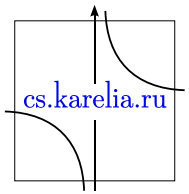
# GetTCP+: Performance Monitoring System at Transport Layer

Aleksandr Sannikov, Olga Bogoiavlenskaia, Iurii Bogoiavlenskii



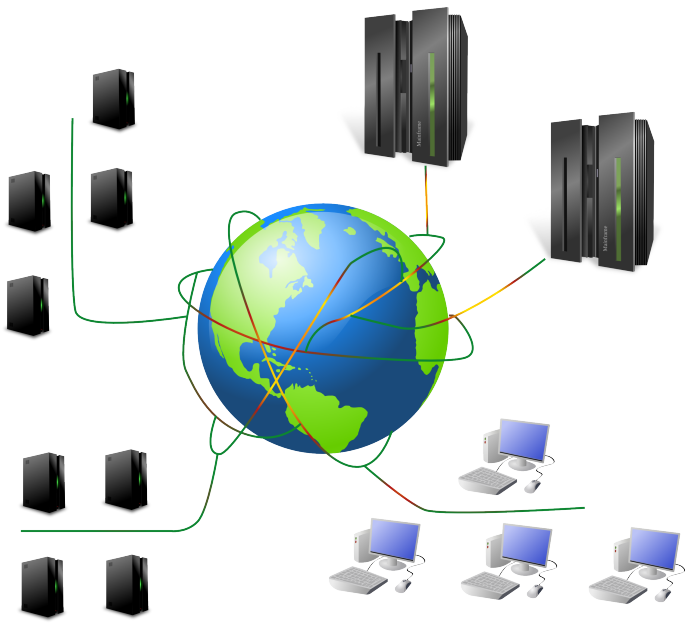

Department of Computer Science

Petrozavodsk State University

# Data communication performance monitoring

- Network design
- Network development
- Network administration

End-to-end path performance is quite necessary for wide class of application from LAN to GRID.



# GetTCP+: Common Points

Aim:

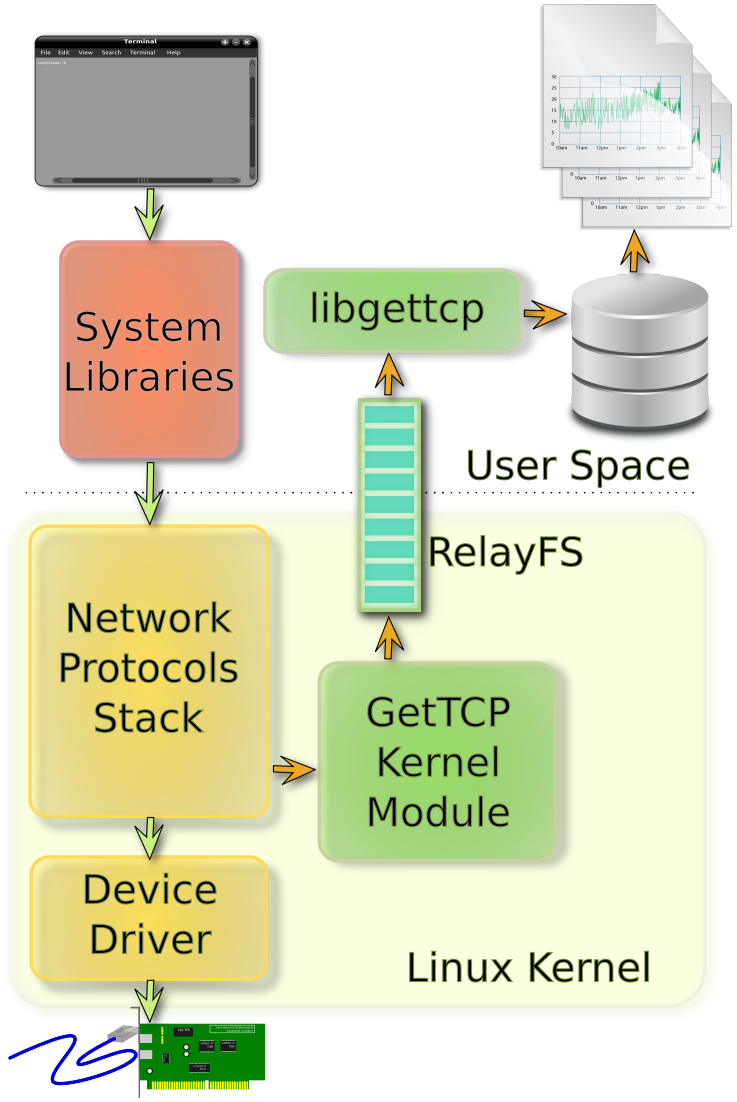
- Development of monitoring system that should be able to provide characteristics of the end-to-end path performance, both general (flow level) and detailed (segment level).

Requirements:

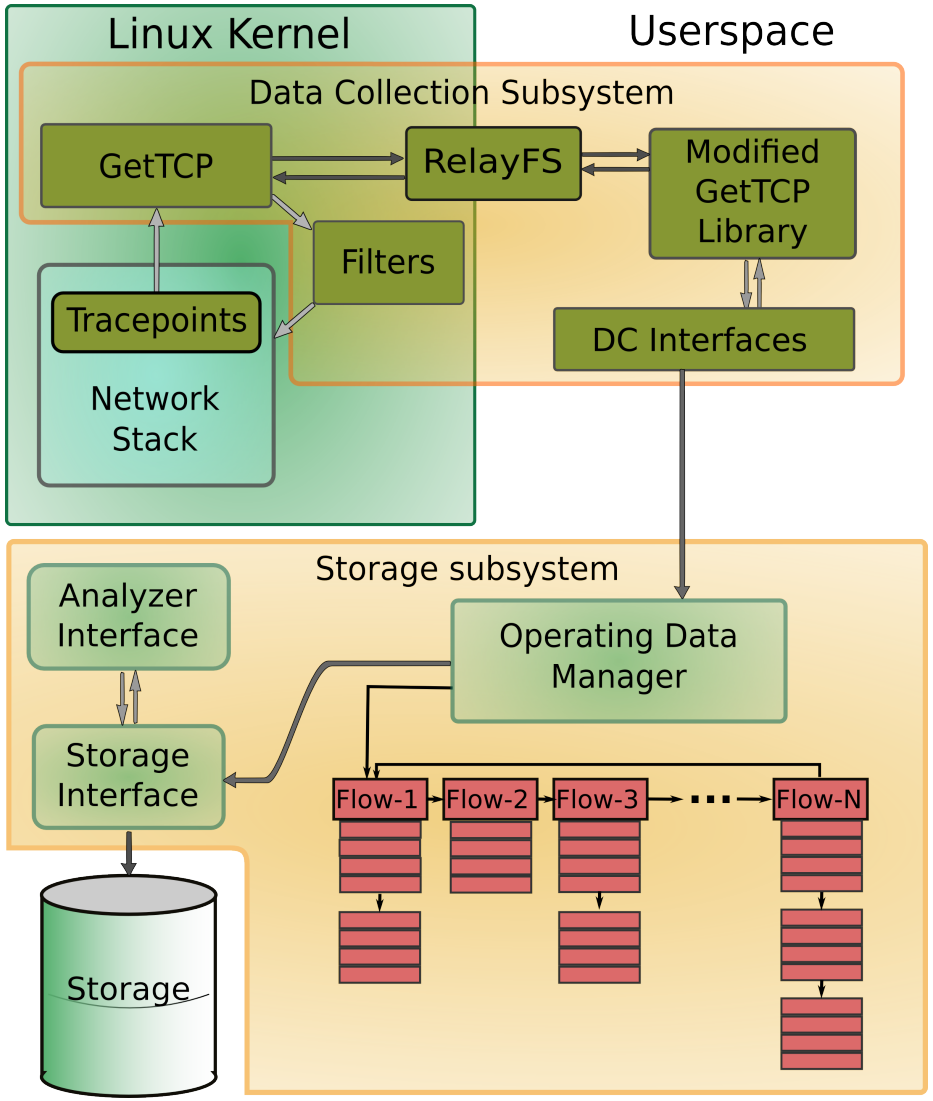
- High performance and low latency
- Flow filtering and clustering flows by destination
- Support of protocol-specific features and extensions
- Easy porting to modern kernel versions
- Real-time flow monitoring
- Providing of all necessary for end user information about flows

# GetTCP

- Provides mechanisms for recording events in Linux kernel network stack and transmits data recorded to the user space
- Operates in real-time without data losses and has no influence to the work the of the network stack



# GetTCP+


Linux Kernel
Userspace
Data Collection Subsystem
GetTCP
RelayFS
Modified GetTCP Library
Filters
Tracepoints
Network Stack
DC Interfaces
Storage subsystem
Analyzer Interface
Operating Data Manager
Storage Interface
Flow-1
Flow-2
Flow-3
Flow-N
Storage

# Storage subsystem

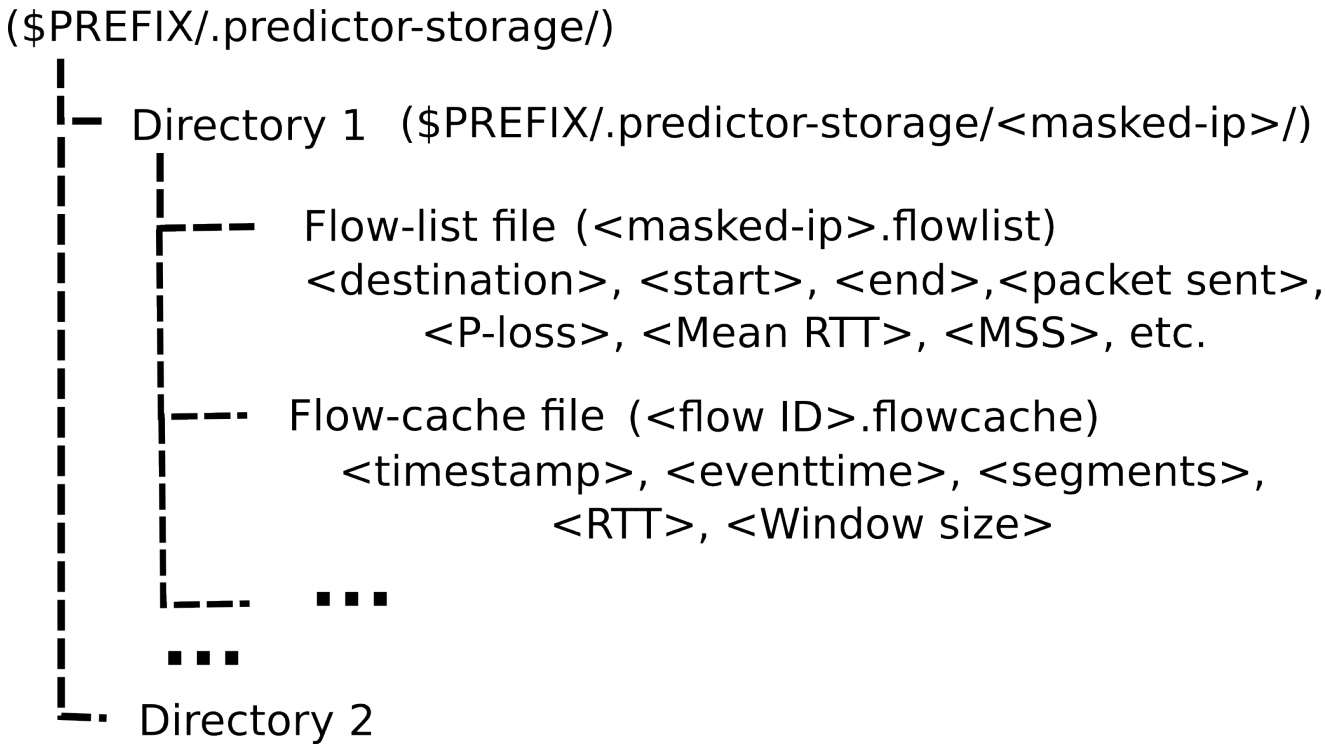
- Operating Data Manager - processing of incoming data with high performance
- Storage - long-term data storing
- Analyzer Interface - data interchange with external tools

