High Frequency sFlow v5 Counter Sampling

Issues and Suggested Resolutions
Version 0.3

Rick Jones rick.jones2@hp.com

This is version 0.3 of Rick Jones' attempt to describe some of the issues he has seen in high frequency sFlow counter sampling. It was last modified on 2011-10-24.

Table of Contents

1 Introduction

sFlow is the name given to a technology which can be used to monitor high speed networks. It seeks to provide visibility into the behaviour and characteristics of networks and has been adopted by a number of vendors and users.

A complete description of sFlow is beyond the scope of this document. A complete description of sFlow can be found at the sFlow website.

One of the goals of sFlow is to be as light-weight as possible - both for the agents - the producers of sFlow data, and the collectors - the consumers of sFlow data. This is reflected in the specification. For most ordinary and mundane uses of sFlow this is sufficient, but some of the specified behaviours result in unacceptable levels of inaccuracy when one wishes to perform "high-frequency" counter sampling - perhaps even as frequent as once per second. This paper will attempt to identify those aspects of the specification which lead to this inaccuracy and provide suggestions as to how they might be resolved.

It is not a goal of this paper to solve issues with sources of error outside the context of the sFlow agent's collecting and transmission of counter samples. For example, it does not seek to solve the problem of variable delays in transmission of sFlow PDUs through a network and/or forwarding agents. However, it is fine if solutions to the problem of agent context counter inaccuracy happen to help with that issue too :)

2 Randomization

Section 3.1 of the sFlow specification (not quoted here for the sake of brevity) goes to some trouble to ensure that flow samples are randomized about the rate specified by the user. It does this to ensure there is no synchronization of the flow samples to traffic flows.

It also suggests that when multiple ports are being counter sampled that they too be scheduled randomly:

> *If the sFlow Agent chooses to regularly schedule counter sampling, then it should schedule each counter source at a different start time (preferably randomly) so that counter sampling is not synchronised within an agent or between agents.*

At the same time, Section 5 of the sFlow specification suggests:

> *The sFlow Agent should try to piggyback counter samples on the datagram stream resulting from Packet Flow Sampling.*

and allows samples (both flow and counter) to be accumulated for upwards of one second to enable filling sFlow PDUs to the greatest extent possible:

> *While the sFlow Datagram structure permits multiple samples to be included in each datagram, the sFlow Agent must not wait for a buffer to fill with samples before sending the sFlow Datagram. sFlow is intended to provide timely information on traffic. The sFlow Agent may at most delay a sample by 1 second before it is required to send the datagram.*

There is only one timestamp in a v5 sFlow PDU, the "uptime" field of the PDU header. The specification suggests:

> *Current time (in milliseconds since device last booted). Should be set as close to datagram transmission time as possible.*

Now, in the absence of flow or multiple-port counter sampling, the upwards of one second delay between the time the counter samples are taken and the time they are transmitted can be considered an acceptable systematic error. Particularly when counter sampling is happening at rather long intervals such as 60 seconds. A one second skew is not that big an error in such cases and only the most picky would be likely to complain :)

However, when the sample interval is in the "high-frequency" range - ie single digit seconds, that one second skew starts to become more meaningful. Particularly when the goal behind having high-frequency counter sampling is to be able to correlate that data with other sampling happening at a similarly high frequency.

While annoying, that is not the main issue. The main issue arises when flow sampling is enabled in conjunction with counter sampling. Perhaps also when multiple ports are configured for counter sampling. When this is done, a "following all the suggestions" sFlow agent configured for high-frequency sampling starts to demonstrate large **random** errors rather than systematic errors. This is doubpleplusungood.

Consider three scenarios. We assume there is always some sample already queued when these counter samples are taken, so the one second coalescing timer is already running. This could be the case with either flow sampling, or multiple port counter sampling.

S is nominally a 1 second sample interval, but randomization or something else has it just a little longer than one second, where the first sample is just before a queued PDU is sent and the second sample is just after. What was just a bit more than one second's worth

of counters is then seen as two seconds' worth. The collector will then think that the rates were actually ~1/2 of reality - ie the error is nearly 50%.

T remains consistent and is thus a boring case.

U is an every two second sample interval. Randomization (or other delays) have made it just a bit shy of two seconds. The first is taken very close to the beginning of a PDU interval, and the second just before the end of the next interval. In this case, what was a nearly two second interval appears to be a one second interval to the collector, who will then report rates nearly 2X reality or an error of 100%.

We have left-out a nominally 1 second counter sample interval that ends-up being less than one second - this would appear as two counter samples for the same port in the same PDU! We have also left-out the nominally 2 second counter sample interval that ends-up being just more than two seconds.

```
                    U       U
                   T    T
                   S      S
Sample taken  0----1----2----3----4----5
by switch

                         U    U
                        T    T
                        S         S
Timestamp in  0----1----2----3----4----5
PDU header

                         U    U
                        T    T
                        S         S
Sample sent   0----1----2----3----4----5
by agent

                          U    U
                         T    T
                         S         S
Sample seen   0----1----2----3----4----5
by collector
```

"But that is only theoretical. Does it really happen?" you might ask. Apart from the author's experience noticing that ntop specified maximums to clip the values being recorded in its RRDs to a presumed link-rate (suggesting at least some were seeing values greater than link-rate), there is also experimental evidence to show it happening in a reasonable test setup.

To that end, we now look at a selection from several plots of "netperf vs sFlow counter" results. Netperf's "demo" mode (–enable-demo when running ./config) is used to emit interim netperf results every second. The sFlow counter results are for counter sampling at intervals of 10, 5 and 2 seconds, along with flow sampling of one out of every 0, 200, 2000 and 20000 frames. To keep some of the graphs from being simply three lines more or less

on top of one another, the y-axis scale is not the same between the graphs. The full set of charts can be seen in Appendix B [Full Set of Charts], page 13.

