Scitags: A Standardised Framework for Traffic Identification and Network Visibility

XRootD and FTS Workshop
13-17 October 2025, PIC, Barcelona, Spain

Shawn McKee[1], V. Acin[1,2], Marian Babik[2], Tim Chown[3], Andrew Hanushevsky[4], Andy Lake[5], Tristan Sullivan[6], Bruno Hoeft[7], James Letts[8], Dale Carder[5], Garhan Attebury[9], Michael Lambert[10], Joe Mambretti[11], Karl Newell[12], Pablo Collado Soto[13]

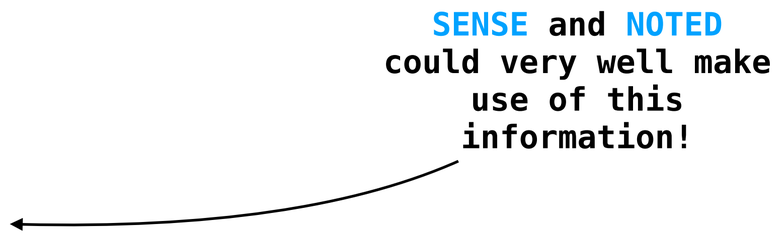
[1]University of Michigan Physics; [2]CERN; [3]Jisc; [4]SLAC National Accelerator Laboratory; [5]ESnet; [6]University of Victoria; [7]Karlsruhe Institute of Technology (KIT); [8]University of California, San Diego; [9]UNL; [10]Pittsburgh Supercomputing Center; [11]Northwestern University; [12]Internet2; [13]Universidad Autónoma de Madrid.

What's the problem?

- **High Energy Physics** (HEP) makes (very) **heavy use** of **networks**:
  - A **huge** amount of **data** is **generated** at the LHC, Belle II and co.
  - With the **HL-LHC** this **data** is set to **increase** by an order of magnitude!
- **Network** data **flows** are **not** very **well understood** though:
  - **Traffic sources** in Research and Education Nets (**RENs**) are **not obvious**.
  - **Without** additional **information**, **analysing** individual **flows** is **challenging**.
  - The **situation** is specially **dire** in the face of **overloaded links**.
- So… let's **try** to **get** some **insights** into what's going on!

What’s the benefit?

- This **enhanced information** will **allow** REN operators and experiments **to**:
    1. **Measure** on-link **performance**.
    2. Make a **more effective use** of existing **links**.
    3. Know **where** the **backbone** needs **upgrading**.
    4. **Automatically act** with **SDN**-capable **backbones**.
    5. **Pinpoint inefficient** experiment **workflows**.
    6. Carry out t**raffic** accounting in shared inter-continental links.
    7. Some **other** use **cases** could pop up: the **information**’s there!
- **Current** tooling includes **overlays** (VRFs) and **network flows**, but **problems** exist.

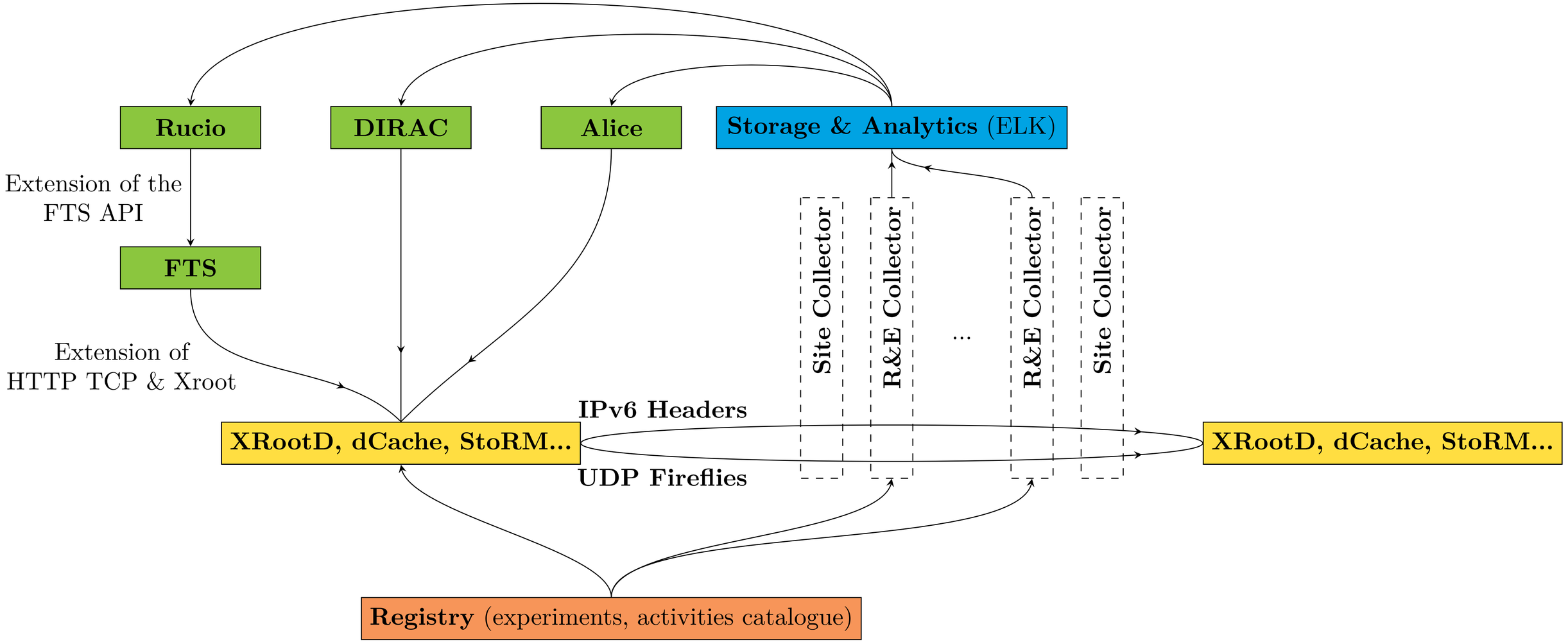


The Scitags project

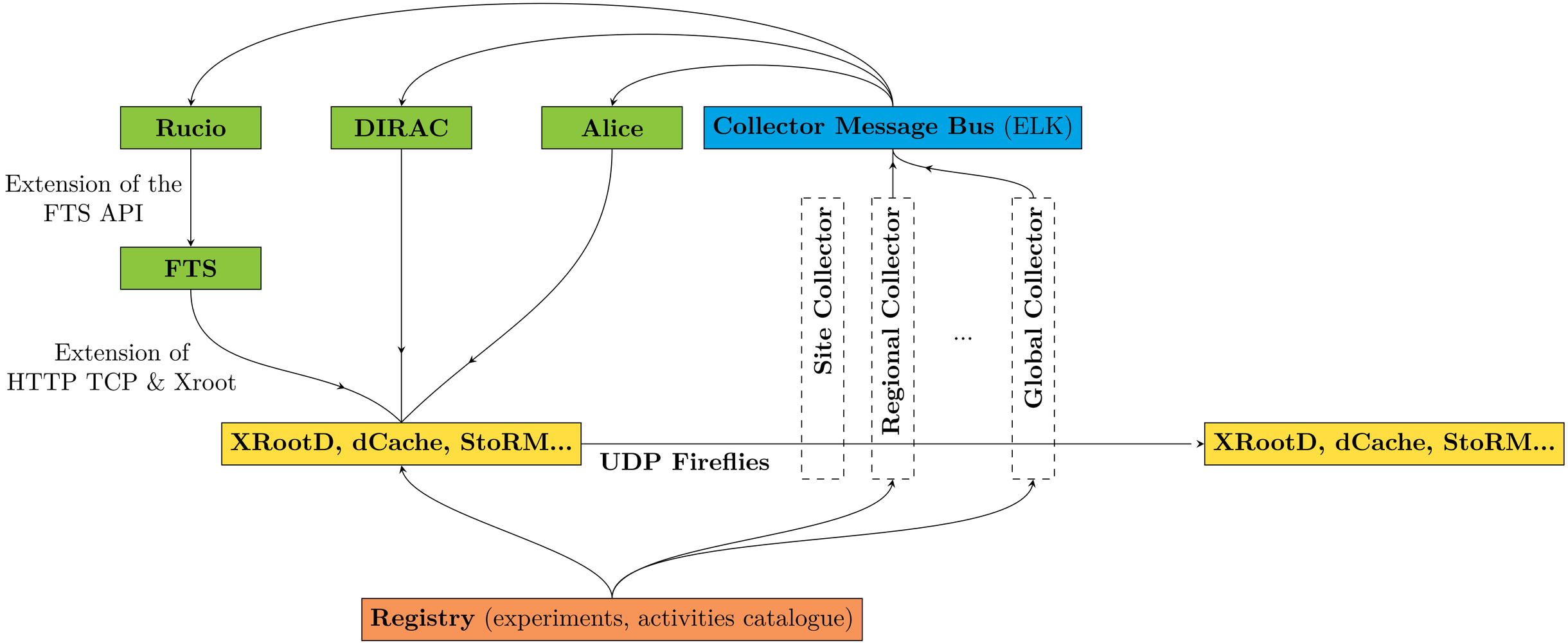
Check the SciTags site for more info!

- The **SciTags** framework:
  - Is an **open platform** to be **used** by any **data-intensive** science **community**. It is **potentially** more **broadly** applicable.
  - Enables the **identification of** the **owner** (experiment) and **purpose** (activity) of traffic.
  - **Defines standards** for **exchanging information** between scientific **communities**, **sites** and **network operators**. This **information** can be **encoded** and **sent** with **different strategies**.
  - **Enables tracking**/correlation with **existing** network flow **monitoring infrastructure** and associated infrastructure already **deployed** by **RENs**.
  - The **technical specification** is readily available **here**.

