Reproducing "FlashRoute: Efficient Traceroute on a Massive Scale[1]"

Ziqi Zhao
Zhengdong Wang
Hao Yin
Liangtai Sun

December 14, 2020

[1] Yuchen Huang, Michael Rabinovich, and Rami Al-Dalky. 2020. FlashRoute: Efficient Traceroute on a Massive Scale. In ACM Internet Measurement Conference (IMC '20), October 27–29, 2020, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3419394.3423619

Table of Contents

1. Motivation
    - Traceroute
    - FlashRoute
2. Implementation
3. Results
    - Performance
    - Impact of Some Features
    - Topology Visualization
4. Appendix

Table of Contents

1 Motivation
- Traceroute
- FlashRoute

2 Implementation

3 Results
- Performance
- Impact of Some Features
- Topology Visualization

4 Appendix

Table of Contents

1. Motivation
   - Traceroute
   - FlashRoute
2. Implementation
3. Results
4. Appendix

Traceroute

- Traditional traceroute

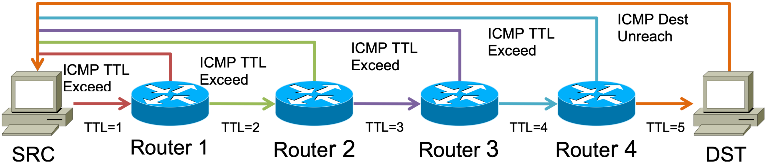


Figure: An example of traceroute[2]

[2] https://major.io/wp-content/uploads/2012/06/RAS_Traceroute_Book_Format.pdf

Traceroute

- Traditional traceroute

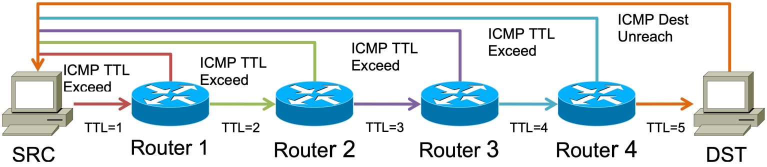


Figure: An example of traceroute[2]

- What if on massive-scale networks, for example, /24 ?

[2] https://major.io/wp-content/uploads/2012/06/RAS_Traceroute_Book_Format.pdf

What if on massive-scale networks?

Some critical issues...

What if on massive-scale networks?

Some critical issues...

- Too slow
  - send & receive synchronously
  - often blocked due to unresponsive servers

What if on massive-scale networks?

Some critical issues...

- Too slow
  - send & receive synchronously
  - often blocked due to unresponsive servers
- Too aggressive
  - routers must responds a huge number of times
  - may be mistaken for a DoS attack

What if on massive-scale networks?

Some critical issues...

- Too slow
  - send & receive synchronously
  - often blocked due to unresponsive servers
- Too aggressive
  - routers must responds a huge number of times
  - may be mistaken for a DoS attack

How can we solve these? FlashRoute.

Table of Contents

1. Motivation
    - Traceroute
    - FlashRoute
2. Implementation
3. Results
4. Appendix

FlashRoute Methodologies

- Asynchronous sending and receiving
  - Encode probing context into the IP and UDP headers

IP Bit Offset	0-3	4-7	8-13	14-15	16-18	19-31
0	Version	Header Length	DSCP	ECN	Total Length Timestamp	6
32	10 Identification		5	1	1 Flags	Fragment Offset
64	Time to Live		Protocol TTL		Header Checksum phase	
96	Source Address					
128	Destination Address					
UDP Bit Offset	0-15 Checksum of Dst IP				16-31	
160	Source Port Number				Destination Port Number	
192	Length				Checksum	
224	Payload					

Figure: Packet encoding

FlashRoute Methodologies (Cont.)

- Doubletree algorithm
  - Split point or TTL
  - Prefer backward probing and reduce redundancy to a minimum

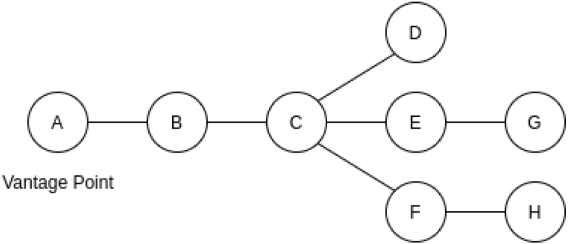


Figure: An ideal topology[3]

[3] Figure 1 in the paper

FlashRoute Methodologies (Cont.)