Processing data have several specific points:

- ”Write-once” and ”Read-only” data
- Sequential data access
- Processing by big slices
- Old data are less important

# Storage organization

- Storage implementation is based on file system objects.
- Flows are grouped by destination
- Storing of common information for each flow and verbose information for each segment



# Storage data

| Start | End | N | Lost | cRTT | W | MSS | RTTn |
|---|---|---|---|---|---|---|---|
| 1321882813612897 | 1321882868341552 | 738881 | 250 | 73229 | 359 | 1448 | 19249 |
| 1321882898261708 | 1321882932915678 | 741536 | 278 | 84433 | 360 | 1448 | 18946 |
| 1321882971857777 | 1321883016067644 | 741536 | 318 | 66094 | 331 | 1448 | 18879 |
| 1321883037896449 | 1321883087628256 | 739916 | 269 | 73361 | 360 | 1448 | 19894 |
| 1321883120172231 | 1321883163385798 | 739717 | 281 | 60854 | 373 | 1448 | 19551 |
| 1321883182680274 | 1321883224470498 | 739220 | 259 | 66542 | 363 | 1448 | 19867 |
| 1321883243410327 | 1321883263863716 | 673184 | 275 | 22489 | 343 | 1448 | 17884 |

Table : Storage data

| RecType | Timestamp | nsegs | lostout | window | RTT |
|---|---|---|---|---|---|
| 0 | 3968509408 | 1 | 0 | 10 | 3804 |
| 0 | 3968621408 | 1 | 0 | 10 | 3769 |
| 0 | 3968621408 | 1 | 0 | 10 | 3769 |
| 0 | 3968621408 | 1 | 0 | 11 | 4052 |
| 0 | 3968621408 | 1 | 0 | 12 | 3994 |
| 0 | 3968621408 | 1 | 0 | 12 | 4204 |
| 0 | 3968621408 | 1 | 0 | 13 | 4073 |
| 0 | 3968621408 | 1 | 0 | 14 | 3658 |
| 0 | 3968621408 | 1 | 0 | 14 | 3782 |

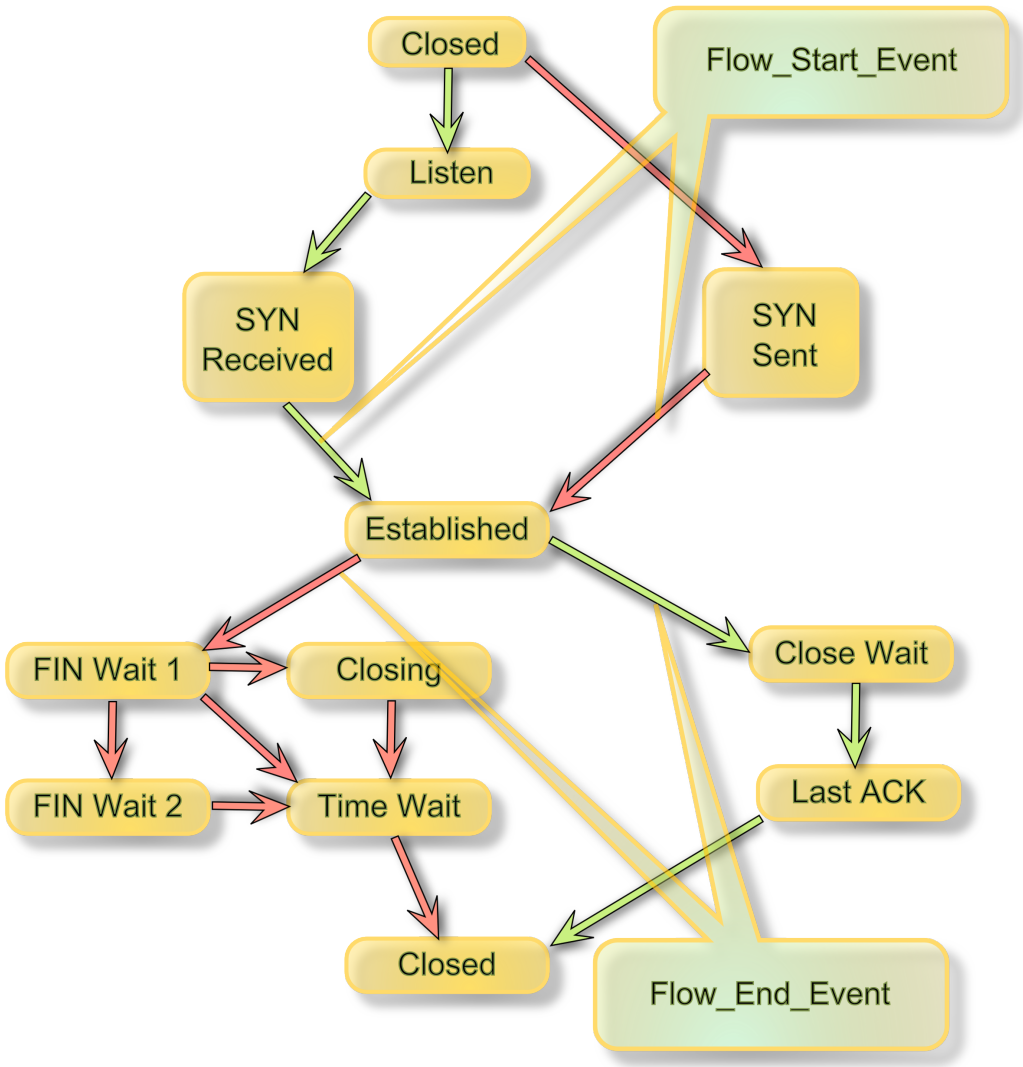
Table : Flow cache file

# Using the Linux Kernel Tracepoints

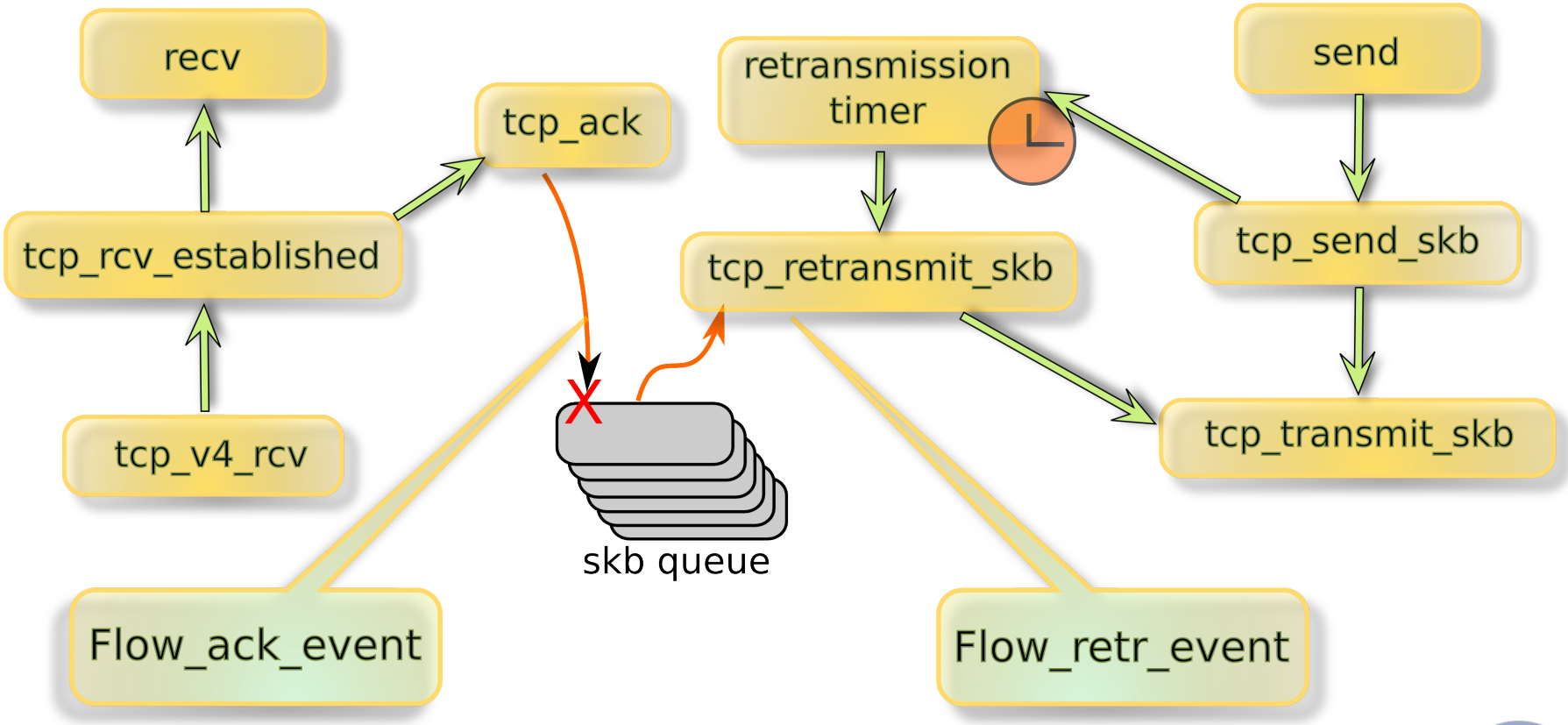
Linux Kernel Tracepoints present the main mechanism of event tracing in GetTCP+.