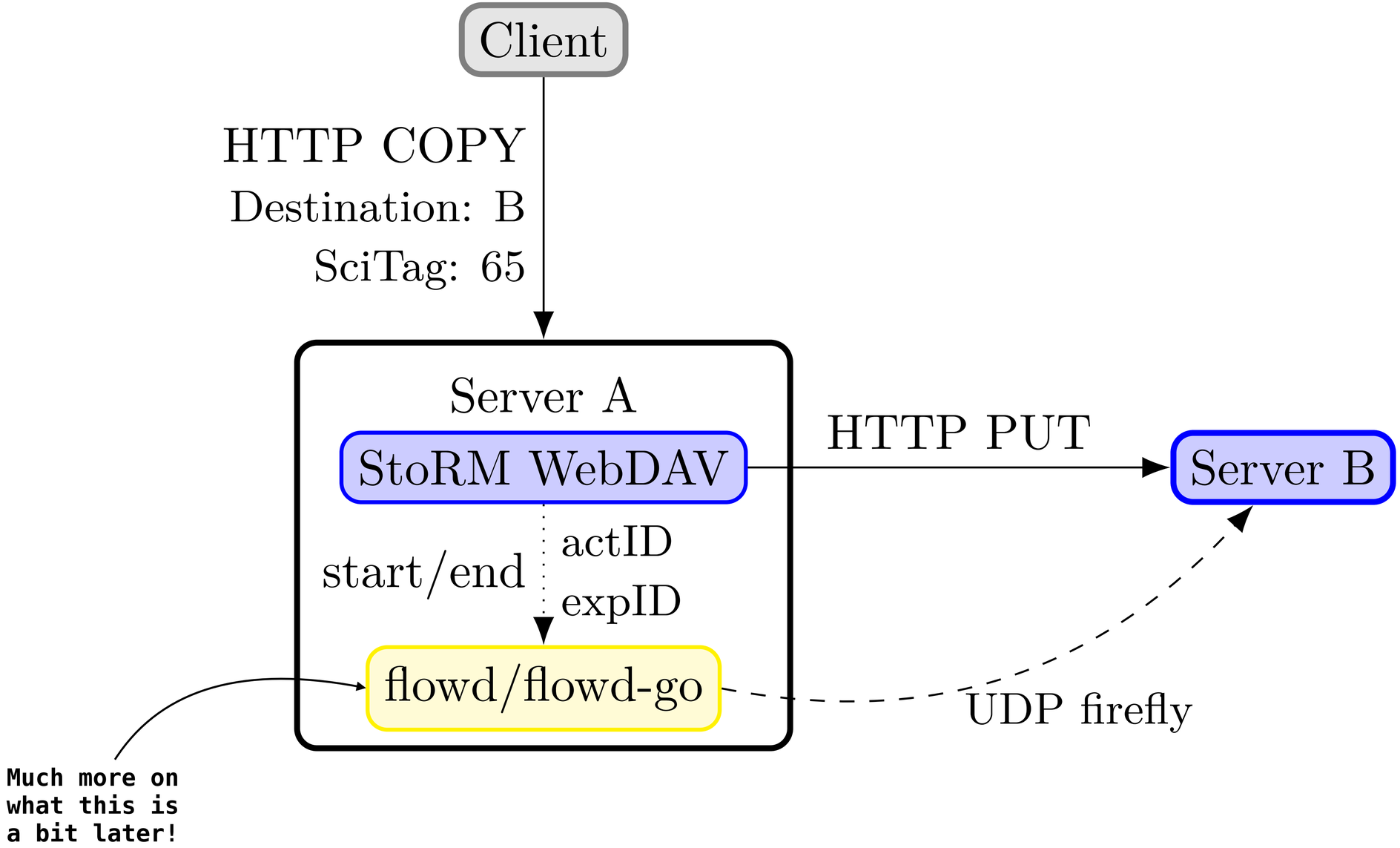
The overall architecture



The overall architecture today



Zooming into a StoRM transfer…



A bit more on the registry

- SciTags strives to **‘mark’ activity**-scoped traffic of an **experiment**.
- Now, **what identifiers** should we agree on? That’s what the **registry** is for:
  - It’s a **JSON document** containing experiment and activity IDs.
  - **Available** at https://www.scitags.org/api.json.
  - The **underlying** source of truth is a **Google Sheet**.
  - It only **provides values**, **not what** to do with them!


```
{
   "experiments": [{
      "expName": "atlas",
      "expId": 2,
      "activities": [
         {
            "activityName": "default",
            "activityId": 1
         },
         {
            "activityName": "perfSONAR",
            "activityId": 2
         },
            "activityName": "Data Brokering",
            "activityId": 3
         },
         {
            "activityName": "Data Consolidation",
            "activityId": 4
         }
      ]
   }],
   "version": 1,
   "modified":
   "2024-02-01T17:32:36.817459+00:00"
}
```


Fragment of ATLAS’ registry data.

The need for collaboration

- The **SciTags** effort **relies** on several key players to **collaborate**:
  - **Storage Elements** should **implement** this marking capability.
  - **Orchestrators** such as **FTS** should add **support** too!
  - **REN operators** must deploy **collectors** along the path…
  - **UDP fireflies** usually call for **firewall reconfigurations**…
  - **Sites** must **deploy flowd** on their machines…

All these organisations are already taking part!

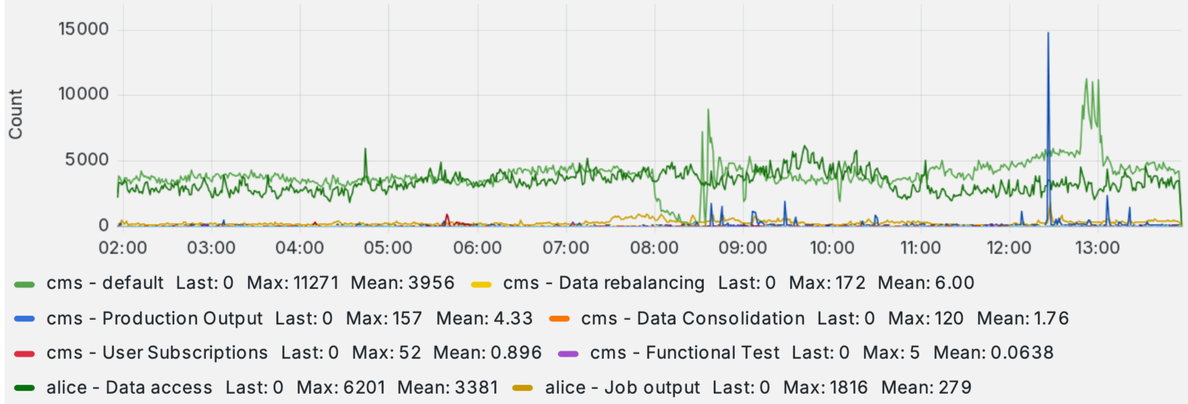
SciTags can be used by any data-intensive community: JP Morgan asked during SC 24!

What's already supported?

Infrastructure Type	Implementation	Supported?	Anything else?
Backbone	**Rucio**	**`32.4.0+`**	-
	FTS	**`3.2.10+`**	-
	GFAL2	**`2.21.0+`**	-
SEs	**XRootD**	**`5.0+`**	In **production** at several **sites**.
	EOS	**`5.2.19+`**	In **production** at **CERN EOS** ATLAS, CMS and ALICE.
	dCache	**`10.0.2+`**	**Static** configuration, **HTTP** and **XRoot** support **ongoing**.
	StoRM	**`1.5.0+`**	In **production** at **CNAF** for all supported **experiments**.
Collectors	**Firefly Collector**	**–**	**Production** deployment at **ESnet** (US) and discussions with **GÉANT**. A **site** collector is running at **CERN**. JISC ran one for 4 UK sites for 5 months.

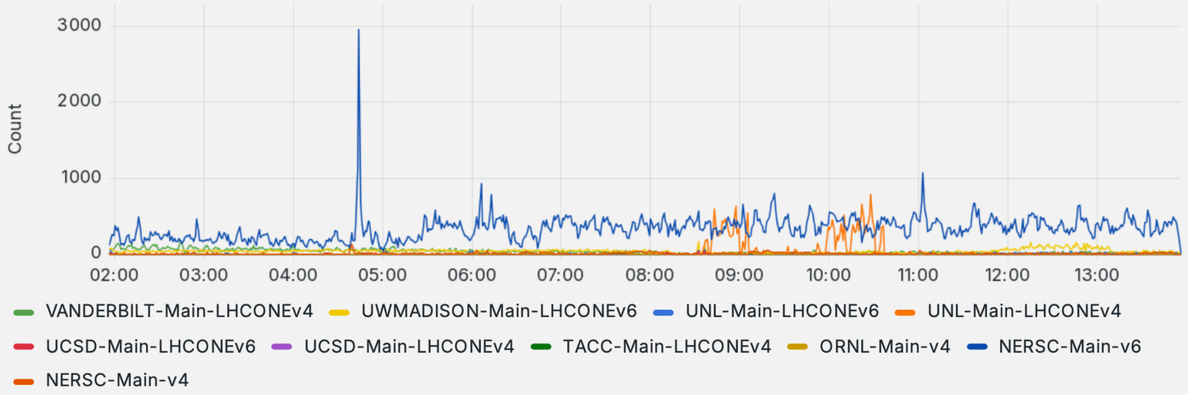
A sample collector dashboard

Total Flows per Exp/Act

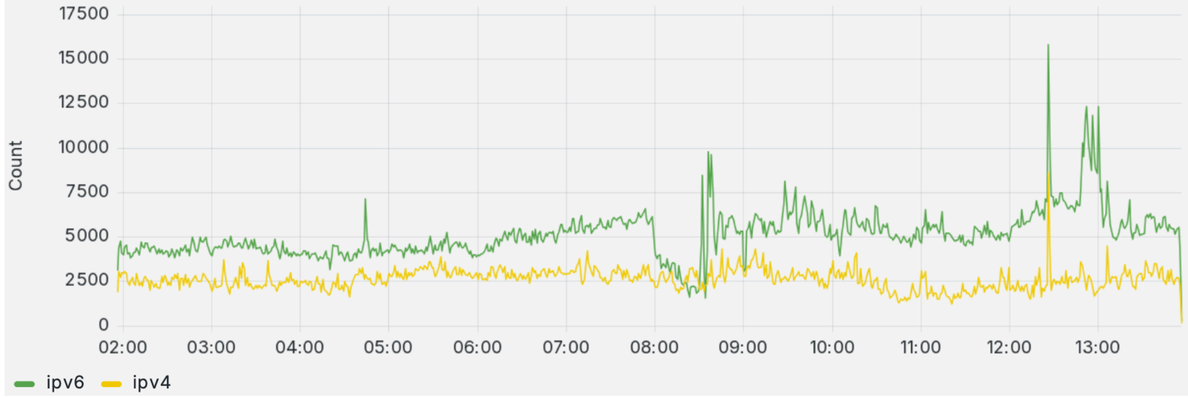


Total Flows per Prefix

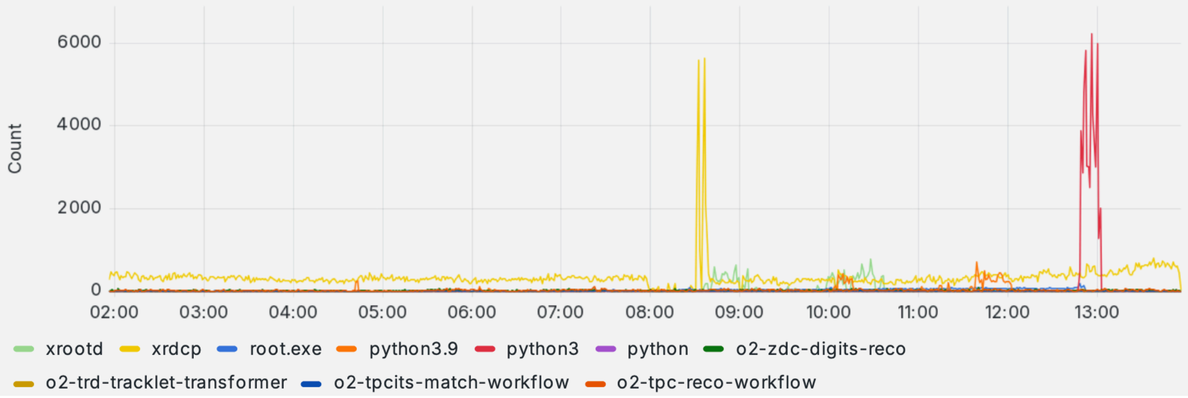
Click me!



Total Flows per IP version



Total Flows per Application (top10)

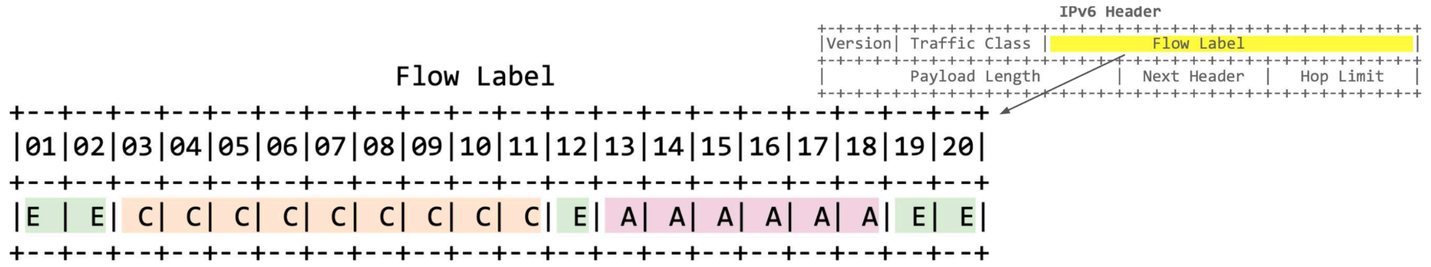


What's a flow and how are they marked?