The reader can see that for some combinations of counter and flow sampling rates, the accuracy of the sFlow counter samples is quite poor indeed.

> Note: Rrdtool is used to hold the data and generate the charts and was configured with a one second step size. Its tendency to "align" things to integral time interval boundaries means that if anything the graphs are understating the magnitude of the errors. For those who wish to see the raw data in full, it can be found under '`data/`' at the online location where the online version of this document is found.

Collector Time	**Delta (ms)**	**Agent Uptime**	**Delta (ms)**	**sFlow ifInOctets**	**Delta (octets)**	**Octets/s (MB/s)**
36.724114	n/a	365016833	n/a	1333740672	n/a	n/a
46.945565	10221	365027054	10221	1457104128	123363456	12.07
56.711708	9766	365036820	9766	1580566104	123461976	12.64
06.393401	9682	365046502	9682	1703880898	123314794	12.74
16.945428	10552	365057054	10552	1827268642	123387744	11.69
26.764257	9189	365066873	9819	1950680618	123411976	12.57

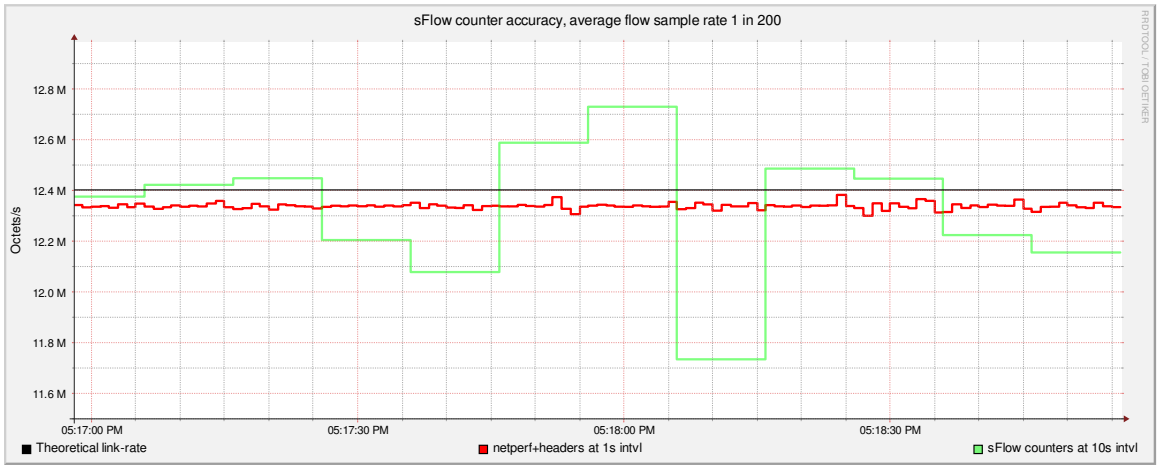

Figure 2.1: Ten second counter samples with flow samples 1 out of every 200 frames

The octet counts over the "intervals" are equivalent to within a fraction of a percent, consistent with the rather uniform nature of the traffic being sent by netperf, but the time deltas based on either the arrival timestamp at the collector or the uptime field of the sFlow PDU vary considerably. This should help confirm that the charts are not simply showing rrdtool artifacts but reflect actual inaccuracies in the sFlow PDUs thanks to over zealous adherence to the specification :) This should be particularly visible in the next two sets of data.

Collector Time	Delta (ms)	Agent Uptime	Delta (ms)	sFlow ifInOctets	Delta (octets)	Octets/s (MB/s)
22.942357	n/a	365723054	n/a	1389790522	n/a	n/a
27.942580	5000	365728054	5000	1451506394	61715872	12.34
32.228223	4285	365732340	4286	1513143266	61636872	14.38
37.942415	5714	365738054	5714	1574831750	61688484	10.80
42.942337	4999	365743054	5000	1636549076	61717326	12.34

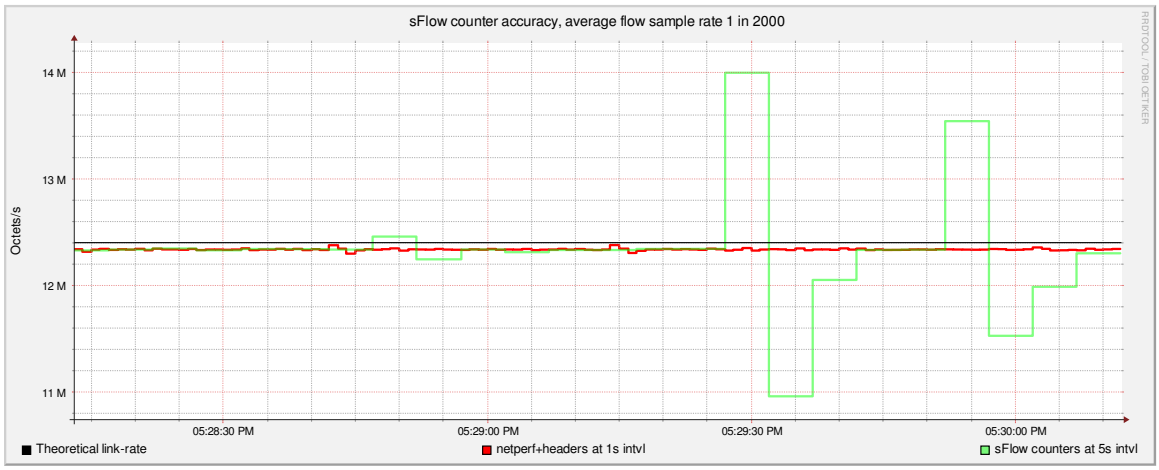

Figure 2.2: Five second counter samples with flow samples 1 out of every 2000 frames

