Task. Considering that FlashRoute is composed by several separated task, we redesign the structure of program in the view of *task*, which is shown in Figure 2. All of the tasks are running in lightweight, coroutine-level parallelism, which make full use of the performance of modern multi-core processors.

Compared with the original implementation, *flashroute.rs* **avoids any use** of mutex or rwlock. All inter-task communications are achieved through message channels or atomic operations, which reduces overheads to a minimum and also makes the program much safer and readable.

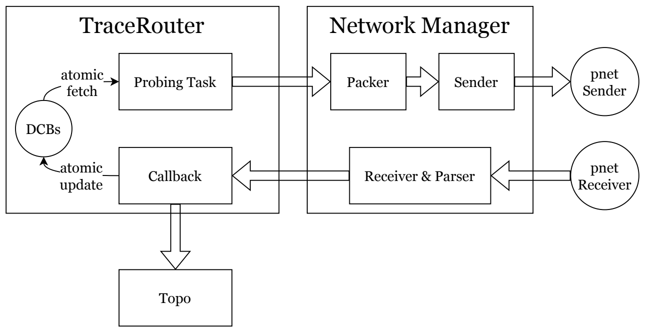

Figure 2: Structure of *flashroute.rs*

Swiss Table Based Data Structure. Originally, probing targets are organized as an array-based implicit linked list, which is enough for the purpose of only scanning /24 internet, but still with poor scalability.

flashroute.rs uses Swiss table, a high-performance hash table, to organize the targets. This design allows customizing probing grain, for example, supporting finer-grained probes in a relatively small subnet using the same algorithm or code, with much lower memory usage compared to the original implementation.

Human-readable Output. Since *flashroute.rs* provides the ability to probe on any subnet, we also propose a method to assemble the traceroute results into a graph, which provides a intuitive display of the discovered network topology. Our *flashroute.rs* is able to dump the graph in *dot* format, and even generate visualization for it. See the next section for details.

4 RESULT

We successfully reimplement the original FlashRoute as *flashroute.rs* using Rust and test its performance in multiple aspects. First, we evaluate *flashroute.rs* by comparing with representative existing alternatives and original FlashRoute. Second, we test the performance difference between the two methods in detail under several identical conditions. It is worth mentioning that we run the tests with different CPU configurations. At last, we provide network topology visualization based on probing results of *flashroute.rs.*

Overall Performance. We test the performance on single target probing from 59.78.X.X to 115.159.1.233 with different existing methods. The results are shown in table 1. Our *flashroute.rs* outperforms all other tools, which runs about **twice** as fast as original FlashRoute.

Table 1: Performance of Single Target Probing

Tool	Probe Time
flashroute.rs	**0:08.34**
FlashRoute	0:15.19
Scamper	0:50.55
traceroute	0:55.10
Yarrp	1:01.51

4.1 Comparison with FlashRoute

In this part, we first test the difference between the two methods at probing rates of 100 Kpps and no limit. And then we evaluate the performance with different CPU configurations. At last, we test the impact of two features: *redundancy removal* during backward probing, and *gap limit.*

Probing rate. We exactly test two different probing rates. First, we evaluate the performance at the probing rate of 100 Kpps with FlashRoute and our *flashroute.rs.* From results shown in table 2 where the suffix means the default split TTL, we can see that *flashroute.rs* runs at the speed which is much closer to the expected probing rate.

Table 2: Performance at Probing Rate of 100 Kpps

Tool	Interfaces	Probes	Scan Time	Probing Rate
flashroute.rs-16	822,527	180,282,633	30:20	99,056
FlashRoute-16	824,923	162,546,005	27:24	98,872
FlashRoute-16*	812,403	97,807,092	17:17	94,317
flashroute.rs-32	818,299	376,085,798	1:02:38	100,076
FlashRoute-32	820,745	342,091,698	57:07	99,822
FlashRoute-32*	807,588	159,185,459	27:32	96,359
Scamper-16	805,970	217,940,057	6:43:53	8,993

The results marked with * are collected from the paper, which performs much better since it preprobes the targets according to the Census Hitlist dataset, instead of a randomly-generated one.

