Final Project Report

Title:	*Latency estimation of IP flows using NetFlow*
Author:	*Santiago Sebio Gallego*
Director:	*Pere Barlet Ros*
Co-Director:	*Josep Sanjuàs Cuxart*
Department:	*Computer Architecture*
Studies:	*Degree in Informatics Engineering (2003)*
Date:	*20th June 2013*

Table of Contents

I. Introduction 5
1.1. Measuring QoS 6
1.2. The paper "Two Samples are Enough: Opportunistic Flow-level Latency Estimation using NetFlow" 7
1.3. Scope of the project 8

II. Technologies used by the method 9
2.1. Cisco's Netflow 9
2.2. Network Time Protocol 10

III. Technologies used for the simulation 11
3.1. Discarded: The Common Open Research Emulator (CORE) 11
3.2. Graphical Network Simulator (GNS3) 13
3.3. Cisco 7200 router with IOS 12.3(22) 14
3.4. Archlinux 15
3.5. Linux kernel tools: tc, tc qdisc and tc filter 15

IV. Main software developed 16
4.1. Library 16
4.2. The rules 17
4.3. Relaxing or replacing the C3 rule 18
4.4. Make C5 more strict (including flows discarded by C1) 20
4.5. Using the initial sequence number and the TCP flags 21
4.6. Exported data 22

V. Software developed/modified for the simulation environment 23
5.1. Executable that calls the library (main) 23
5.2. Traffic-generating script (flowcreation.sh) 23
5.3. Traffic-shaping script (netemvariation.sh) 23
5.4. TCPDUMP and editcap 23
5.5. PCAP_DIFF 24
5.6. PCAP_DIFF-output compiling script 24
5.7. YAFSCII 24

VI. Main code documentation 25
6.1. Main functions, for external call 25
6.2. Flow comparison functions 26
6.3. Flow management functions 27

VII. Evaluation 28
7.1. Evaluation methods 28
7.2. Compatibility tests with GNS3 28
7.3. Performance tests using the CAIDA traces 32

VIII. Schedule and costs 40
8.1. Schedule 40
8.2. Costs 43

IX. Conclusion 44

X. Bibliography 46

I. Introduction

Internet plays a big paper in the modern business. From common services like chat, e-mail, VoIP and website browsing to more advanced ones like remote PC access, shared repositories or GPS-tracking, they have all become indispensable tools, to the point that Internet downtime translates to an interruption of almost all work in some sectors.

But not only downtimes are to be accounted for, because troubles with the Internet connection, even if it does not go down, can also mean a slowdown on the workflow, and alter the mood of the workers. Websites failing to load or loading corrupted, requiring multiple retries to perform one task or suffering unreliable communications can all be infuriating and directly decrease the productivity.

Because of that, measuring the quality of connection towards Internet or even the connection between two places owned by the same entity (for instance, the connection between the database server and the terminals managed by the workers) has become an important task.

That quality of service is determined by how long network packets take to go from the sender to the intended receiver, and if they actually do. In the case of losses, it is obvious that the less the better, but it is not as easy for the latency. A connection with higher average latency than another can be better overall if that latency is consistent, instead of having a wide range of possible packet latencies (jitter). But that heavily depends on the service.

For instance, VoIP/webcam communication deals nicely with losses (since it will only mean a few frames skipped, or in the worse cases a few-second freeze), but is terribly affected by latency (since it will force longer stops while waiting for an answer). Jitter, specially if consistent through a few seconds, can also become bothersome.

On the other hand, most other services are somewhat tolerant to latency, and their tolerance to losses depends on the precautions taken by the programmers. Some of the worst possible cases are unnoticed file corruption and lost messages.

1.1. Measuring QoS

But measuring quality of service is not an easy task. The most objective data (the exact times at which each packet is sent and received) is virtually unobtainable in anything other than closed environments, so any attempt to measure it has to work with more abridged data. And even then, that data is not necessarily easy to collect or use.

This project deals with QoS-measurement through flow-level reports, mostly following the standards set by Netflow v9 and older, but also dealing with the newer IPFIX, which has not been widely adapted yet and does not have much software available. Netflow's latest version, v10, follows the standards set by IPFIX.

These two protocols (Netflow and IPFIX) define a way to describe network traffic as flows, saving a lot of resources while attempting to keep relevant information for network diagnostic. An entry is created for every combination of protocol, ports, source and destination, which also logs the timestamp of the first and last packets, the total amount of packets, total bytes...

Using the information provided by those protocols, this method attempts to match the flows registered at two different places. If two flows (one from each flow-reporting probe) seem to refer to the same collection of packets, the four timestamps (two that mark the beginning and two that mark the end) can be used to tell how long did those packets (the first and the last) took to get from one probe to the other.

Using those matched timestamps, the method makes an estimation of the network status at that time.

1.2. The paper "Two Samples are Enough: Opportunistic Flow-level Latency Estimation using NetFlow"

This paper [1], written by researchers of the Purdue University, describes their implementation and results of a flow-level latency estimator, and the results of their tests. The main contributions of this article are the design of Consistent Netflow and their own Multiflow estimator.

Consistent Netflow explains how it would be possible to synchronize Netflow probes in order to make the estimation relevant and accurate. It mostly describes how to modify Sampled Netflow, a variation of Netflow that only logs a subset of the possible flows. Using regular Netflow is not a possibility when dealing with high-bandwidth connections, so Sampled Netflow is used instead. But the current implementation of Sampled Netflow chooses flows at random, making impossible to correlate flows from different routers. Their design of Consistent Netflow chooses which flows to log according to the hash of a few fields of the first packet of the flow, making synchronization between the routers possible.

This section also talks about how important time synchronization is, and how it can be achieved either by GPS or the modern IEEE 1588 protocol, that allows synchronizations within microseconds.

Then there is the explanation of their Multiflow estimator, a group of rules that match flows and then use the latency extracted from the delay of the first and last packets. When estimating the latency of a specific flow, they propose three methods: using the two timestamps of that flow (accurate only for small flows), using all the timestamps logged for the duration of the flow (most accurate for long flows) and the hybrid method, choosing one or the other depending on the length of the flow.

The main advantage of measuring QoS with this methodology is that it can be retrofit in already deployed systems, saving time and resources. Even though accuracy will get hit, it is still quite good (around 20% median error in flows of more than 100 packets), and it outperforms other methods that do not require many changes.

1.3. Scope of the project

This project has three main goals:

- Implement a program that applies the rules used in the article described above.
- Create a network simulation environment that allows to test this and other similar projects.
- Test the software created with both the created environment as well as with the method used for the paper.

II. Technologies used by the method

These are technologies required for the deployment of the project. Either them or a modern replacement are necessary.

2.1. Cisco's Netflow

Cisco's Netflow [2] is the most known flow reporting protocol. It was originally a method to store routing calculations, so they would not have to be recalculated for successive packets in the same flow. It was only later changed into a protocol to save and export flow reports.

Netflow works checking a few values of the packets that go through a router (protocol, ports, involved addresses), and if they match the ones of an already checked packet, they get grouped in a 'flow', that stores certain characteristics of the packets that have been grouped (mostly total packets, total size, timing of the first and last packets). After a while, these 'flows' describing the traffic are exported to a collector, usually a general purpose computer.

- Sampled Netflow:

Sampled Netflow is only supported by the Cisco 12000 routers, which makes it hard to come by. But even then, that version of Sampled Netflow (the only one, as of now) chooses the packets(not even flows!) at random, so it is completely unusable by this project, since the reports would never match. As said before, it is necessary to pick flows deterministically (depending on hashes).

- Consistency when delivering the reports:

The report delivery from the probes to the collector does not perform any checks, so the reports can suffer loses (either partial or complete) or even corruption. Because of that, to increase the reliability it is recommended to collect the reports with computers directly connected to the routers, and then assemble the report files (with proper transferring methods, making sure nothing is lost or corrupted) for analysis.

– ActiveTimeout:

This parameter determines for how many minutes an active flow (that has not seen any end-of-transmission packets) can live. If it reaches the limit set by this parameter, it will expire (and get split). By default it is set to 30 minutes, but it can be set to as low as 1 minute, increasing the amount of samples taken.

Setting it to one minute does have disadvantages though, because depending on the jitter and packet rate it is likely that a flow will get split at different packets in each router, making that sample unusable for the latency estimation.

2.2. Network Time Protocol

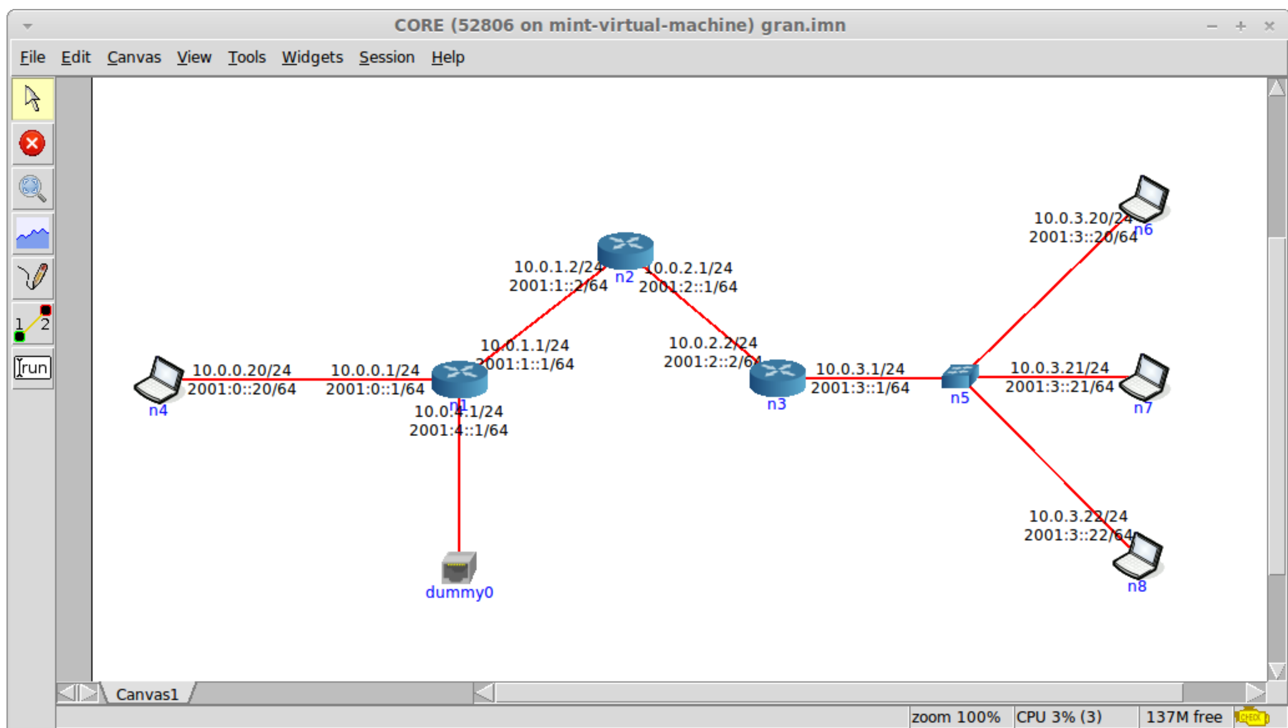
Depending on the network to be analysed, it might be possible to use NTP [3] instead of GPS clock synchronization. NTP is an stratified clock-sync protocol that improves its precision with successive sync-requests. The lowest stratus (stratum 1) is populated by devices directly connected to atomic clocks (these being called stratum 0), and it increases with every step away from them. Usually computers sync to a stratum 3 servers over the internet, so they would be stratum 4 themselves.