Collector Time	Delta (ms)	Agent Uptime	Delta (ms)	sFlow ifInOctets	Delta (octets)	Octets/s (MB/s)
33.783233	n/a	366093896	n/a	1638364736	n/a	n/a
35.206680	1423	366095320	1424	1663674350	25309614	17.77
37.674285	2467	366097787	2467	1688997626	25323276	10.23
39.940985	2266	366100054	2267	1712693606	23695980	10.45
41.486578	1545	366101600	1546	1738035192	25341586	16.39

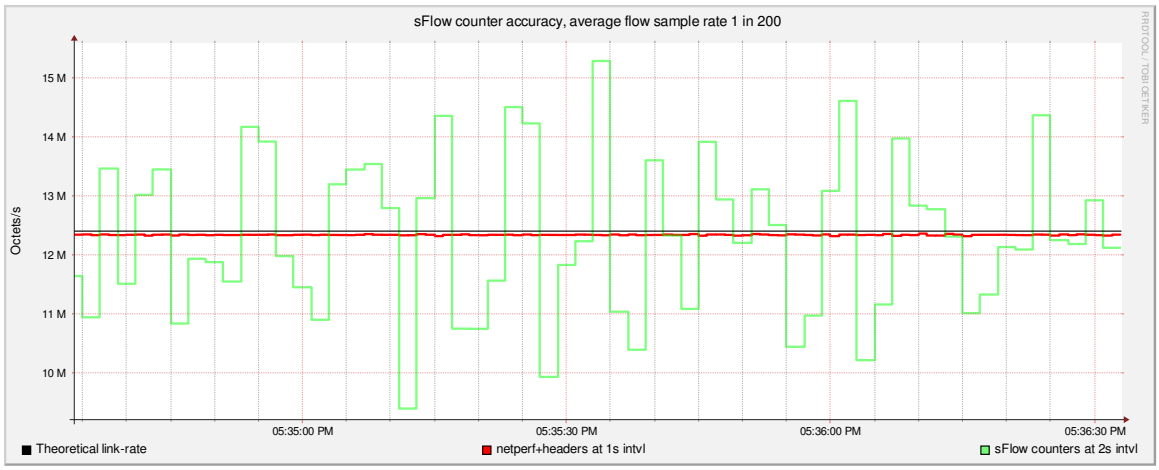

Figure 2.3: Two second counter samples with flow samples 1 out of every 200 frames

The reader can see that as the counter sampling interval shrinks, the magnitude of the error from randomization combined with coalescing increases. This error, being random, cannot be "post processed" out. One does not know how long any given counter sample sat being coalesced. One cannot even use the inter-arrival time as a guess because the sFlow specification allows the agent a rather wide latitude in deciding when to send PDUs:

> *A maximum Sampling Interval is assigned to each sFlow Instance associated with an interface Data Source, but the sFlow Agent is free to schedule polling in order maximize internal efficiency.*

so one cannot even know with particular certainty that a port configured for a counter sampling interval of N seconds is being sampled once every N seconds.

The attentive reader will also notice that the octet delta for the two second counter sampling interval is not always consistent to within a fraction of one percent. Looking at the graph for Figure B.9 one can see some inaccuracy even without flow samples configured. This additional source of inaccuracy is currently believed to be an artifact of the specific agent rather than the sFlow specification. Of course that supposition is subject to change at any time :)

2.1 Solution to Randomization

There are at least two possible solutions to dealing with the unacceptable error magnitude stemming from the combination of randomization of sFlow PDU transmission time and the

single timestamp. One solution would be to include a timestamp with each counter sample. The other would be to remove some of the randomness from counter sampling - particularly in so far as counter samples sit waiting for transmission.

2.1.1 Adding a Timestamp to Counter Samples

One potential solution to the issue of random error would be to include a timestamp with each counter sample. This timestamp would be set to the time the sample was taken. For purposes of computing per-unit-time metrics, the collector would then ignore either PDU arrival time or PDU header timestamp and use this timestamp, and the counter timestamp of the previous sample(s). This would eliminate both the random and systematic error in counter samples.

Timestamps in sFlow have some issues though. First, the uptime, and presumably any added counter sample time, would be defined as being milliseconds since the device last booted, as this is how time is defined in the v5 specification. This means that any sFlow collector must compute and maintain a mapping from agent time to "real" time. This means that a collector must maintain "state" for the agent which may be added and undesirable overhead.

Further, while it is the author's opinion that "time" in an sFlow agent **should** advance as "true" time does, there is no guarantee in the sFlow specification that the time source used by an sFlow agent is all that good. There is no requirement that it be synchronized with a source of "true" time such that it would advance in step with that time. This would put an added burden on the collector because it could not fully trust the time deltas measured from the counter sample timestamps. It would have to start trying to be a "Junior NTP" implementation comparing the rate of increase of its own, presumably synchronized, clock with that of the sFlow agent(s) sending it PDUs.

Further still, timestamps will consume additional space in the sFlow PDU, either further reducing the number of samples per PDU, or lessening the benefit of following the suggestion of removing extraneous data. Chapter 3 [Extraneous Data], page 9.

And perhaps finally, this resolution to the problem requires a revision of the specification for on the wire behaviour, and alteration of both existing agents and collectors.

