Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Threshold Analysis for Low-overhead Control of Load Shedding Mechanisms

Viktor Markopoulos

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	M. Sc. Henriette Röger
Commenced:	March 22, 2021
Completed:	September 22, 2021

Abstract

Complex event processing is a concept and a group of technologies that realize real-time event processing. An input data stream of events are being processed by a CEP system in order to extract patterns and information about the data source. The volume of this data stream can be enormous and the need for a mechanism to ease the CEP system's workload is needed. For this, a load-shedder mechanism is deployed on the system. But in order for the load-shedder to have a clear insight of the CEP system, a monitoring mechanism must be applied, to feed the load shedder with the information needed. But this monitoring solution must not impose additional overhead to the already overloaded system. In this thesis, multiple monitoring strategies are being assessed, by carrying out experiments using a monitoring mechanism with different configurations for sampling, deviation threshold, and scope and slide time frames for sampling, with the help of two projects, i.e. CoDaEFC and Precept. The goal of these experiments is to analyze the monitoring solutions' approach. A mechanism's goal should be to give insight to the system, yet without adding too much workload on the system. The result metrics of the experiments are then being logged and presented in a visual form. Conclusions are being drawn, based on the results of the experiments and how these configurations affect the CEP system's performance.

Contents

List of Figures

1 Introduction

With the rise of IoT, there has been a need for efficient process of the resulting data streams. IoT devices are sensors such as temperature sensors, smart fridges, smart locks, doorbells devices etc. All are event-based systems connected to a network, often the Internet, producing a data stream of events caused by the physical world, e.g. rise of temperature, opening the smart fridge, hitting a smart doorbell etc. These data streams are often valuable to companies for analysis and getting an insight on the devices' usage and behavior. The analysis of a data stream may also be a core part of a device's function. E.g. a temperature sensor that automatically enables the air condition once the temperature is high, should be able to recognize the data stream of continuously rising levels of heat. The aggregation, filtering and analysis of those data streams is realized by the concept of Complex Event Processing (CEP).

CEP is a set of methods and techniques which aim to efficiently process data streams and recognize patterns and interesting situations [FGD+17]. In a distributed CEP there are components called operators. Operators are running within machines described as nodes, which are part of a graph. These operators run the logic behind the processing of a data stream, therefore it is crucial for them to have a noise-free process. The cause of noise ranges from faulty hardware of the nodes, to poorly written code, and to the occasional high volume of input.

An operator can get throttled from an enormous size of input data. One solution for this is to implement a load shedding mechanism, which will decrease the overload of an operator by "dropping" some of the input data stream packets. This mechanism can be implemented based on monitoring data of the operators, which are "fed" to the load-shedder and it acts accordingly, by dropping packet and aiding bottleneck nodes. That way, the normal flow of the network is ensured.

Nevertheless, the monitoring of an operator adds an additional load to its node's performance. It is important to add as little additional overload to a node as possible, so that this overload will not interfere with the operator's processing of the input data stream. Yet, insufficient and poor monitoring performance will result to poor monitoring data, thus affecting the performance of the load-shedder. Therefore it is vital to find a monitoring strategy that conforms the need to accurately shed the input of an operator, while at the same time doesn't take much of the resources the operator needs in order for it to process the data stream.

Structure

The thesis is structured as follows:

Chapter 2 – Basic definitions: In chapter 2 we explain some basic definitions of terms that are used in this thesis, such as CEP and monitoring.

Chapter 3 – Related Work and Problem Statement: We give some insight on the state of the art approaches on monitoring and load-shedding distributed stream processing applications, along with some tools and technologies that are used today for distributed event streaming.

Chapter 4 – Finding the optimal monitoring configuration: In chapter 4 we test out our infrastructure on various monitoring configurations.

Chapter 5 – Evaluation: After experimenting, we lay out the output data and evaluate the course and the results of the experiments.

Chapter 6 – Conclusion: In the last chapter, we conclude.

2 Basic definitions

2.1 CEP

Complex Event Processing is a concept, a computing paradigm, based on events and the event-driven architecture. An event is the result of an action, which causes a state change. It can be the rise of temperature with its action being an outbreak of a fire, or simpler a click of a mouse. In more complex examples, an event can also be a database entry.

In event-driven architecture, an event will trigger a message containing the name of the event and an event body and spread in the components that are so configured to receive such events. According to [Mic06], the event-driven architecture is very loosely coupled, i.e. the source of the events do not have any knowledge of the event's future processing.

There are three types of event processing, namely

- Simple event processing
- Stream event processing
- Complex event processing

In simple event processing, immediate action is taken upon the arrival of an event, as it requires no processing of the event itself. An example would be the change of a constant value, which would result to a fixed, standard action. Or a standard action that leaves little-to-no room for processing, like the order of a product from an e-shop.

Stream event processing includes simple event processing. An incoming event will first be processed and based on the result of this process it can activate a series of downstream actions to respond to it. In Figure 2.1 there is a general illustrative example. As a more specific example, we can think of a CPU temperature censor. The sensor transmits the temperature levels to an event processing component, which filters, i.e. processes, the incoming events. Once the temperature level threshold is exceeded, the component is sending a signal for the fans to work.

Complex event processing includes stream event processing and is the in-depth analysis and processing of a stream of events, in order to extrapolate information, which are not distinguishable when looking at each event individually. This information can "discover" more "complex" events, which can later be the source of other processes. The CEP can be distinguished in two modes, i.e. the online and the offline mode. In the online mode, a data stream of events is being consumed by the system, without it having any prior

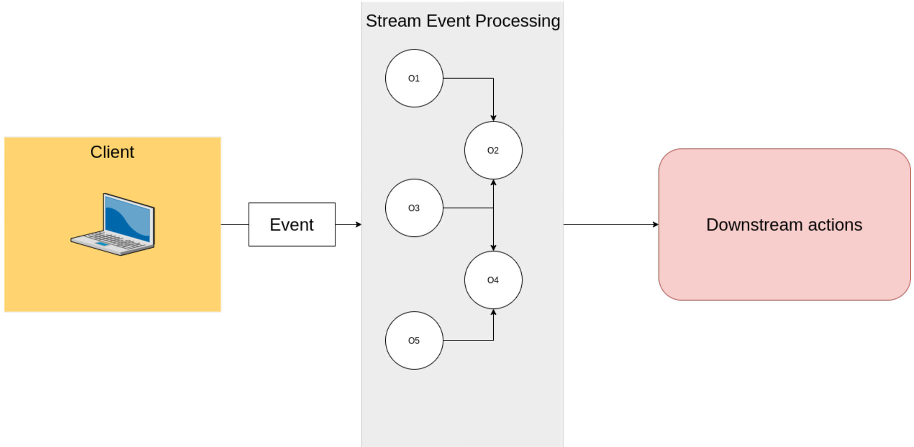

Figure 2.1: Stream event processing

knowledge of the stream itself. In the offline mode the CEP system has complete knowledge of the input stream [Hal17]. The CEP and stream event processing terms are often used interchangeably, although the stream event processing doesn't involve multiple data sources and much complex processing, like CEP.

2.2 Distributed application

Like all of event-driven applications, CEP applications are heavily associated with graphs, as they are most usually distributed among nodes of a graph topology. A distributed application is deployed on a graph, distributing the parts of its logic into smaller parts to achieve scalability and faster calculations. For an application to achieve that, a directed acyclic graph topology is used.

A directed acyclic graph is the tuple $G = (V, E)$, where

- V is the set of vertices, or else called nodes, and
- E is the set of edges, which are actually pair of vertices, defined as

$$E \subseteq \{(x, y) \mid (x, y) \in V \; and \; x \neq y\}$$

In the notation (x, y) x is the source of the edge and y the target, since the graph is directed. In Figure 2.2 there is an example of a directed acyclic graph. Acyclic means that there is no way starting from a node x and by following the edges down all possible paths, to find the same starting node x in any of those paths again.