- A tracepoint placed in the code provides a hook to call a function that can be provided at runtime
- A tracepoint can be ”on” or ”off” at runtime
- They are lightweight hooks that can pass an arbitrary number of parameters
- Handlers can be implemented in kernel module
- Less volume of changes of mainline kernel code is necessary
- New hooks can be added easily

# Events connected with flow state



# Events connected with segment transmission



# Segmentation Offloading

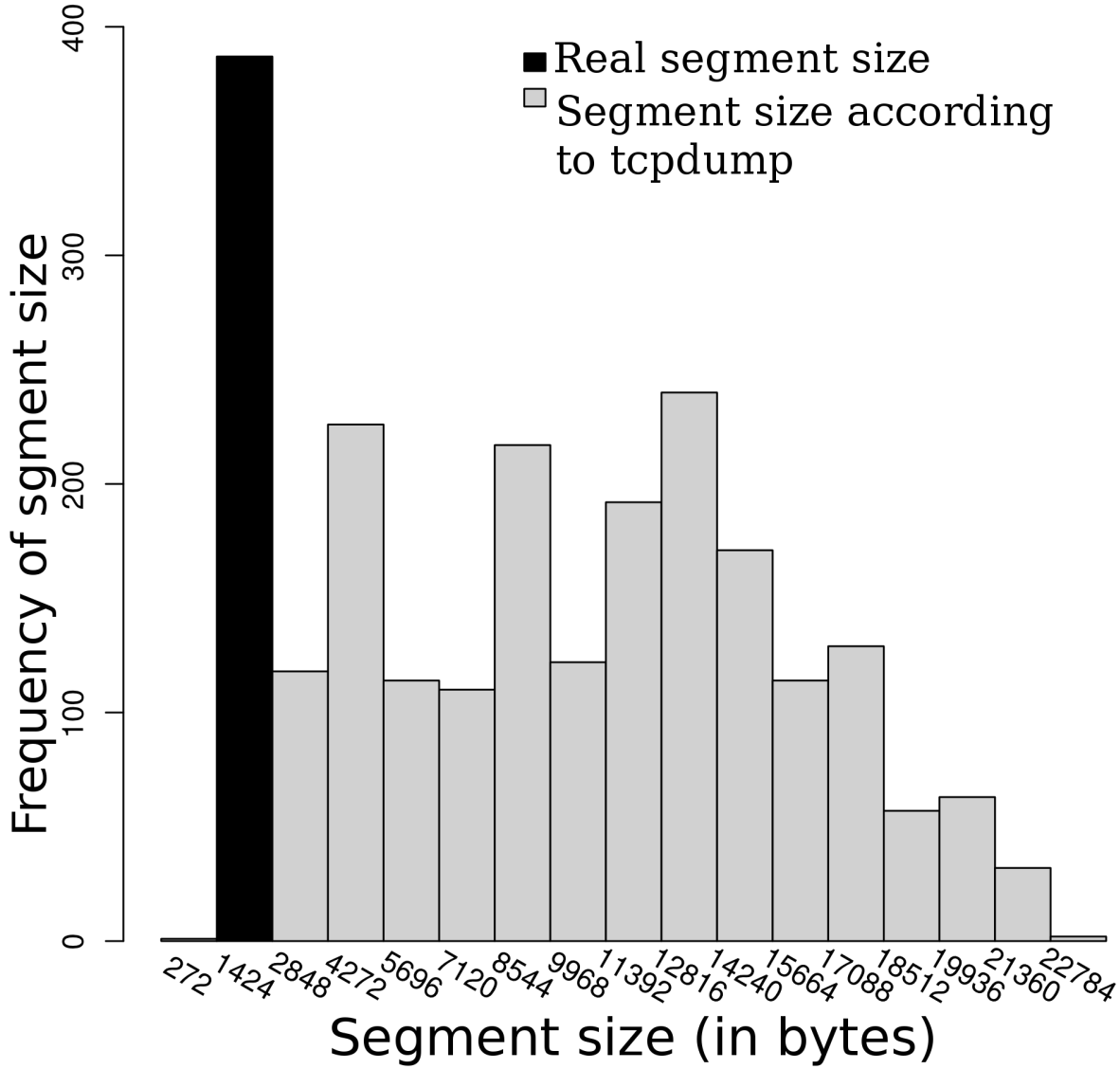
Segmentation - separation of data block on several segments for transmission.

- GSO – Generic Segmentation Offloading
- GRO – Generic Receive Offloading
- TSO – TCP Segmentation Offloading

Problem:

- Information about Segmentation Offloading is unavailable in user space.
- GetTCP+ provides correct information about offloading.