Cisco's implementation of NTP achieves around 100ms precision over the internet, 10ms over stable WANs, and 1ms in LAN. Obviously, to make use of NTP in this case, precisions of more than 10ms are unacceptable, so an almost direct connection between the routers is required.

III. Technologies used for the simulation

These are the essential technologies needed for the simulation environment, which we have found to be the best after trying a few alternatives.

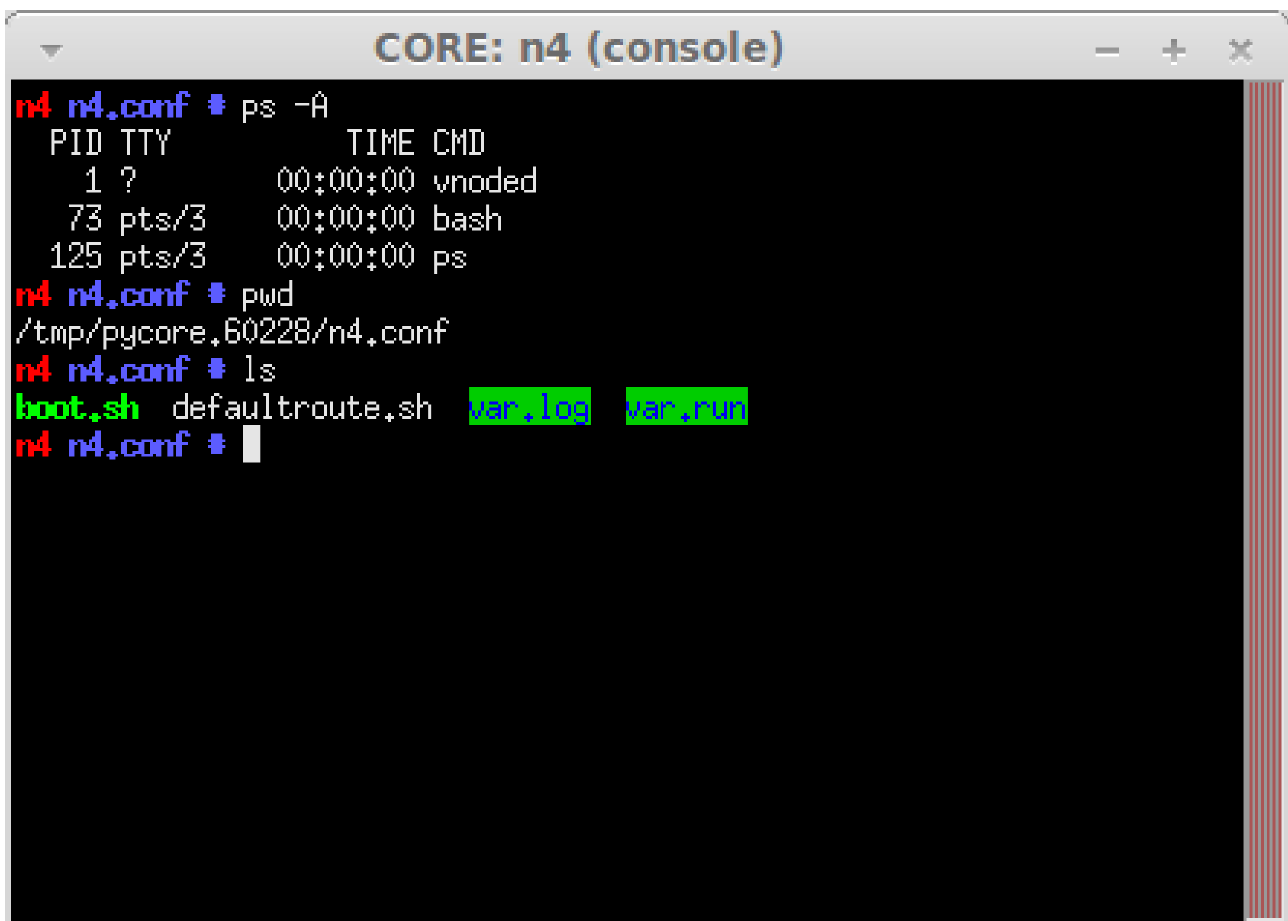
3.1. Discarded: The Common Open Research Emulator (CORE)



Screenshot 1: A sample setup in the CORE environment. The "dummy0" interface allows connecting the virtual network to the host machine.

The Common Open Research Emulator [4], created by researchers of the U.S. Navy, offers a decent amount of features despite being lightweight and really easy to setup. The inner networking is managed with virtual network devices, their output being read and written by virtual nodes, more specifically "Linux network namespaces", a recent feature of the Linux kernel that allows creating independent program and network stacks. Right from the GUI it is possible to open a terminal window in the corresponding node, and execute any software installed in the host machine.

Thanks to the simplicity of its architecture and the lack of hardware virtualization (since all the code is executed natively), even an average machine is able to emulate a network composed by dozens of nodes.



Screenshot 2: Showcase of the basic node processes and filesystem.

But the way the nodes are emulated is both its best feature and its main disadvantage. They are very efficient, and can execute almost everything that the host machine can, but they are exclusively Linux machines, and, after all, they do not even have their own emulated hardware. No being able to incorporate Cisco's IOS (nor any other firmware) to the system puts a limit to what can be tested. Also, the lack of hardware emulation makes it unfit to test, for instance, clock synchronization (because all the nodes use the same clock, the host's clock).

Taking notice of those limitations, we proceeded with Core, despite the possibility of it falling short. At the end, we had to switch to one of the alternatives because of an inconsistency in the Netflow-probe software we were using (softflowd [5]) and preferred to use a platform that supported Cisco's native software.

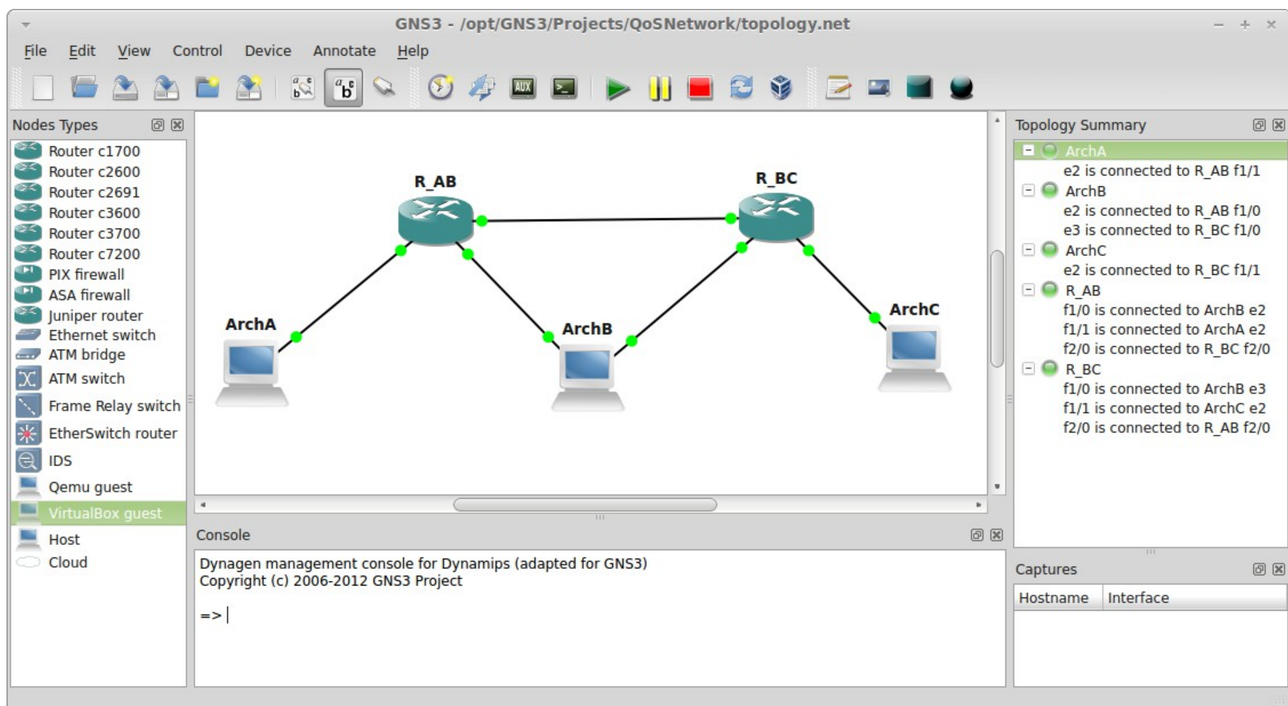
Even if it was more softflowd's fault than Core's, and we could have tested with other Linux Netflow probes (like nprobe), we were fearful of finding other misbehaviours, forcing us to switch environment anyway. All in all, it was nice getting familiarized with CORE, which can work very nicely if the project is not bothered by the lack of device variety.

3.2. Graphical Network Simulator (GNS3)

GNS3 [6] is a complex and feature-rich tool compatible with many devices, thanks to its VirtualBox [7] compatibility and IOS emulation through Dynamips [8] (of which GNS3 is the main platform).

GNS3 supports three device emulators: Qemu, VirtualBox and Dynamips:

- The first is an open source virtualizer, which, while requiring more resources than CORE's machines, it is a lot lighter than full-feature virtualization software.
- VirtualBox, owned by Oracle, is compatible with almost every computer operating system that exists, and many of them are directly supported by the "Guest Additions", that enable many additional features (like dragging files from and into the guest system).
- Finally, Dynamips is able to emulate some of Cisco's routers. Sadly, it does not support most of the newer routers, which also limits the versions of IOS it can run. Still, it is the most developed IOS emulator available, and its range of emulated Routers is enough for the purpose of this project.



