A Practical Evaluation of Load Shedding in Data Stream Management Systems for Network Monitoring

Jarle Søberg, Kjetil H. Hernes, Matti Sickkinen, Vera Goebel, and Thomas Plagemann

University of Oslo, Department of Informatics
P.O. Box 1080, Blindern, N-0316 Oslo
{jarleso, kjetih, sickkine, goebel, plageman}@ifi.uio.no

Abstract. In network monitoring, an important issue is the number of tuples the data stream management system (DSMS) can handle for different network loads. In order to gracefully handle overload situations, some DSMSs are equipped with a tuple dropping functionality, also known as *load shedding*. These DSMSs register and relate the number of received and dropped tuples, i.e., the *relative throughput*, and perform different kinds of calculations on them. Over the past few years, several solutions and methods have been suggested to efficiently perform load shedding. The simplest approach is to keep a count of all the dropped tuples, and to report this to the end user. In our experiments, we study two DSMSs, i.e., TelegraphCQ with support for load shedding, and STREAM without this support. We use three particular network monitoring tasks to evaluate the two DSMS with respect to their ability of load shedding and performance. We demonstrate that it is important to investigate the correctness of load shedding by showing that the reported number of dropped tuples is not always correct.

Keywords: DSMS applications, data stream summaries, data stream sampling

1 Introduction

In this paper, we focus on network monitoring as the application domain for *data stream management systems* (DSMSs). More specifically, we evaluate the load shedding mechanisms of existing DSMSs in the case of *passive network monitoring* where traffic passing by an observation point is analyzed. This field of research is commonly challenged by the potentially overwhelming amounts of traffic data to be analyzed. Thus, data reduction techniques are important. Many of those techniques used by the traffic analysis research community as stand-alone tools are also utilized extensively for load shedding within the DSMSs. Examples include *sampling* [8, 9], *wavelet analysis* [11], and *histogram analysis* [23]. Monitoring tasks may also include timeliness constraints: Storing measurements and performing "off-line" analysis later is impossible in many monitoring scenarios where analysis needs to happen in near real-time. For example, an intrusion detection system (IDS) based on passive traffic measurements needs to generate alerts immediately in the case of a detected intrusion. Internet service providers (ISPs) are interested in real-time measurements of their backbones for traffic engineering purposes, e.g., to re-route some traffic in case congestion builds up in certain parts of the network. In addition, network traffic rarely generates a constant stable input stream. Instead, traffic can be bursty over a wide range of time scales [14], which implies that the monitoring system should be able to adapt to a changing load. For all these reasons, it is very interesting to study the applicability of DSMSs equipped with load shedding functionalities in this context.

The STREAM [1, 3, 4] and Gigascope [7, 6] projects have made performance evaluations on their DSMSs for network monitoring. In both projects, several useful tasks have been suggested for network monitoring. Plagemann et al. [17] have evaluated an early version of the TelegraphCQ [5] DSMS as a network monitoring tool by modeling and running queries and making a simple performance analysis.

With respect to load shedding, Golab et al. [10] sum up several of the different solutions suggested in the DSMS literature. Reiss et al. [22] evaluate sampling, multi-dimensional histograms,

wavelet-based histograms, and fixed-grid histograms in TelegraphCQ. They also suggest solutions for stating queries that obtain information about the shedded tuples.

We have used the two public-domain DSMSs; TelegraphCQ v2.0 and STREAM v0.6.0, for network monitoring. We have run different continuous queries and have measured system performance for different network loads. We choose to use these two systems since they are public available and have been used for network monitoring in earlier experiments. TelegraphCQ v2.0 supports load shedding, and STREAM v0.6.0 has not this functionality implemented. To our knowledge, no practical evaluations have been performed with focus on load shedding in DSMSs for network monitoring.

Our contribution is to propose a simple experiment setup for practical evaluation of load shedding in DSMSs. At this point, the experiment setup can be used for measuring *tuple dropping*, which is one of the load shedding functionalities. This is useful for systems that do not support load shedding. For measuring the tuple dropping, we suggest a metric; *relative throughput*, which is defined by the relation between the number of bits a node receives and the number of bits the node is able to compute. Our experiment setup can also evaluate the correctness of systems that support tuple dropping. Here, we show that TelegraphCQ reports less dropped tuples than what seems to be correct. We also use the experiment setup to show how TelegraphCQ behaves over time, and see that load shedding and query results reach an equilibrium. Second, we suggest a set of three network monitoring tasks and discuss how they can be modeled in TelegraphCQ and STREAM.

The structure of this paper is as follows: In Section 2, we describe the tasks and the queries. While discussing the tasks, we try to introduce system specific features for the two DSMSs. Section 3 presents the implementation and setup of our experiment setup, and Section 4 shows the results from our experiments. In Section 5, we summarize our findings and conclude that the experiment setup is useful for performance evaluations of network monitoring tasks for DSMSs. Finally, we point out some future work.