# TCP Segment size: real and according to tcpdump

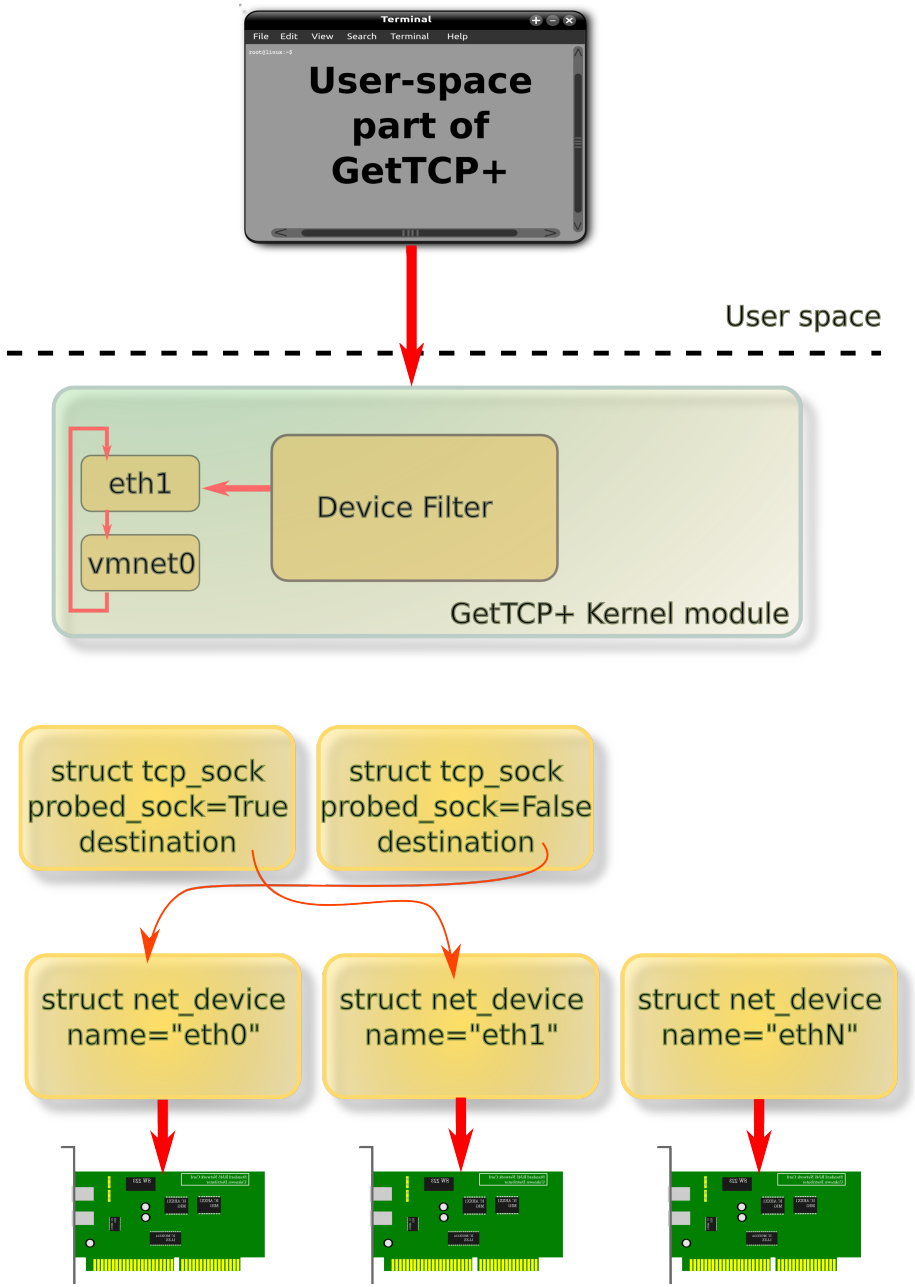


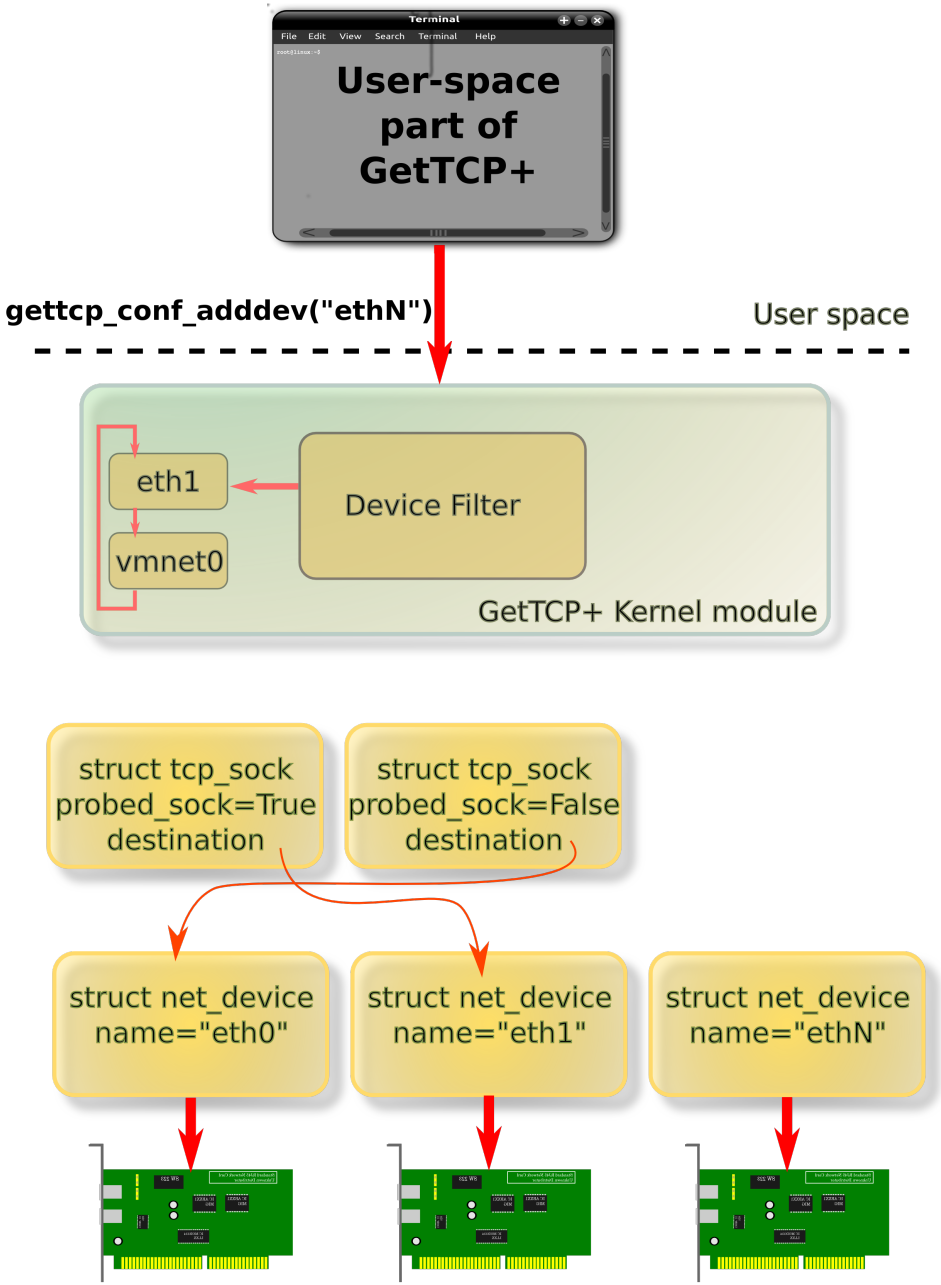
# Filtering

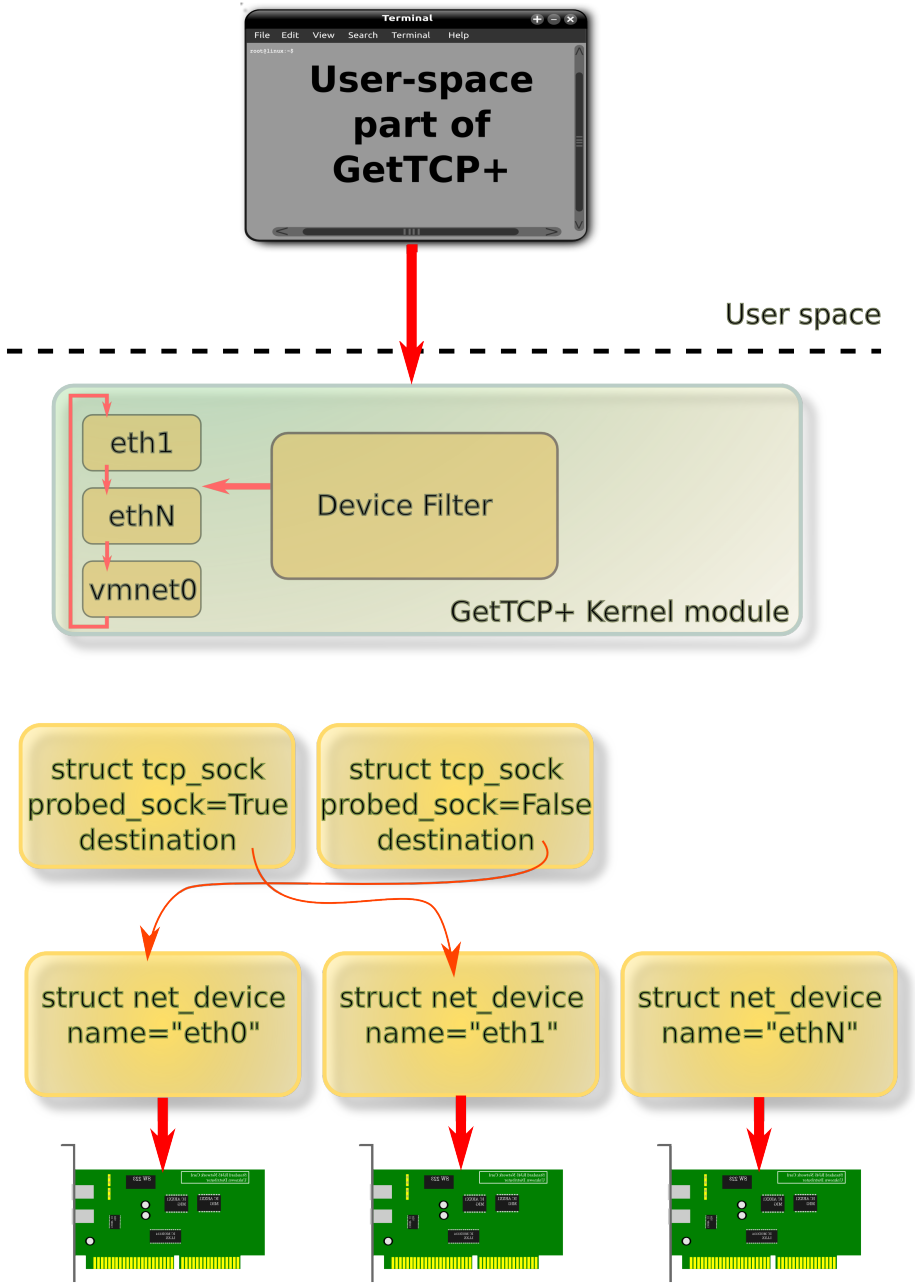
Not every flow is important for analysis, especially in the context of the distributed systems.
Two types of filtering:

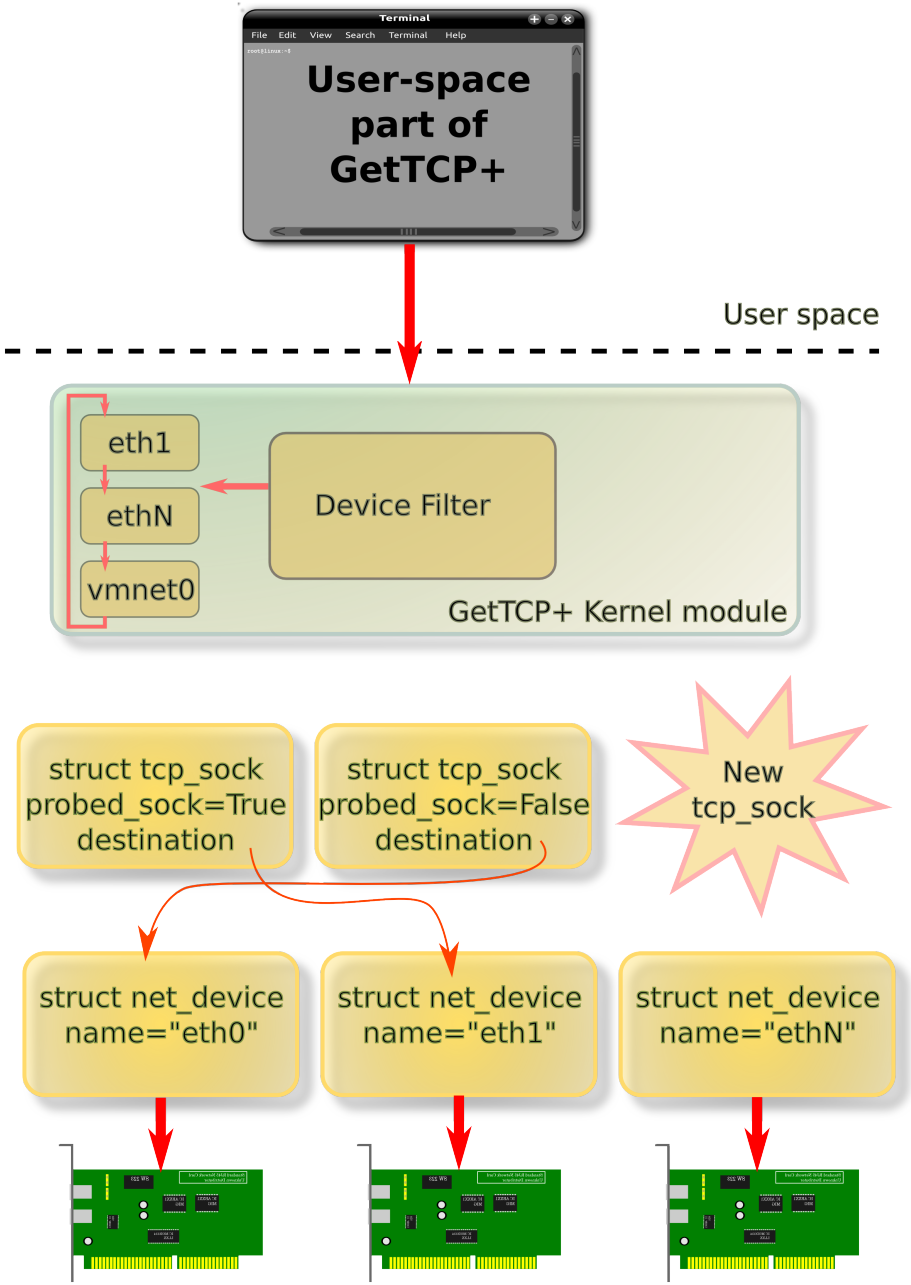
- By using device
- By destination host/subnetwork address (currently IPv4 only)

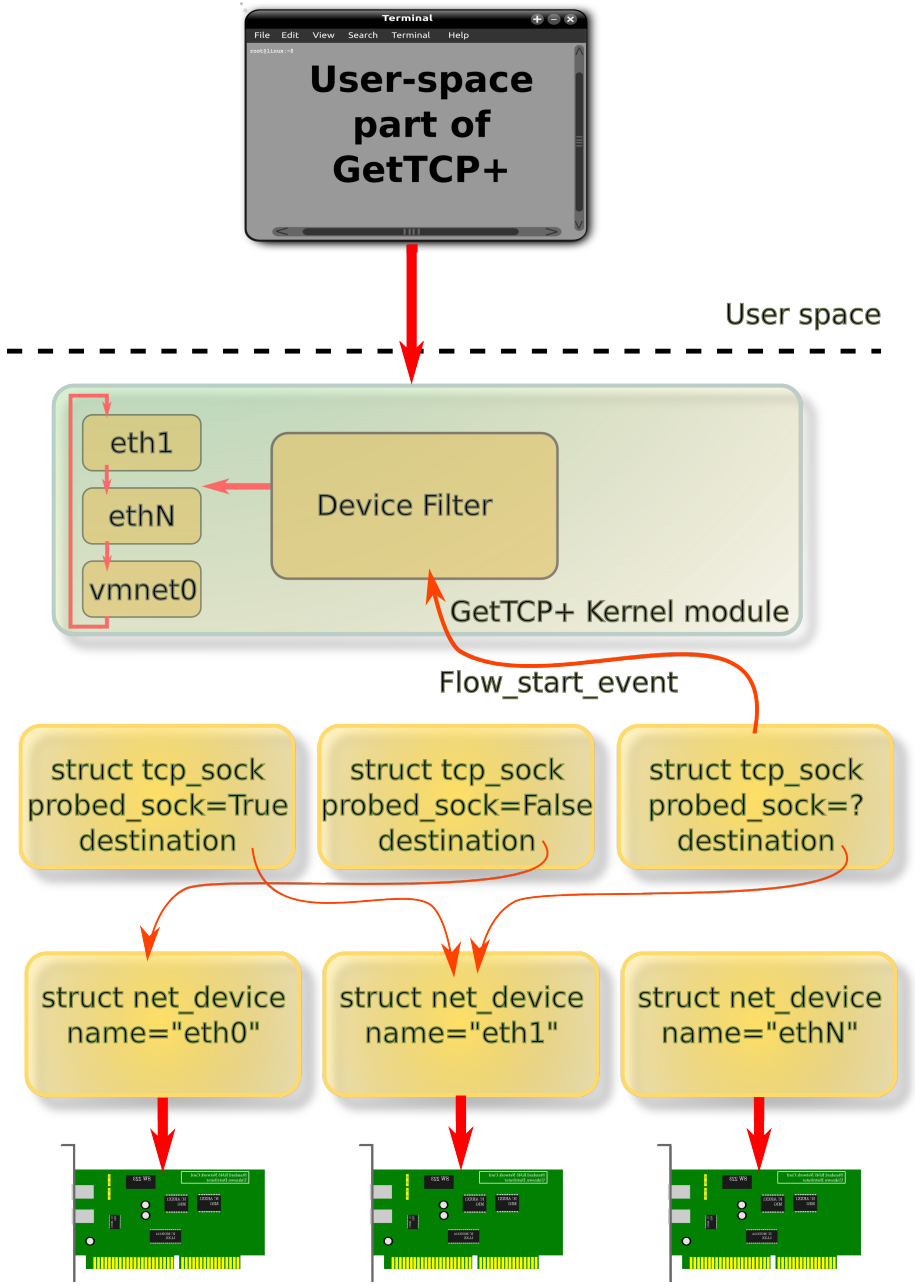
Filtering is made for every flow and necessity of monitoring is defined by tcp_sock→probed_sock field of tcp_sock structure.