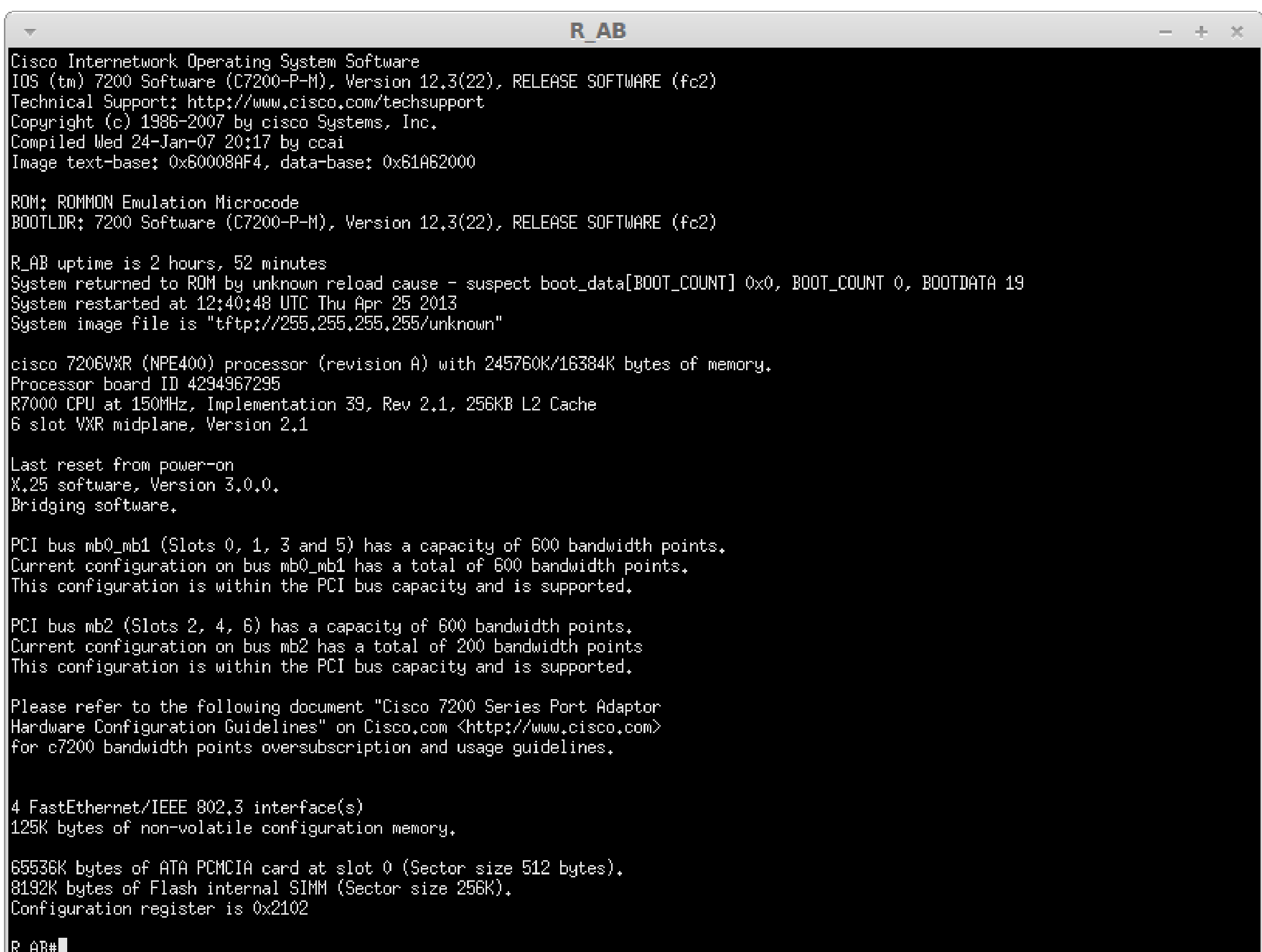
Screenshot 3: A GNS3 topology, showing the connections between the interfaces on the right.

3.3. Cisco 7200 router with IOS 12.3(22)

Dynamips supports Cisco's 1700, 3600, 3700, 2600 and 7200 and series. We tested for a while with a 3620 router, since the 3600 series seem to be the most popular in the GNS3 community. But it lacked some features present in the 7200 series, even when packed with the same IOS versions (12.3). We confirmed that lack by checking Cisco's website, where numerous hardware-dependant differences (even within the same IOS version) are documented.

This specific router-IOS combination supports Netflow 5, 6 and 9, which proved to be ideal for this project. Netflow 9 is the newest version, and supports egress traffic, while version 6 is the simplest and most stable version that has enough features to perform the tests. Netflow 5 does not allow to customize timeouts, which is a relatively new feature, added in the 12.3(7)T IOS version, the 'T' indicating that it is a test version.

Regarding performance, we did not find any noticeable difference between the few 3620 and 7200 images tested.



Screenshot 4: One of the emulated Cisco routers showing the output of the "show version" command.

3.4. Archlinux

For the computers part of the GNS3 topology we chose Archlinux [9] because of it being a minimalistic distribution, which most likely would not have changes that could prevent using the kernel features we intended to. It lacking many common repositories did not slow down the setup, because most of the software we used already required downloading, configuring, compiling and installing the packages manually.

3.5. Linux kernel tools: tc, tc qdisc and tc filter

Traffic Control [10], invoked with the command "tc", allows to configure the way network traffic is routed, and allows to apply a wide set of rules, some made specifically for testing purposes, without any practical application (like forcing delay, packet loses or corruption). Conventional networking hardware usually lacks such features, and they are really simple when they do include them.

"tc qdisc", short for "queueing discipline", allows to create multiple "paths" packets can go through, while being applied rules that might change the packets themselves, re-route them, or, in this case, delay or lose them.

"tc filter" sets the rules that will determine what patch do the packets go through. You can filter by both packet contents and header. Combined with tc qdisc, it allows to set the QoS parameters usually seen in domestic routers (like prioritizing certain protocols or ports).

IV. Main software developed

This section will describe the most relevant features of the software, the first two describing the basic behaviour, and the other three describing modifications by us to the original behaviour that can improve the results.

4.1. Library

The documentation of this library can be found in the corresponding chapter. Here we will comment on the main idea and some of the changes.

The library has four main tasks:

- Store “flowinfo_t” structs that each contain the information of a flow, including the ports, IP addresses, exporter, timings, packets, bytes...
- Process the flows of one of the exporters one by one, attempting to match them according to the set of rules defined by the article. Once two flows are matched, there is a reasonable chance that the start and end timestamps correspond to the same packets, so the differences can be used to estimate the latency.
- The latency timings from the last step are processed, generating additional information like jitter and minimum, maximum and average latencies. It is also possible to generate the values for specific ports, IP addresses, protocols...
- The information extracted in the last step is written into files in csv (comma-separated values) format, making it reasonably easy to read and import.

4.2. The rules

These are the original rules used to pair the flows. All the changes and tests are related to them, plus they will be referenced by their number, so it is necessary to remember them to understand the rest of the paper.

- C1: The first and last packet times must be compatible. That is, considering latency, processing delay, and timing differences, the difference between the start of the flow of one exporter and the other have to be within a margin.

- C2: The bytes and packets of the two flows have to match.

- C3: Rules out flow pairings that might include packets that one of the probes split into another flow.

- C4: Rules out flows that might be missing packets in the sender due to inactive timeout, but that due to jitter are found in the receiver. (Irrelevant in our tests, since the inactive timeout is 15 seconds, which is much less than the maximum delay).

- C5: If a pair of flows was previously discarded due to the C2-C4 rules, discard the following flows with the same key until there is enough separation (an amount of time larger than the inactive timeout).

These rules work nicely most of the time, but we found two changes that perform better in some situations, explained in the next sections.

4.3. Relaxing or replacing the C3 rule

The C3 rule can remove all the samples from certain services, preventing the estimation of those. Services with lengthy connections and high packet-rate (for instance, VOIP) are very likely to be ruled out by C3, because they will spawn multiple flows cut by the active timeout, and the last packet logged at the receiver might belong to the second half of the flow.

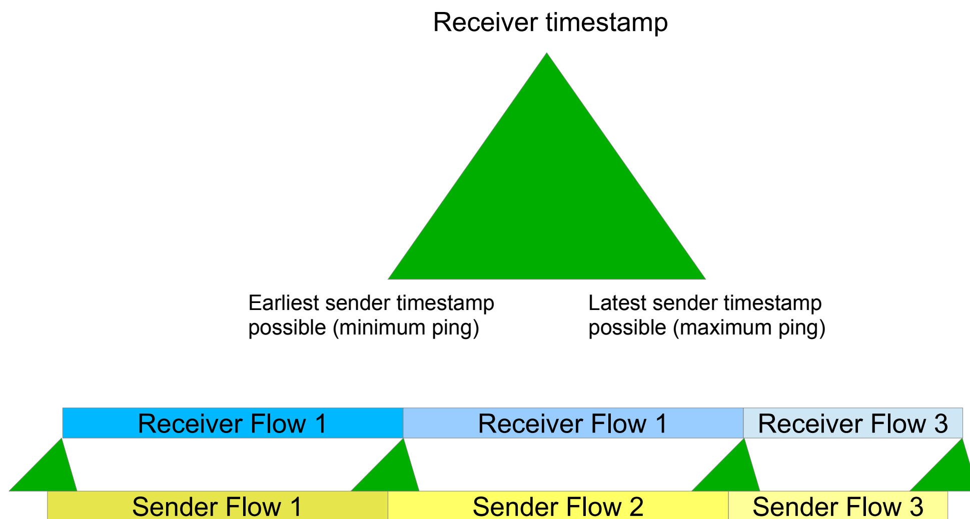


Figure 1: Diagram illustrating the problem associated with flows split by Active Timeout.

As figure 1 shows, when dealing with flows split by Active timeout, the final packet logged in a receiver flow might correspond to a packet of a sender flow that is not the one we are attempting to match. Obviously it would not pass C2 in the first place, assuming no losses, but let us say Sender Flow 1 has 50 packets, one was lost, but a packet from Sender Flow 2 is included in Receiver Flow 1. Then, both SF1 and RF1 would have 50 packets, but the ending timestamps would refer to different packets, and the latency sample would be incorrect. C3 prevents those scenarios.