2.1.2 Prioritizing Counter Samples

A second possible solution, and one preferred by the author, is to simply minimize the randomization experienced by counter samples to the point of the added error being epsilon. This can be accomplished by "prioritizing" counter samples such that they never sit being coalesced. This should virtually eliminate both the random and systematic errors of coalescing by the agent.

It is of course, still desirable to maximize the number of samples per PDU, particularly under load. So rather than take the naive path of simply flushing a PDU for each individual counter sample a somewhat more sophisticated algorithm is employed. This also allows retention of at least some of the randomization of counter sampling intervals. Instead of thinking of counter sampling in the context of a single port, we consider what might be called "PDU groups." PDU groups are sampled together and sized based on how many counter samples will fit in a single PDU.

As ports are configured for sampling at a given maximum interval, they are assigned to a PDU group for that interval. The agent can start with N PDU groups, spreading

ports across them evenly and filling them all before adding further PDU groups. If the agent starts with N=1 this will start with maximizing samples per PDU though minimizing randomization of sample times between ports (within a PDU group). Starting with a value of N = MaxPossiblePorts/SamplesPerPDU will start with maximum randomization between ports and minimum samples per PDU. Once all ports are configured for sampling the two schemes converge.

It would be left up to the agent whether it occasionally "shuffles" ports among PDU groups.

The sampling timer is set for the PDU group, with whatever randomization is desired, and when the timer expires samples are taken and handed to the entity building PDUs. If samples are handed one at a time, a flag is included to indicate whether the code accumulating samples into PDUs must flush what it has. For the first M-1 samples of the PDU group, this flag will be clear. The code building the PDUs is then free to continue accumulating samples. The last sample from the PDU group will have the "flush" flag set. The entity building PDUs must then flush the PDU containing any accumulated samples and this sample. This flush is what ensures counter samples do not sit for any appreciable length of time being coalesced into a PDU.

If samples of the PDU group are presented to the entity "en mass" rather than one at a time, the entity must treat things as if the flush flag was set for the last sample in the one at a time case - one implementation might be to simply flush whatever is currently accumulated and then send the counter samples as one PDU (remember the PDU group is defined as generating no more samples than fit in a PDU). Or it may behave as if it was called with samples one at a time, filling the the currently accumulating PDU, sending it, and then sending a partial PDU containing the remaining samples.

This all presumes that the act of building/flushing the PDU takes comparatively little time relative to the advancing of time, allowing the PDU timestamp to be set by the entity building the PDUs. If this is not the case there can still be skew error and it will be necessary to address that by judicious manipulation of the PDU timestamp - perhaps sending N PDUs with the same timestamp (selected by the time the first counter sample is given to the entity building PDUs) but increasing sequence numbers.

The first and foremost advantage to this method of resolving the random error issue is it does not require any changes to collectors. sFlow collectors will see no change in the on-the-wire protocol. They can continue to process PDUs as they have before, only the data will be considerably more accurate. All that is required is an update to the agents, which would be required of any protocol change anyway.

It retains a degree of randomization at the PDU group level. Further it does not significantly increase the number of PDUs being sent by the agent and processed by the collector. Flow samples will still continue to be accumulated and at most only one additional PDU per "burst" of PDUs will be sent. In fact, since it does not increase the space consumed per sample as adding timestamps would, it is quite possible, though not worked-out yet by the author, that this proposal would actually result in a higher average samples per PDU than adding counter timestamps. Especially when many ports are being sampled per agent.

3 Extraneous Data

The sFlow specification suggests:

> *Wherever possible, the if_counters block should be included. Media specific counters can be included as well.*

which, sadly, has been taken rather fully to heart by sFlow implementers. At one level, and certainly at one time, including media-specific counters may have been all well and good. For example, when sFlow was first specified, Ethernet was still "real Ethernet" with its half-duplex CSMA/CD operation, and so media-specific counters like:

- dot3StatsSingleCollisionFrames
- dot3StatsMultipleCollisionFrames
- dot3StatsDeferredTransmissions
- dot3StatsLateCollisions
- dot3StatsExcessiveCollisions

could still increment. Some, such as single and multiple collision and deferred transmission could be expected to increment routinely and perhaps serve as an indication of link loading. However, with today's ubiquity of full-duplex operation, those counters are vestigial at best. There are no collisions in a full-duplex Ethernet. In fact, the "error only" nature of the other media-specific Ethernet counters are such that during "normal" operation the chances of any of them actually counting is minimal. And if they do increment, at least one of the generic error counters should increment as well, serving as the proverbial canary in the coal mine. As such they are effectively wasted space in the sFlow PDU, limiting the number of counter samples which can fit and so requiring more PDUs per given number of sampled ports.

3.1 Solution to Extraneous Data

Extending the sFlow MIB to allow control over inclusion of media-specific counters in counter samples would allow a non-trivial increase in the number of counter samples per PDU, which would reduce the per-PDU cost of sampling. Based on eyeballing packet traces it would seem that the maximum number of counter samples per PDU with Ethernet and a 1400 byte PDU limit is seven counter samples. Removing the media-specific counters would save 13 * 4 * 7 = 364 bytes of data alone plus another 7 data_format structures for another 28 bytes. The 392 bytes saved should allow another two counter samples to be included per PDU, which would be a corresponding decrease in the per-PDU costs in both the agent and collector.

4 Lack of Coalescing

The “cost” of sFlow processing can be thought of as having two components - the first being “per-sample” and the second being “per-PDU.”