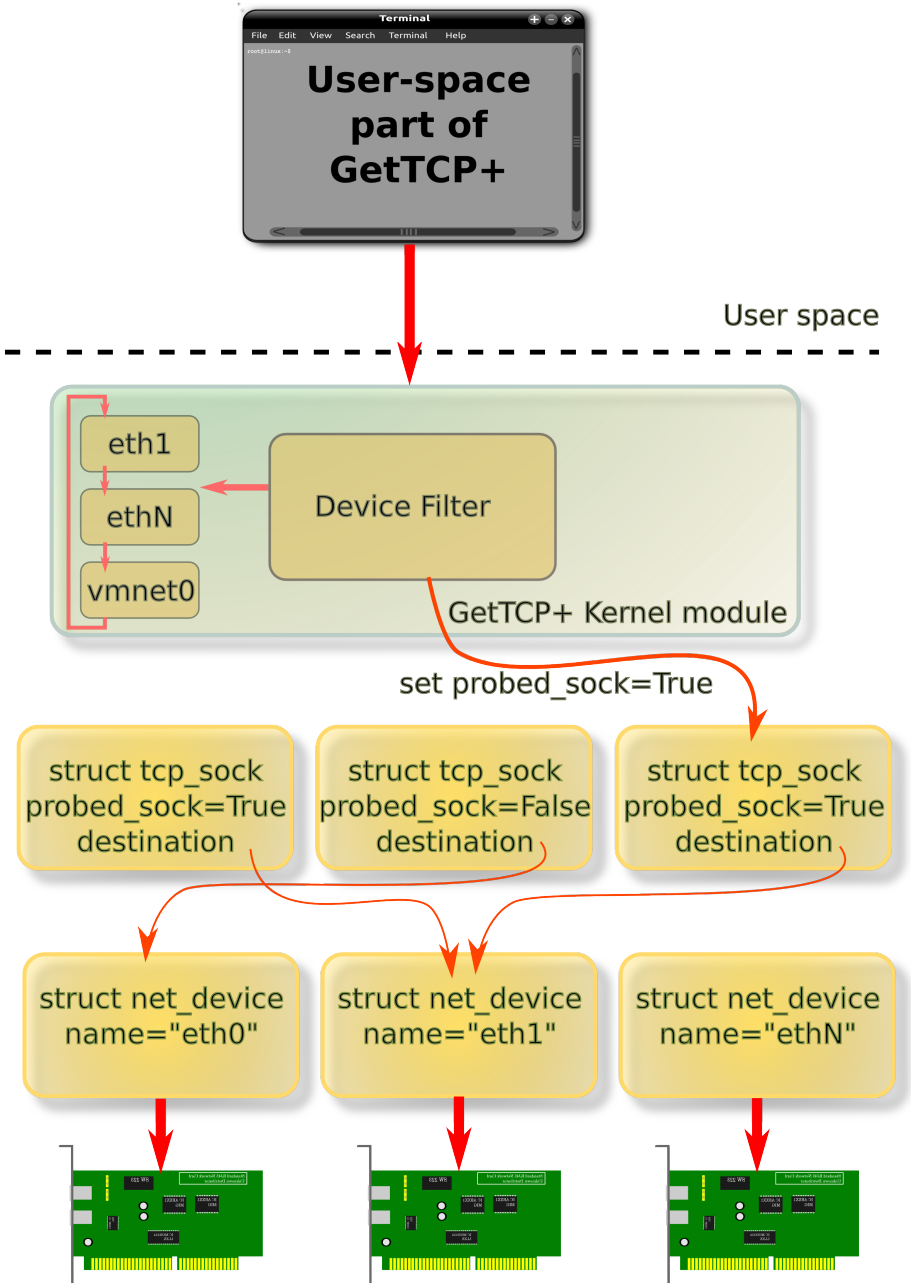

Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
User space
eth1
vmnet0
Device Filter
GetTCP+ Kernel module
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"


Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
gettcp_conf_adddev("ethN")
User space
eth1
vmnet0
Device Filter
GetTCP+ Kernel module
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"


Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
User space
eth1
ethN
vmnet0
Device Filter
GetTCP+ Kernel module
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"


Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
User space
eth1
ethN
vmnet0
Device Filter
GetTCP+ Kernel module
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
New tcp_sock
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"


Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
User space
eth1
ethN
vmnet0
Device Filter
GetTCP+ Kernel module
Flow_start_event
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
struct tcp_sock probed_sock=? destination
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"


Terminal
File Edit View Search Terminal Help
User-space part of GetTCP+
User space
eth1
ethN
vmnet0
Device Filter
GetTCP+ Kernel module
set probed_sock=True
struct tcp_sock probed_sock=True destination
struct tcp_sock probed_sock=False destination
struct tcp_sock probed_sock=True destination
struct net_device name="eth0"
struct net_device name="eth1"
struct net_device name="ethN"

# Filtering by destination host/subnetwork address

Address filtering works by the same way to device filtering.

IP $\Leftarrow$ (inet_sock(tcp_sock)$\rightarrow$inet_daddr);
**for** *current* $\in$ *filtered addresses list* **do**
  **if** *(IP & (current$\rightarrow$network_mask) = current$\rightarrow$address)* **then**
    tcp_sock$\rightarrow$probed_sock $\Leftarrow$ True
  **end**
**end**

If both of filtering types are enabled:

tcp_sock$\rightarrow$probed_sock $\Leftarrow$ (result of filtering by device) & (result of filtering by address)

# Testing

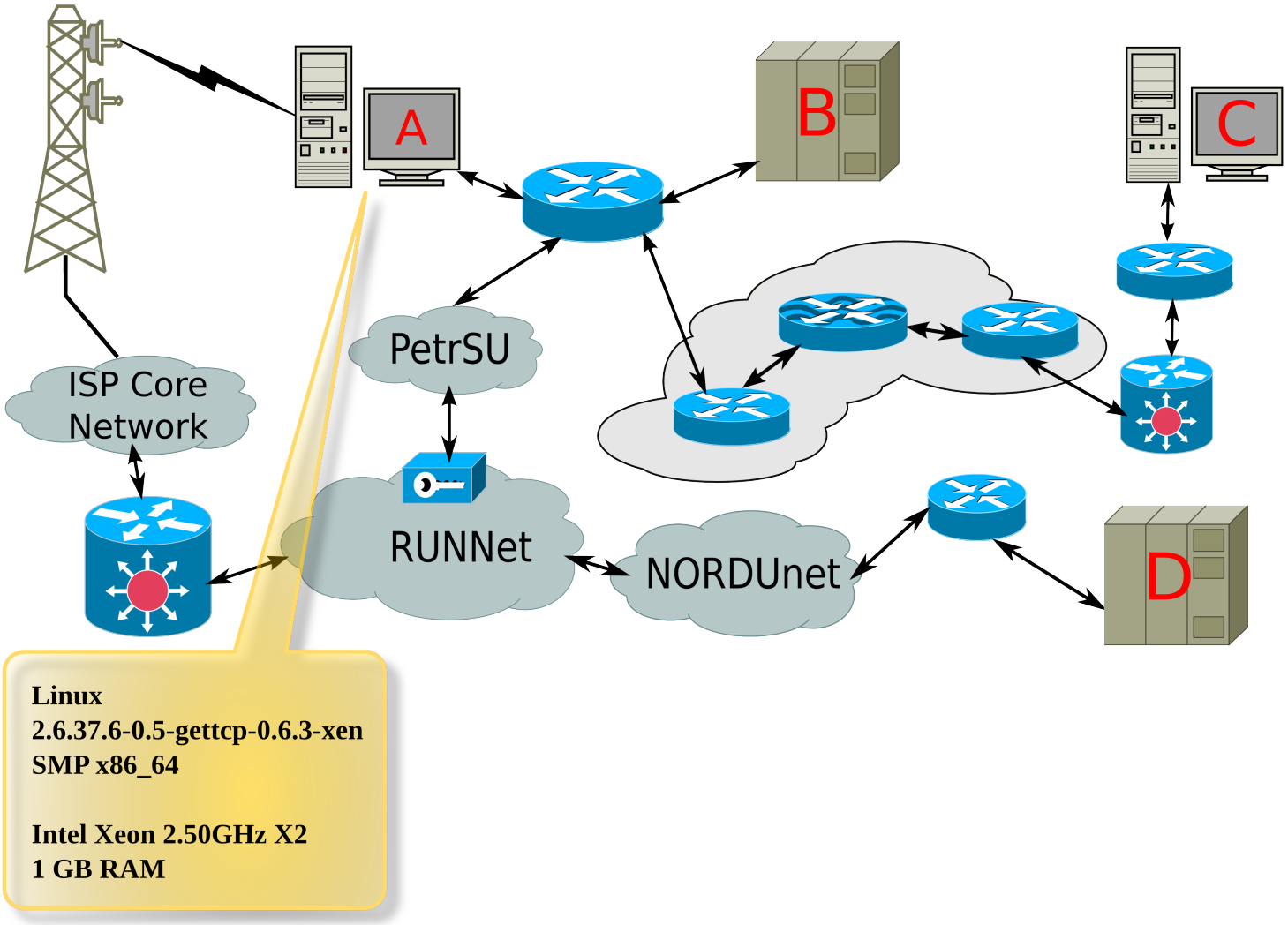
GetTCP+ was tested for several network configuration:

- high and low throughput
- small and relatively large RTT
- low and high loss rate