But, as we pointed in the introduction of this section, C3 can be too strict. Let us say there is no packet loss at all in the scenario shown in Figure 1. C3 would still rule out SF1-RF1 and SF2-RF2. SF-3RF3 would pass C3, because it is shorter and is not cut at the end by Active timeout, but would be ruled out by C5 because the two previous pairings have the same key and were ruled out by C3. If these three pairings are the only samples of certain service, the excessive cautiousness of C3 might have removed important information.

To prevent this, we propose this change to the rules:

- The removal of C3, allowing chained flows to be used in the sampling.
- Successive flows of the same key will be checked before any of them are used in the sampling, and the C5 rule can be applied retroactively.

Consider this scenario:

- SF1 has 40 packets.
- SF2 has 40 packets. The first packet corresponds to the last in RF1, and the last to the penultimate in RF2.
- SF3 has 17 packets. The first packet corresponds to the last in RF2, and the last corresponds to the last in RF3.
- RF1 has 40 packets, but one packet logged in SF1 was lost, and the 40th packet of RF1 corresponds to the first in SF2.
- RF2 has 40 packets, the first corresponds to the second in SF2, and the last to the first in RF3.
- RF3 has 16 packets. The first corresponds to the second in SF3, and the last to the last in SF3.

Without C3, the first two pairings would pass, generating incorrect latency samples. But with the retroactive C5, the check between SF3 and RF3 would halt at C2, and the two previous pairings would be discarded.

The implementation of a retroactive C5 requires a few changes in the structure of the program and also slows down a bit the execution, but allows more sampling, plus the new structure (that stores each 'valid' pairing) makes the library easier to extend with new features.

Both the removal of C3 and the retroactive C5 will be evaluated in the corresponding section.

4.4. Make C5 more strict (including flows discarded by C1)

A basic flaw we found in the original set of rules is that a pair discarded by C1 is not considered for C5. There is a good reason for that, and it is that two flows ruled out by C1 might be very far (in time) from each other. But at the same time, it allows scenarios such as the one explained below to add incorrect samples:

- SF1 has 40 packets.
- RF1 has 38 packets, the first one and another during the flow were lost. Also the separation between the first and the second is enough for the flow to get discarded by C1.
- SF2 has 10 packets.
- RF2 has 10 packets. The first one corresponds to the last one in SF1, and one included in SF2 was lost. Due to jitter or a change in the packet rate, the pairing is not discarded by C1 this time. We now have a pair that has a correct ending sample but incorrect starting sample.

Even if this scenario might look picky, it is not that unlikely (and the disparity between SF1 and RF1 does not have to be so small, it could be that a few dozen packets were lost, and we might still find this scenario).

To prevent it, we propose using pairs discarded by C1, but that are close enough to be referencing the same packets, in the checks made by C5. In the example scenario, SF2-RF2 would be discarded because SF1-RF1 were more than likely of referencing the same packets, but were unsynchronized because of packet loss.

This strict C5 will be evaluated in the corresponding section.

4.5. Using the initial sequence number and the TCP flags

There are two useful fields of information that were not used initially, and those are the sequence numbers and TCP flags. While Netflow v9 and previous versions do not provide them, IPFIX does export them with the rest. This opens the possibility of using them for stricter pairings.

- "Initial sequence number" corresponds to the sequence number field in the first packet of the flow. The straight-forward, drawback-less usage is making sure the sequence numbers match during a pairing, and if they do not, discard the pair. This could help in the unlikely situation of a pair meeting all the rules despite referencing different packets, and this last fact being revealed by the non-matching sequence numbers.
- Regarding the TCP flags, it is possible to use the strict ruling of only using the first and last timestamps if the SYN and FIN flags are present, respectively. That will eliminate many timestamps that might have been correct, but the chance of incorrect pairs should decrease quite a bit.

Both these modifications will be evaluated in the corresponding section.

4.6. Exported data

Once the flows have been paired and we are reasonably sure of having useful samples, it is time to decide what exact information should the program export.

The basic information is the latency of the packets associated with certain key. So the most verbose output would be printing the two latency values of each paired flows, together with the flow information (IP addresses, ports, protocols, start and ending time at the sender, amount of packets...). In the article this method is called "endpoint estimator".

An extension to that output is the multipoint estimator, which is the main feature in the article. Instead of providing the latency at the boundaries of a flow, the multipoint estimator assigns to each flow the average of the latencies found between the start and the end of the flow. According to their results, this method performs way better than the endpoint estimator for longer flows.

The per-flow output is the best for accuracy tests, as well as for software that works with flows, but it is not too readable for the end-user or for simpler pieces of software that does not require such in-depth information. In that case, it is possible to output grouped data; that data could be information regarding the latency suffered by the packets during an specific timeframe.

If the flows have been previously been categorized, the output can include the parameters of the latencies of those categories. Either way, the parameters in a per-interval output should include the maximum, minimum and average latencies, as well as the jitter.

For the initial tests with the software we applied different policies to certain port ranges. Those port ranges were assigned to a certain category, and the per-minute output of the program had to estimate correctly the policies.

Later, in order to roughly reproduce the tests performed in the article, we used the per-flow output.

V. Software developed/modified for the simulation environment

5.1. Executable that calls the library (main)

It has the code responsible of reading the data, parsing it into the correct data-structures and feeding it to the library, which will processs the data and generate the output. It supports two formats: flow-tools' [11] output and YAFSCII's ASCII output, which is a tool included in YAF that translates YAF's IPFIX-compilant format to readable ASCII.

5.2. Traffic-generating script (flowcreation.sh)

This shellscript loops and calls hping3 (a tool that creates TCP/UDP connections) with a randomized destination and source ports, amount of packets, packet-rate, size of the packets and destination address. Those vales are randomized between the desired intervals. As a result, by calling this script a computer generates multiple varied TCP connections, which can be routed through the Netflow probes, generating flow reports. It has a used port-check to prevent having multiple hping3 instances attempting to use the same ports.

5.3. Traffic-shaping script (netemvariation.sh)

This shellscript modifies tc qdisc parameters so the traffic conditions vary over time, either for all the packets or only for certain intervals of ports or IP addresses.

5.4. TCPDUMP and editcap

With TCPDUMP [12] at the boundaries of the analysed network it is possible to log every single packet, which can later be used for precise latency calculation. But since the devices at the boundaries are linux machines that are not perfectly synchronized, editcap [13] is needed to synchronize the packet dumps a posteriori.

5.5. PCAP_DIFF

This tool [14] compares two packet dumps in pcap format (generated by TCPDUMP), and if a packet from one dump matches a packet from the other one, a line is printed indicating the nature of the packet and the timestamp difference (precise latency).

We had to add a null-check in a loop's header because it was resulting in segmentation fault errors. As far as we know, the lack of this check is an oversight unlikely to get corrected, since the project has increased its scope, has been renamed to TPCAT and includes a GUI and a few new features. It lacks, however, pcap_diff's command-line capabilities, and does not seem to be useful for large captures.

5.6. PCAP_DIFF-output compiling script

PCAP_DIFF's output as-is is not useful to check if the latency is being correctly estimated, so we programmed an awk script (a tool made to deal with tasks that need to do an action per-line in a target file). This script can group the packets in intervals, and output the maximum, minimum and average latency of the packets. It can also output the latency of packets sent through specific port intervals.

5.7. YAFSCII

Before describing the change we had to perform on YAFSCII [15], we will define POSIX or Unix time. This standard counts the time passed since the 1st of January 1970, at 00:00 UTC. It is widely used in all Linux-based software. When represented in seconds, it has the disadvantage requiring almost all of the bits in a 32-bit integer, so there is almost no room for extra precision. In fact, in 2038 not even the seconds will be storable in a 32-bit integer.

Now onto YAFSCII, the time output was highly inconvenient for our purposes since the format was complete, with numbers for year, month, day, hour, minute... we modified its source so the time output was in milliseconds since Epoch. The only trouble with this format is that it requires a 64-bit integer to store it, since the current time in that format requires 40 bits. But since the format used by the code is a 2-element struct (with one element for seconds since Epoch and another for microseconds), adapting this format is quite straight-forward.

VI. Main code documentation

6.1. Main functions, for external call

*void *initialize_qos(int out_type):*

Initializes the struct that stores the status of the processing and handles a pointer to it, which is required for every other external call.

'out_type' chooses which kind of output will be performed:

- '0': One line for each of the timestamps paired and the corresponding delay.
- '1': One line for each flow paired, including the start time, end time, the delay by endpoint-estimation, the delay by multi-flow estimation and the real average delay of the flow.
- '2': One line per export_period (a constant, set to 60 seconds). Each line contains the jitter, average, maximum and minimum delays of all the flows during the period, plus each of those values for a number of applications/categories specified by a config file.

*void process_flow(void *state, flowinfo_t *f):*

Processes and stores a copy of the flow pointed by 'f', which will be used for the calculation.

*int output(void *state):*

Outputs all the safe output possible. By "safe", it means that it is unlikely that relevant information is not being ignored due to it not being processed yet. So the output will only use the timestamps that are separated enough (in time) from the newest flows processed. That separation is defined by a constant and should be longer than the longest export delay possible. That way, we can be relatively sure that we did not omit useful information that was going to be processed later.

*int output_all(void *state)*

Unlike the regular output function, this one does not restrict itself to 'safe' output, and uses all the flows that have been processed. It should be called once all the flows in a finite trace (not a continuous stream) have been processed by the library.

6.2. Flow comparison functions

*int startdiff(flowinfo_t *f1, flowinfo_t *f2)*

Gives the difference in microseconds between the start of the flow pointed by f1 and the one pointed by f2. Returns a positive value if f1's start was earlier than f2's.

*int enddiff(flowinfo_t *f1, flowinfo_t *f2)*

Gives the difference in microseconds between the end of the flow pointed by f1 and the one pointed by f2. Returns a positive value if f1's end was earlier than f2's.

*int startenddiff(flowinfo_t *f1, flowinfo_t *f2)*

Gives the difference in microseconds between the start of the flow pointed by f1 and the one pointed by f2. Returns a positive value if f1's start was earlier than f2's end.

*char same_key(flowinfo_t *f1, flowinfo_t *f2)*

Returns 1 if the key of the flow pointed by f1 matches the key of f2. The key is defined as the two IP addresses and two ports (source and destination). The source of one flow has to match the source of the other flow, it won't return 1 for flows in opposite direction. If the key doesn't match, it returns 0.

6.3. Flow management functions

void make_invalid(flowinfo_t f)*

Makes the flow invalid for future pairings. Used after a flow has been paired, so it does not get paired multiple times.