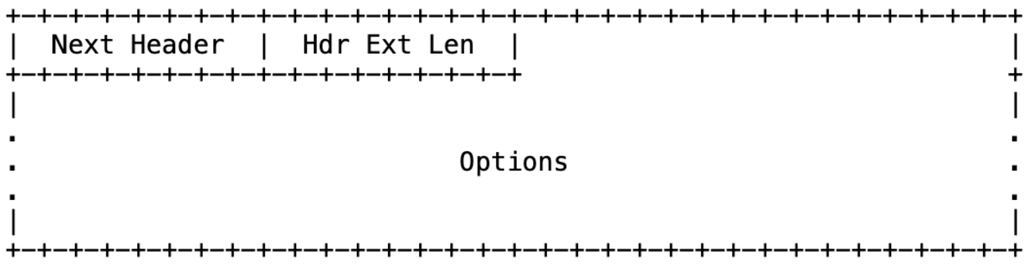
- **Generally**, a **flow** consists of **{src,dst} IPvX** and **{src,dst} port**.
- To **identify** them in the **backbone**, we **mark datagrams** by:
  1. Setting the **Flow Label** in **IPv6** datagrams.
  2. Sending a **UDP firefly** with connection metadata to a **collector** or **SE**.
  3. **R&D** is ongoing to leverage **IPv6 Extension Headers**.
- The **program** in charge of this **marking** is **flowd**.
- At **UAM** we **reimplemented flowd** in Go. This became **flowd-go**.

A deeper look at IPv6-based marking

- One **strategy leverages** the **IPv6 Flow Label**.
  - This **flow label** is **very coveted**…
  - An **IETF RFC** Informational **Draft** is available.
- New **options** based on **IPv6 Extension Headers** (EH) are **available**! See **ebpf-PDM** for a great discussion. **Destination Options** (DO) headers are preferred.



	Purpose	Example
Community ID	Who are you affiliated with?	2 (ATLAS)
Activity ID	What are you doing within your community?	3 (data brokering)
Entropy	Comply with **RFC 6437**	10010



And what do fireflies look like?

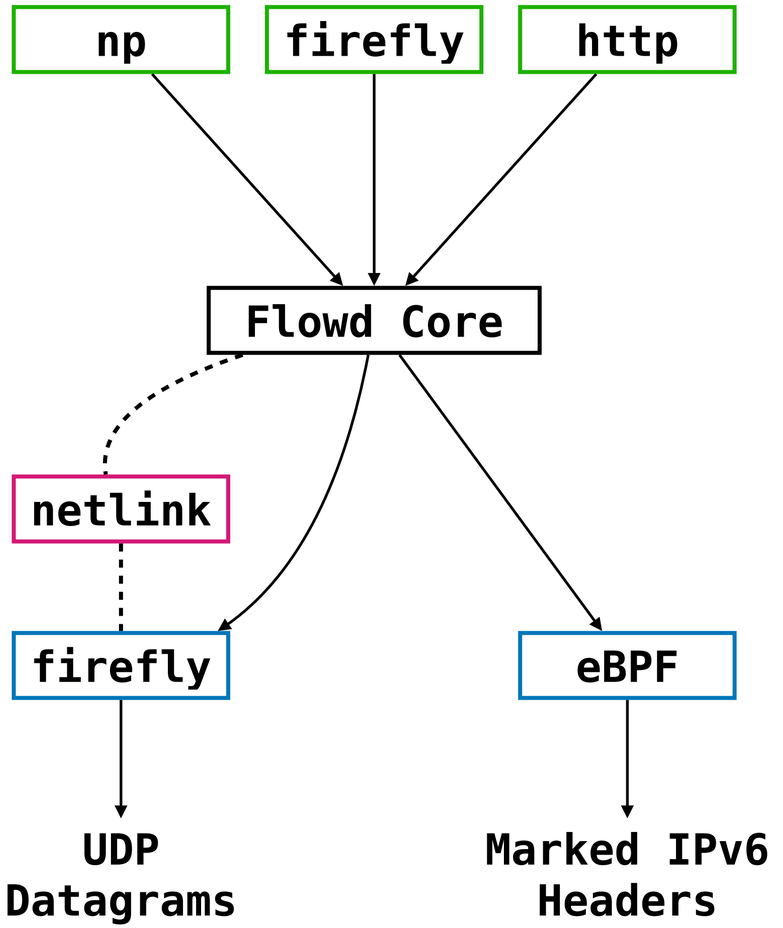
- Put simply, **fireflies** are **JSON documents**.
- They **carry** transfer **metadata**, experiment and activity **IDs**…
- These form the **payload** of **UDP** datagrams.
- They include a **syslog header** to make **ingestion** by **logstash** (ELK stack) **easier**.
- **They** should be **sent** to:
  1. The **transfer's destination**.
  2. **Optionally** to a regional/global **collector**.
- **Fireflies work** on top of **IPv{4,6}**!


```
{
   "flow-id": {
      "protocol": "tcp",
      "afi": "ipv6",
      "dst-ip": "2001:67c:1bec:236::9",
      "src-port": 30175,
      "src-ip": "2001:48a8:68f7:1:192:41:231:128" ,
      "dst-port": 50152
   },
   "version": 1,
   "flow-Lifecycle": {
      "state": "end",
      "start-time": "2021-09-21T17:46:43.406370+00:00",
      "end-time": "2021-09-21T17:46:43.406370+00:00"
      "current-time": "2021-09-21T17:46:43.406352+00:00"
   },
   "usage": {
      "received": 4234,
      "sent": 34343443
   },
   "netlink": {
      "rtt": 33.4
   },
   "context": {
      "experiment-id": 16,
      "application": "flowd vo.0.1",
      "activity-id": 1
   }
}
```

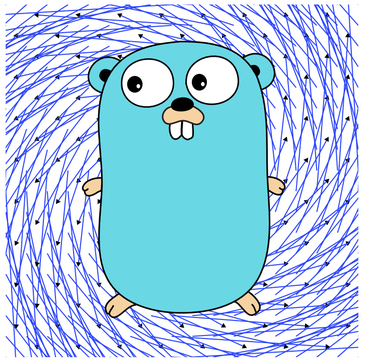

An example firefly.
The syslog header is not included.

That’s great, but who does all this?

- The **backbone** of the **SciTags** initiative is **flowd**, it:
  - Is a **Python** program.
  - Showcases a **plugin** and **backend**-based structure:
    - **Plugins** accept **flow definitions**.
    - **Backends** implement the actual packet **marking**.
  - Is **capable** of **enriching** flows by leveraging **netlink**.
  - Must be **deployed** on site **Storage Elements**.
  - **Already** made an **appearance** on **slide 6**!
  - Is **available** on **scitags/flowd**.

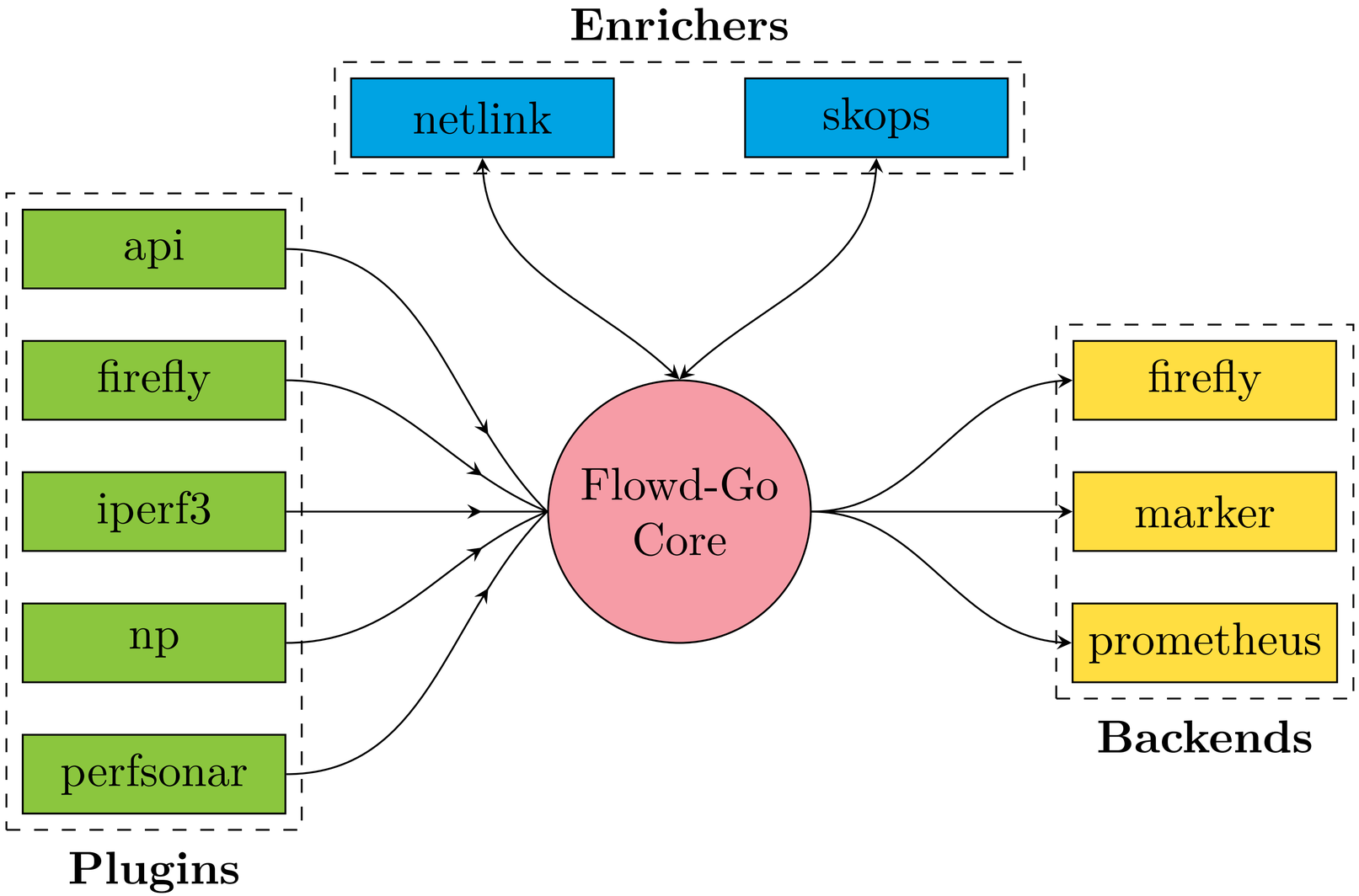


flowd + go = flowd-go!



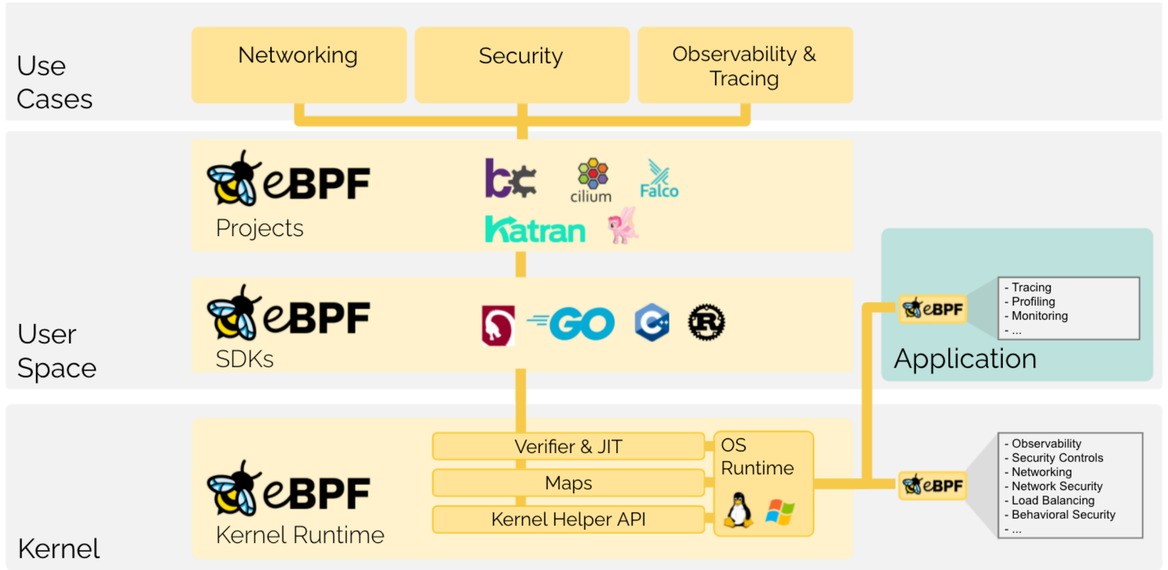
- **Flowd** is **great**, but it has some **shortcomings**:
  - **Performance** is **limited** by **Python** itself.
  - The **netlink** subsystem was **based** on **parsing** text!
- We developed **flowd-go**, a new **implementation** from **scratch** in **Go**:
  - It brings **Go's strengths**: built-in **concurrency**, **static** binaries...
  - Our **approach** to **eBPF** is radically **different**:
    - We use **cilium** instead of **BCC**: we can leverage **CO:RE**!
    - This comes with an (**expected**) **performance increase**!