A CEP system is consisted of channels and operators. The operators are the components that process the stream of events, i.e. the actual logic behind the application, and can be deployed on the graph's node. Note that a node may have more than one operators deployed. The event data stream channels that flow between the nodes are the graph's edges.

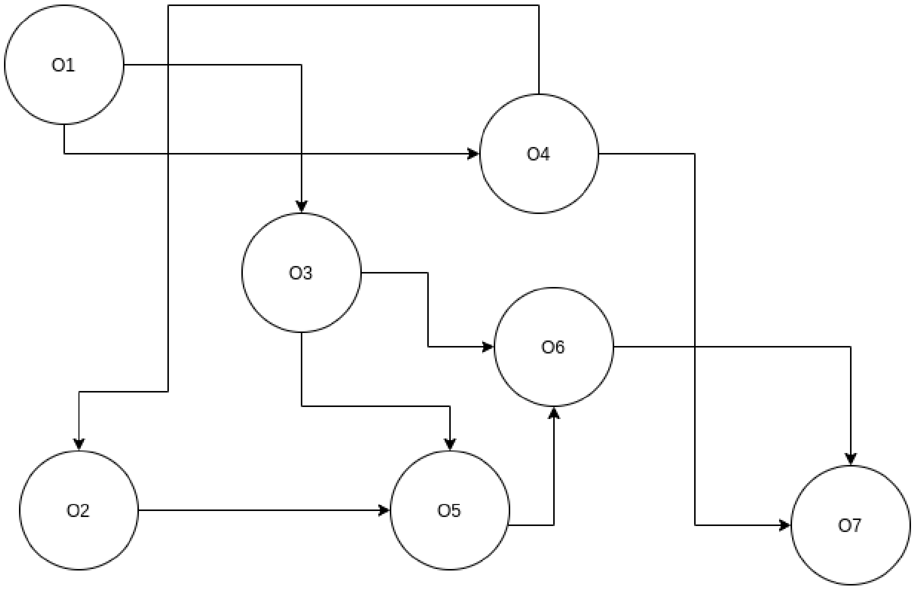

Figure 2.2: Directed acyclic graph

2.3 Monitoring and Load Shedding

The term monitoring is revolved around the observation and aggregation of a system's behavior metrics, while performing its usual tasks. In the concept of CEP monitoring is focusing on the performance of the operators, i.e. how they respond to the input data stream, how long it takes them to process it etc.

A load shedder is a mechanism that helps the CEP system to respond to a volume of input data that surpasses the capabilities of its operators. The load shedding usually takes place by dropping incoming events, and easing the CEP system's workload.

3 Related Work and Problem Statement

The concept of CEP has been around since the nineties. According to [Lea09], Rapide, a project created at Stanford University in 1995 to monitor hardware events to find errors and weaknesses, is the first project that coined the term "complex event processing". Over the years the topic of CEP has been extensively researched, which lead to a number of scientific papers, as well as tools and frameworks for realizing the concept of CEP. Apache is a company that has released multiple frameworks for realizing CEP. E.g. Apache Flink [Fou21a] is a framework designed to process both bounded and unbounded data streams. Bounded streams are data streams with defined beginning and end, whereas unbounded streams have no defined end, thus events must be processed and injected at real time.

3.1 Monitoring

A part of CEP which has been the center of research is its performance monitoring. In [VSI+16], the authors attempt to find a cloud-based solution. According to them, a monitoring framework must have distributed operations and be able to handle the distributed nature of the application and must be non-intrusive, i.e. the monitoring solution must not make code-level changes. It also has to be independent from the application itself, as well as acquire, publish, manage and analyze the monitoring data.

In comparison to [VSI+16], in [FTG+14], the authors use an solution integrated into the application's code, instead of having a monitoring overlay. They heavily use the subscribers and publishers output rate and decide on the monitoring overhead based on the precision or imprecision of the metrics.

Yuen et al. in [YC12] follow this distributed nature of monitoring by utilizing a peer-to-peer approach, where client sensors, who produce the input, communicate with inner nodes called proxies and are responsible for aggregating and distributing metrics data to the roots of the graph, i.e. the nodes responsible for monitoring. Metrics travel through that network to reach one of the multiple monitors. The network also updates itself and looks for better performance while working at the same time.

In [LIH+12], the authors follow the CEP paradigm for monitoring cloud services, by using mostly time-based monitoring metrics such as the average number of new data items processed per second, making the monitoring of a CEP a universal concept that can be applied on other architectures as well.

3.2 Load Shedding

The monitoring data produced must be passed to the load-shedder, in order for it to have insight of the CEP system's work. Thus a proper load-shedding mechanism should be implemented. In [SBR19] the authors build a load-shedding mechanism that uses a probabilistic model, based on event windows. They raise important questions regarding load-shedding solutions, e.g. what events should the load-shedder drop, how many events should be dropped, and what is an efficient way to drop them.

Tatbul et al. in [TÇZ07] propose two approaches, one centralized solver-based solution and a distributed approach. Both of the approaches are based on pre-planned solutions, which is later decided were and when to be used. The distributed solution consists of four phases, i.e. advance planning, where a shedding solution is being planed, load monitoring, where the monitoring of the CEP system takes place, plan selection, to select the shedding solution based on the monitoring data and the plan implementation, where the solution itself is being implemented. When the approach is centralized, the first three phases are performed on the central server, and a coordinator server is applying the remaining step. In the distributed approach, all steps take place on all of the nodes in a coordinated fashion.

3.3 Problem Statement

It is important for a CEP system to be able to work without any interruptions and at full potential, especially when it comes to critical infrastructure. In order to prevent such hindrances, a good load-shedding solution must be applied.

For the load-shedding mechanism to work and to be efficient, it has to aggregate system metrics and monitoring data, in order to have proper insight of the system's workload. This data feed coming from the monitoring mechanism should reveal potential bottleneck nodes with excessive workload. This should give important information the load-shedder's mechanism, which should later decide on to which or how many events should be dropped.

Nevertheless, a monitoring mechanism can impose additional overhead to an already overloaded operator of a CEP system. Especially in geographicaly distributed nodes, the operator's performance and network latency must be kept at a lower possible level. And the monitoring solution's goal is to feed its aggregated metrics to the load-shedder, to ease the workload. Thus the questions are raised:

What is the trade-off between close monitoring and the performance overhead imposed by it? What is the sweet spot of sufficient monitoring data and additional overhead?

A proper monitoring solution should

- Impose the least possible additional overhead to the operators
- Give the most possible insightful metrics to the load-shedder

To analyze possible solutions to this problem, we conduct multiple experiments to assess a monitoring strategy, based on a fixed number of configurations.

4 Finding the optimal monitoring configuration

4.1 Environment

For realizing the topology of this research, two projects were used, i.e. CoDaEFC and Precept [Stu21]. As mentioned, our environment is a distributed network of CEP operators and a load-shedder. The CoDaEFC project is responsible for the source of the input data stream, the data sink and the load-shedder. A part of the Precept project is realizing the distributed network of the operators, as well as the logic behind them, i.e. the application itself. The nodes of the graph receive the input from CoDaEFC's source, they process it, and send it to CoDaEFC's sink. In Figure 4.1 you see a graphical representation of how the two projects interact with each other.