- Preprobing
  - Split TTL too large $\rightarrow$
    mistaken for a DoS attack
  - Split TTL too small $\rightarrow$
    miss probes-saving opportunities that backward probing exploits

FlashRoute Methodologies (Cont.)

- Preprobing
  - Split TTL too large $\rightarrow$
    mistaken for a DoS attack
  - Split TTL too small $\rightarrow$
    miss probes-saving opportunities that backward probing exploits
  - One-probe hop-distance measurement
  - Provide an estimate of distances to targets!

Table of Contents

1 Motivation
- Traceroute
- FlashRoute

2 Implementation

3 Results
- Performance
- Impact of Some Features
- Topology Visualization

4 Appendix

flashroute.rs: A Rust Implementation

Why Rust[4]?

- A modern system programming language

[4] `https://rust-lang.org`

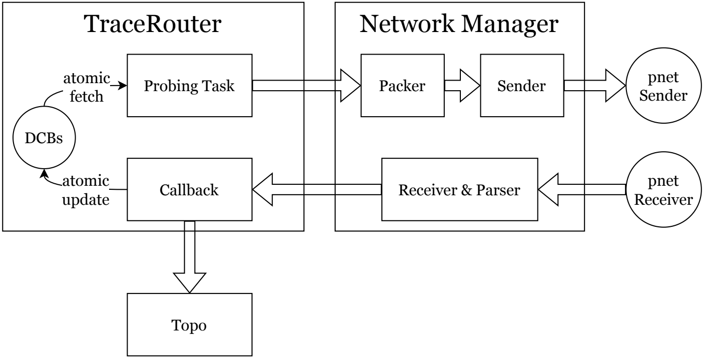
flashroute.rs: A Rust Implementation

Why Rust[4]?

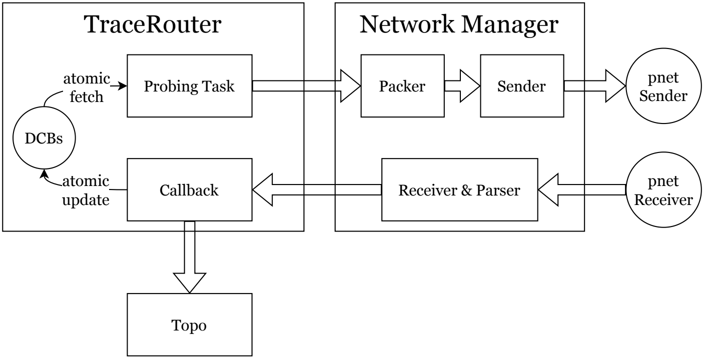
- A modern system programming language
- Performance: blazingly fast and memory-efficient
- Reliabilty: **ownership model** guarantees memory-safety and thread-safety
- Productivity: great community provides a lot of high-performance 3rd-party libraries