Flowd-go's architecture

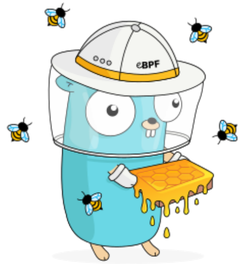


What's this eBPF thing?

- **eBPF** is a **technology allowing** us to:
  - **Run** programs in a sandbox in **kernel space**.
  - **Extend** the **kernel without** modules and having to **recompile**.
- **Communication** between kernel and user space **hinges** on **maps**.
- This **technology** is **applicable** to a **myriad** use cases including **networking**, low-level **monitoring** and more!



Our eBPF loaders



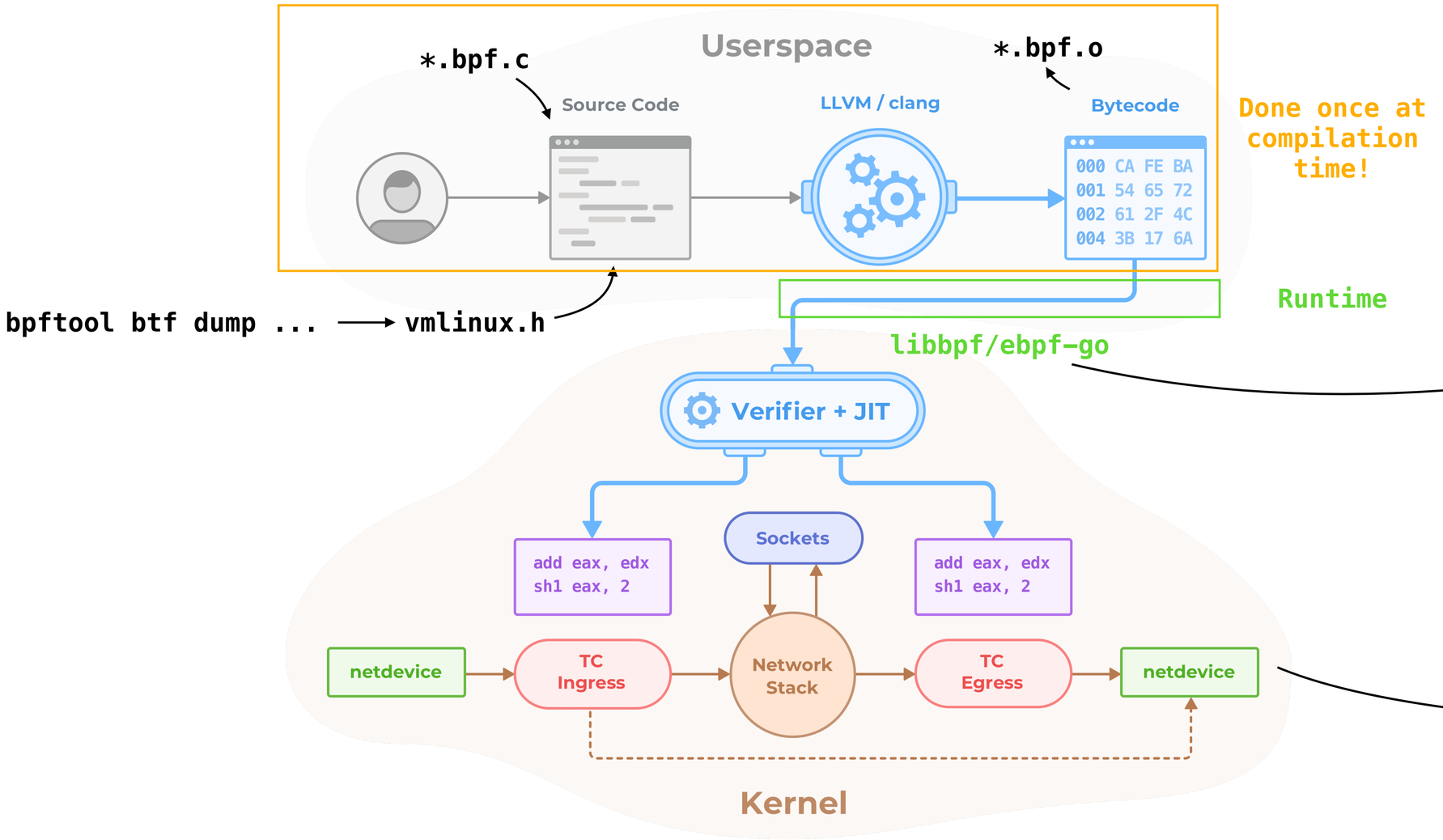
- eBPF **programs** need to be **loaded**, **attached**, **interacted** with...
- This is usually **done** through the `bpf(2)` **syscall**.
- **Calls** are **abstracted** away by **'loaders'**. We've **worked** with **two**:
    1. **libbpf** (C): de-facto **standard developed** hand in hand with the **kernel**.
        - New **features trickle down** to other loaders: this happened with **CO:RE**!
    2. **ebpf-go** (Go): Go-native alternative **leveraged** by **large projects** (**cilium**).
- **Libbpf** was **abandoned** because it **'imposes'** the use of **cgo**...
- The **key feature** is **CO:RE** (Compile-Once:Run-Everywhere).

We leveraged the `libbpfgo` wrapper

What was flowd's approach to eBPF?

- **Flowd** leveraged the **BPF Compiler Collection** (BCC).
- Put simply, **BCC** hinges on **compiling** the **BPF** program at **runtime**:
  - It **imposes Clang/LLVM** as a **dependency** on the end user.
  - The only **available interfaces** are **Python** and **Lua**.
  - Several **C types abstract** away the actual **kernel definitions**...
- BCC's **Python interface** has been **deprecated**: they're **migrating** to **libbpf**!
- **BCC**-based solutions are **still** in **production** at **Netflix** and co.!

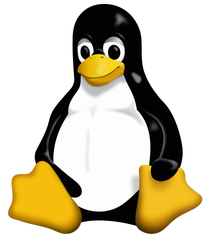
What does it look like in the end?

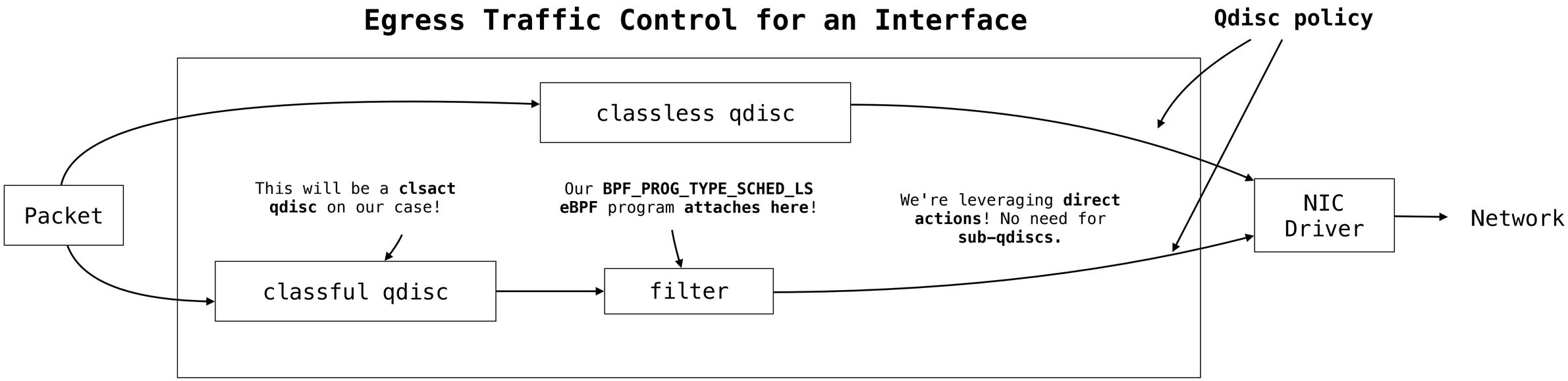


CO:RE leverages **BTF**-encoded **information** in the **bytecode** to **adapt data structures** across **kernel versions**! We'll be **mostly** working with kernel **v5.14** (**AlmaLinux 9**).

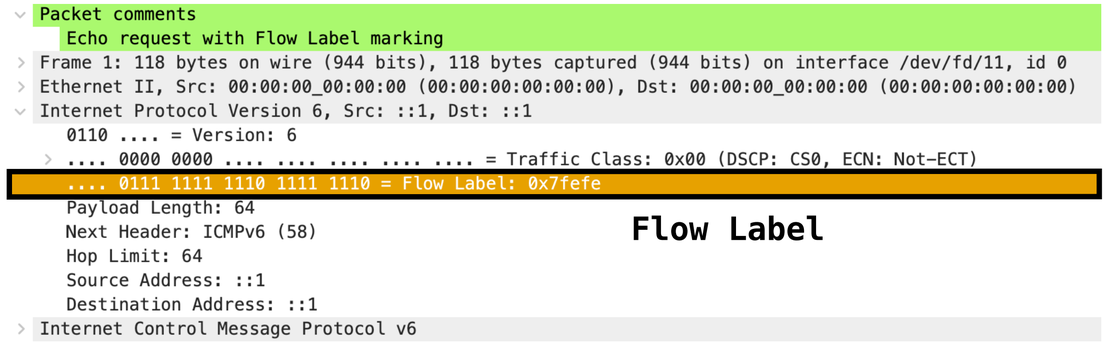
Different eBPF **program types** will **attach** to **different** kernel **subsystems** in with **different methods.**

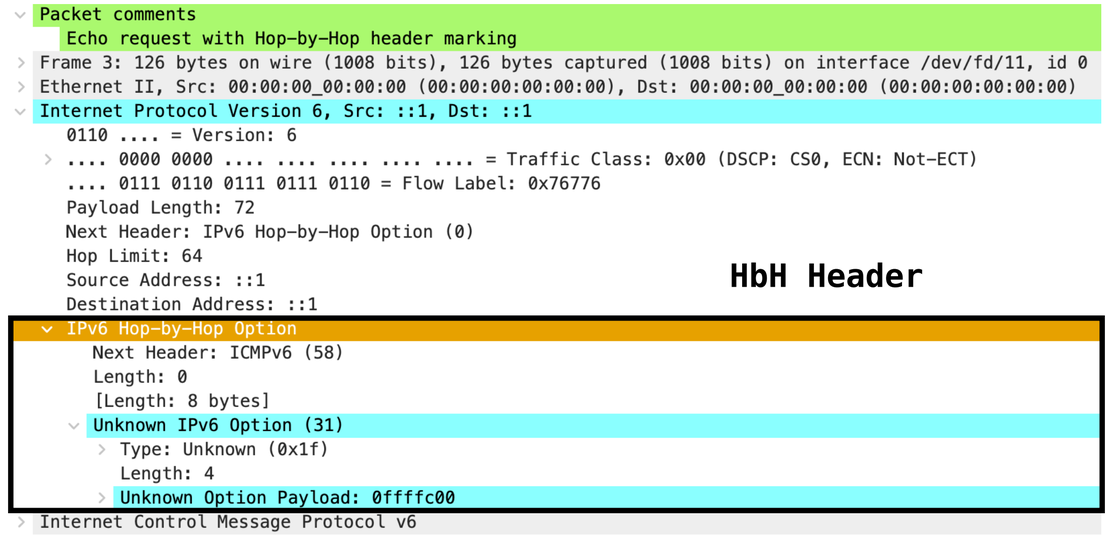
Where does our marker fit in tc(8)?