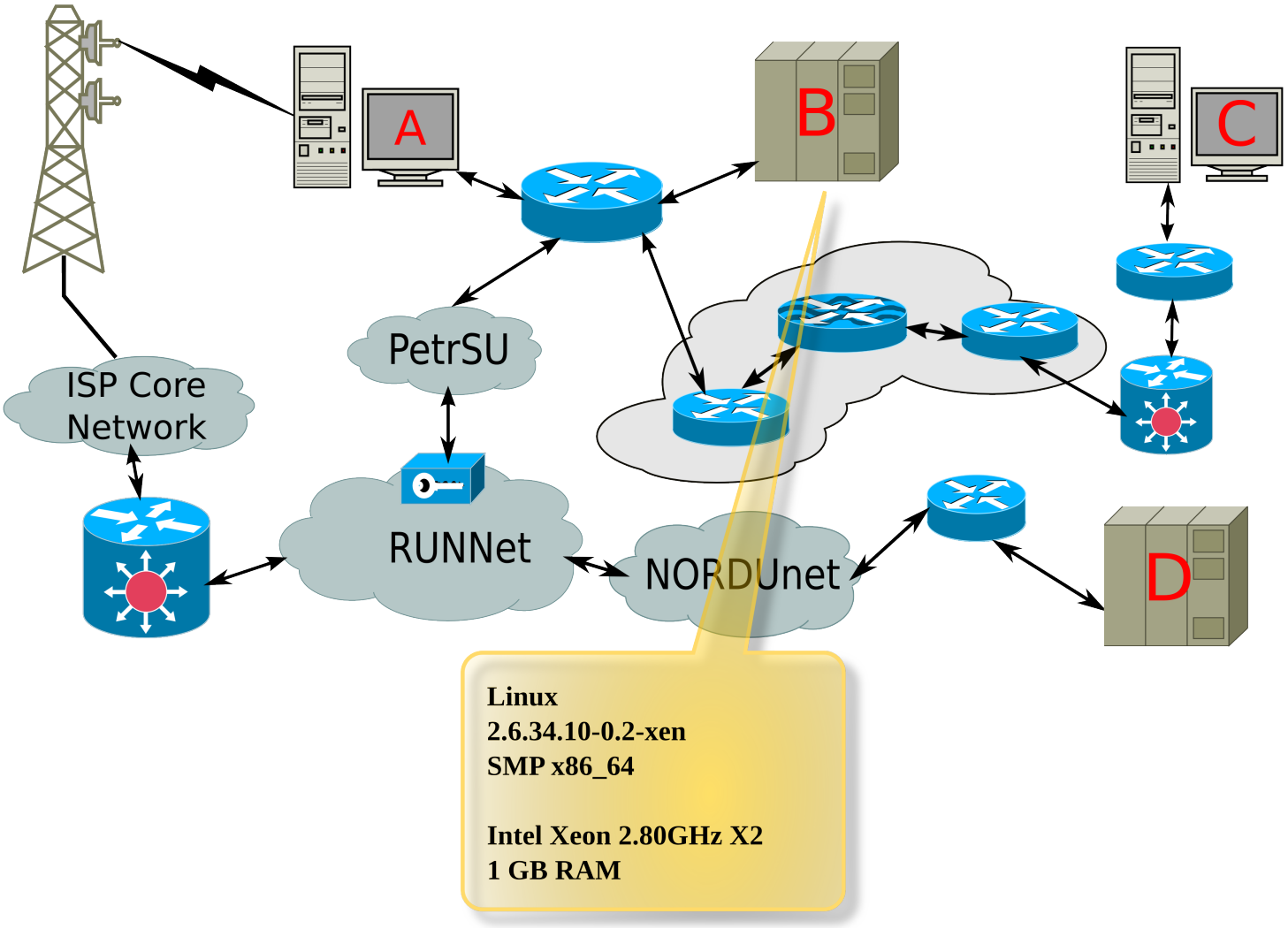
Traffic generator: **i**perf

- constant size of transferred data: 200 mb (exclude load testing)
- sequential, disjoint flows in each test
- used congestion control algorithm: TCP NewReno

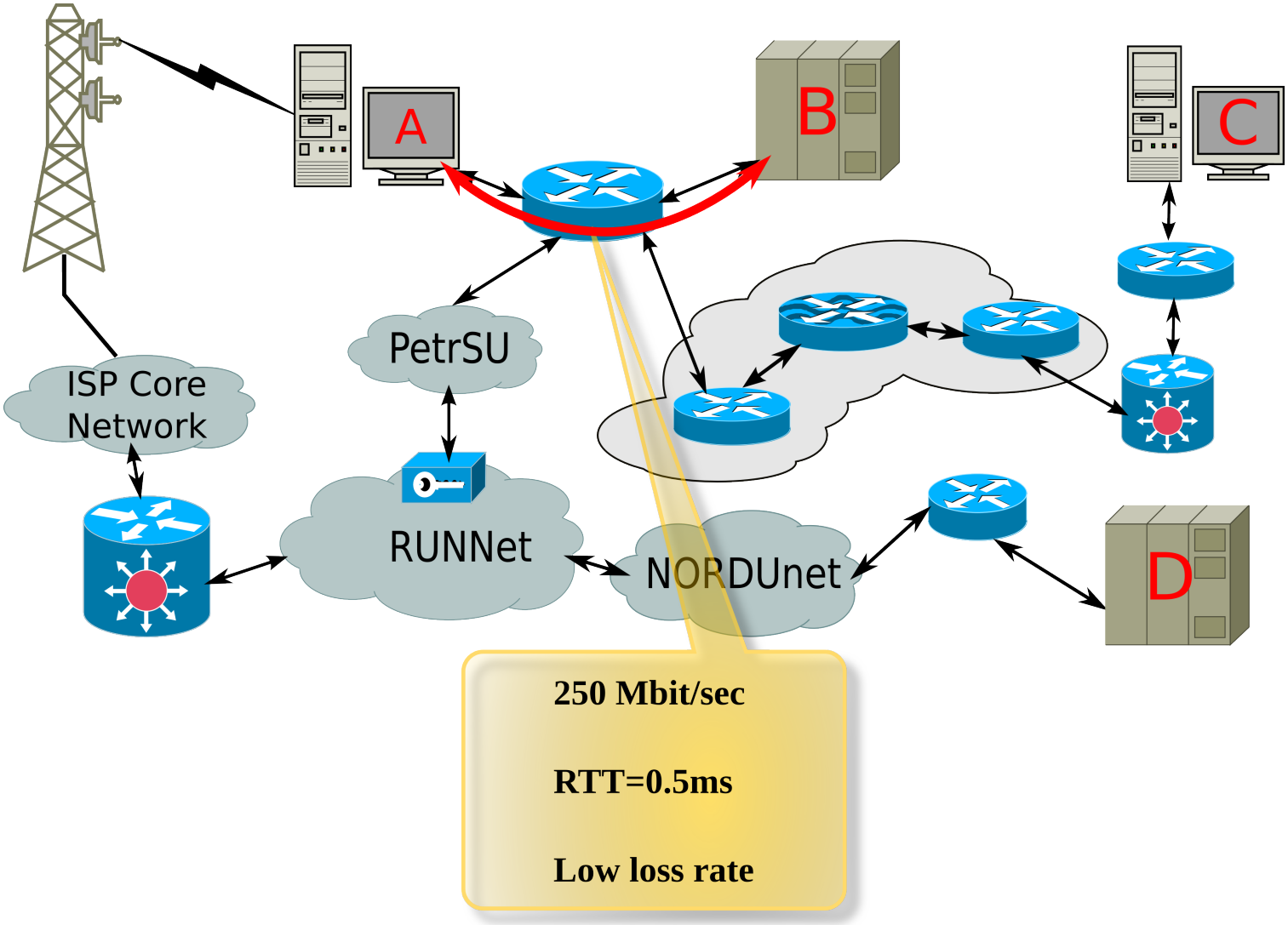
# The Source Host Configuration



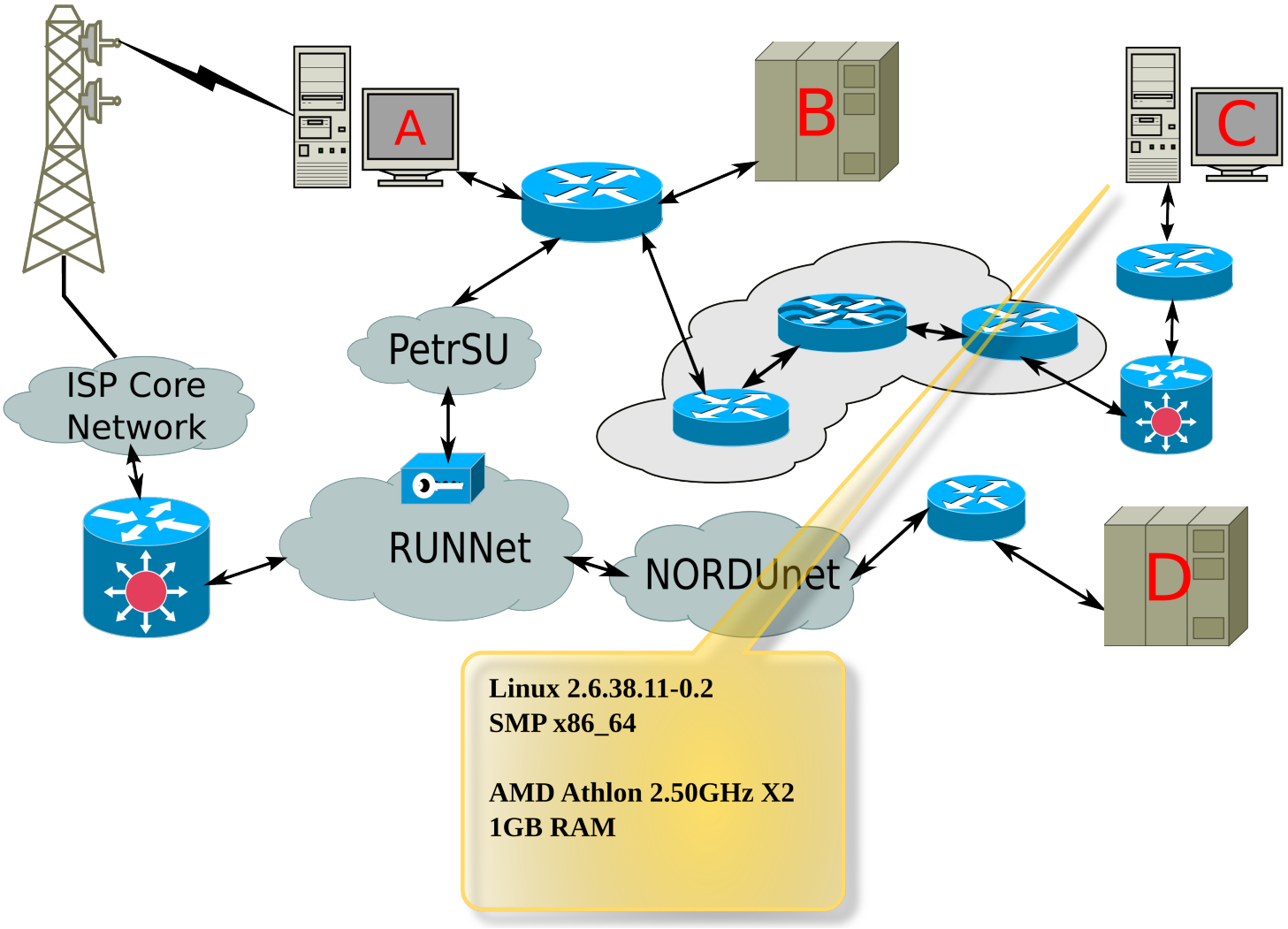
# The First Destination Host Configuration