With all the trouble taken by the author to suggest changes which would lessen the length of time counter samples would sit being coalesced it might seem odd to see lack of coalescing mentioned as an issue in high frequency counter sampling. However, during his trip through several switches’ sFlow behaviour, the author has encountered at least one (which is not the one used for the rest of this writeup) where each port of the switch was associated with its own sub-agent and as such, there was no coalescing of the sFlow counter samples whatsoever, save perhaps with flow sampling.

While it is likely such a scheme has minimal agent-induced random error, as counter sampled port counts increase, the PDU-per sample nature of such behaviour becomes quite expensive. It maximized the per-PDU cost of sFlow processing.

This is not a specification failure as much as it seems to be an implementation failure, but it is mentioned here for completeness.

5 Conclusion

Based on the data presented, as well as the remainder of the data collected but not summarized here, it should be fairly clear that there are indeed fundamental issues in the sFlow specification which lead to considerable inaccuracy in high-frequency counter sampling. These issues can be and are perhaps best addressed with agent implementation changes which do not require any change to the "on the wire" protocol, nor any changes to collectors.

The sFlow specification should have an errata published suggesting the proposed implementation of "PDU groups" and an addition to the specification to allow control, via the sFlow MIB and/or agent command-line, over the inclusion of media-specific counters should be brought-forth.

These changes should increase the accuracy of high-frequency sFlow counter sampling without adding any significant burden on either agents or collectors.

Appendix A Test Configuration

The data presented here were collected using a small, two-node test setup consisting of one Hewlett-Packard Z400 workstation with a single, four-core Intel ® W3550 processor, 12 GB of RAM and running running Ubuntu 11.04 bits using one port of an HP NC364T to connect to the switch under test which was then connected to a venerable HP Compaq 8510w laptop running Ubuntu 11.10. The Z400 acted as both sFlow collector and traffic source via netperf with the laptop as the traffic sink. There was nothing else connected to the switch under test.

Apart from the undesirable random error behaviour of sFlow/the switch, the switch under test had a further bug in its sFlow implementation - it wrapped the ifInOctet and ifOutOctet fields of the generic interface counters at 32 bits rather than 64. To workaround this bug the interface from the Z400 to the switch was configured to run at only 100 Mbit/s rather than 1 Gbit/s. In this way the counters would take 10x longer to wrap.

The switch is not named here because the author is not interested in pointing fingers at any specific switch vendor. At least not in the context of this paper :) He will note that he has access to a variety of switches, across a number of vendors, so any guess as to the switch vendor is just that - a guess. And quite possibly wrong :)

Netperf interim results and sFlow counter samples were stored in round-robin databases using '`rrdtool`' and '`librrd4`' version 1.4.3-1ubuntu4. Traces of the sFlow PDUs were captured with '`tcpdump`'. All of this, and control of the sFlow agent on the switch under test was accomplished via a script which the author is willing to provide upon request. It may also be present with the raw data when/if that gets exported along with this paper.

A netperf UDP_STREAM test is used to ensure that there is no queuing of sFlow PDUs behind other traffic. The flow of UDP datagrams is from the Z400 to/through the switch, and the sFlow PDUs are from the switch to the Z400 and the links were all full-duplex. Thus the odds of sFlow PDUs being queued behind other traffic (such as TCP ACKs) was epsilon. The Linux networking stack provides "intra-stack" flow control when sending UDP datagrams so we know that the interim results reported by netperf are accurate in terms of what was sent out the Z400. We also know that there were few or no drops anywhere along the way by comparing the final result, which includes what the laptop actually received.