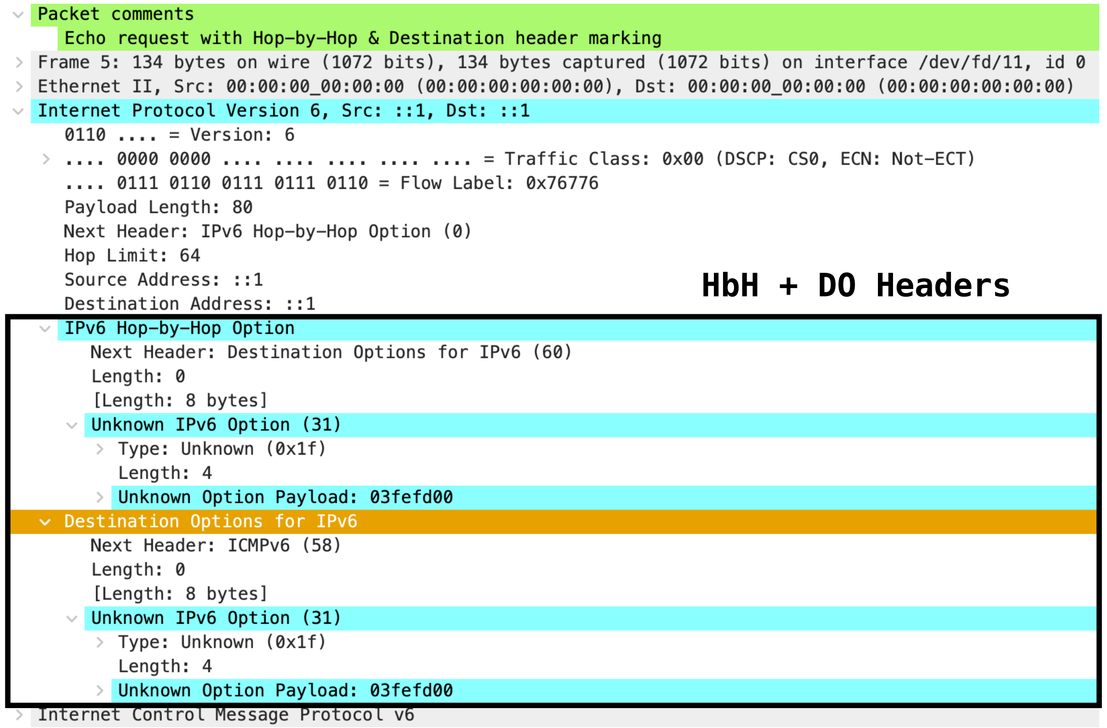




Demonstrating IPv6 marking

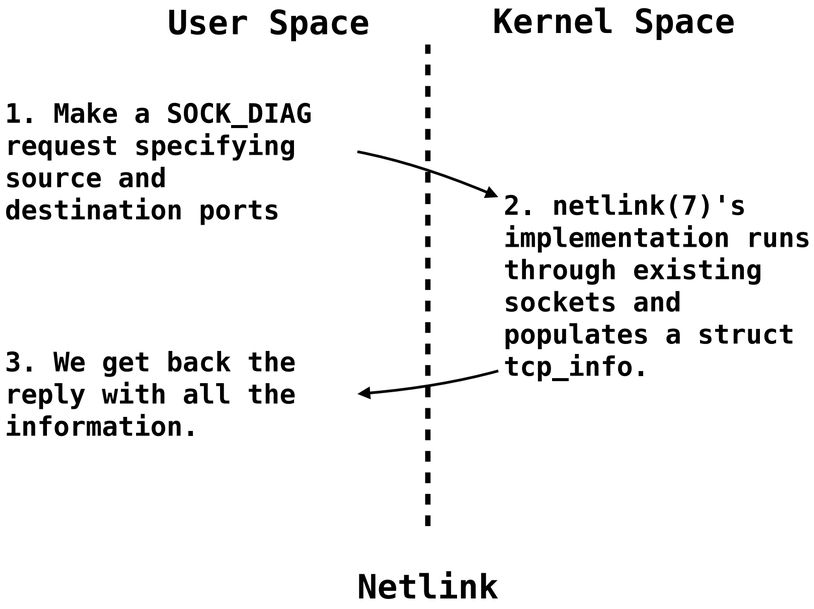


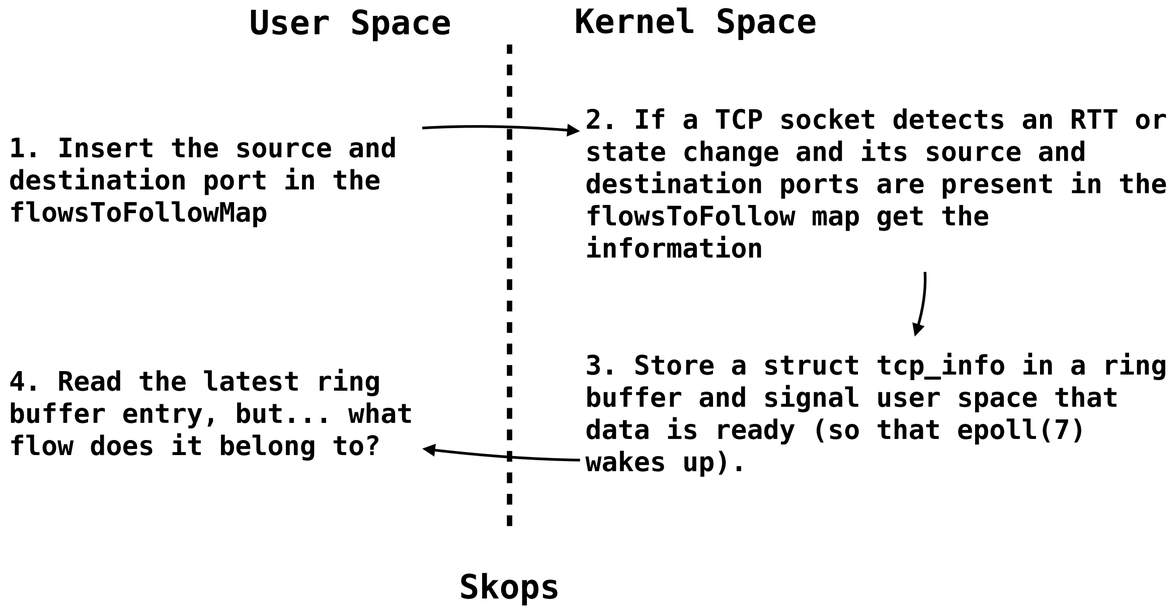




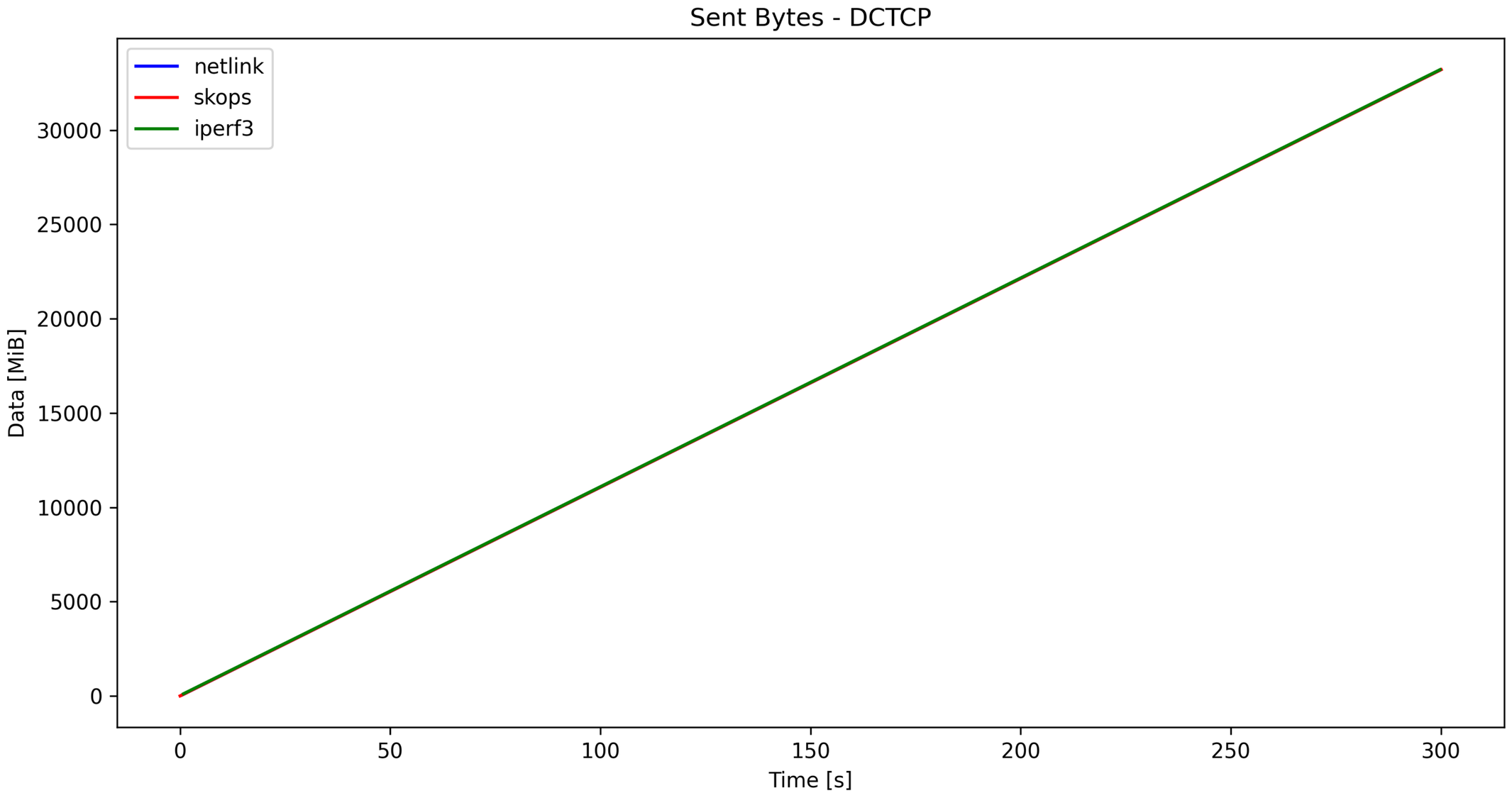
Extracting TCP-level information

- TCP-level information can be **acquired** and **embedded** in **periodic fireflies.**
- We support **two** data **sources**: **netlink** and **eBPF**-based **socket introspection.**