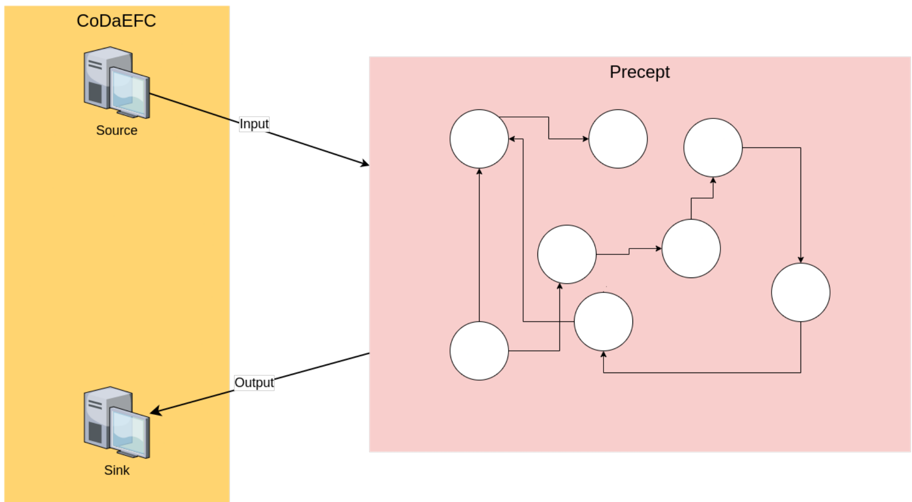

Figure 4.1: CoDaEFC and Precept

CoDaEFC is using the open source project Apache Kafka [Fou21b]. Kafka is a distributed event streaming platform, using the paradigm of producers and consumers. A Kafka producer is used on the operators to publish their data and a metrics consumer is used to consume and aggregate the metrics.

For the experiment's topology, one node was used with one operator deployed on it, as seen in Figure 4.2. The input data stream is a series of 100.000 events of 0's and 1's. The operator's work is to search for a specific pattern of these data, i.e. the pattern "001".

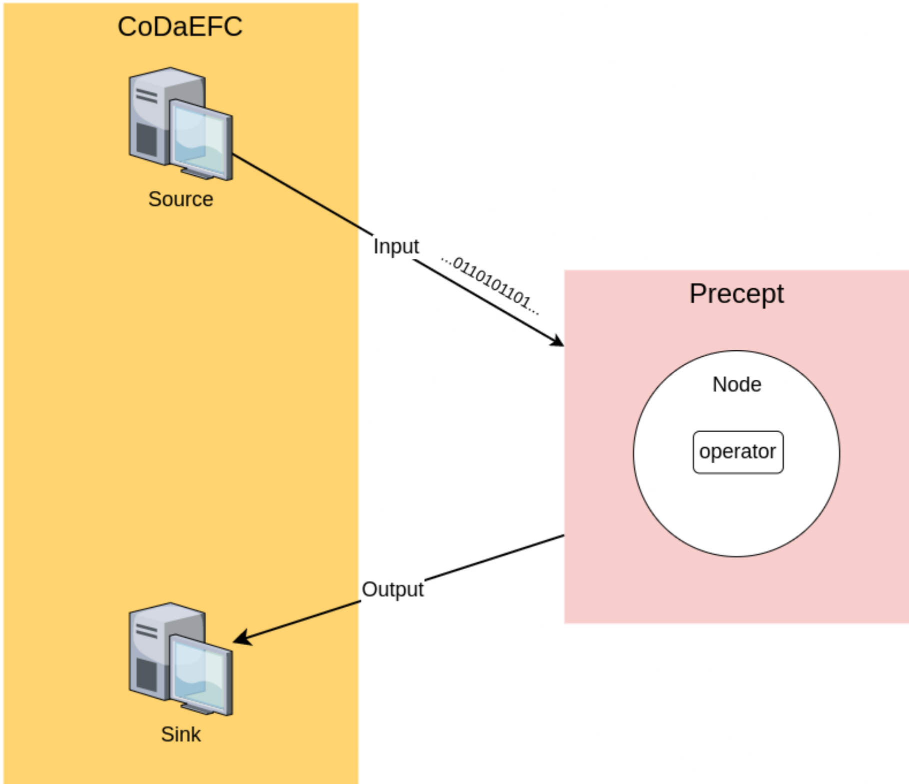

Figure 4.2: The experiment's topology

In order to have more reliable results, an environment as realistic as possible must be created. A real production environment is far more unpredictable than a "sanitized", lab environment. It can have delays and hindrances, occurred by different sources and for different reasons, e.g. a delay because of a malfunctioning hardware or a sudden burst of input data.

In need for such an environment, two layers of delay were added to simulate the conditions of a production system. The first layer is cycling through the values of $0.03, 0.015, 0.045$, and 0.035 seconds. This change from one delay to another is taking place every 1000 events. Figure 4.3 shows the graph of the first layer of delay imposed to the operator.

With the first cycle of delays, some realism is achieved. Nevertheless, we need to add a random variable to the equation to simulate the unpredictability of a production system. In addition to the first delay layer, in order to add the feeling of randomness, a small random delay of up to 0.005 seconds was imposed on every event. Figure 4.4 shows the fluctuation the second layer achieves. Note that this plot is not static, i.e. the fluctuation seen here is not the same for every experiment but depends on the randomness of the second delay.

With these additions, we attempt to simulate the variable of unpredictability of a real life CEP production environment.

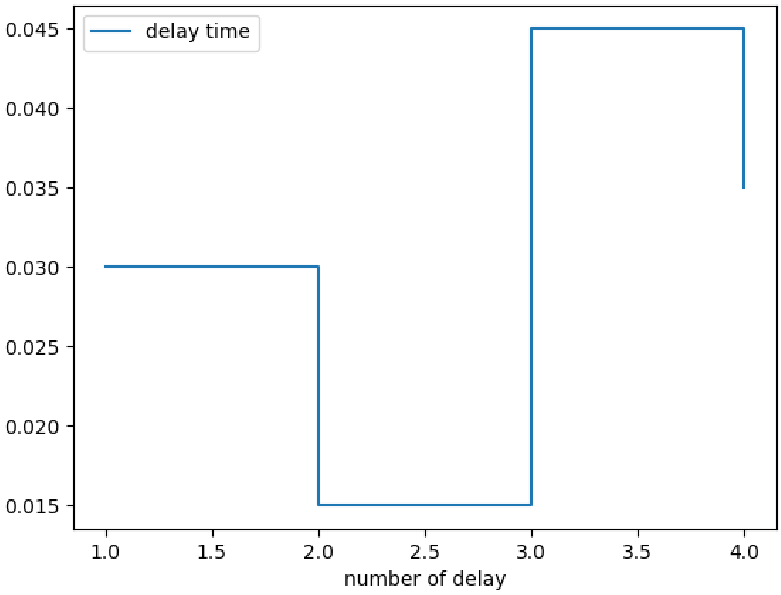

Figure 4.3: Result of the first layer of delay

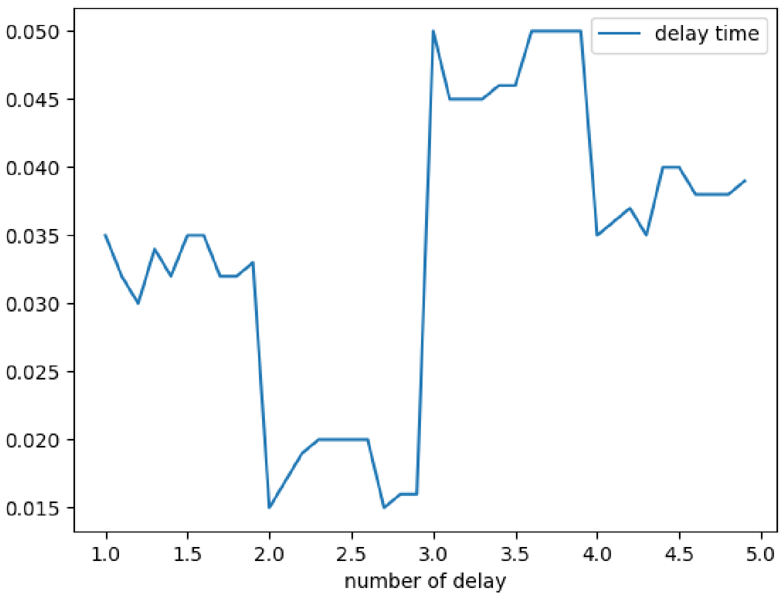

Figure 4.4: Result of the second layer of delay

4.2 Configurations

Our goal is to determine how well we can monitor an operator by logging data and drawing conclusions out of the monitoring results by having the least possible overhead imposed by the monitoring. In order for us to find out, we have to conclude multiple experiments with different logging configurations and compare the results at the end.