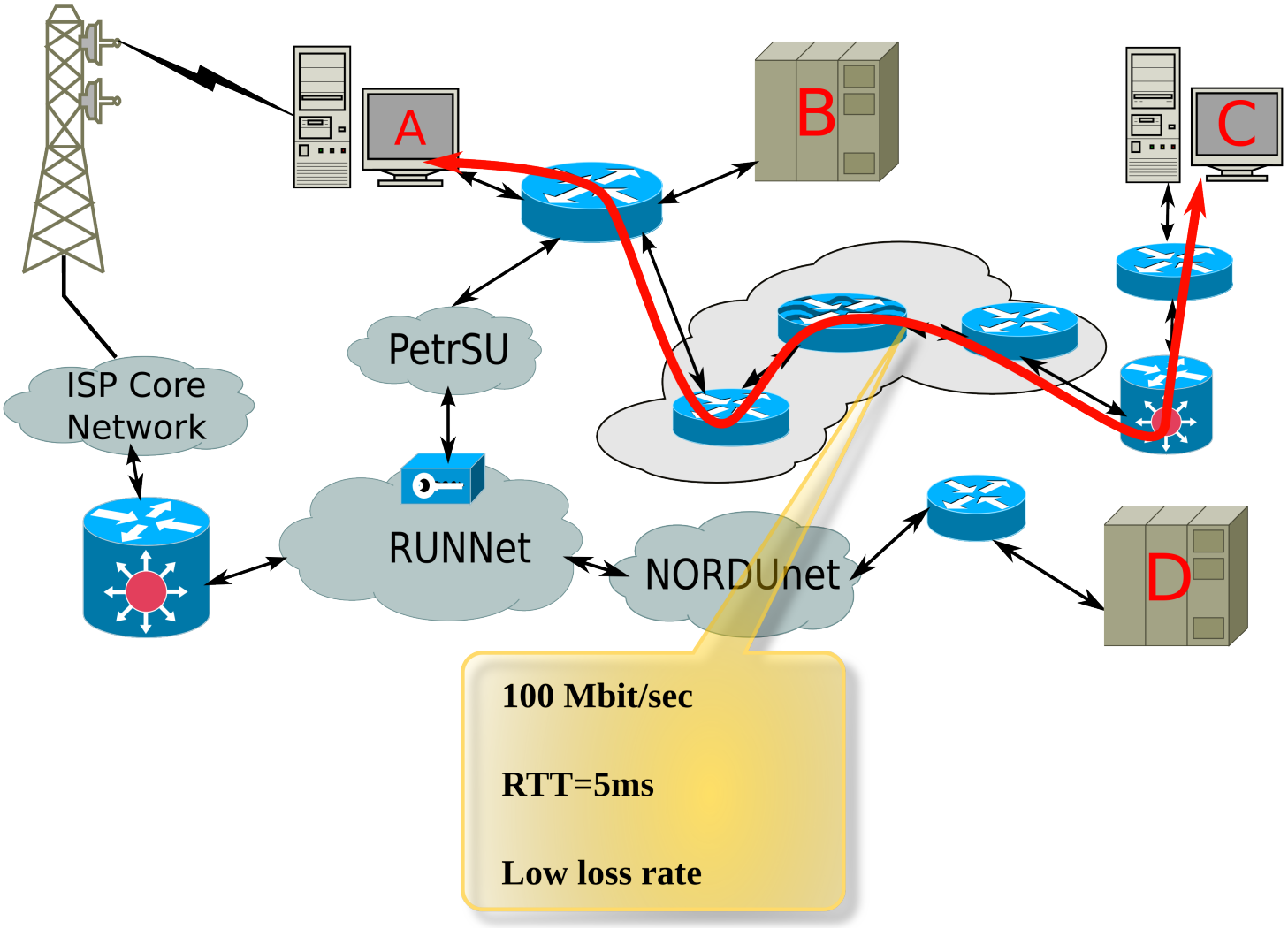
# Debug and Load Testing



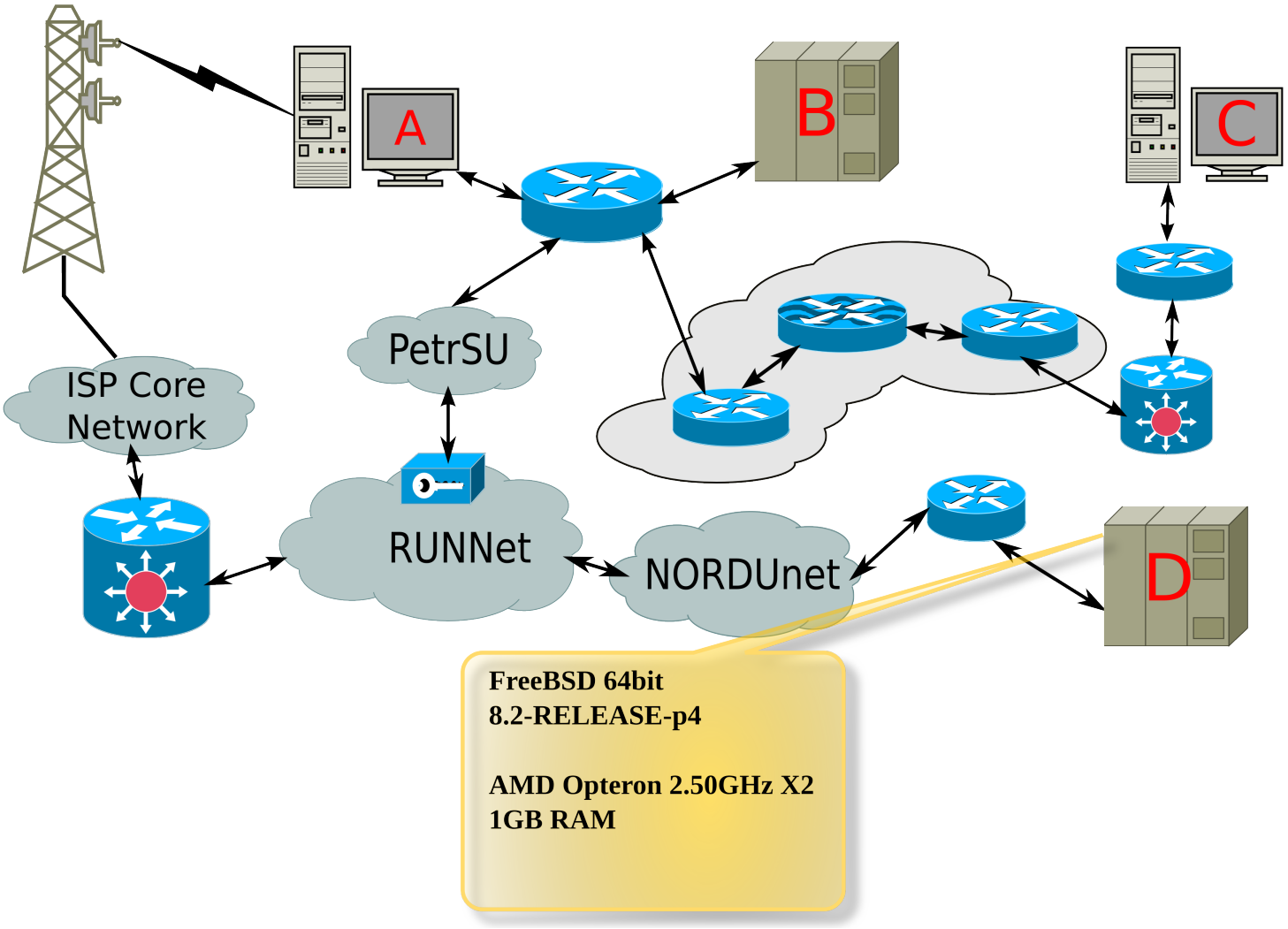
# The Second Destination Host Configuration