2 Network Monitoring Tasks and Query Design

In this section, we discuss three network monitoring tasks. We give a short description of each task and show how to model the queries. We also argue for their applicability in the current domain. The first two tasks are used in our experiments, while the third task is discussed on a theoretical basis. For that task, we show that some tasks may give complex queries that depend on correct input and therefore can not allow load shedding, since important tuples might be dropped.

For both systems, TelegraphCQ and STREAM, the tasks are using stream definitions that consist of the full IP and TCP headers. These are called `streams.iptcp` and `S`, respectively. The definition is a one-to-one mapping of the header fields. For example, the `destination port` header field is represented by an attribute called `destPort int`. We have tried to match the header fields with available data types in the two systems. For example, TelegraphCQ supports a data type, `cidr`, which is useful for expressing IP addresses.

2.1 Task 1: Measure the Average Load of Packets and Network Load

To measure the average load of packets and the network load, we need to count the number packets obtained from the network and calculate the average length of these packets. In the IP header, we can use the `totalLength` header field for this purpose. This header field shows the number of bytes in the packet. Having the information from this task may be of great importance for the ISPs to look at tendencies and patterns in their networks over longer periods. In TelegraphCQ, we model the query as follows:

```
SELECT COUNT(*)/60, AVG(s.totalLength)*8
FROM streams.iptcp s [RANGE BY '1 minute' SLIDE BY '1 second']
```

We see that for each window calculation[1], we count the number of packets each second by dividing by 60. We multiply the `totalLength` with 8 to get bits instead of bytes. The network load is estimated by finding the average total length of the tuples presented in the window. The query returns two attributes that show the average per second over the last minute. Alternatively, this query can be stated by using sub-queries, but because of space limitations, we do not discuss this solution for TelegraphCQ here.

The STREAM query is syntactically similar. We have chosen to split up this query into two queries, to show how they express network load in bytes per second (left) and packets per minute (right):

```
Load(bytesPerSec):
RSTREAM(SELECT SUM(totalLength)
        FROM S [RANGE 1 SECOND])

RSTREAM(SELECT AVG(bytesPerSec)
FROM Load [RANGE 10 SECONDS])
```

```
Load(packetsPerMinute):
RSTREAM(SELECT COUNT(*)
        FROM S [RANGE 1 MINUTE])

RSTREAM(SELECT AVG(packetsPerMinute)
FROM Load [RANGE 10 SECONDS])
```

Both queries use a sub-query to perform a pre-calculation on the tuples before sending the results to the main query. For ISPs, it would probably be interesting only to be notified when the load exceeds certain levels. This can be added as a `WHERE`-clause, stating `AVG(bytesPerSec) > 30 Gbits`, for example.

2.2 Task 2: Number of Packets Sent to Certain Ports During the Last n Minutes

In this task, we need to find out how the destination ports are distributed on the computers in the network, and count the number of packets that head for each of these ports. The number of packets are calculated over an n minutes long window.

This is an important feature, since there might be situations where there is a need for joining stream attributes with stored information. In network monitoring, this task can be used to show the port distribution of packets on a Web-server. Additionally, it is possible to re-state the task to focus on machines in a network instead of ports, i.e., how many packets have been sent to certain *machines* during the last n minutes. To solve this, we can look at the distribution of destination IP addresses instead of ports. For instance, if one Web-server seems to be very active in the network, a proxy might need to be installed to balance the load.

We choose to extend the query by joining the data stream with a table that consists of 65536 tuples; one for each available port. We do this to show the joining possibilities in a DSMS. Thus, prior to the experiments, we create the table *ports* that contains port numbers, as integers, from 0 to 65535, in both DSMSs. We only state the TelegraphCQ query here:

```
SELECT wtime(*), streams.iptcp.destPort, COUNT(*)
FROM streams.iptcp [RANGE BY '5 minutes' SLIDE BY '1 second'], ports
WHERE ports.port = streams.iptcp.destPort
GROUP BY streams.iptcp.destPort
```

`wtime(*)` is a TelegraphCQ specific function that returns the timestamp of the current window. The STREAM query is almost equal, except that it has to project the `destPort` tuples in a sub-query before the join. This is because STREAM only allows 20 attributes for aggregations, something which is statically defined in the code, and that we choose not to change in our experiments.

2.3 Task 3: Number of Bytes Exchanged for Each Connection During the Last m Seconds

When a connection between two machines is established, we are interested in identifying the number of bytes that are sent between them. In order to manage this, we have to identify a TCP connection

[1] `RANGE BY` describes the window range while `SLIDE BY` describes the update interval.

establishment, and sum up the number of bytes for each of these connections. As stated above, this task is only making a qualitative comparison for both query results, and we outline the ideas of how to model this query.