There were three kinds of configurations used, namely

- the update step i.e. how many events do we log,
- the deviation threshold of the experiment, and
- the metrics slide and metrics scope.

These three points are the configurations, which we set accordingly for each experiment. All of the configurations revolve around the processing time metric, since our goal is to give insights on the overload of the operator, which affects the processing time.

4.2.1 Update Step

In order to pick different update steps, we need to come up with a sampling strategy of the messages sent to the operator. A solution to this problem is to find a proper strategy, where the minimum sample size will provide the most insightful output about the operator.

Type of sampling

When we are talking about sampling methods there is a distinction between probability sampling and non-probability sampling. The first sampling method relies on probabilistic calculations, whereas the latter relies on the researcher's experience and heuristics.

Probability Sampling:

With this method, every element of the population has a chance to get picked as a sample based on random selection. These methods have a greater chance to end in a generalized outcome of the whole population.

Probability sampling is divided into four categories, namely simple random sampling, systematic sampling, stratified sampling, and cluster sampling.

Simple random sampling: It is the most simplistic method of sampling. The sample is chosen based on a random selection of nodes from the network. This approach is achieving complete randomness of the node sample, thus a more generalized overview of the network. While this reduces the selection bias, the randomly selected nodes can end up not fulfilling certain sample characteristics that should exist, e.g. not choosing different kinds of operators.

Systematic sampling: In systematic sampling, the researcher sets a random selection process based on a formula, e.g. select a node every 3 nodes. After putting the node population in a certain order so that the process can enumerate it, we let the process choose the sample. In this method, the researcher gains a little control over the selection method by creating a process that will fulfill some prerequisites, e.g. select 2 nodes from every clique of a graph.

Stratified Sampling: With stratified sampling, the node population is divided into subgroups with similar characteristics, e.g. group of nodes with the same workflow or with a similar topology like having two exit nodes. These characteristics as well as the number of subgroups are subjective to the research's nature and purpose. After dividing the population, a sample is chosen from each group. Ideally, the size of each sample is proportional to the size of the subgroup it's selected from. This method is reducing the sampling bias but it requires more specific knowledge of the population and its characteristics.

Cluster Sampling: The researcher can combine more than one method. Here the researcher creates node subgroups like in stratified sampling but the subgroups must share similar characteristics with the whole sample, i.e. the cluster should ideally be a miniature of the network. Ideally, if the cluster sizes are small and practically possible, the researcher can select whole clusters as samples, which would lead to more reliable results. If not, then they can pick certain elements from the clusters by applying any of the already mentioned sampling methods or even any of the non-probability sampling methods.

Non-probability sampling:

As the title suggests, non-probability sampling methods are considered those that do not apply any mathematical or probabilistic knowledge on selecting a sample rather the researcher selects a sample based in more of a manual manner. Non-probability sampling methods are more time- and cost-efficient than probability sampling methods, as they do not process all possible candidates from a population. But due to this lack of randomness, it tends to produce less generalized and representative results.

Non-probability sampling methods are convenience sampling, purposive sampling, quota sampling, voluntary response sampling, and snowball sampling. Due to the nature of the research, only convenience, purposive, and quota sampling is worth mentioning.

Convenience Sampling: This is the "lazy" sampling, meaning that the researcher selects their sample based on costs, accessibility, or any other parameter that makes it "easier" for them. While it doesn't apply any randomness to the selection, it is much more cost-efficient than probability sampling.

Purposive Sampling: In purposive sampling, the selection is solely based on the researchers' expertise on the matter of research. They handpick the sample, based on personal estimations. In our case, the researcher can possibly choose nodes that are in a critical place of the network.

Quota Sampling: Similar to stratified sampling, quota sampling is grouping the population into subgroups with the same characteristics and choose the sample in a similar, proportional manner to stratified sampling. The difference is that in quota sampling, the selection of the sample is non-random, i.e. the researcher is manually choosing the sample based on other criteria, like the convenience and low costs.

Because of the nature of the experiment and population, i.e. a data stream of incoming events, we have to rule out the non-probabilistic approaches, especially the ones were the free will is given to the population itself, i.e. the voluntary response sampling. And because we don't have access to the whole population at once, we can only follow the systematic approach of sampling.

Sample Size

After deciding the method of sampling it is important to set the sample size. The most common method used in deciding a sample size is Cochran's formula. This formula is based on some quality parameters that the researcher has to decide before using it. The researcher has to consider three key parameters when deciding on the size, namely the confidence level, the margin of error, and the target proportion.

The confidence level describes the level of certainty that our sample is accurately representing the whole population. The most common confidence level used is 95%.

Also, a typical value for margin of error is 5%. This value represents the accepted error difference between the sample and the actual outcome of the research if we had used the whole population as a sample.

Finally, if we are looking for a certain attribute in the data input, target proportion is the percentage that our sample has the same attribute as the whole population. When in doubt or there is no attribute in question then this value is set to 50%, which is the worst-case scenario.

After deciding these three values, Cochran's formulas can be applied:

The first formula is the ideal sample size we are looking for.

Z-score is, in simple terms, an estimation of how far from the mean a value is. The score can either be looked up in a z-score table or calculated. Example in Python:

```python
import scipy.stats as st
st.norm.ppf(1-(1-cl)/2)
```

$$n_0 = \frac{Z^2 p(1-p)}{e^2}$$

$$n = \frac{n_0}{1 + \frac{n_0 - 1}{N}}$$

Figure 4.5: Cohran's Formulas

The second formula adjusts our ideal sample to the size of our population. It is used for a smaller population, where the size is limited.

According to Cochran's formula, the ideal sample for a population of 100.000 events would be 383. This means that we have to log a total of 383 events.

$$update\_step = 100000/383$$

By dividing 100.000 to 383 we get 261. At a step of 261 events, we'll log an event and update the average processing time. In the end, we will have logged approximately 383 evenly distributed events, which follows Cochran's formula indication.

Before starting the experiment we set a variable $update\_step$ to an integer n. This variable instructs the code to log and update the processing average time of the experiment every nth event. By changing this configuration, we observe how the processing average time grows over time, since we do not log every single event that the operator is receiving. Of course, the bigger the $update\_step$ value is, the "cheaper", in terms of resources, it is for us to monitor the network, as we log fewer events. After logging the average processing time, the processing time average is being updated and stored in a CSV file, which later we plot by using python and the matplotlib and pandas libraries.

For the experiments, values of 10, 50, 100, 261, and 500 were used to systematically sample the incoming data stream.

4.2.2 Deviation Threshold

The deviation threshold corresponds to the "tolerance" the system shows to processing time changes. Similarly to [FTG+14], where the authors also use a threshold configuration to measure the imprecision of the specific metric, deviation threshold sets the average processing time limit which if the current event processing time rate exceeds, then the running processing time average gets updated by activating the $sent\_update$ flag in the code. In Figure 4.6, there is a graphical representation of the said threshold, for better understanding. The term "tolerance" comes from the fact that if the value is high enough, then the average update won't get updated, as it "tolerates" higher processing times.