And then we try to test under maximum probing rate to evaluate the performance difference of *flashroute.rs* and the original version. From results shown in table 3, we can see that the performance of our method is always better than original under different grains (8, 16, 32) with more probes. Especially under the grain of 8, the probing rate of *flashroute.rs* is even 50% higher than that of the original version. As a consequence, *flashroute.rs* utilizes the probing time more efficiently.

Utilization of multicore processors. In this part, we test the performance of *flashroute.rs* in comparison with the original version under two CPU configurations.

- 4 CPUs: Intel Xeon (4) @ 2.0 GHz
- 8 CPUs: AMD EPYC 7B12 (8) @ 2.25 GHz

Table 3: Performance at Maximum Probing Rate

Tool	Interfaces	Probes	Scan Time	Probing Rate
flashroute.rs-8	840,437	136,655,629	**9:09**	**248,917**
FlashRoute-8	847,343	127,698,272	12:39	168,245
flashroute.rs-16	834,601	180,874,492	**12:01**	**250,866**
FlashRoute-16	837,672	163,091,546	15:41	173,317
flashroute.rs-32	830,889	377,175,922	**22:52**	**274,910**
FlashRoute-32	835,343	342,231,468	30:29	187,113
Scamper-16	805,970	217,940,057	6:43:53	8,993

From table 4, we can see that the probing rate of our method is 15% higher than the former one, because our method can make better use of the advantages of multiple CPUs and improve the computational efficiency.

Table 4: Performance for Different CPU Configurations

Tool	CPUs	Interfaces	Probes	Scan Time	Rate
flashroute.rs	4	803,248	203,197,113	**21:00**	**161,267**
FlashRoute	4	804,711	185,251,917	24:54	123,997
flashroute.rs	8	834,601	180,874,492	**12:01**	**250,866**
FlashRoute	8	837,672	163,091,546	15:41	173,317

4.2 Impact of some features

In this part, we mainly test the impact of two important features, redundancy removal during backward probing, and the gap limit.

Redundancy Removal. As shown in the table 5, we compare the performance difference with and without RR (Redundancy Removal) option. We can find that the number of unnecessary probes can be significantly reduced with RR on, which indirectly improves the performance of *flashroute.rs* while there is no significant reduction in the number of detected interfaces.

Table 5: Impact of Redundancy Removal during Backward Probing

Tool	RR[1]	Interfaces	Probes	Received	Time
flashroute.rs-8	On	840,437	**136,655,629**	**26,746,697**	**9:09**
flashroute.rs-8	Off	838,082	204,739,320	60,359,387	12:54
flashroute.rs-16	On	834,601	**180,874,492**	**14,491,139**	**12:01**
flashroute.rs-16	Off	845,655	266,010,541	60,904,277	18:39

Gap limit. Then we test the impact of gap limit on our method *flashroute.rs* whose results are shown in Fig 3. We can see that the probing time and the number of detected interfaces increase as the value of Gap-limit increments.

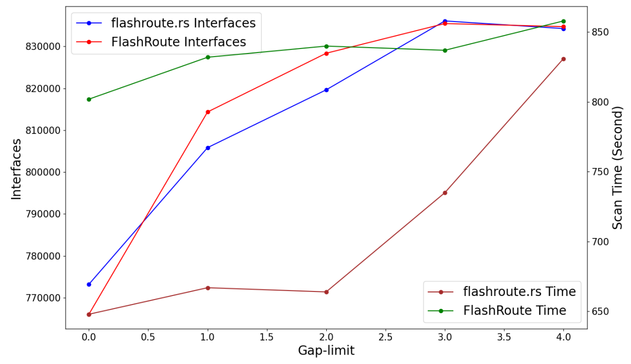

Figure 3: Impact of Gap-limit

4.3 Topology Visualization

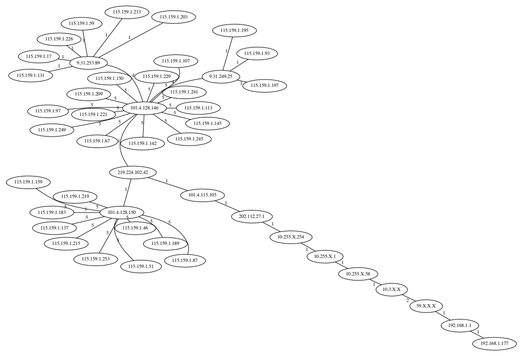

Figure 4: Topology of some hosts @115.159.1.0/24

Figure 4 illustrates the network topology of some hosts in subnet 115.159.1.0/24. From the figure, we can see that the hosts indeed form a tree-like structure as we mention before.