In TCP, connections are established using a 3-way handshake [19], and it is sufficient to identify an established connection with a tuple consisting of source and destination IP addresses and ports. For expressing this task, we first need to identify all three packets that play a role in the handshake. The most important attributes to investigate are the SYN and ACK control fields, as well as the sequence number and the acknowledgment number. To achieve this, we state two sub-queries. One selects TCP headers that have the SYN option field set, while the other selects the headers having the SYN *and* ACK option fields set. In a third query, we join these headers with headers having the ACK option field set as well as joining on matching sequence and acknowledgment numbers. In the 3-way handshake, timeouts are used in order to handle lost packets and re-transmissions. We model this using sliding windows. If a tuple slides out, a timeout has been violated. To identify the number of bytes that are exchanged on a connection, we have to join all arriving tuples with one of the connection tuples. This means that the connection tuples have to be stored for a limited period of time, for example in a window. We can also construct queries that, in a similar way as the ones described above, identify the connection tear down.

A sub-task would be to identify all the connection establishments that have not joined the final ACK packet, and therefore times out. In SYN-flood attacks this timeout can be exploited by a client by not sending the final ACK to the server after sending the initial SYN packet [12]. This behavior is important to identify, and is also an important task for IDSs. In this case, when we use windows to model timeouts, we want to identify the tuples that are deleted from the window. Here, STREAM has the possibility of defining a `DSTREAM`, i.e., a *delete-stream*, which satisfies this requirement.

What is interesting in this task, is that it needs to obtain information from all the packets to function correctly. For instance, sampling of only important tuples can not be used, because the three packets depend on each other in both control fields and sequence and acknowledgment numbers. Thus, the sampler becomes a query processor itself, something that only pushes the problem one level closer to the data stream.

3 Experiment Setup

In this section, we show the design of our experiment setup for our practical evaluation of DSMSs for network monitoring (see Figure 1).

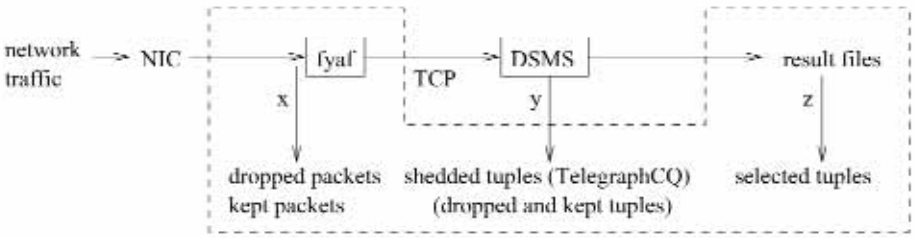


Fig. 1. The experiment setup.

We have developed a program called `fyaf`, which reads network packets from the network interface card (NIC). For packet capturing, `fyaf` uses the `pcap` library, which reports the number of kept and dropped packets after the capturing is completed. This is shown by the arrow denoted x in the figure. Kept packets are denoted x_k, while dropped packets are denoted x_d. Both TelegraphCQ and STREAM process data in a comma separated value (CSV) fashion, and `fyaf` is used to

transform the TCP/IP headers to the current DSMS's CSV stream scheme. `fyaf` also removes the payload from the packets, and sends the headers as tuples to the DSMS in the same frequency as they arrive. `fyaf` sends the tuples to the DSMS over a TCP socket, to guarantee that all tuples arrive. For this purpose, we have to change parts of STREAM so that it manages to accept socket requests. The advantage of using a local TCP socket is that possible delays in the DSMS affects the performance of `fyaf`, and thus, the packet capturing mechanism. In addition, TelegraphCQ reports the number y of kept and dropped tuples, denoted y_k and y_d respectively. In practice, TelegraphCQ's shedded tuples are reported as streams that can be queried as shadow queries of the task query [22]. The result tuples from the DSMS, z, are stored to file. An invariant is that systems that provide load shedding should have $x_d = 0$, while a system that does not support load shedding should have a varying number of dropped packets depending on queries and network load. A packet that is dropped from the NIC can not be turned into a tuple, so we consider that tuple as dropped.

The experiment setup consists of only two computers; A and B, which form their own small network. Each computer has two 3 GHz Intel Pentium 4 processors with 1 GB memory, and runs with Linux SuSE 9.2. The experiment setup and the DSMSs are located at node B. We have chosen to use two metrics; *relative throughput* and *accuracy*. We define these two metrics, as well as the term *network load*, to avoid misunderstandings.