[4] `https://rust-lang.org`

Redesign in the view of Task - Overview

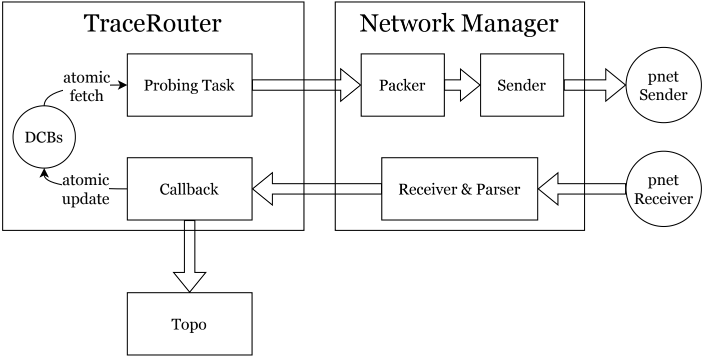


Redesign in the view of Task - Overview



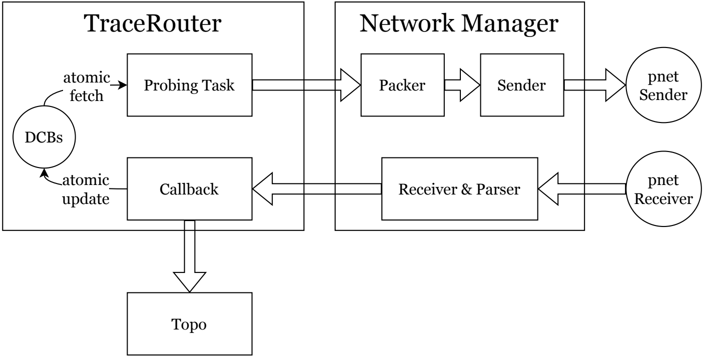
- Lightweight, coroutine-level, automatic task scheduling.
- Make full use of the performance of modern multi-core processors.

Redesign in the view of Task - Thread-safety



- Mutex or rwlock *FREE*. All inter-task communications are achieved through message channels or atomic operations.

Redesign in the view of Task - Thread-safety



- Mutex or rwlock *FREE*. All inter-task communications are achieved through message channels or atomic operations.
- *Absolute* thread-safety thanks to Rust's ownership model.
- Much lower overhead.

Swiss Table Based Data Structure

- Originally targets are organized as an array-based implicit linked list
  - which is enough for the purpose of only scanning /24 internet...
  - but with poor scalability

Swiss Table Based Data Structure

- Originally targets are organized as an array-based implicit linked list
  - which is enough for the purpose of only scanning /24 internet...
  - but with poor scalability
- Swiss table: a high-performance hash table
- Advantages
  - allow customizing probing grain
  - probe partial internet with much lower memory usage

Human-readable Output

- *flashroute.rs* provides the ability to probe on any subnet

Human-readable Output

- *flashroute.rs* provides the ability to probe on any subnet
- Propose a method to assemble traceroute results into a graph

```
graph {
    0 [ label = "192.168.1.177" ]
    1 [ label = "115.159.1.131" ]
    2 [ label = "9.31.253.89" ]
    ...
    2 -- 1 [ label = "1" ]
    12 -- 11 [ label = "5" ]
    7 -- 13 [ label = "4" ]
    ...
}
```

Human-readable Output

- *flashroute.rs* provides the ability to probe on any subnet
- Propose a method to assemble traceroute results into a graph

```
graph {
    0 [ label = "192.168.1.177" ]
    1 [ label = "115.159.1.131" ]
    2 [ label = "9.31.253.89" ]
    ...
    2 -- 1 [ label = "1" ]
    12 -- 11 [ label = "5" ]
    7 -- 13 [ label = "4" ]
    ...
}
```

- Generate visualization on small-scale topology

Table of Contents

1. Motivation
   - Traceroute
   - FlashRoute
2. Implementation
3. Results
   - Performance
   - Impact of Some Features
   - Topology Visualization
4. Appendix

Table of Contents

1 Motivation

2 Implementation

3 Results
- Performance
- Impact of Some Features
- Topology Visualization

4 Appendix

Overview of *flashroute.rs*

Our goals:

- Enhance FlashRoute in Rust
  - originally implemented in C++ using 2.4k LoC[5]
  - expectations
    - fewer codes
    - even better performance
    - asynchronous operations instead of manually-handled mutex

[5] `https://github.com/lambdahuang/FlashRoute/`

[6] Tested on AMD EPYC 7B12 (8) @ 2.25 GHz

Overview of *flashroute.rs*

Our goals:

- Enhance FlashRoute in Rust
  - originally implemented in C++ using 2.4k LoC[5]
  - expectations
    - fewer codes
    - even better performance
    - asynchronous operations instead of manually-handled mutex

Our results:

- *flashroute.rs* succeeds to probe 800,000 interfaces as FlashRoute does

[5] https://github.com/lambdahuang/FlashRoute/

[6] Tested on AMD EPYC 7B12 (8) @ 2.25 GHz

Overview of *flashroute.rs*

Our goals:

- Enhance FlashRoute in Rust
  - originally implemented in C++ using 2.4k LoC[5]
  - expectations
    - fewer codes
    - even better performance
    - asynchronous operations instead of manually-handled mutex

Our results:

- *flashroute.rs* succeeds to probe 800,000 interfaces as FlashRoute does
- *flashroute.rs* implements almost all features in only **1.2k LoC**