5 CONCLUSION

In this project, we reimplement FlashRoute as *flashroute.rs* in Rust, which reduces the number of LoC and improves the performance significantly. Furthermore, we also present a visualization method for network topology, making the analysis of dynamic network state more intuitive. Finally, we successfully reproduce the results in FlashRoute paper.

APPENDIX A CONTRIBUTION MATRIX

Item	Z. Zhao	Z. Wang	H. Yin	L. Sun
Coding	●			
Code Review		●	●	●
Code Test	●	●		
Experiment	●	●		
Data Processing	●	●	●	●
Slides	●	●	●	●
Report	●	●	●	●
Google Cloud	●			

REFERENCES

[1] Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clémence Magnien, and Renata Teixeira. 2006. Avoiding Traceroute Anomalies with Paris Traceroute. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC '06)*. Association for Computing Machinery, New York, NY, USA, 153158. https://doi.org/10.1145/1177080.1177100

[2] B. Augustin, T. Friedman, and R. Teixeira. 2007. Multipath tracing with Paris traceroute. In *2007 Workshop on End-to-End Monitoring Techniques and Services*. 1–8. https://doi.org/10.1109/E2EMON.2007.375313

[3] Robert Beverly. 2016. Yarrp'ing the internet: Randomized high-speed active topology discovery. In *Proceedings of the 2016 Internet Measurement Conference*. 413–420.

[4] Robert Beverly, Ramakrishnan Durairajan, David Plonka, and Justin P Rohrer. 2018. In the IP of the beholder: Strategies for active IPv6 topology discovery. In *Proceedings of the Internet Measurement Conference 2018*. 308–321.

[5] Benoit Donnet, Bradley Huffaker, Timur Friedman, and K. C. Claffy. 2007. Increasing the Coverage of a Cooperative Internet Topology Discovery Algorithm. In *Proceedings of the 6th International IFIP-TC6 Conference on Ad Hoc and Sensor Networks, Wireless Networks, next Generation Internet (NETWORKING'07)*. Springer-Verlag, Berlin, Heidelberg, 738748.

[6] Benoit Donnet, Philippe Raoult, Timur Friedman, and Mark Crovella. 2005. Efficient algorithms for large-scale topology discovery. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 327–338.

[7] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. 2013. ZMap: Fast Internet-wide scanning and its security applications. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 605–620.

[8] Yuchen Huang, Michael Rabinovich, and Rami Al-Dalky. 2020. FlashRoute: Efficient Traceroute on a Massive Scale. In *Proceedings of the ACM Internet Measurement Conference*. 443–455.

[9] Young Hyun, Bradley Huffaker, Dan Andersen, Matthew Luckie, and Kimberly C Claffy. [n. d.]. The IPv4 Routed/24 Topology Dataset, 2014. *URL: http://www. caida. org/data/active/ipv4_routed_24_topology_dataset. xml* ([n. d.]).

[10] Yuchen Jin, Sundararajan Renganathan, Ganesh Ananthanarayanan, Junchen Jiang, Venkata N Padmanabhan, Manuel Schroder, Matt Calder, and Arvind Krishnamurthy. 2019. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM Special Interest Group on Data Communication*. 104–116.

[11] B. Li, J. He, and H. Shi. 2008. Improving the Efficiency of Network Topology Discovery. In *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*. 189–194. https://doi.org/10.1109/GPC.WORKSHOPS.2008.34

[12] Matthew Luckie. 2010. Scamper: A Scalable and Extensible Packet Prober for Active Measurement of the Internet. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. Association for Computing Machinery, New York, NY, USA, 239245. https://doi.org/10.1145/1879141.1879171

[13] Yuval Shavitt and Eran Shir. 2005. DIMES: Let the Internet measure itself. *ACM SIGCOMM Computer Communication Review* 35, 5 (2005), 71–74.

[14] Neil Spring, Ratul Mahajan, and David Wetherall. 2002. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 133–145.