Definition 1 Network load *is specified by the number of observed bits per second on a link.*

Definition 2 *The* relative throughput *of node N, denoted RT_N, is the relation between the network load node N receives and the network load it manages to handle. (In this paper, we use both terms "relative throughput" and "RT" for the relative throughput.)*

Definition 3 *The* accuracy *is measured as percentage of the computed result related to the correct result.*

4 Experiments and Results

In this section, we show the factors we use to verify that the experiment setup works and to run the tasks. After a practical verification, we show the results from Task 1 and Task 2. The experiments investigate the DSMSs' behavior at varying network load. The load is generated by the public domain traffic generator TG [16], which we set to send a constant rate of TCP packets with segment size of 576 bytes. We keep the rate constant to see how the DSMS behaves over time. This contradicts a real world scenario where load may change over a 24 hours period. However, we want to initiate the load shedding of TelegraphCQ, i.e., start load shedding instead of adapting to the stream, which is one of TelegraphCQ's strengths [2, 20, 15]. The traffic is generated over a time period called a *run*. Mainly, the network load is measured at 1, 2.5, 5, 7.5, and 10 Mbits/s, and for each network load, we perform 5 runs. This implies that `fyaf` has to compute data at a higher network load than the number of bits received on the NIC, to avoid becoming a bottleneck. We also assume that `fyaf` performs faster than the current DSMS since `fyaf` is less complex. This is verified by `fyaf`'s log files. All results are average values over the given number of runs. `fyaf` is set only to send the TCP/IP packets to the DSMS as tuples. Other packets, e.g. control packets in the network, are written to file for further analysis and identification.

The STREAM prototype does not support load shedding, and we use this system to investigate if it is sufficient to use the experiment setup for addressing the number of dropped tuples.

We define one query for this test, $Q1$ – `SELECT * FROM S`. $Q1$ simply projects all the tuples that are sent to the DSMS. This makes it easy to verify that the number of tuples sent from `fyaf` and number of tuples returned from $Q1$ are equal, i.e., $x_k = z$ in Figure 1. If this is the case, we can say that the relative throughput of STREAM is equal to the relation between dropped and kept packets from the NIC, i.e., $RT_{STREAM} = \frac{x_k}{x_d + x_k}$.

In TelegraphCQ, a separate wrapper process, the *wrapper clearing house* (WCH), is responsible for transforming the arriving tuples so that they are understood by the continuous query handling process; the *back end* (BE) process. The WCH aims to receive and obtain tuples from one or more external sources [13], which is a necessity for TelegraphCQ to handle real-time streams from several sources. The information about the shedded tuples can be accessed by using *shadow queries*. Thus, the accuracy metric can be used to evaluate TelegraphCQ's load shedding. TelegraphCQ provides several load shedding techniques [21, 22], but for simplicity, we only look at the number of dropped and kept tuples. By adding these two numbers together, we get a good estimate of the total number of tuples TelegraphCQ receives each second.

Since TelegraphCQ uses load shedding, we have the possibility of using the experiment setup to investigate the correctness of the load shedding functionality. The idea is simple: We know the number of tuples sent to TelegraphCQ, x_k, and the number it returns, z. The difference between those two results should be equal to the number of tuples dropped by the shedder, i.e., $y_d = |x_k - z|$. TelegraphCQ's relative throughput is denoted RT_{tcq}, which means that $RT_{tcq} = \frac{y_k}{y_d \mid y_k}$.

For investigating RT_{tcq}, we have chosen to use a query that is equal to $Q1$. By using such a query, we can investigate the correctness of the load shedding, since all tuples are selected:

```
SELECT <X>
FROM streams.iptcp
```

We vary the number of attributes, to see if this gives different results. Thus, in this query, `<X>` can be either "`*`", "`sourceip, destip, sourceport, destport`", or "`destip`". We can also observe how many tuples TelegraphCQ drops while the queries run. We execute the queries over streams with duration of 60 seconds and vary the network load.

Following are the results from these experiments. For STREAM, we investigate the result files from $Q1$, where we observe a small deviation from the expected results. We observe that $z < x_k$. `fyaf`'s log files reveal that ARP [18] packets compose a small part of the traffic. `fyaf` only sends TCP/IP headers to the DSMS and filters off other packets, like ARP. Since we use these statistics to evaluate the results from our experiments, we need to investigate the margin of error composed by the ARP packets. We calculate the margin of error by comparing the number of ARP packets to the total number of packets captured by `fyaf`. We observe that the margin of error is negligible, as it is always less than 0.025 %. Consequently, we consider this margin of error as insignificant. Thus, we conclude that the experiment setup can be used to measure the number of dropped packets.

After running the TelegraphCQ queries, we observe that the relative throughput decreases as the network load increases above 5 Mbits/s, and we observe - as expected - that the task projecting "`*`" has a lower relative throughput than the other two. At 20 Mbits/s we observe that the relative throughput is down to 0.1 for all three queries. Up to 20 Mbits/s, we see that $\frac{x_k}{x_d + x_k} = 0.999$. On higher network loads, the experiment setup starts to drop a significant amount of tuples. Consequentially, we have a maximum of 20 Mbits/s to run the TelegraphCQ queries on. The remaining 0.001 of dropped tuples from the experiment setup are interesting and will be subject for future work.