Appendix B Full Set of Charts

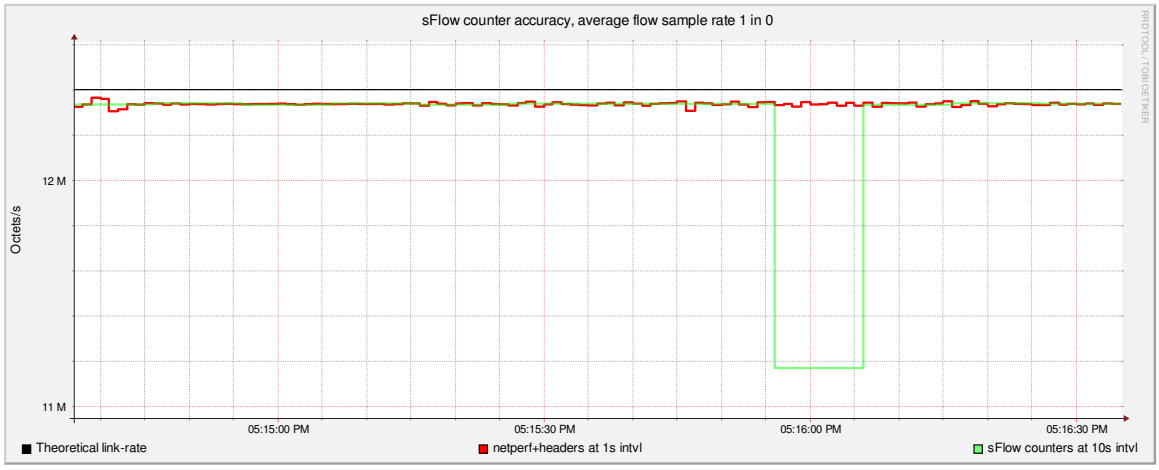

Figure B.1: Ten second counter samples with flow samples 1 out of every 0 frames

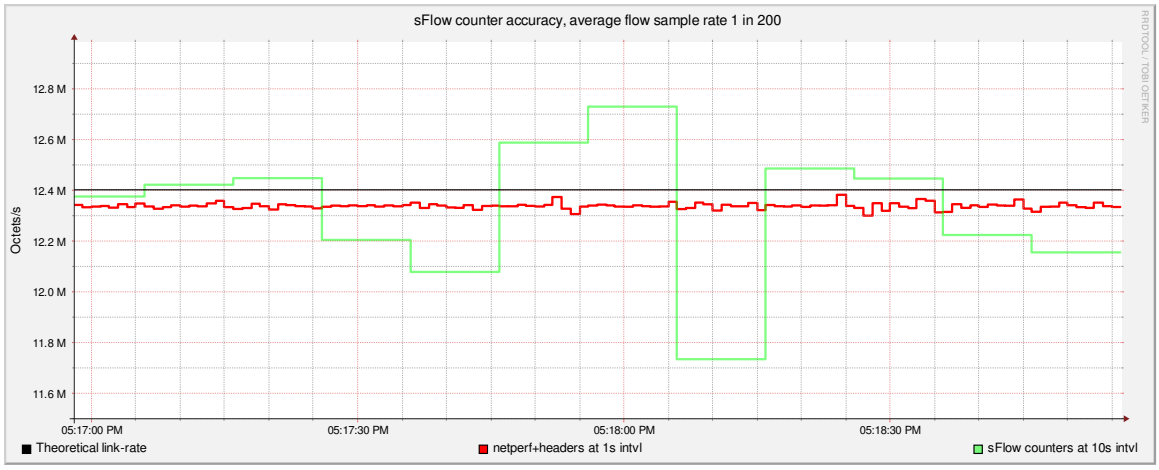

Figure B.2: Ten second counter samples with flow samples 1 out of every 200 frames

Figure B.3: Ten second counter samples with flow samples 1 out of every 2000 frames

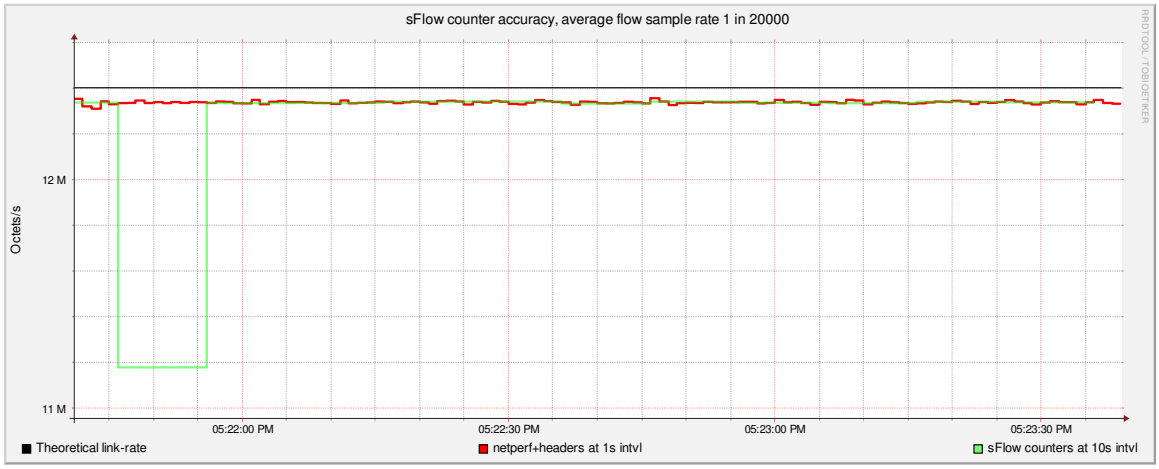

Figure B.4: Ten second counter samples with flow samples 1 out of every 20000 frames

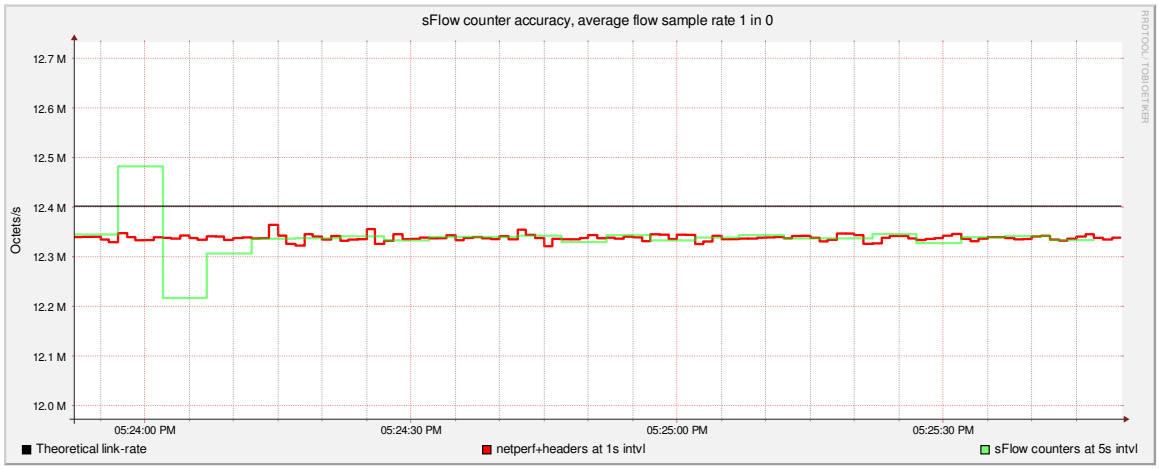

Figure B.5: Five second counter samples with flow samples 1 out of every 0 frames