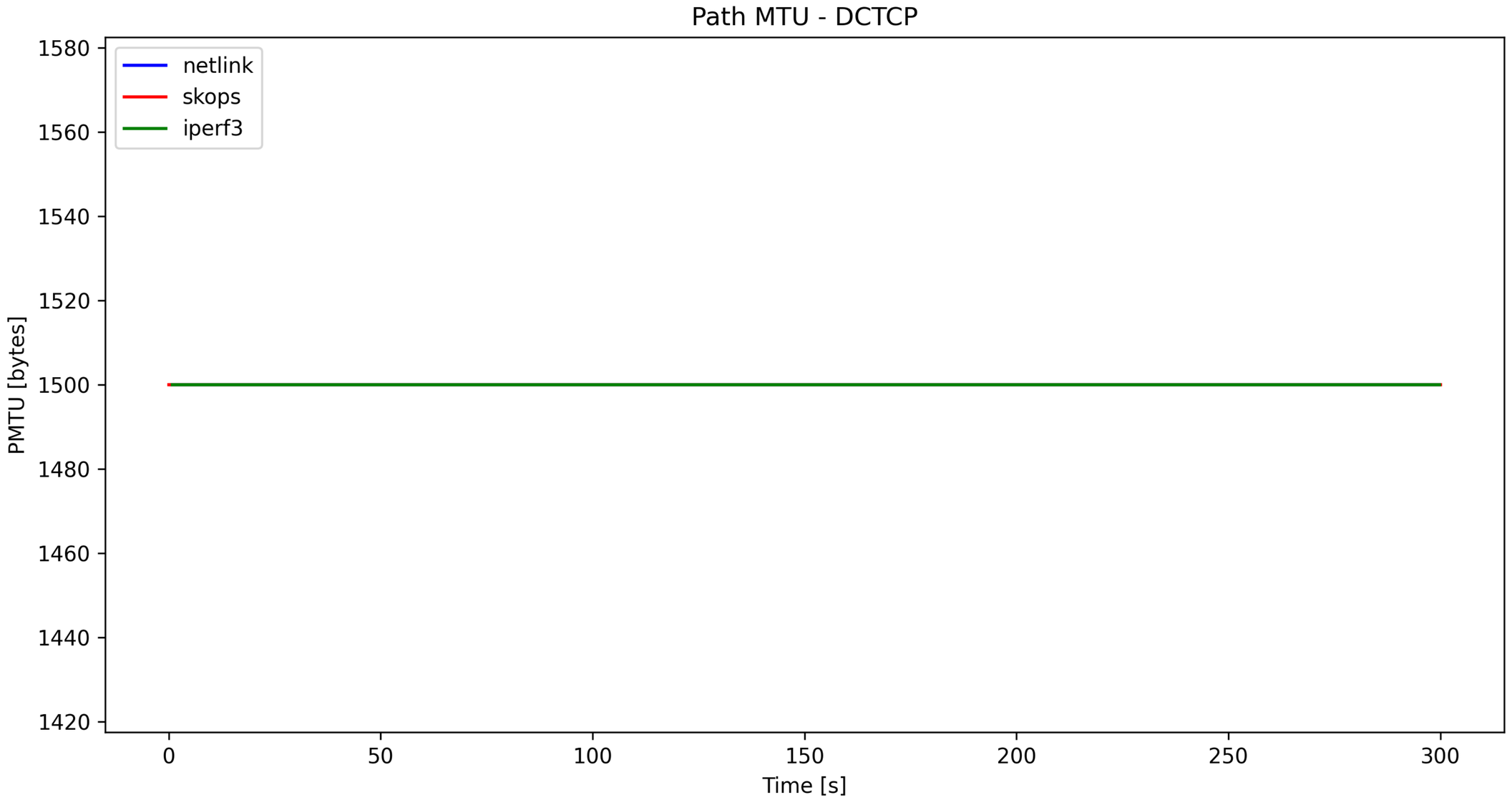




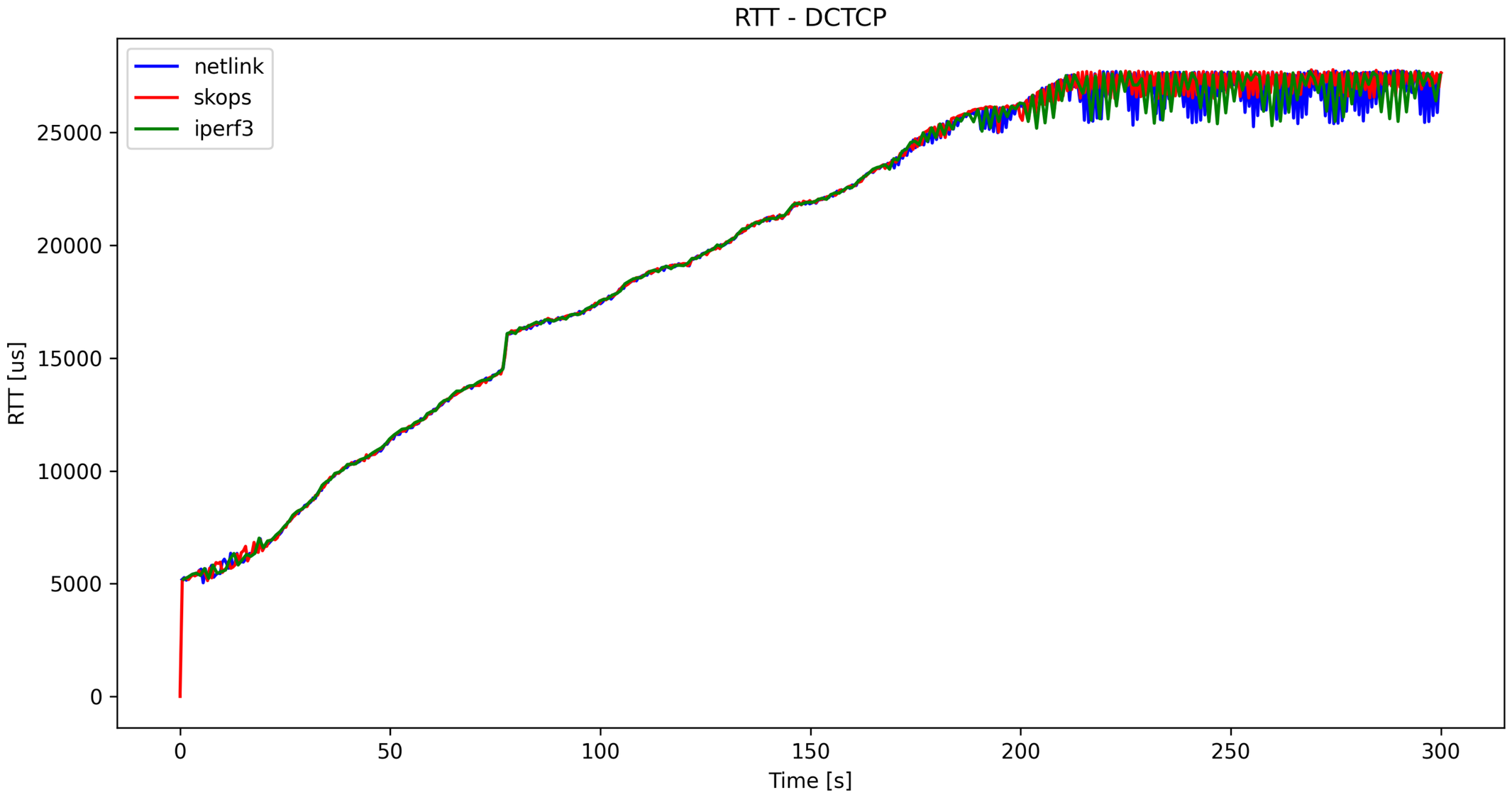
Great, but does it work?



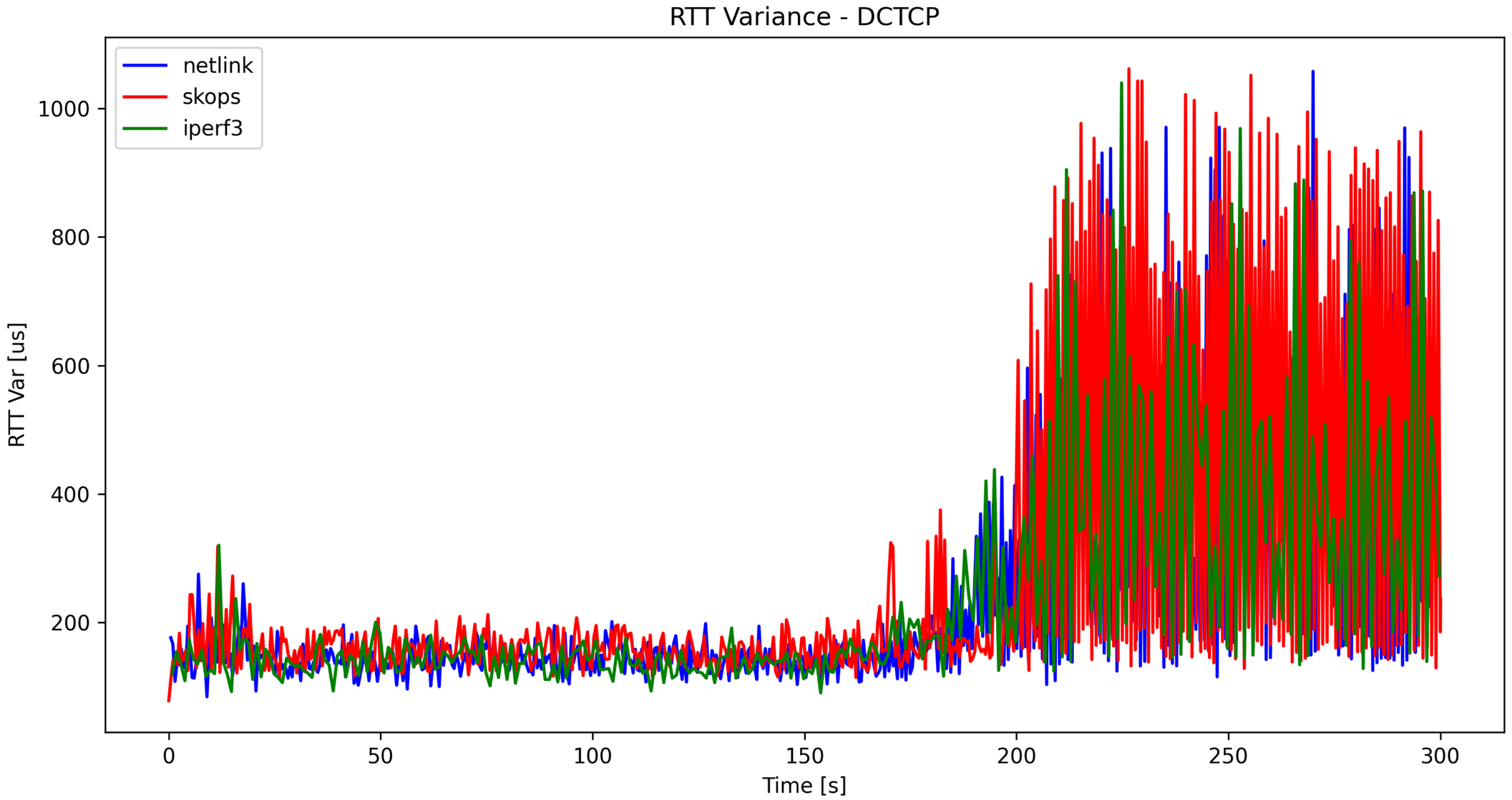
Great, but does it work?



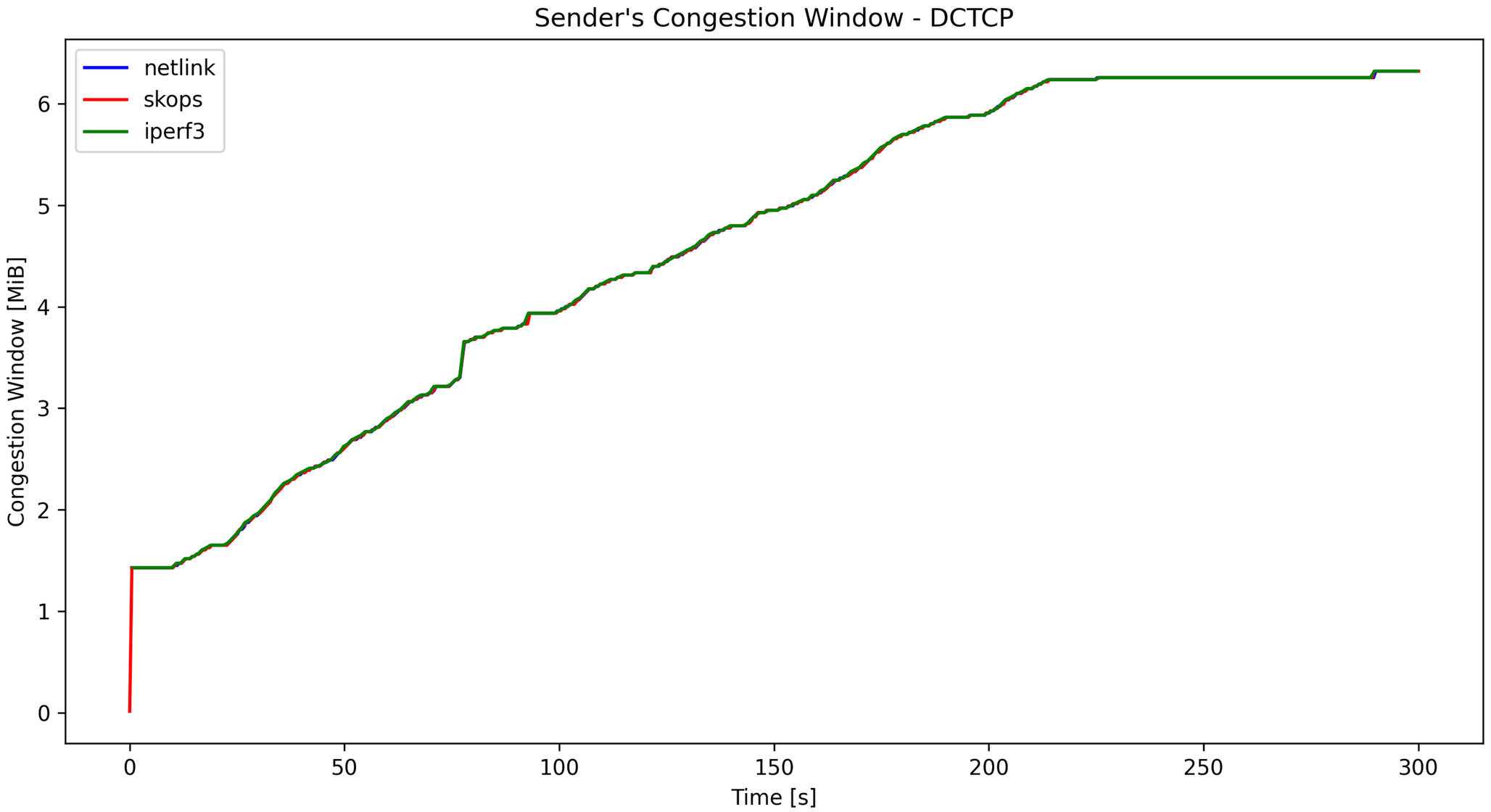
Great, but does it work?