Following, we discuss the correctness of TelegraphCQ's load shedding functionality. We set RT_{es} to be the relative throughput that is expected from TelegraphCQ based on the data we obtain from the experiment setup, i.e., $RT_{es} = \frac{z}{x_k}$. For correct results, we have that $RT_{es} = RT_{tcq}$. The results from running $Q1$ shows that TelegraphCQ seems to report a higher relative throughput than what seems to be correct, which means that $RT_{tcq} > RT_{es}$. In Figure 2, we show the relative error by calculating $\frac{RT_{es}}{RT_{tcq}}$. What is interesting, is that the relative error does not decrease as the network load increases. Instead, we have a minimum at 10 Mbits/s before all three query results seem to converge to 1. At the current time, we can not explain this behavior, as all known sources of errors in our experiment setup have been investigated. To investigate this even further is topic of ongoing work. Thus, we see that TelegraphCQ seems to report a more optimistic load shedding than what is true when it comes to tuple dropping. Based on these results, we conclude that the experiment setup can be used for reporting dropped packets for DSMSs that do not have this functionality

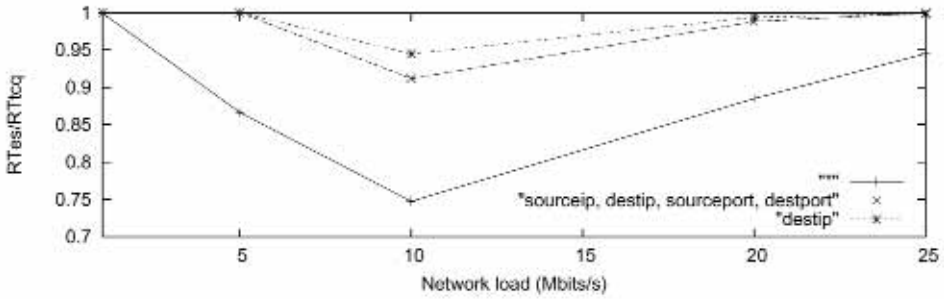


Fig. 2. Relative error in RT_{tcq}.

implemented. Since this is possible, we use this experiment setup to investigate the correctness of the reported load shedding, and we assume that TelegraphCQ does not report a correct number.

4.1 Task 1

For Task 1, we continue investigating TelegraphCQ's accuracy in the query results. For Task 2, we show the difference between STREAM and TelegraphCQ, as well as discussing the behavior of TelegraphCQ in a one hour run. For Task 1 and Task 2, each run lasts for 15 minutes, and the upper limit for the network is 10 Mbits/s.

Based on the findings from the prior experiments, we wish to continue investigating TelegraphCQ's accuracy. Instead of using RT_{cs}, as we did above, we now use the real network load for correcting. We have verified that TG's network load is correct.

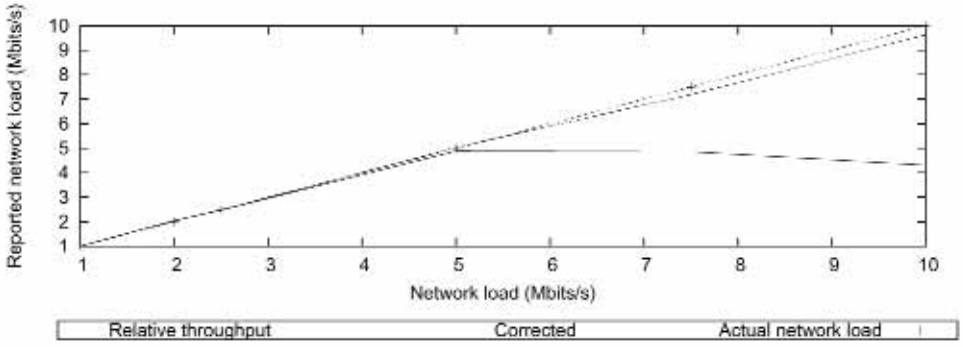


Fig. 3. Relative throughput as reported and corrected.

As Task 1 aims to measure the number of packets and network load, it is easy to estimate the correctness of the results. Figure 3 shows the correct network load as the cross dashed diagonal line. The relative throughput is decreasing just after 2 Mbits/s and is reduced considerably when the network load is 10 Mbits/s. We use the information from the query to correct the output by adding the shedded tuples and multiply with the average packet size. We see that after 5 Mbits/s, the corrected results start to deviate from the actual network load. At 10 Mbits/s, the correctness is 95.1 %. This is expected and coincide with what we observed in the prior experiment. Since the maximum network load is 10 Mbits/s, we do not know how accurate these results are at higher speeds, but we should observe that the deviation is reduced as the network load increases.

4.2 Task 2

For Task 2, we start by comparing the relative throughput from the two DSMSs. RT_{tcq} is computed by calculating the shedded tuples from TelegraphCQ, while RT_{STREAM} is computed by counting the number of dropped packets from the NIC. Since one packet is equal to one tuple, and that we have shown that it is possible to use our experiment setup for this purpose, it is interesting to look at the differences between the two systems. Figure 4 shows the relative throughput for the two systems.