*void discard(void *state, int flownumber)*

Adds a flow to the list of discarded flows (flows that could not be paired). This list is checked during the pairings, using check_discard(), to prevent possible unsynchronized flows from being paired.

*char check_discard(void *state, flowinfo_t* f)*

Returns 1 if there is a flow with the same key as 'f' in the discarded list, and 0 otherwise.

*void clear_discarded(void *state, int flownumber)*

Removes the references to the flows up to 'flownumber' from the discarded list. Cleanup function, used after older flows that are not going to be used get removed.

*void clear_pairs(void *state, int flownumber)*

Removes the pairs that reference to the flows up to 'flownumber' from the discarded list. Cleanup function, used after older flows that are not going to be used get removed.

*char insert_exporter(void *state, int exporter)*

Inserts a new exporter in a list of exporter addresses, used to determine which is the main exporter (the one whose flows will be checked one by one, and only once each, when attempting to pair flows).

VII. Evaluation

7.1. Evaluation methods

We used two ways to test the software developed: one of our own design, trying to use tools as close as reality as possible, making sure that the software could be of some use in a real environment (albeit much at a much smaller scale). The main objective of these test is to make sure everything is compatible.

The other way is inspired by the article and attempts to be as close as possible to it, using very large and dense Internet traces, and aims to test the performance of the program.

7.2. Compatibility tests with GNS3

The main benefit of playing around GNS3 was learning about Cisco's IOS, which is, after all, very likely to be involved in any deployment of the method this paper is about. We learned that not every version of this operating system is suited for the purpose, and of those that are suited, some are better than others.

There are a few features or characteristics to look for in a router that is going to export Netflow traffic:

Customizable timeouts: introduced in IOS 12.3(7)T and only available when using Netflow v6 or higher, it allows lowering the maximum duration of a flow (after that, it gets split). Setting it to a lower value than the default 30 minutes allows increasing the time-stamp density, specially for services that tend to stablish long connections. It also allows customizing the seconds before a flow expires (and therefore, any packets that might have been part of that flow will generate a new flow), which is set by default to 15 seconds.

Capturing outbound traffic: Until Netflow v9, it was only possible to capture inbound traffic, that is, traffic that is entering through interface, but not traffic coming out from one. Because of that, it might be needed to capture useless traffic when using versions previous to v9, as explained below and shown by the figure 2.

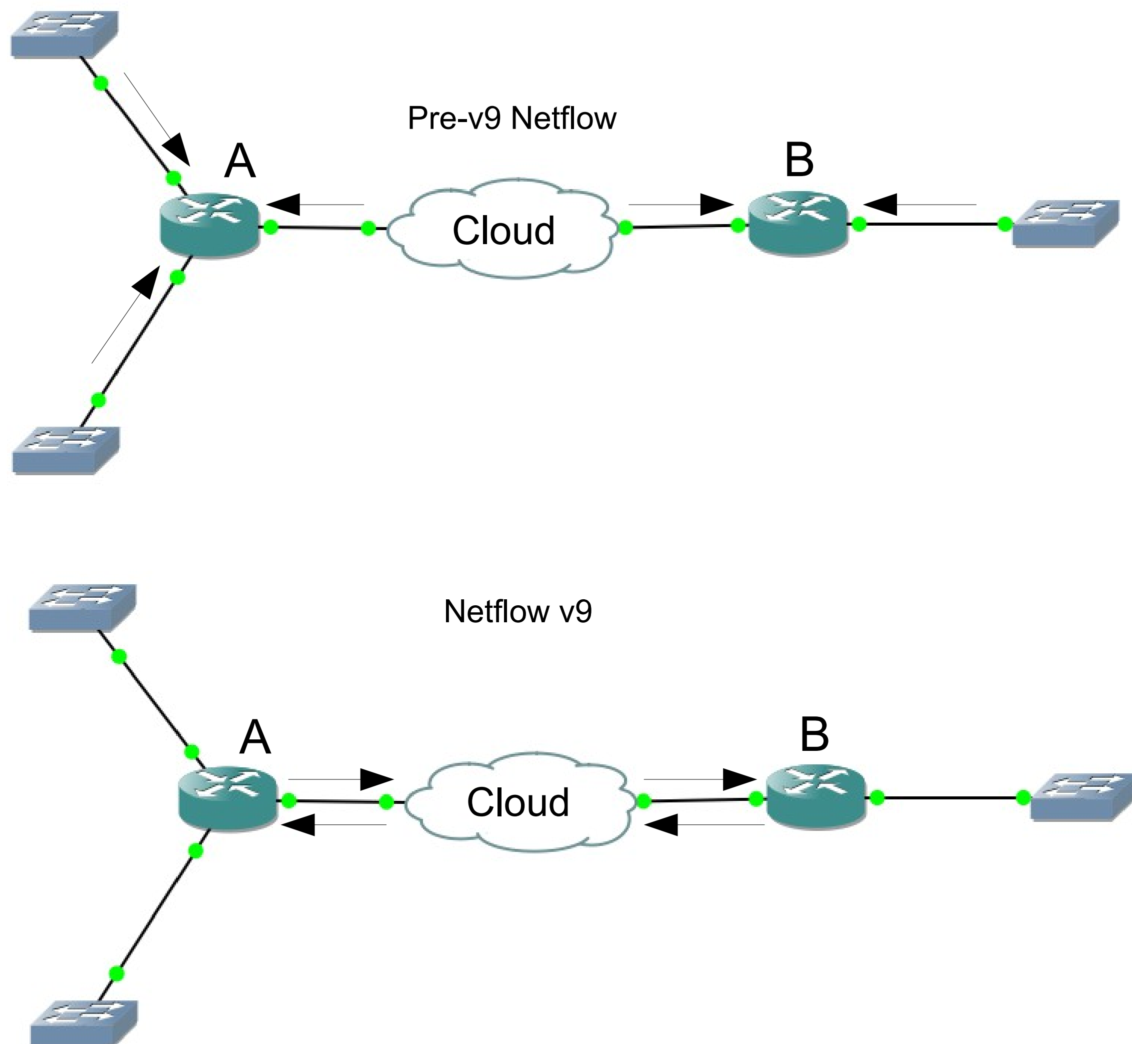


Figure 2: Comparison of the traffic that has to be captured in Netflow v9 versus previous version.

If we wanted to analyse the traffic going through the cloud, from A to B, with older Netflow versions it is necessary to capture the ingress traffic in all of the interfaces of the routers (which would include the traffic going from one of the subnets of A to the other, as well as traffic directed to the routers themselves). On the other hand, with Netflow v9 it is possible to capture exclusively the traffic going through the interfaces directly connected to the cloud.

Consistency on delivery: Another thing that has to be considered when creating a Netflow setup is the report delivery. Netflow does not have any kind of security protocol, so they can be lost or corrupted without notice. The exporters do not even check if the collector is up. Because of that, it is recommended that there is a collector in the local network of each of the Netflow probes, and the reports would get merged later. The only security measured included was added in Netflow v9 and consists in a export number at the beginning of each export, which allows to detect loses of entire exports, but there is no way to ask for a retry to the probe, and it does not protect from partial losses or corruption.

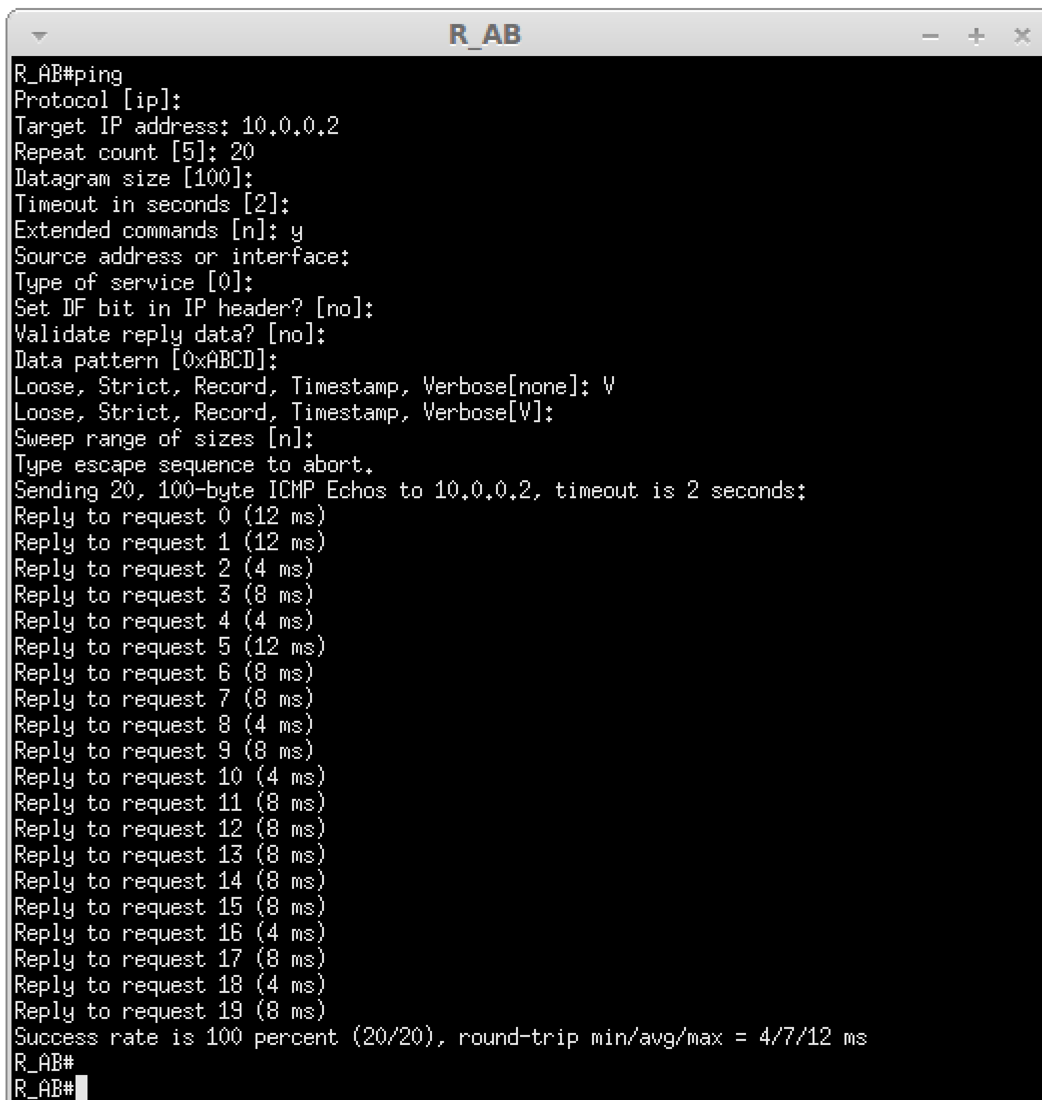
After investigating those matters, we generated traffic within the simulated topology with a script, which got modulated by a virtual machine between the routers, and those routers reported the traffic through the Netflow protocol.