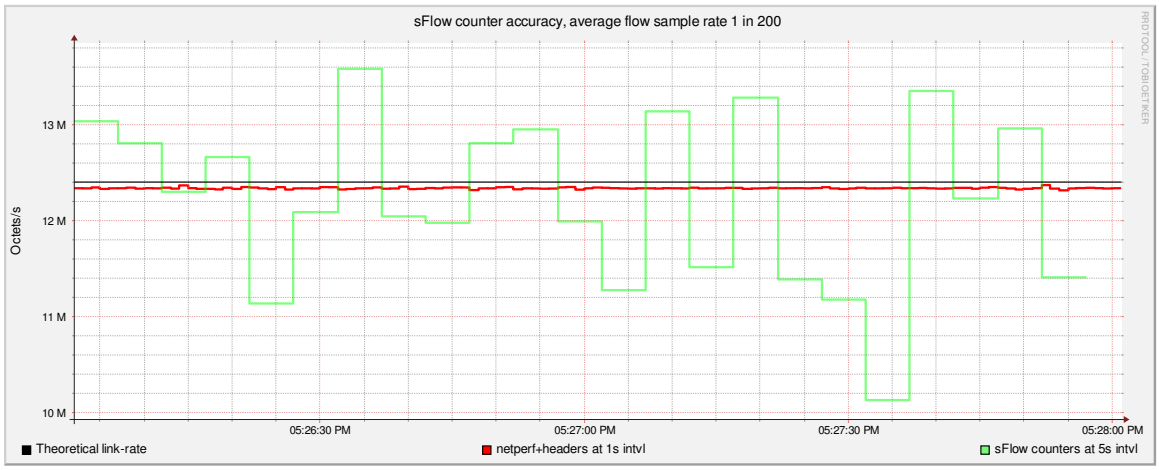

Figure B.6: Five second counter samples with flow samples 1 out of every 200 frames

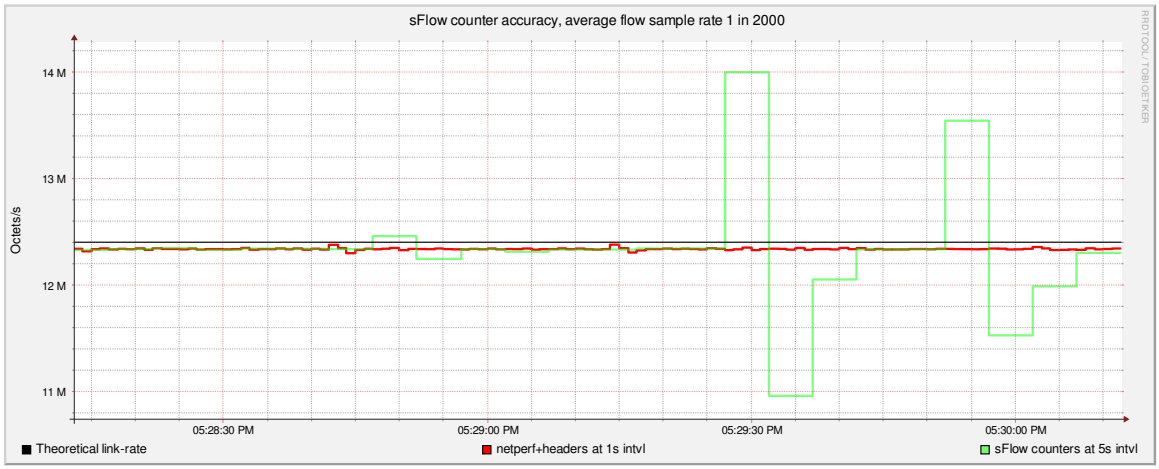

Figure B.7: Five second counter samples with flow samples 1 out of every 2000 frames

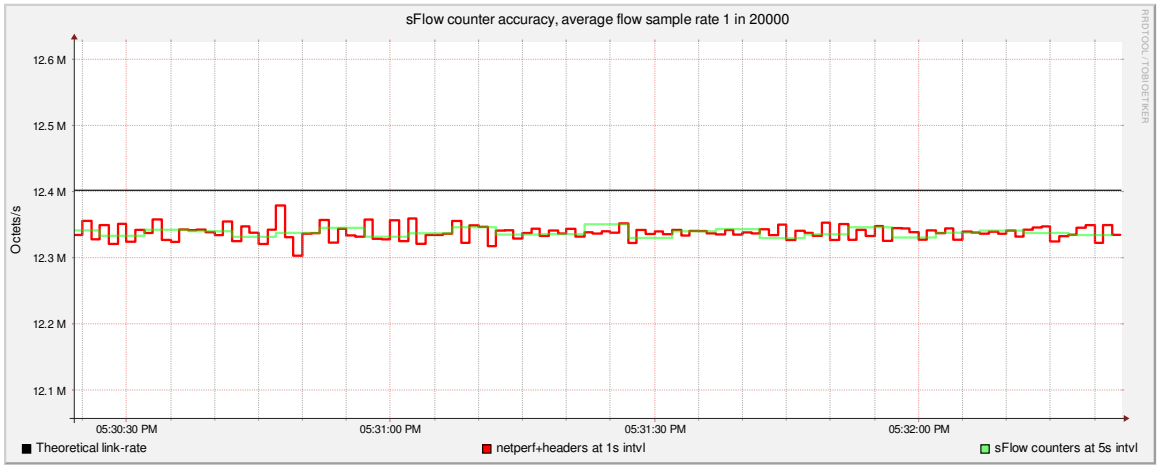

Figure B.8: Five second counter samples with flow samples 1 out of every 20000 frames