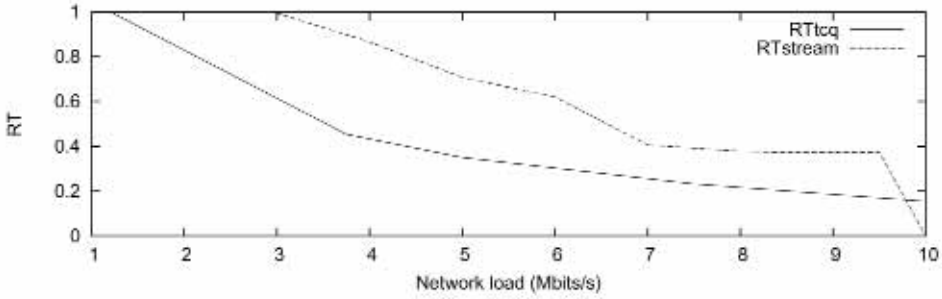


Fig. 4. A comparison between TelegraphCQ and STREAM.

We see that STREAM manages to compute more tuples than TelegraphCQ does, but that both systems have a significantly lower relative throughput when the network load increases to 10 Mbits/s. In our experiments, STREAM did not have any correct runs at 10 Mbits/s, thus, the relative throughput is set to 0. An alternative way of comparing between the two systems, is to turn off TelegraphCQ's load shedding, something that is possible, but we have chosen not to do this in our experiments.

Because of space limitation, we do not discuss the comparison between the two systems any deeper. Instead, we focus on looking at the relative throughput and results from TelegraphCQ's in Task 2.

One interesting issue with investigating streams of data, is to see how the systems behave over a time interval. In Figure 5, we observe TelegraphCQ's average relative throughput over 5 Mbits/s runs that lasted for one hour. The upper graph shows the plot of the relative throughput, while the lower graph shows a plot of the number of packets that are destined for the open port at machine B. We see that the output does not start until after the window is filled. What is interesting to see is that the relative throughput drops significantly after the first five minutes. This also implicate that TelegraphCQ waits until the first window is filled up before starting the aggregations and outputs. Note that five minutes of TCP/IP headers is much data to compute. We also see that the relative throughput is only approximately 0.8 to begin with, meaning that TelegraphCQ already is under overload before starting the calculations.

A consequence of the sudden drop at five minutes is that the result starts to decrease dramatically. For example, after approximately ten minutes of low relative throughput, the number of result tuples reaches zero. But after one hour, we see that the two processes cooperate well, and the plot shows a behavior that looks like a harmonic reduction. One possible solution is that we observe that since the BE has few tuples to process, the WCH is allowed to drop fewer tuples, something which results in more tuples to process, and thus, lower relative throughput. This mutual fight for resources converges to a final relative throughput. As future work, we will look at the possibilities of even longer runs, to see if this equilibrium is stable.

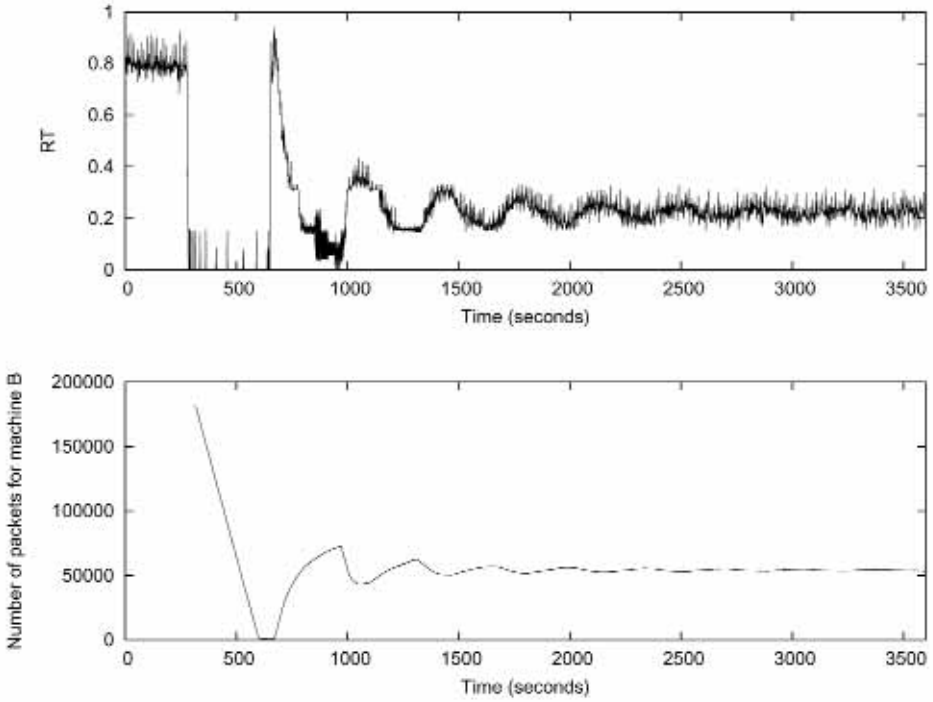


Fig. 5. The relative throughput and number of packets from a query lasting for an hour.