Figure 4.6: Deviation threshold representation

For this configuration, intuitive values of 0.1 and 0.3 were used.

4.2.3 Metrics scope and metrics slide

The processing time average is being calculated based on time frames. For that, two layers of time frames are added, namely the metrics scope and the metrics slide.

The metrics scope is the wider time frame, which sets the time limits to the overall calculation of the processing time average. For this fixed scope size, smaller fixed time frames within this scope, called metrics slides, are being used. Within those smaller time frames a running processing time average is being calculated based on the samples events we pick from the update step configuration. After we calculate the processing time average of all slides within the scope, an average value is being calculated from those values. In Figure 4.7, there is a graphical representation of the configuration.

For this configuration, values of 10, 60, and 90 were used for the metrics scope and values of 5, 10, 30, and 60 for the metrics slide. Experiments were carried out for all scopes paired with all slides, whose value is smaller or equal the scope value. A slide greater than the scope wouldn't make sense, because the scope needs the calculated average of its slides, in order to calculate its own average. Having a slide greater than the scope would mean that the scope average is calculated before the slide's time frame has passed.

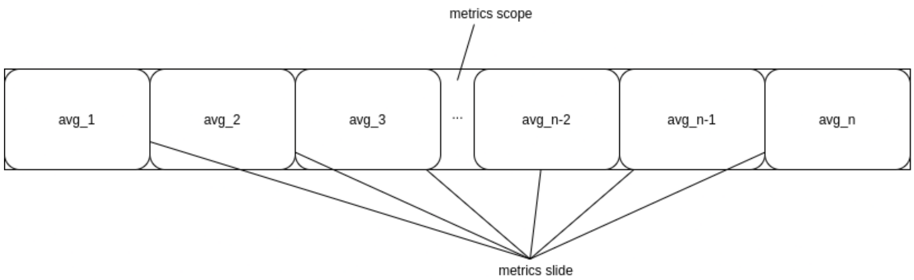

Figure 4.7: Metrics scope and metrics slide representation

5 Evaluation

The purpose of the above mentioned configurations is to help us assess the operator's workload. Each configuration imposes different amount of overhead to the operator and our goal is to find the trade-off between workload and sufficient monitoring.

To get an insight on those assessments, we log

- The sink output rate,
- the update rates of the average processing time, and
- the running average processing time during the experiment,

plot this output, and present them with some conclusions that are drawn out of them.

5.1 Sink Output Rate

The sink output rate corresponds to the experiment's quality. Because of the load-shedding's actions, some input data may be dropped, resulting to less input for the operator to work with. When the operator receives fewer data then it is expected from it to have less output. The more output the sink has, the more data has the operator received. This means better insight of the data stream, thus better output quality.

Nevertheless, it is important to remember that if the operator has a lot of input, then this means that the load-shedder allowed many events through, leading to the conclusion that our monitoring isn't sufficient.

In Figure 5.1 we see the top 15 experiments with the highest output rate. Taking a closer look at the configurations, we see that the all of experiments have a metrics slide value of 60, paired with either a scope of value 60 or 90. This pattern holds for the following 30 experiments. In the following results in Figure 5.2 we see that this changes and the metrics slide of value 30 is following. Lastly, in Figure 5.3 we see the last spots of our experiments, were the experiments with metrics slide of value 10 and 5 are placed.

One can notice that the higher the metrics slide value is, the more output the sink has. We can also conclude that the difference between a metrics slide of value 10 and one with value 5 is negligible, while we see a bigger difference on the experiments that use a metrics slide of value 30 and those of value 60. The last conclusion probably happens because of their small numerical difference between 10 and 5 in comparison to 60 and 30.

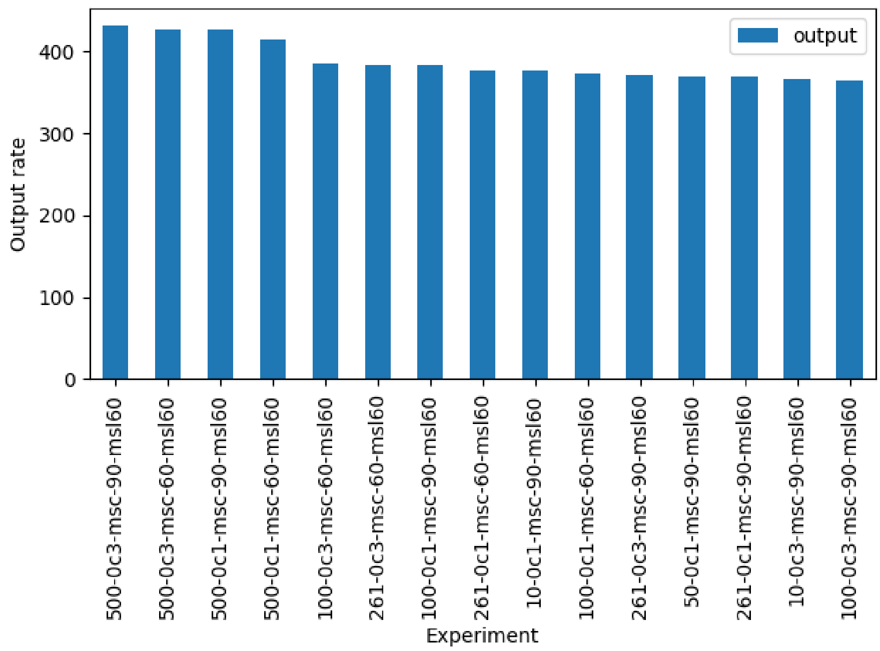

Figure 5.1: Top 15 experiments with the highest sink output rate (step-dev_thr-scope-slide)

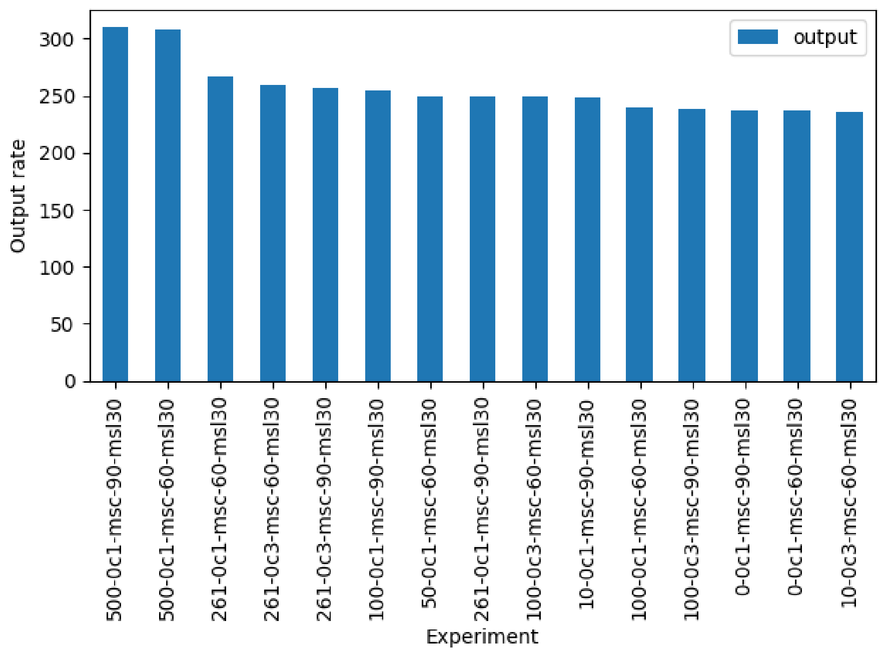

Figure 5.2: Experiments 31-45 with the highest sink output rate (step-dev_thr-scope-slide)