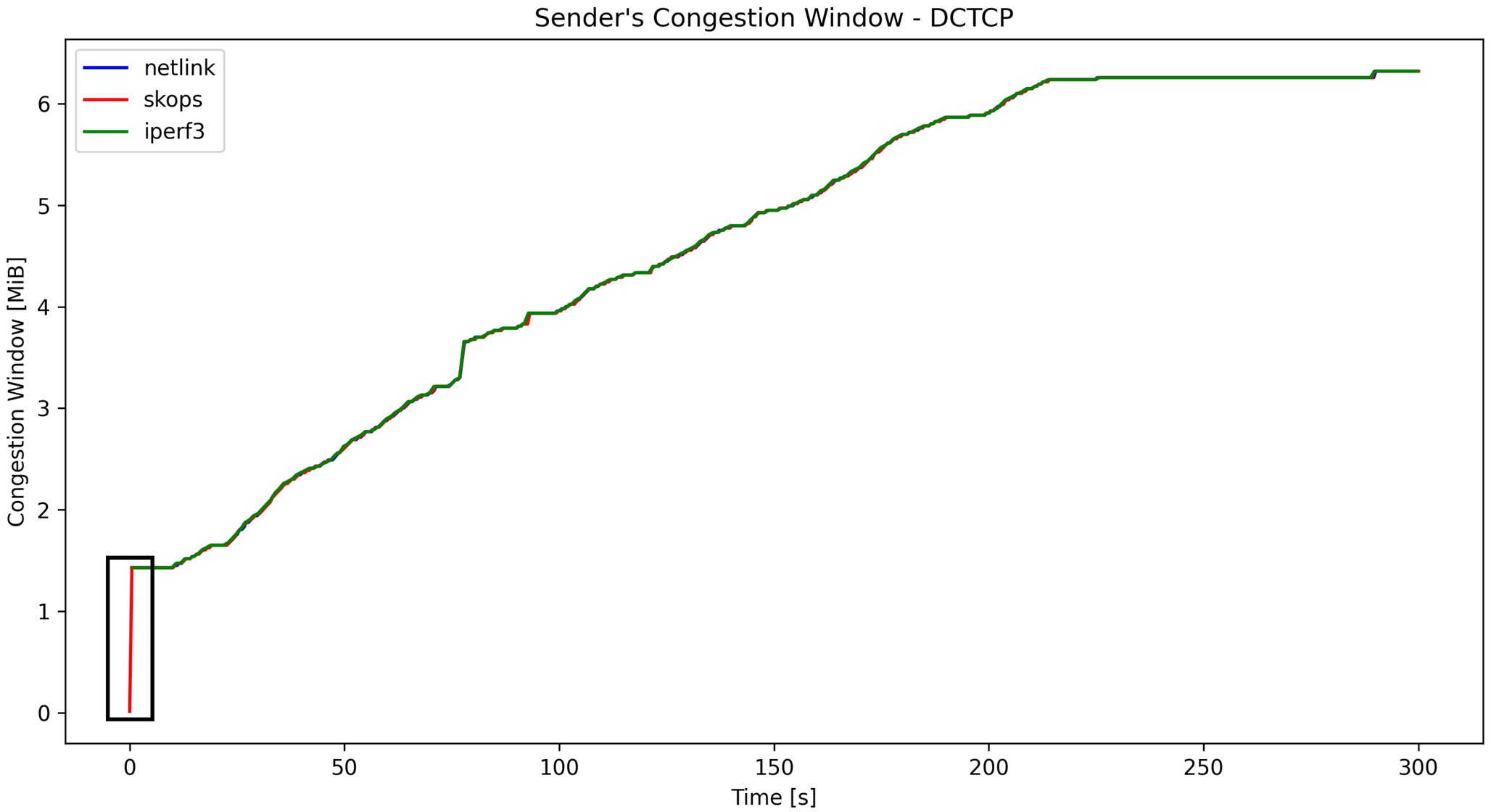
Great, but does it work?



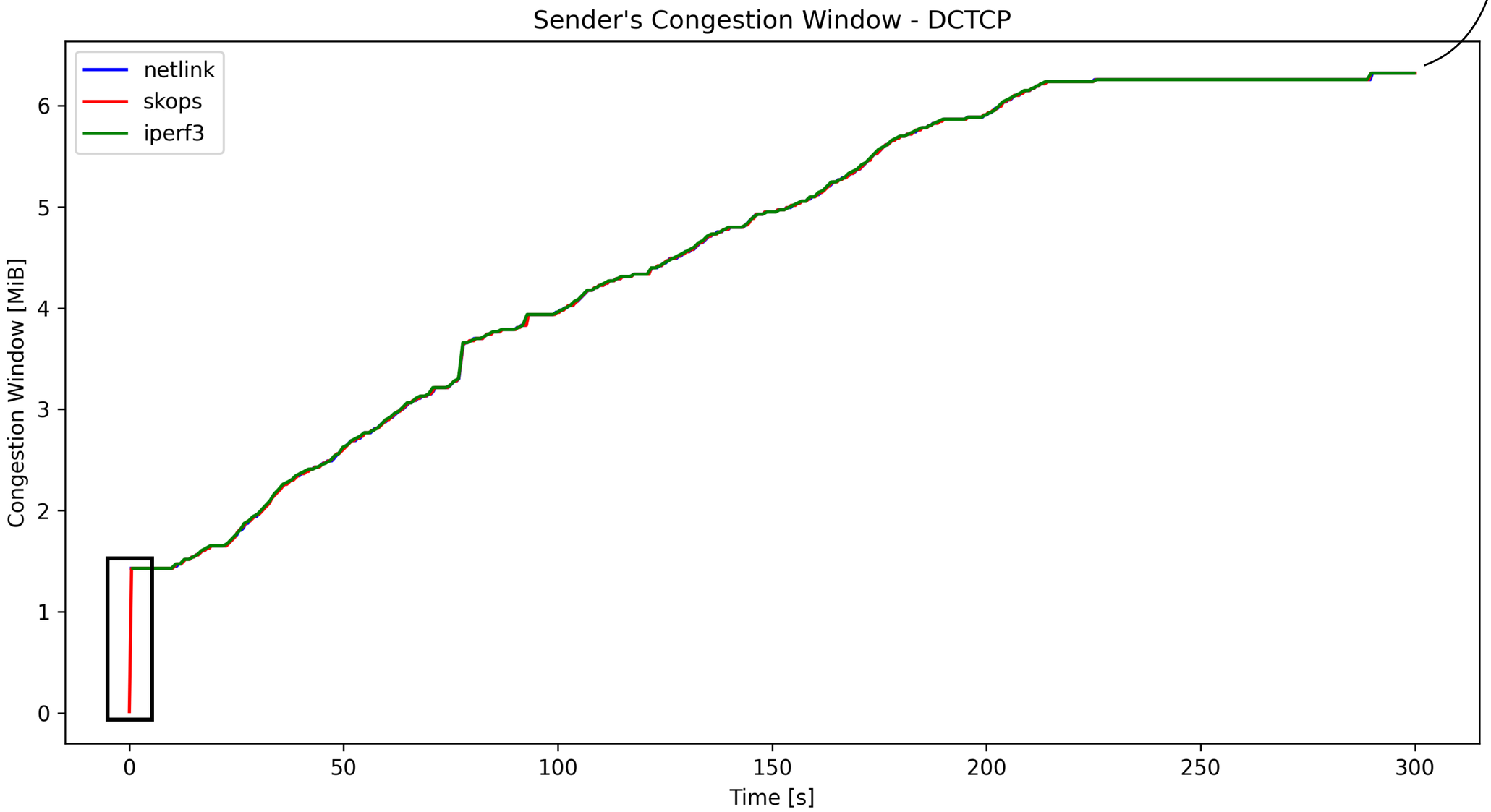
Great, but does it work?



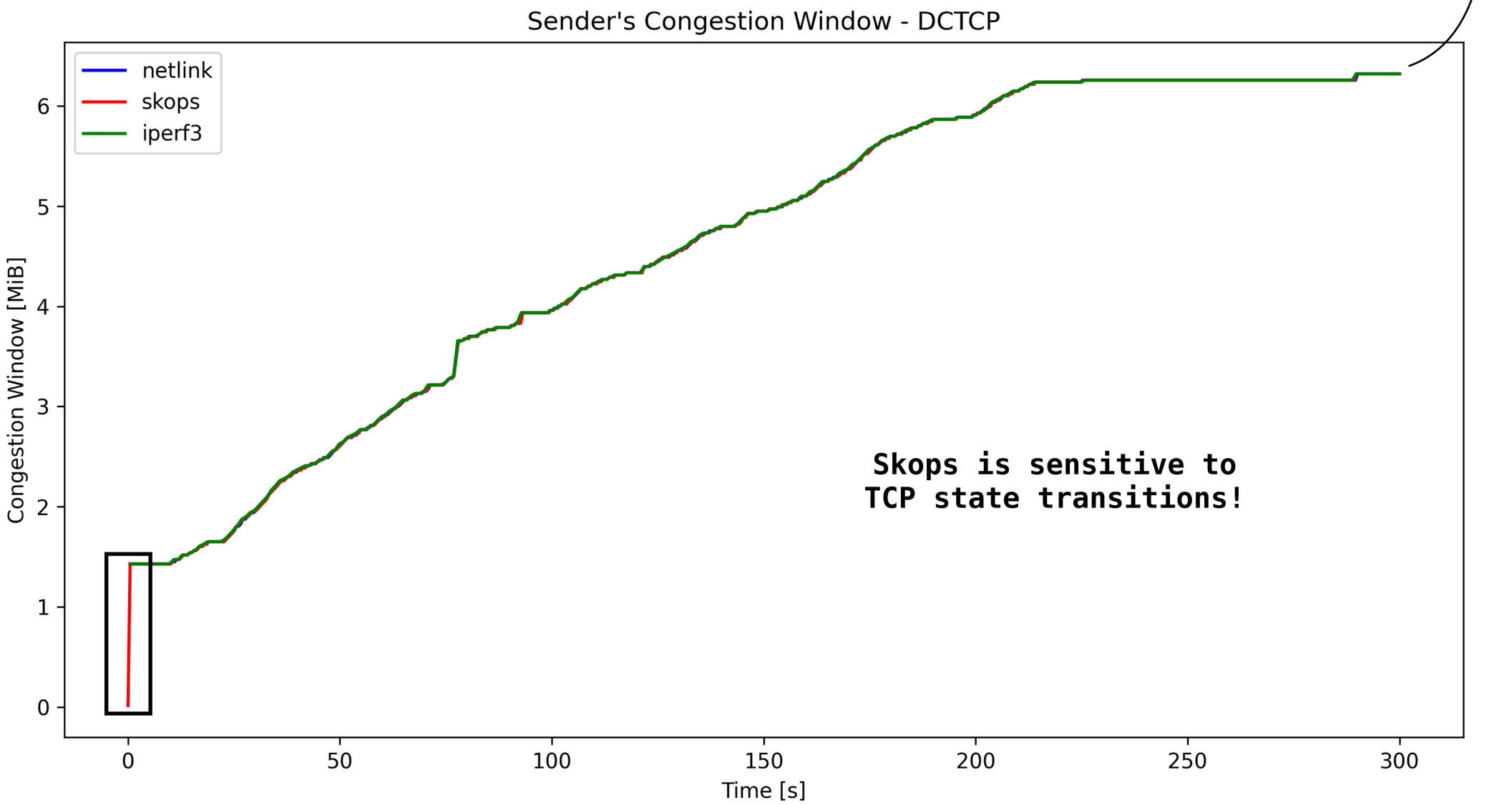
Great, but does it work?