Both the reports and the program proved to be reliable within certain limits – limits imposed by the simulation itself. We captured the actual packets with TCP at the sender and receiver (virtual PCs directly attached to the routers) and synchronized the traces with editcap (when needed, since the virtual machines used by GNS3 are not always synchronized).

Using the pcap_diff tool we described previously, we managed to obtain the real delays suffered by the packets, and compared them to the ones extrapolated from Netflow. And despite that both were somewhat consistent, showing correctly variations over time in the latency and jitter, there was always certain delay from the Netflow capture to the TCPDUMP capture, and it was not always the same.

As it happens, GNS3 topologies have an innate delay when tunneling packets, which can be reduced to very few (less than 2) milliseconds when there are enough spare CPU cycles, but varies between that and ~10 milliseconds during heavy load.

That would not be such a big problem if it was not due to how Dynamips (the Cisco IOS emulator) works, since in a clean, under-loaded PC the tunneling is sufficiently fast and consistent. But Dynamips' emulates IOS through an infinite loop without any kind of pause or sleep, using as much CPU power as it is provided to it. Even in multicore machines such as the one we used for the tests this intensive work made the whole GNS3 environment unreliable.



Screenshot 5: A capture of 'ping' performed by IOS between two directly connected routers, showing GNS3's tunneling behaviour during moderate CPU load.

The feature that solves this problem, 'Idle PC' (part of GNS3 and Dynamips), picks an instruction in the IOS code and sleeps whenever it reaches it. When using it, the CPU usage drops up to twenty times what it was. But this does not solve the problem at all, since it generates Netflow inconsistencies (the timestamps are not precise, plus some packets seem to be ignored) and even bigger latency inconsistencies in certain cases.

Because of that, we abandoned the intent of using this environment for performance tests, and restricted its use to compatibility tests and the study of IOS.

7.3. Performance tests using the CAIDA traces

Following the method used by the authors of the article, we requested access to the Anonymized Internet Traces 2008 from CAIDA (Cooperative Association for Internet Data Analysis [16]). These traces were captured by monitors on OC192 Internet backbone links in Chicago and San Jose, and only include the header of the packets, which have also been modified so they can not be related to real, specific IP addresses.

The Chicago trace includes 13 million packets in 60 seconds, which correspond to 1.2 million flows.

In the article they created a queue-simulator that let packets through according to various delay models, being weibull the most favoured.

Sadly, we found this queue-simulator to be outside of the reach of this project, since it would require us to get familiarized with the PCAP library (LIBPCAP), program our own queuer, which would emulate the time spent processing the packets and write them on the output file with the corresponding delay.

We had to restrict the project to simpler delay models. We were surprised at finding that we would have to program those ourselves too, since the only tool that did something like that (editcap) could only add a fixed delay to all packets at once.

What we ended up doing was modifying editcap's source code in order to add variable delay and chance-based packet losses (by default it can only remove packets specified in the command line). Any function can be chosen for the delay, as long as the delay difference applied at timestamps T1 and T2 is never bigger than the time difference between those two. If that happened, the order of the packets could get changed, which can not be easily corrected as far as we could tell, other than using the “strict time option” with editcap, that does not really reorder packets, but instead increases the timestamp to make sure they are strictly increasing.

We chose to use trigonometric functions to apply the delays, more specifically the sine function. The delay applied to a packet with our modified editcap is:

$$100\text{ms} + 200\text{ms} * sinus\left(\frac{(timestamp.seconds \,\%\, 10 + timestamp.miliseconds / 1000) * pi}{10}\right)$$

Since the value passed to sin always goes between 0 and pi, the result of the sin function is always between 0 and 1. Because of that, the delay varies between 100ms and 300ms, varying slowly enough so that packets never get reordered.

After applying these varying delays, whole packets are skipped from writing if the result of rand() is bigger than a certain value, simulating packet loss.

Then both the original trace and the fabricated trace are processed by the program, tagged as if they were produced by two different exporters. The output used in this case is the "per-timestamp output", which writes the time and delay of every beginning and ending of paired flows.

Then, the timestamps exported by the program are checked against the formula, in order to know if the program exported the right timestamps (matched timestamps that represented the same packets) or not.

There are 1.071.683 flows in the original trace, but many of them were removed if they were not relevant in some of the tests (specially non-TCP connections). The resulting trace has the following characteristics:

- 632.741 total flows.
- 127.338 one-packet flows.
- 409.227 flows between 2 and 10 packets.
- 96.176 flows of more than 10 packets.
  - Of those, 17230 flows over 100 packets.
  - 1332 flows over 1000 packets.

The maximum timestamps possible in a result are 505403*2 + 127.338 = 1.138.144 (since the 1-packet flows can only provide one timestamp).

So we ran tests with these versions, combining the modifications explained in section IV:

- A regular version, without any modification other than the removal of C3 (which we used as the standard, since C3 is questionable to begin with).
- The exact version from the article, with C3.
- A version with the retroactive C5 (discarding a pairing because a future flow with the same key could not be paired) plus stricter C5 (flows are more likely to be added to the discarded pool that is checked in C5).
- A version that checks for the sequence numbers before pairing flows.
- A version that checks the tcp flags, only using the initial timestamps of flows with SYN, and the final timestamps of flows with FIN.

Every version was tested with 3 different traces: one without packet losses, one with 5% losses and one with 10% losses. The results were the following:

No packet loss:

Version	Samples	%Samples	Misses	%Miss	Average error	Maximum Error
Regular	1.134.910	99.72%	0	0.0000%	0 ms	0 ms
With C3	890.030	78.20%	0	0.0000%	0 ms	0 ms
Retroactive C5	1.134.898	99.71%	0	0.0000%	0 ms	0 ms
Seqnum check	1.134.910	99.72%	0	0.0000%	0 ms	0 ms
Strict Flags	412.861	36.27%	0	0.0000%	0 ms	0 ms

- 'Samples' refers to the amount of unique delay samples (timestamp plus delay found) provided by the execution.
- %Samples' refers to the percentage of samples provided compared to the maximum of samples in a lossless trace (1.138.144).
- 'Misses' corresponds to the amount of miss-samples provided in the result, which are samples product of flows wrongly paired.
- '%Miss' corresponds to the percentage of misses compared to the amount of samples provided.
- 'Average error' indicates the how far, on average, were the miss-samples from the correct value.
- 'Maximum error” indicates how far was from the correct value the worst sample in the result.

5% packet loss:

Version	Samples	%Samples	Misses	%Miss	Average error	Maximum Error
Regular	820.184	72.06%	319	0.0389%	20 ms	191 ms
With C3	694.360	61.01%	305	0.0439%	24 ms	191 ms
Retroactive C5	818.398	71.91%	31	0.0038%	28 ms	141 ms
Seqnum check	820.184	72.06%	319	0.0389%	20 ms	191 ms
Strict Flags	309.625	27.20%	73	0.0236%	31 ms	165 ms

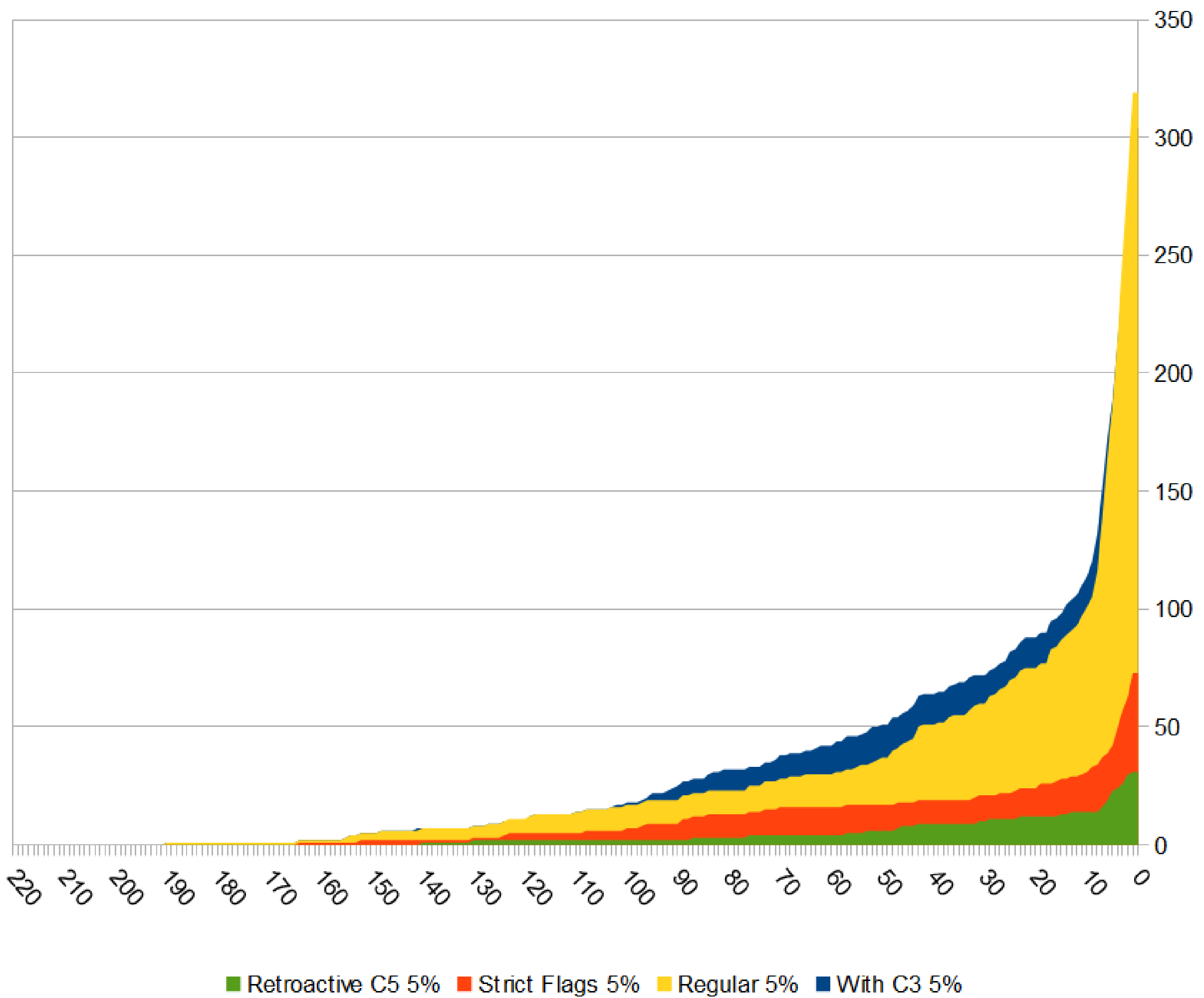


Figure 3: Cumulative distribution of errors, the Y axis shows the amount of errors bigger than the miliseconds marked by the X axis. The seqnum check version is missing because it has the exact same results as the regular version.