5 Conclusions

In this paper, we perform a practical evaluation of load shedding in DSMSs for network monitoring. We argue that DSMSs can be used for this application by referring to related work, as well as stating a set of network monitoring tasks and suggesting queries that solve these tasks. We also investigate and outline a solution for a more complex task, which aims for identifying TCP connection establishments.

For the practical evaluation, we suggest a simple experiment setup that can be used to add tuple dropping in DSMSs that do not support load shedding, as well as evaluating the tuple dropping accuracy in DSMSs that support this functionality. We also suggest a metric; *relative throughput*, which indicates the relation between the number of bits a node receives and number of bits the node is able to compute. By running simple projection tasks on two DSMSs - STREAM and TelegraphCQ - we verify that the experiment setup works as intended. We also observe that TelegraphCQ seems to report higher relative throughput than what is actually reached. Then, we evaluate the two first network monitoring tasks, and show TelegraphCQ's correctness and a comparison between STREAM and TelegraphCQ's relative throughput. Finally, we discuss the results from an one-hour run of TelegraphCQ.

Our conclusion is that load shedding is a very important functionality in DSMSs, as the network load often is higher than what the system manages to handle. We also conclude that network monitoring is an important application for DSMSs and that our experiment setup is useful for evaluating the DSMSs in network monitoring tasks. We argue for this by showing some results from our experiments. We see that our simple experiment setup can be used to investigate the correctness of tuple dropping in DSMSs.

Since network monitoring is a promising application for DSMSs, our future work is to investigate the expressiveness of the Borealis stream processing engine as a network monitoring tool, as well as evaluating its performance in our experiment setup. We will also extend our experiment setup to evaluate the correctness of load shedding techniques such as *sampling* and *wavelets*.

References

1. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Department of Computer Science, Stanford University*, 2004.
2. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.
3. S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic management. In *Proceedings of the 2001 Workshop on Network-Related Data Management (NRDM 2001)*, May 2001.
4. S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.
5. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. *Proceedings of the 2003 CIDR Conference*, 2003.
6. C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spataschek. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data / Principles of Database Systems*, June 2002.
7. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM Press.
8. N. Duffield. Sampling for passive internet measurement: A review. *Statistical Science*, 19(3):472–498, 2004.
9. N. Duffield, C. Lund, and M. Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions in Information Theory*, 51(5):1756–1775, 2005.
10. L. Golab and M. T. zsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
11. P. Huang, A. Feldmann, and W. Willinger. A non-instrusive, wavelet-based approach to detecting network performance problems. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 213 227, New York, NY, USA, 2001. ACM Press.
12. T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. In *Proceeding of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2005)*, August 2005.
13. S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: An architectual status report. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2003.
14. W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1 15, 1994.
15. S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. *ACM SIGMOND*, June 2002.
16. P. E. McKenney, D. Y. Lee, and B. A. Denny. *Traffic generator release notes*, 2002.
17. T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, and E. W. Biersack. Using data stream management systems for traffic analysis - a case study. April 2004.
18. D. Plummer. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware. RFC 826 (Standard), November 1982.
19. J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.
20. V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. *Report No. UCB/CSD-03-1231*, February 2003.
21. F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. Technical report, February 2004.
22. F. Reiss and J. M. Hellerstein. Declarative network monitoring with an underprovisioned query processor. In *The 22nd International Conference on Data Engineering*, April 2006.
23. A. Soule, K. Salamatia, N. Taft, R. Emilion, and K. Papagiannaki. Flow classification by histograms: or how to go on safari in the internet. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 49–60, New York, NY, USA, 2004. ACM Press.

UNIVERSITY OF OSLO

A Practical Evaluation of Load Shedding in Data Stream Management Systems for Network Monitoring

Jarle Søberg, Kjetil Hernes, Matti Siekkinen,
Vera Goebel, Thomas Plagemann
University of Oslo, Department of Informatics
P.O. Box 1080, Blindern, N-0316 Oslo
{jarleso, kjetih, siekkine, goebel, plageman}@ifi.uio.no

- Outline
  1. Overview
  2. Motivation
  3. Experiment setup
  4. Tasks and results for TelegraphCQ
  5. Conclusion and future work

UNIVERSITY OF OSLO

Overview

- TelegraphCQ as a network monitoring tool
- Investigate load shedding in TelegraphCQ
  - "Black box"-testing
  - Two tasks
    - Design
    - Results

UNIVERSITY OF OSLO

Motivation: Passive network monitoring with DSMSs

- Capture data packets from the network
  - No insertion of own packets, just listen!
  - Obtain information from the captured data
- Send packets as *tuples* to the DSMS
  - A data packet can be viewed as a *tuple*
  - The packet's header fields can be viewed as *attributes*