It is safe to assume that the higher metrics slide value is used, the more output the sink has, thus better quality. But a greater metrics slide means that the workload of calculating the average processing time is less, as explained in 4. So it is a matter of how strict we want

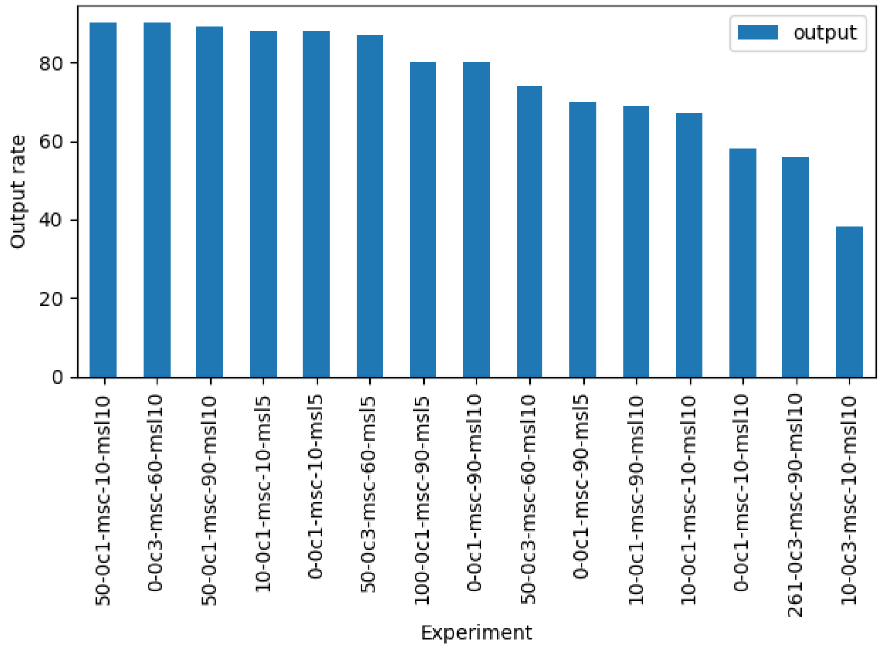

Figure 5.3: Experiments with the lowest sink output rate (step-dev_thr-scope-slide)

to be with our monitoring. Note that the more workload a node has, the more obliged is the load-shedder to take action and drop incoming events. A smaller metrics slide value results to operations being carried out more often in the node, in order to calculate the processing time average, thus more workload. Another conclusion is that the update step doesn't seem to play much of a role in this output.

5.2 Processing Time Average Update Rates

It generally holds true that the fewer running processing time average updates there are, the less overhead there is to the node. An update of the running processing time average means that the new running average should be calculated, thus running more operations to the node where the operator is deployed.

Taking a look at Figure 5.4, we see that the metrics slide and scope influence is similar to the influence it has to the sink output rate. The higher pair value of metrics scope and metrics slide results to fewer updates being carried out, which means less overhead. It was the similar case for the sink output rate, where a higher value for of metrics scope and especially of metrics slide resulted to more output, meaning better output quality and less overhead. As expected, when higher values produce less overhead, lower pair values will lead to more updates and more overhead, as seen in Figure 5.5.

Another conclusion drawn out of the results is that the higher the value of the deviation threshold is, the fewer updates are being calculated. In Figure 5.5, we see that the value

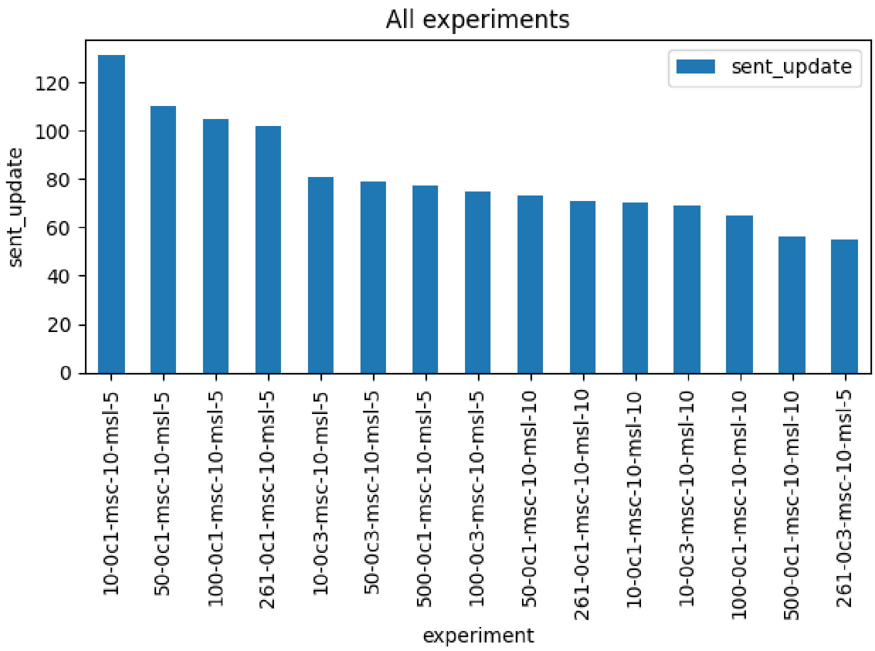

Figure 5.4: Top 15 experiments with the highest processing time average update rate (step-dev_thr-scope-slide)

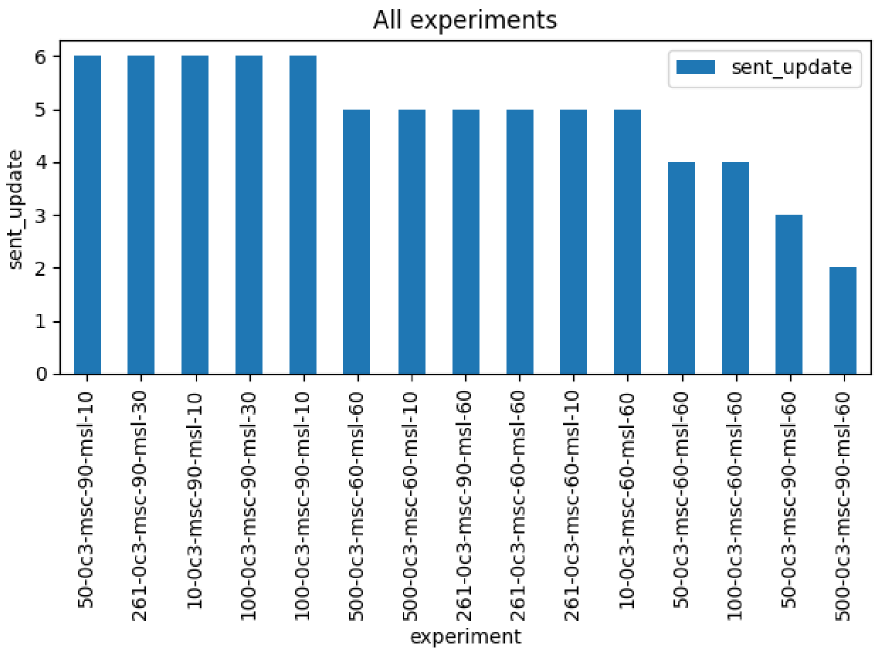

Figure 5.5: Experiments with the lowest processing time average update rate (step-dev_thr-scope-slide)

of deviation threshold $0,3$ is reigning on the last spots. That is a reasonable conclusion, since the deviation threshold corresponds to the tolerance of the new running processing time average. The lower the deviation threshold is, the smaller changes the monitoring logic imposes to the processing time average. When the event processing time exceeds this threshold, the processing time average gets updated. This configuration seems to be the one mostly in control of this output. Again, this output seems to be independent of the update step configuration.