10% packet loss:

Version	Samples	%Samples	Misses	%Miss	Average error	Maximum Error
Regular	648.966	57.02%	417	0.0643%	25 ms	211 ms
With C3	560.805	49.27%	414	0.0738%	26 ms	213 ms
Retroactive C5	646.569	56.81%	39	0.0060%	27 ms	156 ms
Seqnum check	648.966	57.02%	417	0.0643%	25 ms	211 ms
Strict Flags	243.894	21.43%	101	0.0414%	35 ms	211 ms

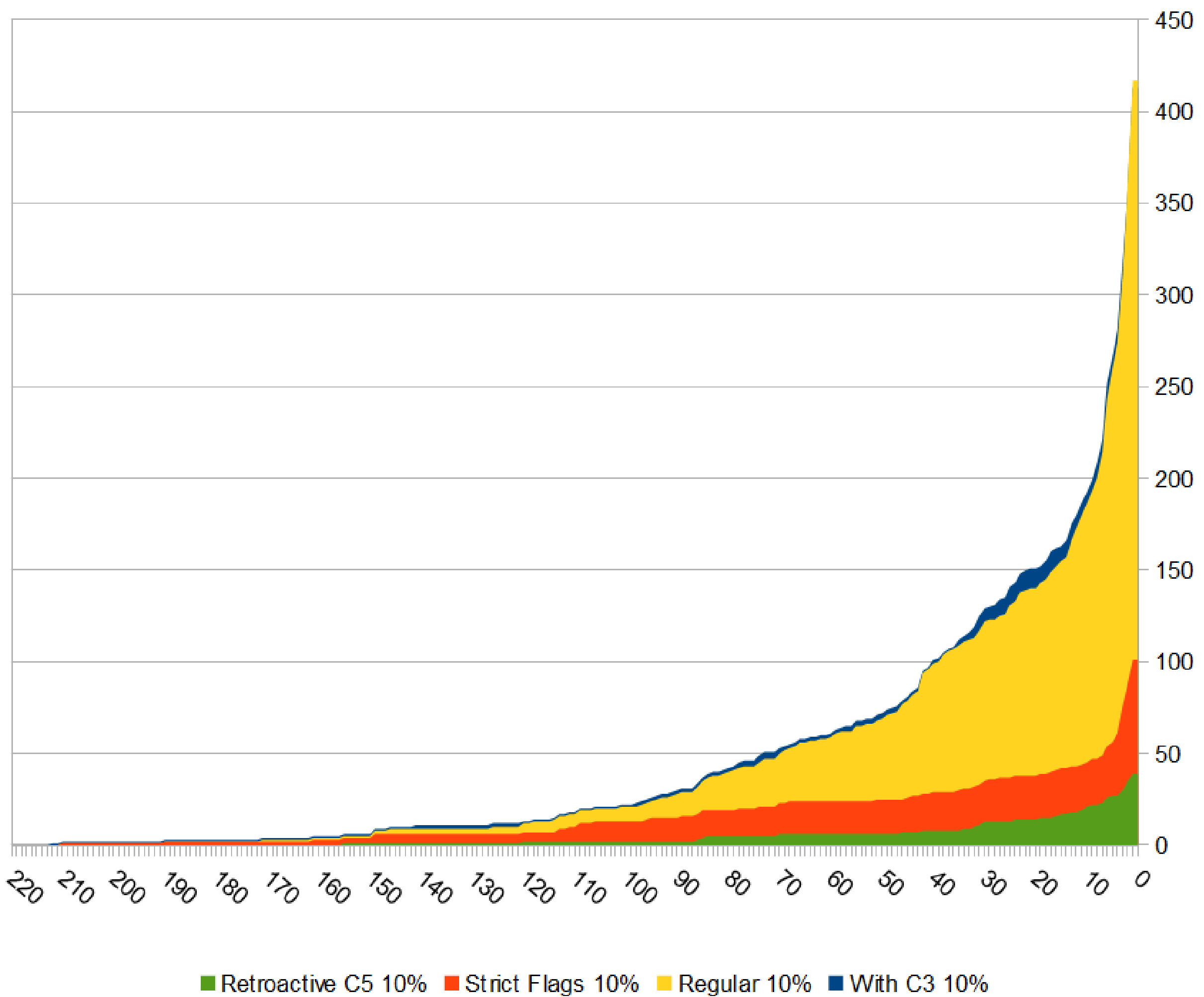


Figure 4: Cumulative distribution of errors, the Y axis shows the amount of errors bigger than the miliseconds marked by the X axis. The seqnum check version is missing because it has the exact same results as the regular version.

Observations on the results:

- The regular version performs quite well by itself giving quite precise measures of the ping during the trace (25ms average error * 0.0643% is almost negligible).

- C3 does not really solve any of the misses in the regular version, as we suspected, and in fact removes a bigger percentage of good samples than bad ones. Plus, looking at the graphs, it seems to produce bigger (though less) mistakes than the regular version.

- Retroactive C5 performs superbly, eliminating about 90% of the misses of the regular version and only a small percentage of valid samples.

- The version with sequence number check does not vary at all from the regular version, so there is no pair that would be approved by the rest of the rules but not by the sequence number check.

- 'Strict flags' cuts the amount of samples by a lot, but manages to remove a bigger percentage of misses than samples, so it could be circumstantially useful.

After checking the precision of the timestamps provided by the method, we compared the two flow-level delay estimation methods: the endpoint estimator (using only the two timestamps of the flow) and the multiflow estimator (using all the timestamps within the duration of the flow). To check that, the program outputs one line for each flow, with the three delays (endpoint, multipoint and real) on top of other data that helps identifying the flow.

The tests were done on the sample with 5% losses with the best version (retroactive and strict C5), and the output was done for one of every 30 flows, without favouring longer flows. This resulted in 11.712 flows having output.

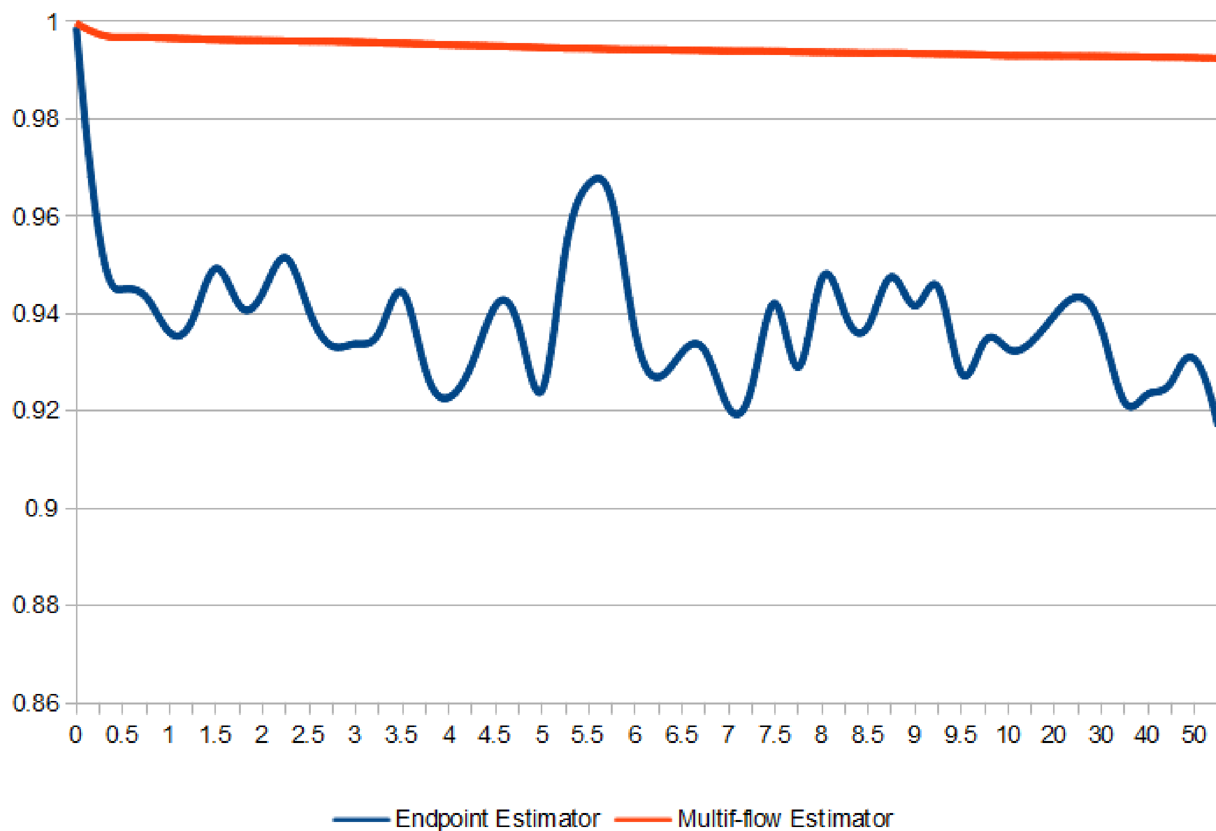


Figure 5: Graph showing the precision of both estimation methods, the Y axis shows how close are the estimations to the real value (1 being the exact value) and the X axis shows the flow length in seconds. From 10 seconds and on the scale is 20 times as dense (each notch represents 5 seconds instead of 0.25).

- The multiflow estimator averages a 99.65% precision. Precision is defined as how close were the estimated values from the real values
- The endpoint estimator averages a 94.13% precision.
- The biggest delay overestimation of the Multiflow estimator was 5.82% higher than the real delay, and the biggest underestimation was 5.09% lower than the real delay.
- For the endpoint estimator they were 39.93% higher and 55.79% lower, respectively.
- The multiflow estimator's precision does not vary much with flow length, other than improving sightly for longer flows.
- On the other hand, the endpoint estimator gets 99.45% precission for flows under 2 seconds, 96.4% for flows under 10 seconds, and 83.48% for flows over 10 seconds.