- Real-time
  - Intrusion detection
  - Traffic engineering

UNIVERSITY OF OSLO

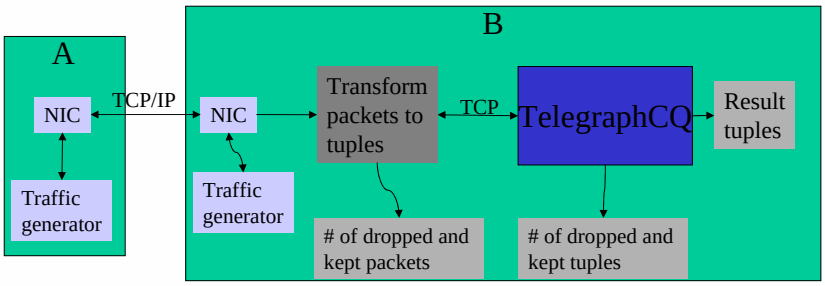
Motivation: Passive network monitoring and load shedding

- Reduces number of packets to be processed
- Network characteristics
  1. A considerable amount of traffic data
     - Gbit/s backbones used by ISPs
  2. Peak periods
     - Low activity at night
  3. Non-interesting data
- Several techniques
  - Simplest: Packet dropping

UNIVERSITY OF OSLO

Experiment setup

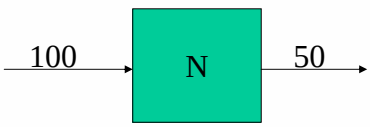
- How does the tuple dropping start?
  - Stress the DSMS
- Generate TCP/IP traffic between two machines
  - Linux SuSE 9.2 (vanilla installation)
  - Two 3 GHz Intel Pentium 4 processors
  - 1 GB of memory



UNIVERSITY OF OSLO

Experiment setup: definitions

1. *Network load* is specified by the number of observed bits per second on a link.
2. The *relative throughput* of node N, denoted RT_N, is the relation between the network load node N receives and the network load it manages to handle.

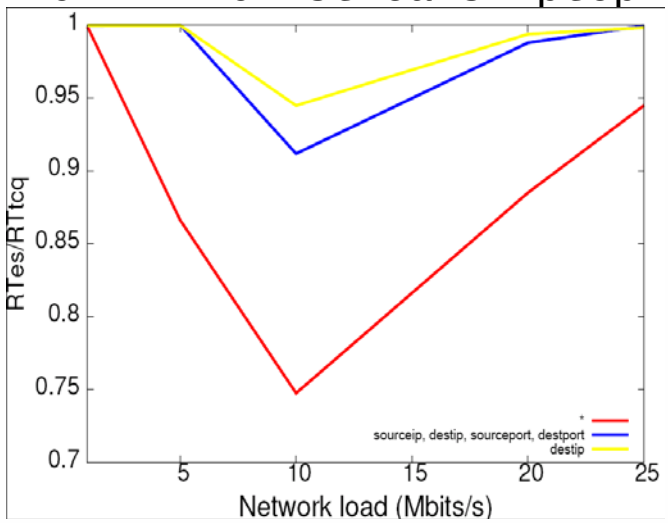


$RT_N = 0.5$

UNIVERSITY OF OSLO

TCQ load shedder correctness:

`SELECT X FROM streams.iptcp`



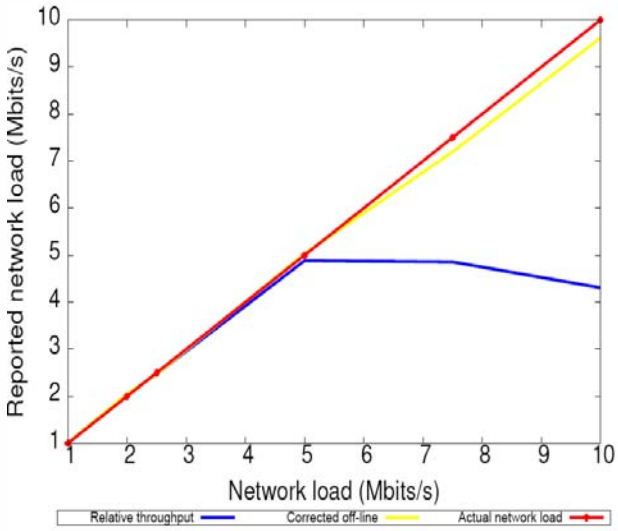
UNIVERSITY OF OSLO

Average Load of Packets and Network Load

```
SELECT
  COUNT(*)/60, AVG(s.totalLength)*8
FROM
  streams.iptcp s [RANGE BY '1 minute'
  SLIDE BY '1 second']
```

UNIVERSITY OF OSLO

Average # of Packets and Network Load



UNIVERSITY OF OSLO

Conclusion and future work

- TelegraphCQ uses a simple and declarative query language
- TelegraphCQ only manages low network loads
  - Starts dropping tuples at 5 Mbits/s
- TelegraphCQ seems to report higher relative throughput than what is actually reached

- Currently, we run tasks on the Borealis stream processing engine
- We will extend the packet-to-tuple translator with more sophisticated load shedding techniques

- Questions?