5.3 Running Processing Time Average

The last output logged is the running processing time average. The results here are plotted with all update steps in one plot, then each plot with a different pair of metrics scope and slide and then each of those plots with a different deviation threshold.

As in the previous two outputs, the update step configuration doesn't affect the running processing time average much. Looking Figure 5.6 and Figure 5.7 individually, the update step plays little to no role in the processing time behavior. Some steps may show some abnormalities and not properly represent the processing time average, like here in the experiment with an update step of 500. But this does not depend on the configuration itself, rather than in occasional operator disruptions.

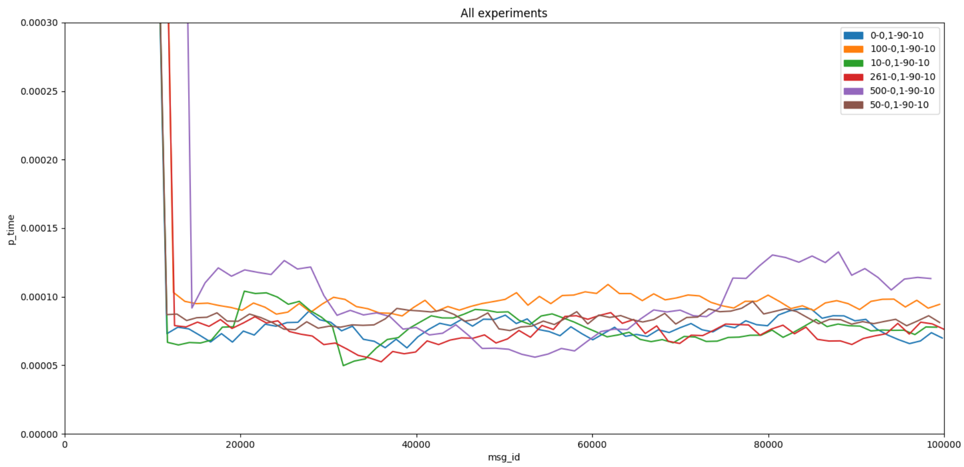

Figure 5.6: Running processing time average during the experiment with all steps, dev. thr. 0.1, m. scope 90 and m. slide 10 (step-dev_thr-scope-slide)

In Figure 5.6 and Figure 5.7, we have the results of two experiments. The experiment in Figure 5.6 has all the steps, a deviation threshold of 0.1, metrics scope of value 90 and metrics slide value of 10. In Figure 5.7, we have an experiment with the same configurations but with a deviation threshold of 0.3. Looking at the lines, we see that the plotting in Figure 5.7 is smoother than the one in Figure 5.6. The deviation threshold causes a smoother

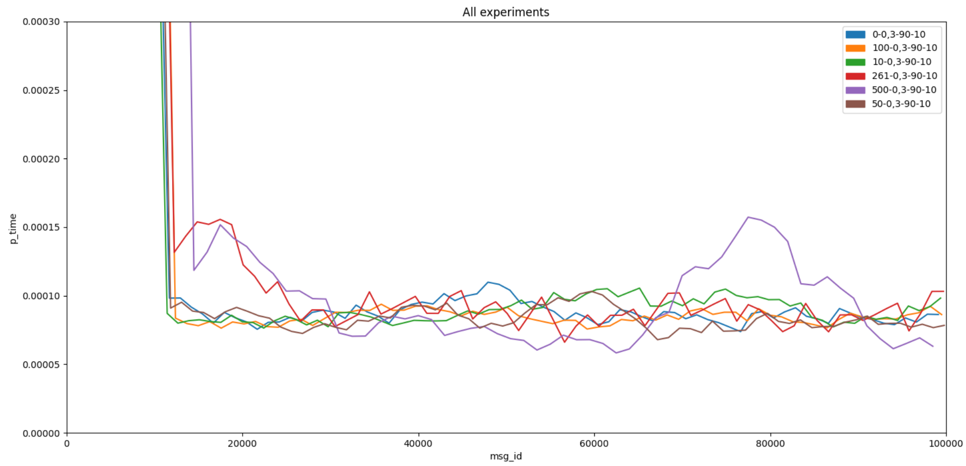

Figure 5.7: Running processing time average during the experiment with all steps, dev. thr. 0.3, m. scope 90 and m. slide 10 (step-dev_thr-scope-slide)

fluctuation in the processing time average, because of the tolerance is shows. We don't see many "up and downs" when the deviation threshold is high, because of the fewer updates it makes to the average processing time value, as seen in section Processing Time Average Update Rates.

This behavior can appears on different metrics slide as well as different metrics scope values. When comparing two experiments with the same deviation threshold, the same metrics scope but with hugely different metrics slides, we see that the fluctuation of the higher metrics slide is smoother and less detailed than the one with the lower. The two experiments in Figure 5.8 and Figure 5.9 have a deviation threshold of 0.3, and a metrics scope of 60. The one in Figure 5.8, with the more detailed plot, has a metrics slide of value 5, whereas the one in Figure 5.9, has a metrics slide of value 60. When the metrics slide value is high, we measure the running processing time average less often within a predefined scope, than when the value is low. That's why we see a less detailed result.

This behavior is also noticed on different metrics scopes. Comparing two experiments with the same configurations but with different metrics scopes in Figure 5.6 and Figure 5.10, we see the big difference in the detail of the processing time average.

5.4 Evaluation conclusion

As a whole result, we can say that the combination of high metrics scope and high metrics slide values lead to more sink output, which means less overhead. The deviation threshold is the one configuration that mainly controls the average processing time update rate, followed by the metrics slide and the metrics scope configuration. Lower values for the metrics scope and metrics slide lead to a more detailed insight of the running processing time average

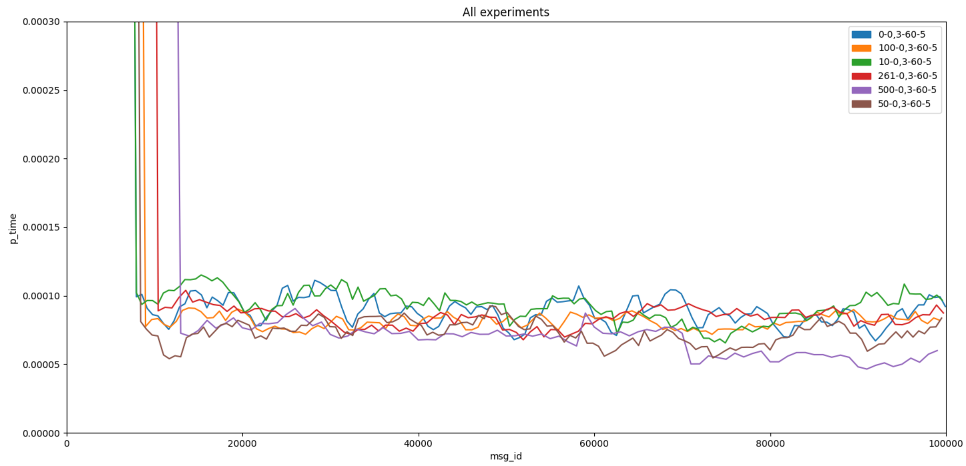

Figure 5.8: Running processing time average during the experiment with all steps, dev. thr. 0.3, m. scope 60 and m. slide 5 (step-dev_thr-scope-slide)