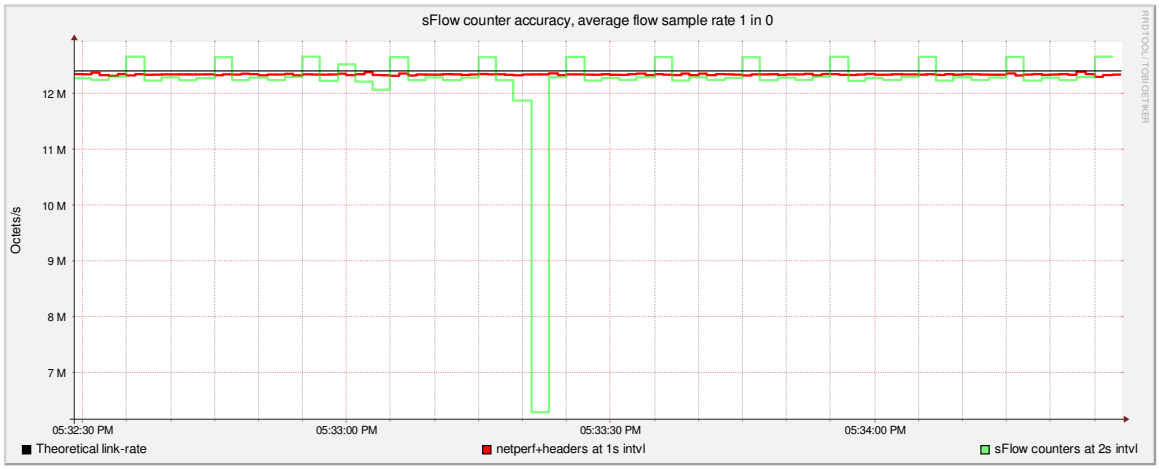

Figure B.9: Two second counter samples with flow samples 1 out of every 0 frames

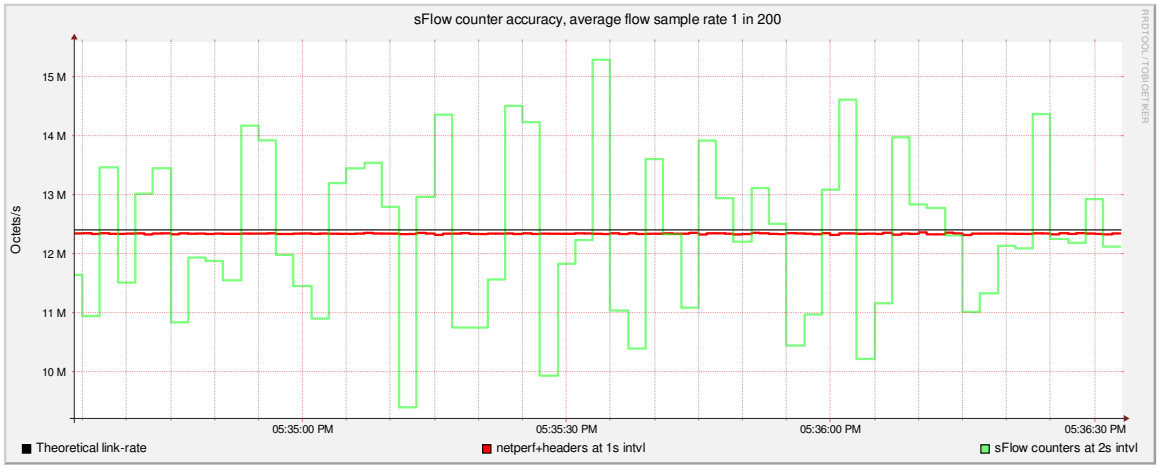

Figure B.10: Two second counter samples with flow samples 1 out of every 200 frames

Figure B.11: Two second counter samples with flow samples 1 out of every 2000 frames

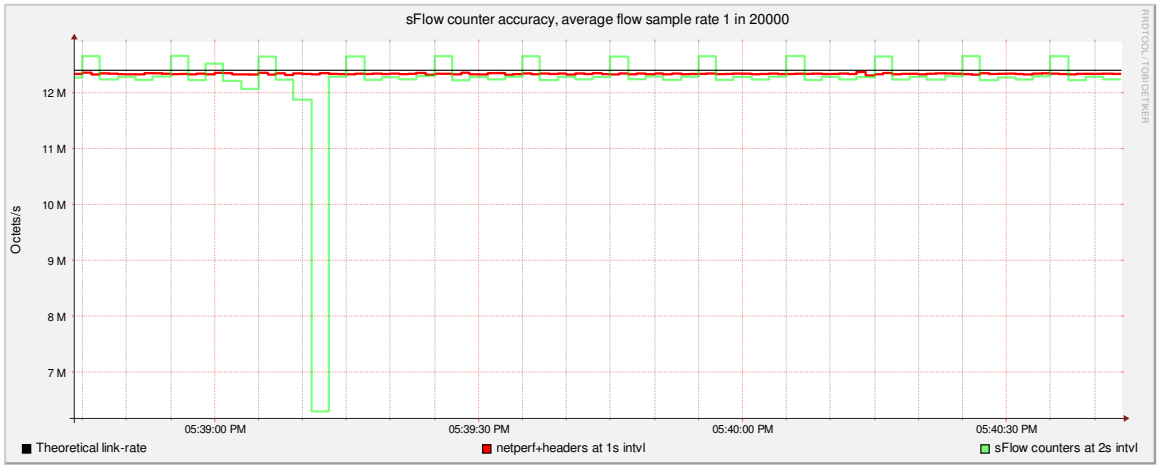

Figure B.12: Two second counter samples with flow samples 1 out of every 20000 frames

List of Figures

Index