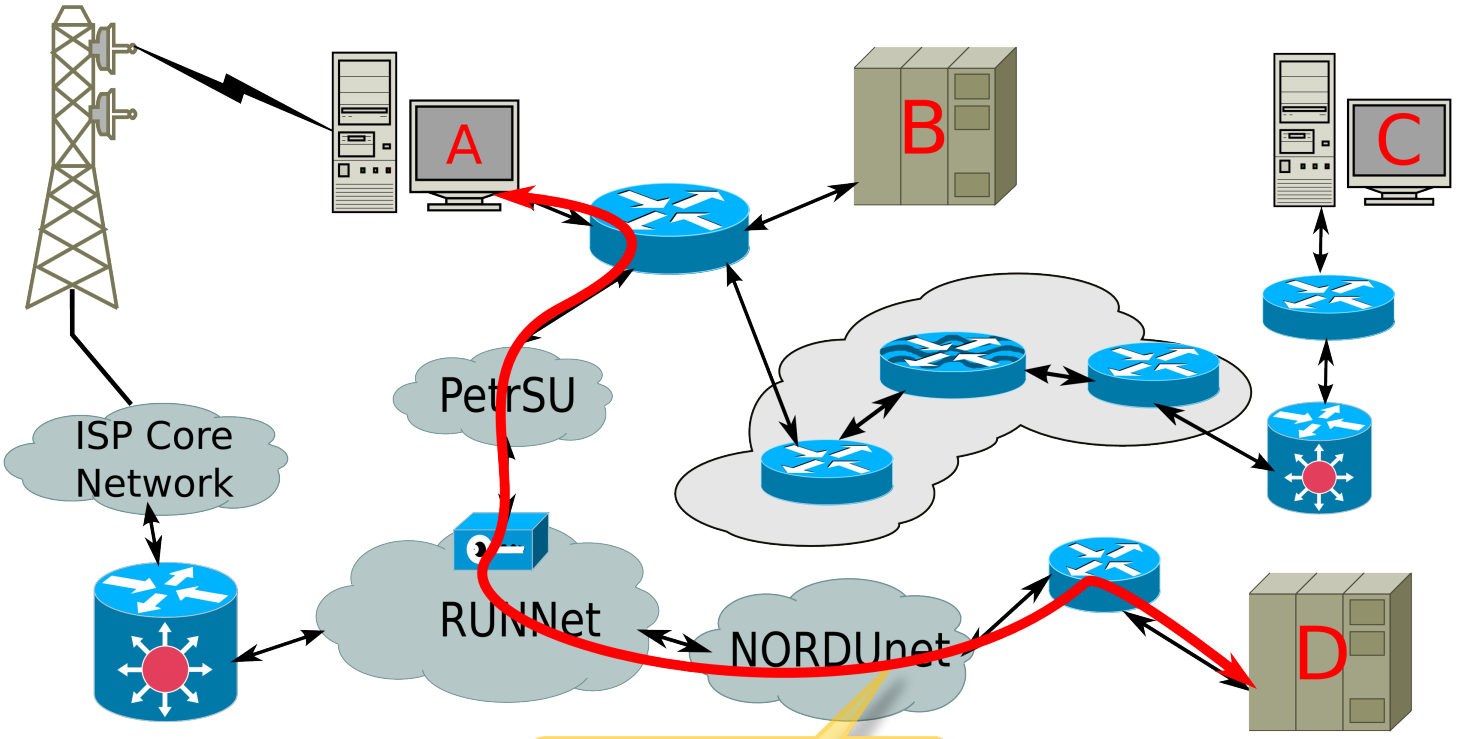
# Testing in the Local Network



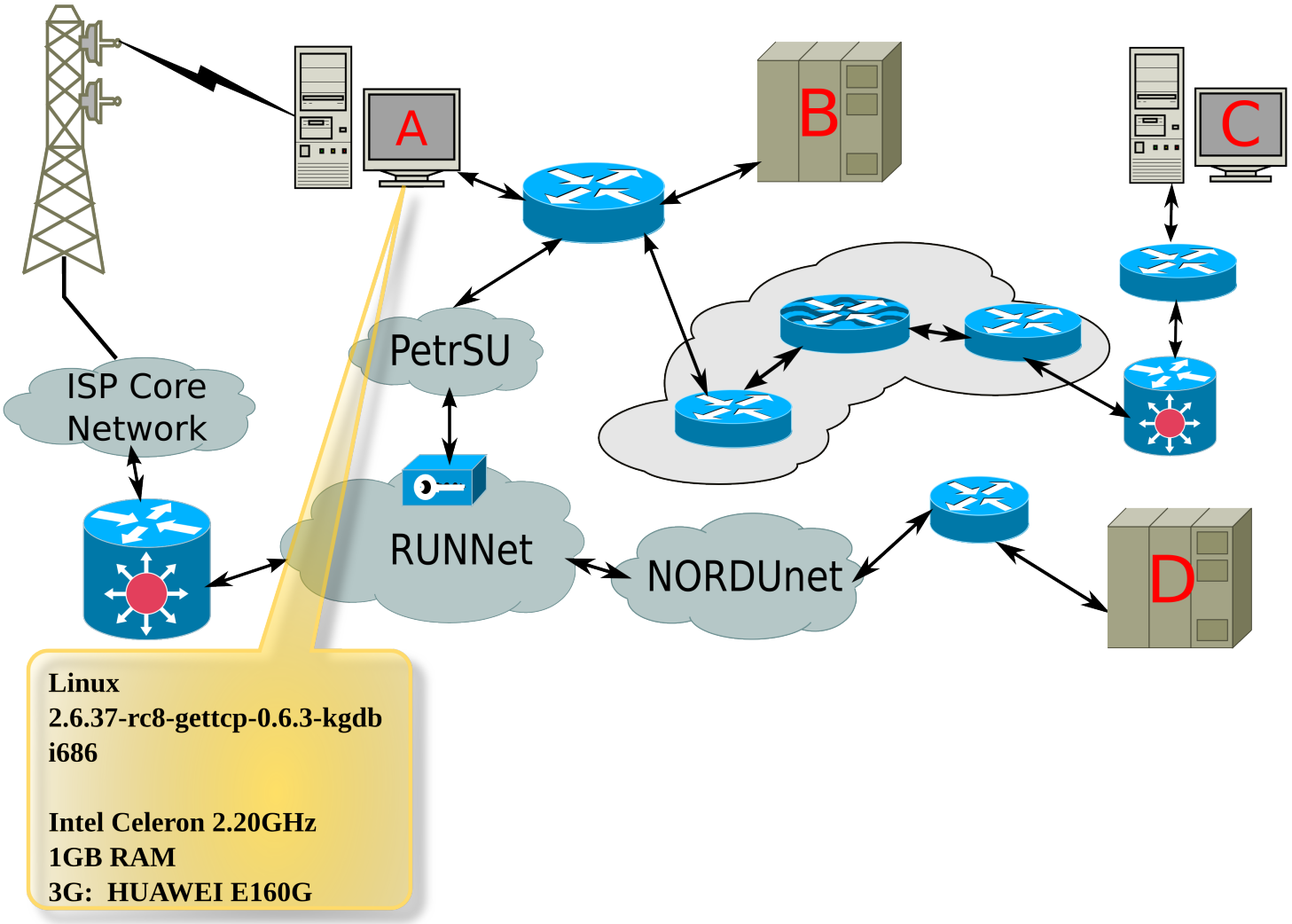
# The Third Destination Host Configuration



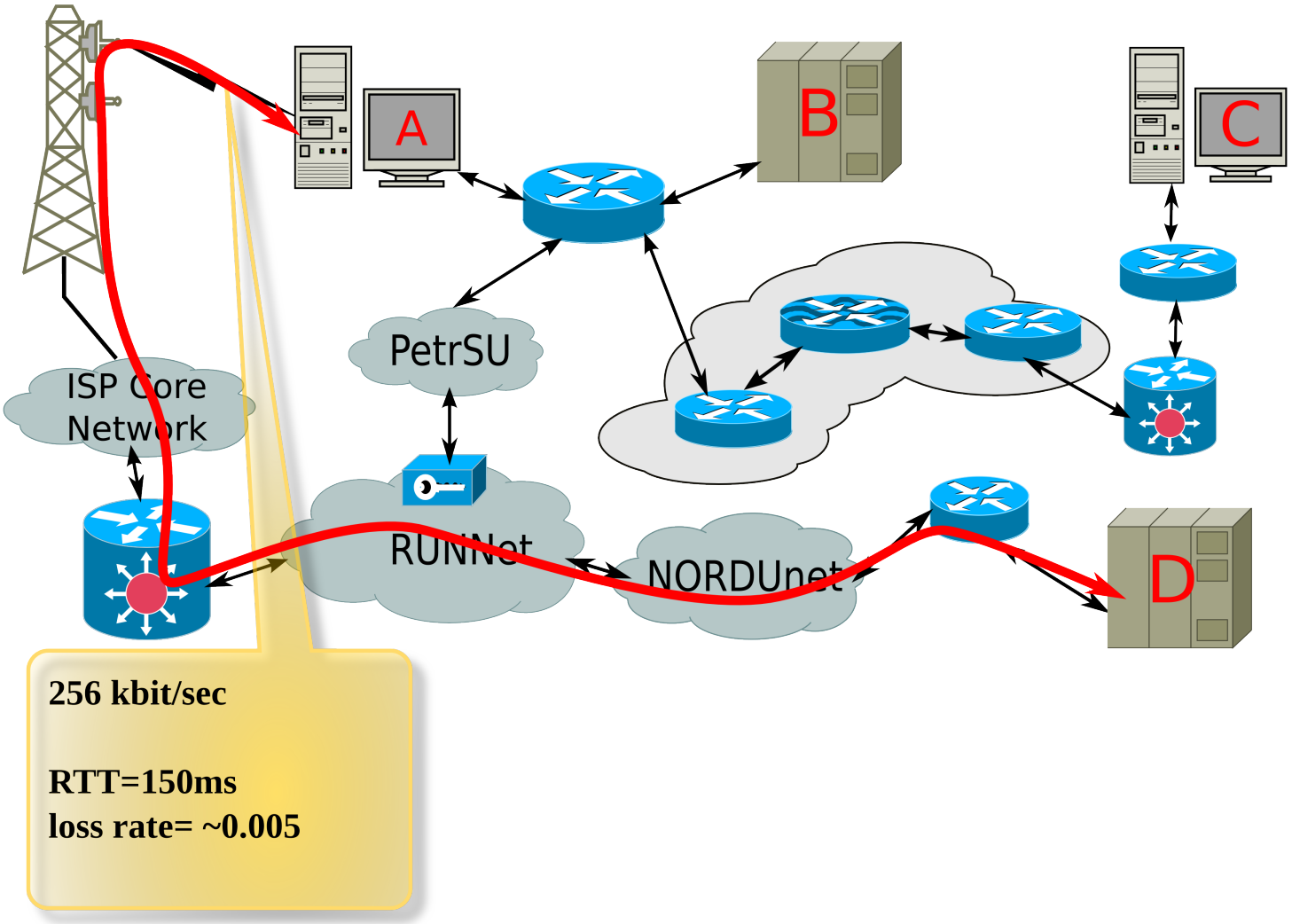
# Testing on the Long Route



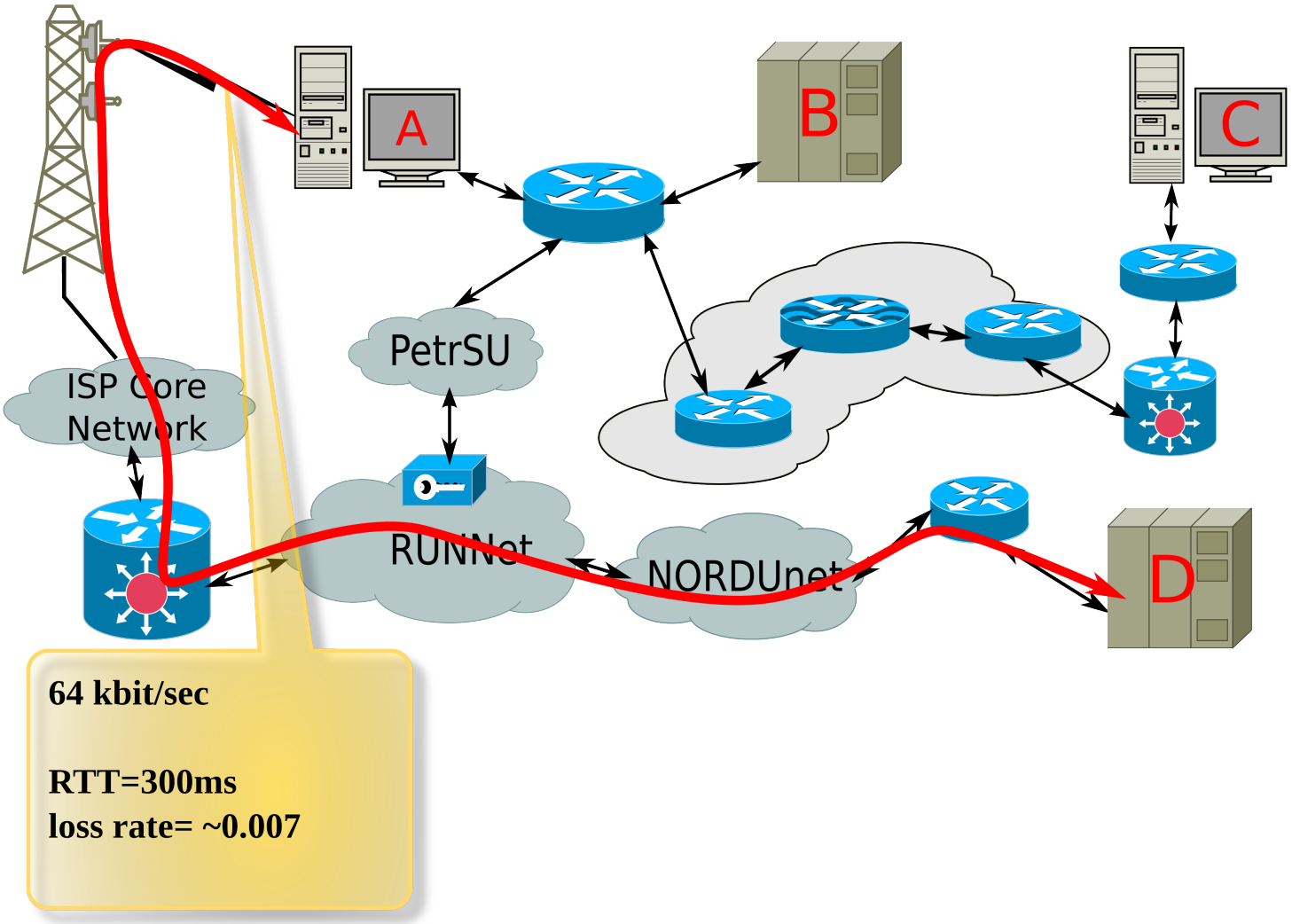
# The Source Host Configuration for Work with 3G



# Testing into the 3G-Network (256kbit/sec)



# Testing into the 3G-Network (64kbit/sec)



# Delays inspired by GetTCP+

Whole system shouldn't produce essential overhead. So the delays inspired by GetTCP+ has been estimated.

Ftrace - tracing utility built-in into Linux kernel. It can provide not only function traces or call graph, but latency measures for specified functions.

Ftrace latency output

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |    |                    |   |   |   |
 0)    1.282 us    |  tcp_ack_event();
 0)    0.258 us    |  tcp_ack_event();
 0)    1.187 us    |  tcp_ack_event();
 0)    0.285 us    |  tcp_ack_event();
 0)    0.292 us    |  tcp_ack_event();
 0)    0.898 us    |  tcp_ack_event();
```

# Mean delays

tcp_ack_event handler is invoked for each segment, so it should invoke low delays. Other handlers are less critical.
Latency measurements were performed on the path from **A** to **B**.

| Event | Handler | Mean delay |
|---|---|---|
| *flow_start_event* | *tcp_start_event()* | 16.904 usec |
| *flow_ack_event* | *tcp_ack_event()* | 0.628 usec |
| *flow_retr_event* | *tcp_retr_event()* | 1.172 usec |
| *flow_end_event* | *tcp_end_event()* | 2.480 usec |

# Conclusion

Results:

- Prototype of GetTCP+ system was developed
- Was provided mechanisms for GetTCP+ control
- Support of monitoring on kernel level provides correct and detailed information
- System was tested on parts of real network with different characteristics

Aims:

- Improvement of analytical component
- Improvement of external interfaces for various tools
- Support of modern networking technologies and new versions Linux kernels