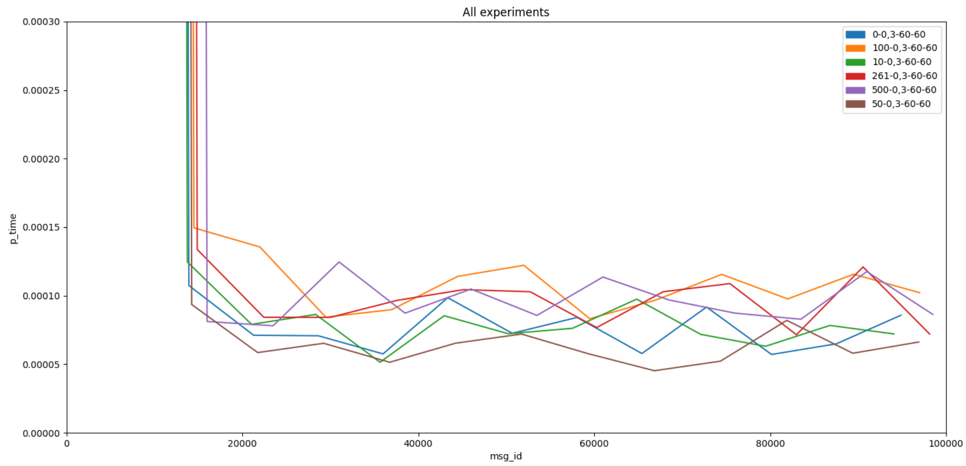

Figure 5.9: Running processing time average during the experiment with all steps, dev. thr. 0.3, m. scope 60 and m. slide 60 (step-dev_thr-scope-slide)

but this comes with more node overhead. And it was also noticed that the update step plays no important role when it comes to monitoring the operators performance.

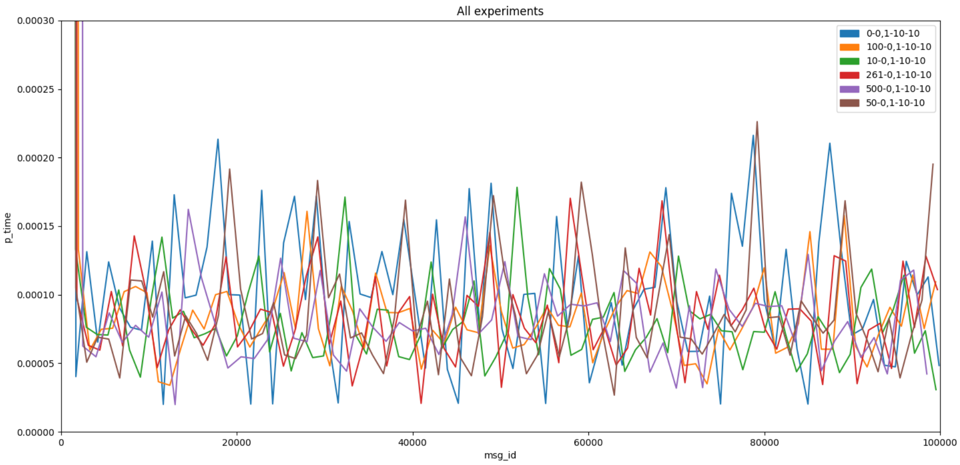

Figure 5.10: Running processing time average during the experiment with all steps, dev. thr. 0.1, m. scope 10 and m. slide 10 (step-dev_thr-scope-slide)

6 Conclusion

The goal of this thesis was to analyze the monitoring of a CEP system, in order to find a monitoring configuration that optimizes the quality of the monitoring results yet minimize the overhead.

All of the work was based on two projects that helped realize the topology and the environment, i.e. CoDaEFC and Precept. For the monitoring experiments carried out, a realistic environment was created with two layers of imposed delay, to simulate production environments. To sample the events of the input data stream, a systematic sampling strategy was followed. A certain deviation threshold was applied to the update mechanism of the average processing time. To find this average processing time, time frames were used, where the average of the average processing time within those time frames was calculated. After the experiments were carried out, there were outputs of the output rate, the average processing time, and the update-threshold rate.

The results of the experiments were then plotted and compared with one another for better understanding. Some results were drawn out of them, as to the effect the configurations had to the processing time, i.e. how much overhead they imposed. The results showed that the sampling rate didn't show any differences when compared with each other. As expected, the deviation threshold appeared to influence the average processing time update mechanism, whereas the smaller the time frames, the more detailed results we get, yet with more overhead.

6.1 Future work

Future researchers could test out more configurations and see how they affect the overhead and the quality of the results. An interesting path to go down to, would be to see how the non-probability sampling strategies can be implemented when monitoring data streams. Another thing to look would be to find a different approach to calculate the average processing time, either time-based or not time-based. If that is achieved, maybe find a way to alternate between time and non-time based approaches, based on the current overhead. A comparison between centralized and decentralized monitoring approach can be the center of a research. Finally, future work can also include CEP systems of a bigger scale, for more generalized results. The concept of CEP monitoring can also be researched on different technologies and paradigms, like hardware, cloud infrastructure, or general distributed and parallel computing.

Bibliography

[FGD+17] I. Flouris, N. Giatrakos, A. Deligiannakis, M. Garofalakis, M. Kamp, M. Mock. "Issues in complex event processing: Status and prospects in the big data era." In: *Journal of Systems and Software* 127 (2017), pp. 217–236 (cit. on p. 9).

[Fou21a] A. S. Foundation. *Apache Flink*. 2021. URL: https://flink.apache.org/ (cit. on p. 15).

[Fou21b] A. S. Foundation. *Apache Kafka*. 2021. URL: https://kafka.apache.org/ (cit. on p. 19).

[FTG+14] S. Frischbier, E. Turan, M. Gesmann, A. Margara, D. Eyers, P. Eugster, P. Pietzuch, A. Buchmann. "Effective runtime monitoring of distributed event-based enterprise systems with ASIA." In: *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*. IEEE. 2014, pp. 41–48 (cit. on pp. 15, 25).

[Hal17] S. Hallé. "From complex event processing to simple event processing." In: *arXiv preprint arXiv:1702.08051* (2017) (cit. on p. 12).

[Lea09] N. Leavitt. "Complex-event processing poised for growth." In: *Computer* 42.04 (2009), pp. 17–20 (cit. on p. 15).

[LIH+12] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, S. Dustdar. "Application-level performance monitoring of cloud services based on the complex event processing paradigm." In: *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2012, pp. 1–8 (cit. on p. 15).

[Mic06] B. M. Michelson. "Event-driven architecture overview." In: *Patricia Seybold Group* 2.12 (2006), pp. 10–1571 (cit. on p. 11).

[SBR19] A. Slo, S. Bhowmik, K. Rothermel. "espice: Probabilistic load shedding from input event streams in complex event processing." In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 215–227 (cit. on p. 16).

[Stu21] U. Stuttgart. *CoDaEFC Project*. 2021. URL: https://gitlab-vs.informatik.uni-stuttgart.de/ (cit. on p. 19).

[TÇZ07] N. Tatbul, U. Çetintemel, S. Zdonik. "Staying fit: Efficient load shedding techniques for distributed stream processing." In: *Proceedings of the 33rd international conference on Very large data bases*. Citeseer. 2007, pp. 159–170 (cit. on p. 16).

[VSI+16] M. Vögler, J. M. Schleicher, C. Inzinger, B. Nickel, S. Dustdar. "Non-intrusive monitoring of stream processing applications." In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2016, pp. 162–171 (cit. on p. 15).

[YC12] C.-H. P. Yuen, S.-H. G. Chan. "Scalable real-time monitoring for distributed applications." In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2330–2337 (cit. on p. 15).

All links were last followed on September 16, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, den 16.09.2021

place, date, signature