VIII. Schedule and costs

8.1. Schedule

The project has been split in four phases, each composed by various steps. This section will name each step and its corresponding professional category.

1. Project definition:
   - Initial organization (project manager)
   - Research the delay-estimation method through Netflow described by the article (analyst)
   - Research Netflow, IPFIX and other flow-level reporting tools (analyst)
   - Research hardware capable of using the previous features (analyst)
   - Research suitable simulation environments (analyst)
   - Project organization (project manager and analyst)

2. Development:
   - Split the required features and responsibilities in different C modules or script files (analyst)
   - Design and develop the C module that processes flow files (analyst and programmer)
   - Design and develop the C functions needed to store and manage flows and the auxiliary data structures (analyst and programmer)
   - Design and develop the function that processes flows and stores pairings (analyst and programmer)

- Design and develop the module that outputs the results (analyst and programmer)
- Perform the changes necessary to the source code of the tools that will be used during testing, so that their output is compatible with the rest of the environment (programmer)
- Design and develop the scripts needed to manage, adapt and test the output of the different tools (analyst and programmer)

3. Testing and deployment:
    - Test the software in a simulated environment with generated traffic (analyst and programmer)
    - Deploy and configure the hardware needed (technical support)
    - Deploy and configure the software (technical support)
    - Test the software in the real environment with controlled traffic (analyst and programmer)
    - Adjust the initial traffic constants used by the software (programmer)
    - Test the software with real traffic (analyst and programmer)
    - Review the results (analyst)

4. Documentation
    - Write the final report (analyst)
    - Final verification and approval (analyst and project manager)

- Project definition
  - Initial organization
  - Research the delay-estimation method
  - Research Netflow
  - Research capable hardware
  - Research suitable simulation environments
  - Project organization
- Development
  - Split the features
  - Develop module that processes flow files
  - Develop functions needed to store and manage data structures
  - Develop the function that processes flows
  - Develop the module that outputs the results
  - Adapt third-party software
  - Develop the scripts needed to manage the different outputs
- Testing and deployment
  - Test in a simulated environment
  - Deploy hardware
  - Deploy software
  - Test the software with controlled traffic
  - Adjust the parameters to the final environment
  - Test the software with real traffic
  - Review the results
- Documentation
  - Write the final report
  - Final verification and approval

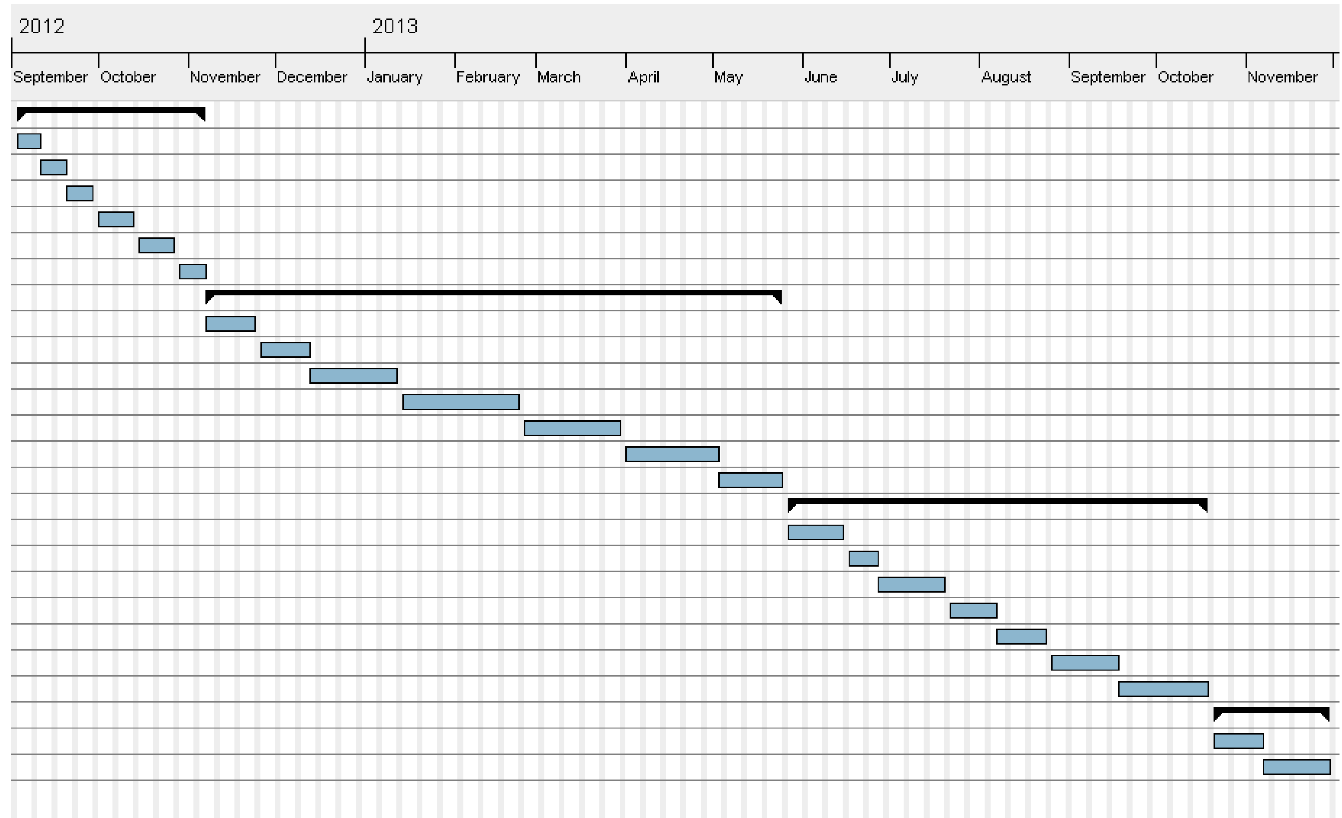


Figure 6: The project's Gantt chart

8.2. Costs

– Human costs:

Category	Salary/Hour	Hours	Total salary
Project manager	€28.00/h	60h	€1,680.00
Analyst	€22.00/h	480h	€10,560.00
Programmer	€16.00/h	440h	€7,040.00
Technical support	€15.00/h	100h	€1,500.00
Total		1080h	€20,780.00

– Hardware and license costs:

Developing the software did not require any paid licenses or specific hardware, but for the deployment it is necessary to have compatible hardware available. Given that this project's objective is mostly to retrofit latency estimations to Netflow-able routers, it is not expected to purchase specific hardware for it. The cost of a brand new compatible Cisco router varies from $1,000 to up to $10,000 or even more, depending on the modules and capabilities, so the hardware cost heavily depends of the size of the network.

IX. Conclusion

Developing this project has taught me quite a bit about Cisco's IOS and hardware, its legacy and its future. I was surprised at how inflexible this operating system can be, but I understand that it is a trade-off necessary when prioritizing reliability. Still, I expect that with the newer versions, the strictness of the commands and configurations will get loosened. Specifically for this purpose, IPFIX looks to be much easier and powerful than current version, specially if they include (or allow to add) hash-based sampled netflow, which is crucial.

I have also learned quite a bit about simulated network topologies, but I admit to have been disappointed by GNS3's current state, which can only be reliably used to roughly test commands and configurations, being is unable to represent many of the intricacies of a real environment.

Regarding the delay-estimation program, I have been satisfied by the performance shown, both at being able to match many of the flows even with high loss levels, and the accuracy of the samples provided. I was happy at finding how effective it is to check newer flows before exporting samples (the “retroactive C5”), an idea I had when reading the article for the first time. I remember being surprised by it not being mentioned, even if there were performance concerns.

Speaking of which, I had to optimize many of the data structures and loops for the Caida traces, since their massive size and density plus the multiplicative nature of some of the loops could make some of the executions quite long. Then there was also the memory requirements, since 2 million flows correspond to around 300MB (a bit unreasonable for such a simple program); so I programmed buffers for the flows and every reference to them, which get cleared periodically and help with that regard.

Finally, regarding the method this project revolves around, I consider its results very successful and proof of how it can be very useful in the proper circumstances. The evaluation confirms the conclusions drawn by the research team of the Purdue University, about how it's not that hard to pair flows generated by real traffic even in very high-bandwidth topologies.

The multi-flow estimator performed better in all situations compared to the endpoint estimator (which was suggested but not confirmed by the paper). The advantages of the endpoint estimator are limited to convenience and performance, since it requires less processing power and coding, but will be always out-classed by the multi-flow estimator regarding precision.

About it's current usefulness, this method is way too dependant on a consistent sampling of the traffic, a convenience pretty much all contemporary routers lack. Because of that, it would be hard to find an scenario where its application is warranted.

On the other hand, its future use might be brighter, depending on the capabilities of the next generation of routers. If they continue the trend of not being specially useful for the diagnostic of quality of service issues, but continue to provide support to flow-level reports (and specially, if the comply with IPFIX's standard), it would not surprise me if this method becomes a very convenient way to diagnose networks. But as of now, IPFIX is just a recently shipped specification, and its true potential remains to be researched once the hardware and software that features it becomes popular.

X. Bibliography

[1] Myungjin Lee, Nick Duffield and Ramana Rao Kompella,Purdue University, "Two Samples are Enough", 2010, http://www.cs.purdue.edu/homes/kompella/publications/infocom10netflow.pdf

[2] Cisco IOS NetFlow, http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html

[3] NTP: The Network Time Protocol, http://www.ntp.org/

[4] Common Open Research Emulator, http://code.google.com/p/coreemu/

[5] softflowd - fast software NetFlow probe, https://code.google.com/p/softflowd/

[6] Graphical Network Simulator, http://www.gns3.net/

[7] Oracle VM VirtualBox, https://www.virtualbox.org/

[8] Dynamips, http://www.gns3.net/dynamips/

[9] Arch Linux, a lightweight and flexible Linux distribution, https://www.archlinux.org/

[10] Linux Traffic Control, http://lartc.org/manpages/tc.txt

[11] flow-tools - Tool set for working with NetFlow data, http://linux.die.net/man/1/flow-tools

[12] tcpdump, a powerful command-line packet analyzer, http://www.tcpdump.org/

[13] editcap - Edit and translate the format of capture files, http://www.wireshark.org/docs/man-pages/editcap.html

[14] pcap_diff, http://sourceforge.net/projects/pcapdif/

[15] YAFSCII, http://tools.netsa.cert.org/yaf/yafscii.html

[16] CAIDA: The Cooperative Association for Internet Data Analysis, http://www.caida.org/home/