Great, but does it work?



Great, but does it work?

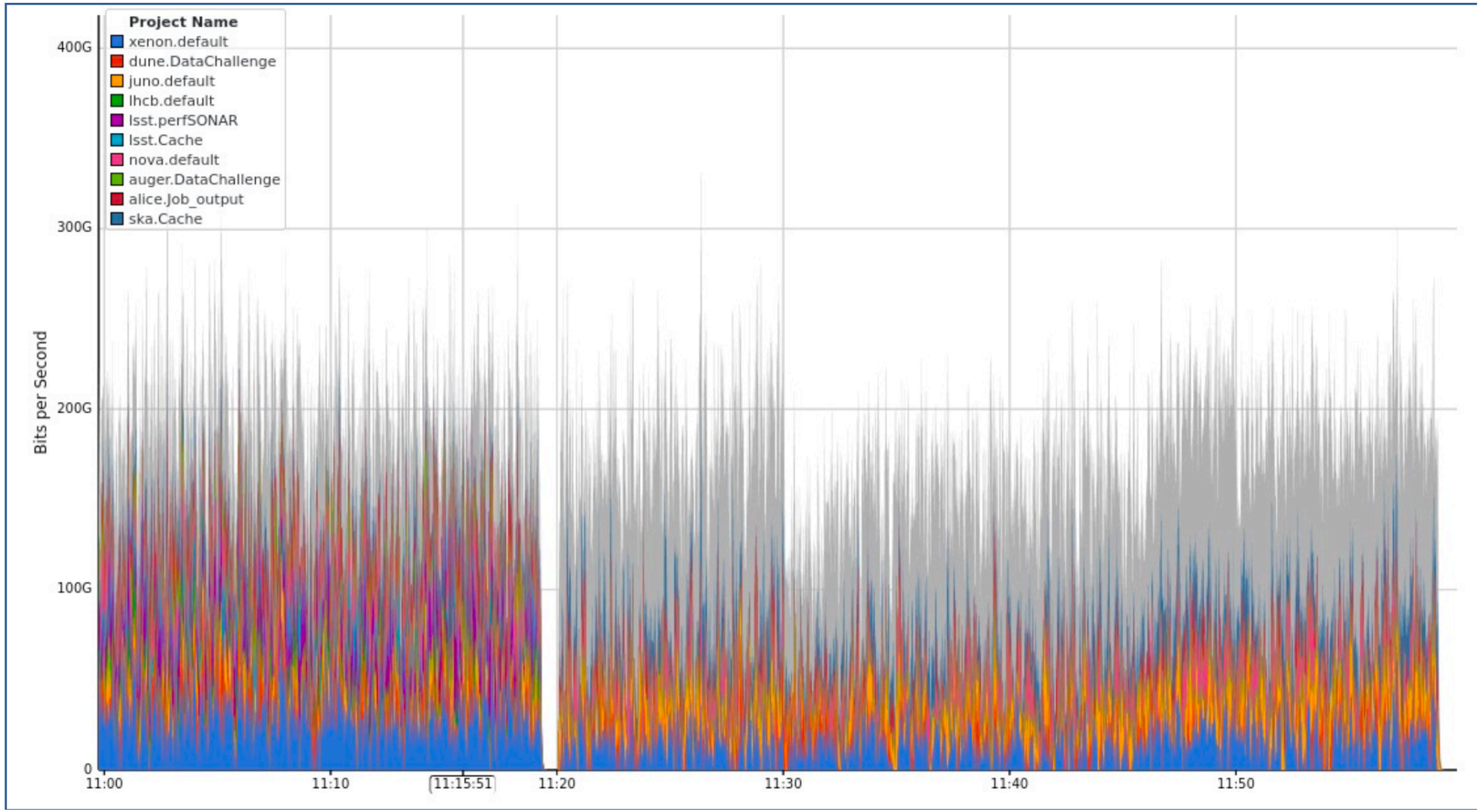


Demo at SuperComputing

- **SciTags** was **demonstrated** both at **SuperComputing** (SC) **23** and **24**.
- It'll be demoed at **SC25** too!
- The **summary** of **SC 24** can be found **here**.
- Note **flowd** was used instead of **flowd-go**.
- **Flowd-go** will be used for **SC25** though!

Demo at SuperComputing

Demo at SuperComputing



How can I try it out?

- **Flowd-go** is **distributed** as a **zero-dependency RPM** that:
  - **Embeds** the **eBPF programs** it uses.
  - Is **built** by CERN's **Koji** instance for both **x86_64** and **aarch64**.
  - Is **published** to CERN's **RPM repositories**.
  - Is **integrated** with **systemd(1)**.
  - **Includes** a **sample YAML configuration** file.
  - **Ships** with a **comprehensive manpage**: **flowd-go(1)**.
- We **encourage** you to **give** it a **go** (pun intended).

Summing up…

- The **SciTags** initiative is in **production use**:
  - Major **LHC experiments** are already running it in production.
  - **Sites** are **encouraged** to **enable** it in production.
  - **Network providers** are **rolling** out **collectors**.
  - **Adoption** during **Data Challenge 2024** shows great **potential**.
- We're **open** to **collaborate** with other **science** domains: **reach out**!
- **Flowd-go** is poised to become the **standard implementation**.
  - **IPv6** header **marking** should be **tested** in **production** this **year**.

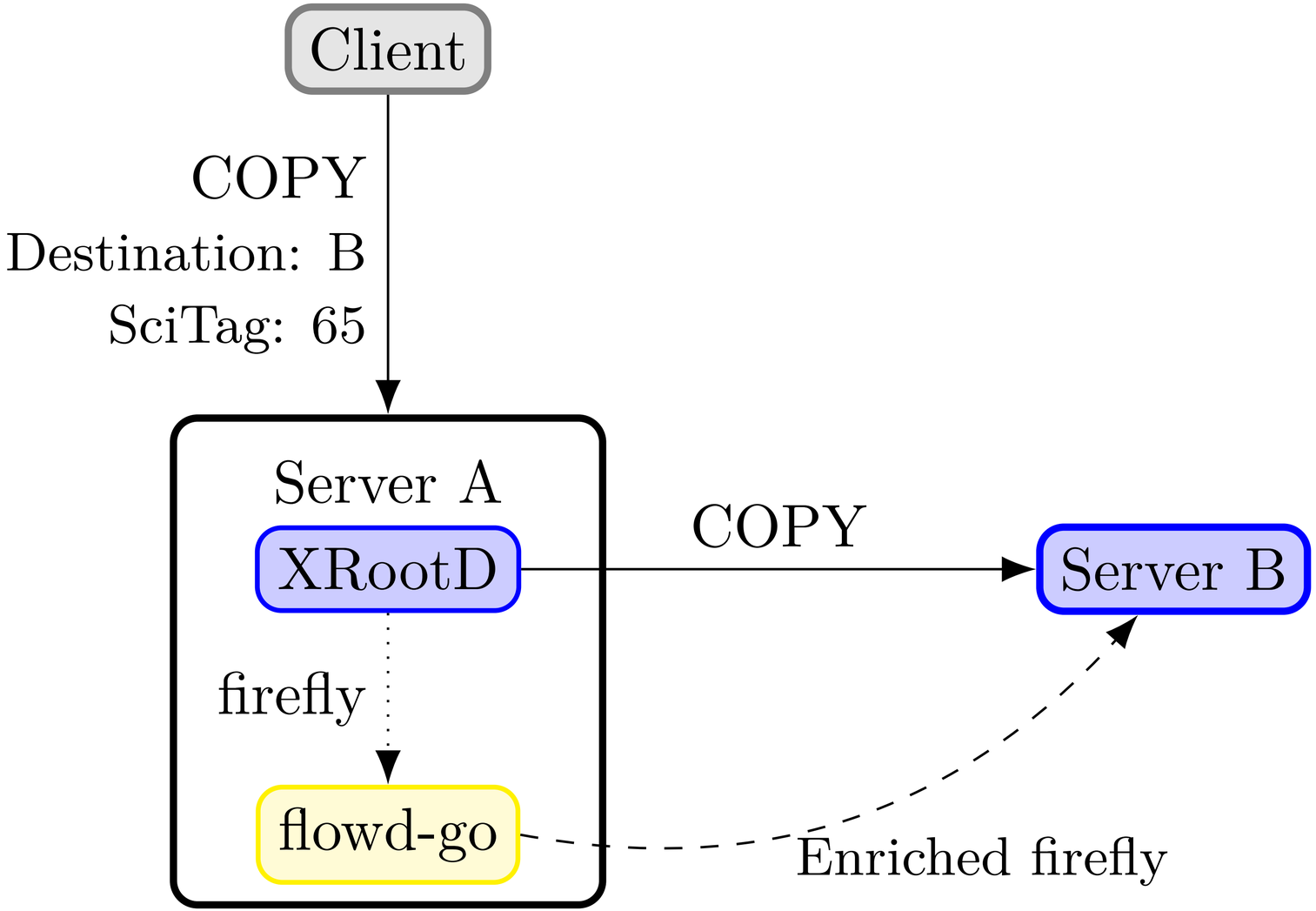
Questions?

Backup

An XRootD use case for flowd-go I

- Firefly **marking** is **controlled** through the **pmark setting**.
- **XrootD** can itself **send fireflies**... but the **information** it can **add** is **limited**!
  - The **getsockopt(2)** syscall provides **information**.
  - **Extending** it **requires recompiling** and **upgrading XRootD**...
  - But **what if** this **firefly** could be **enriched**?
- **Flowd-go** allows for a **workflow** where:
  1. **XRootD sends** a **firefly** upon connection **start/end** to **flowd-go**.
  2. Flowd-go **gathers** connection **information** on its **own**!

An XRootD use case for flowd-go II



An XRootD use case for flowd-go III


```
void XrdNetPMarkFF::SockStats(struct sockStats &ss) {
#ifndef __linux__
   memset(&ss, 0, sizeof(struct sockStats));
#else
   EPName("SockStats");
   struct tcp_info tcpInfo;
   socklen_t  tiLen = sizeof(tcpInfo);

// The data returned is from the server's perspective. This must be
// resolved by the caller as to which perspective should be presented.
// Note that for put requests the source is the client.
//
   if (getsockopt(sockFD, IPPROTO_TCP, TCP_INFO, (void *)&tcpInfo, &tiLen) == 0)
      {ss.bRecv = static_cast<uint64_t>(tcpInfo.tcpi_bytes_received);
       ss.bSent = static_cast<uint64_t>(tcpInfo.tcpi_bytes_acked);
       ss.msRTT = static_cast<uint32_t>(tcpInfo.tcpi_rtt/1000);
       ss.usRTT = static_cast<uint32_t>(tcpInfo.tcpi_rtt%1000);
      } else {
        memset(&ss, 0, sizeof(struct sockStats));
        DEBUG("Unable to get TCP information errno=" << strerror(errno));
      }
#endif
}
```

What if we're running on different kernel versions?

What if we want to extract more detailed information?

Can we only poll for information?

src/XrdNet/XrdNetPMarkFF.cc