[5] https://github.com/lambdahuang/FlashRoute/

[6] Tested on AMD EPYC 7B12 (8) @ 2.25 GHz

Overview of *flashroute.rs*

Our goals:

- Enhance FlashRoute in Rust
  - originally implemented in C++ using 2.4k LoC[5]
  - expectations
    - fewer codes
    - even better performance
    - asynchronous operations instead of manually-handled mutex

Our results:

- *flashroute.rs* succeeds to probe 800,000 interfaces as FlashRoute does
- *flashroute.rs* implements almost all features in only **1.2k LoC**
- *flashroute.rs* gains probing performance improvement of up to **40%**[6]

[5] https://github.com/lambdahuang/FlashRoute/

[6] Tested on AMD EPYC 7B12 (8) @ 2.25 GHz

Performance of Single Target Probing

Results of probing single target 115.159.1.233 from 59.78.X.X:

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

Performance at Probing Rate of 100 Kpps

Results with different split TTL, 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[7], instead of a randomly-generated one.

[7] USC/LANDER project. Jan 30 2019. Internet Addresses Hitlist Dataset. USCLANDER/internet_address_hitlist_it84w-20190130/rev10351.

Performance at Maximum Probing Rate

Results with different split TTL, 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

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

Results are tested on

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

Table of Contents

1 Motivation

2 Implementation

3 Results
- Performance
- Impact of Some Features
- Topology Visualization

4 Appendix

Impact of Redundancy Removal during Backward Probing

Tool	RR[8]	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

[8] Redundancy Removal

Impact of Gap-limit

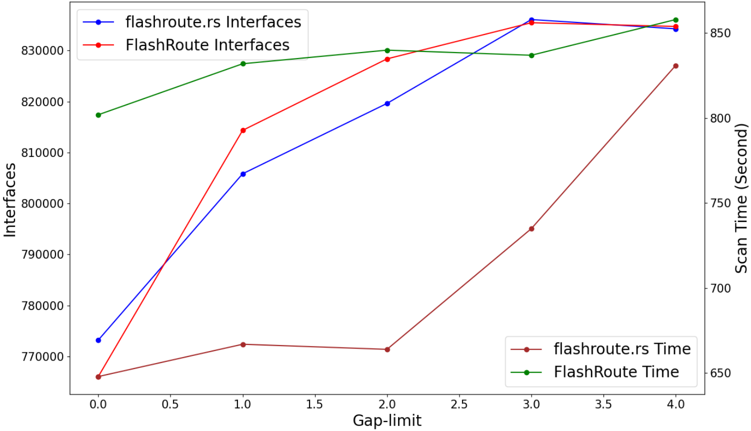


Figure: Interfaces and Scan Time under Different Gap-limit

Table of Contents

1. Motivation
2. Implementation
3. Results
   - Performance
   - Impact of Some Features
   - Topology Visualization
4. Appendix

Visualization

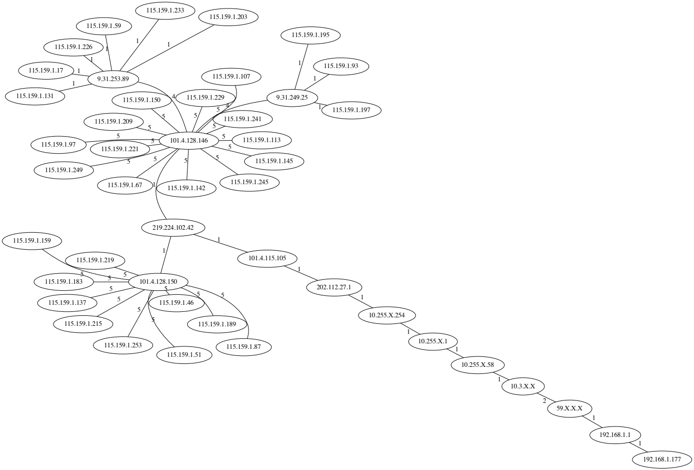


Figure: Topology of some hosts @115.159.1.0/24

Table of Contents

1. Motivation
   - Traceroute
   - FlashRoute
2. Implementation
3. Results
   - Performance
   - Impact of Some Features
   - Topology Visualization
4. Appendix

Contribution Matrix

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

Sources

All sources of *flashroute.rs* can be found at:

`https://github.com/BugenZhao/flashroute.rs`

Thank You