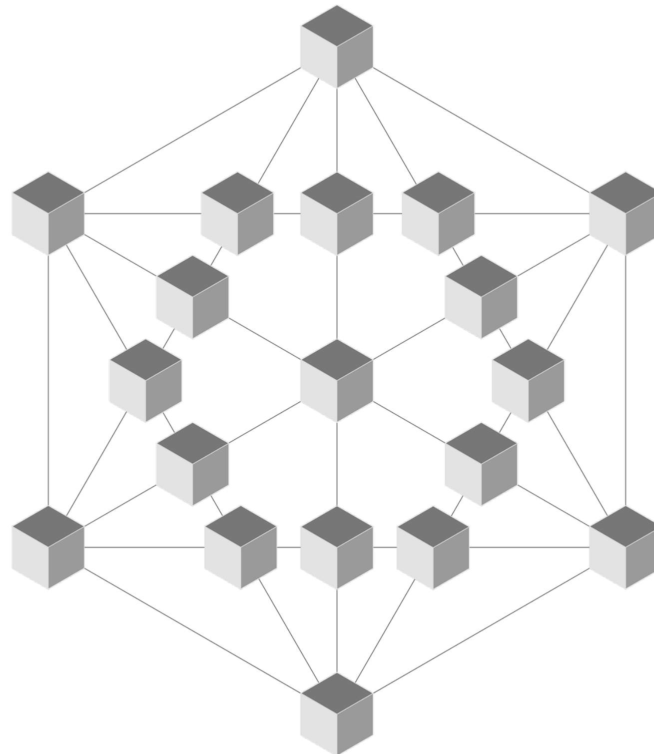

EPGTOP: A tool for continuous monitoring of a distributed system

Master's thesis in Computer Systems and Networks

Konstantinos Peratinos, Sarkhan Ibayev

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

EPGTOP: A tool for continuous monitoring of a distributed system

Konstantinos Peratinos, Sarkhan Ibayev

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

EPGTOP: A tool for continuous
monitoring of a distributed system
KONSTANTINOS PERATINOS
SARKHAN IBAYEV

© KOSTANTINOS PERATINOS & SARKHAN IBAYEV, 2019.

Supervisor: Romaric Duvignau, Computer Science and Engineering
Advisor: Eric Nordstöm, Ericsson AB
Examiner: Marina Papatriantafilou, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Networking Connections, artistic impression by GDJ.

Typeset in LaTeX
Gothenburg, Sweden 2019

EPGTOP: A tool for continuous monitoring of a distributed system
Konstantinos Peratinos, Sarkhan Ibayev
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Monitoring is fundamental to provide operational support for online systems and has been an integral part of most computer systems for decades. Kernel-level counters maintaining statistics such as the number of accepted/dropped packets are examples for classic monitoring. The ever-increasing number of connected devices has affected the scale of computer systems. Distributed systems are now inherent to most large-scale computer systems and require adjustments to existing monitoring algorithms since monitoring statistics are no longer retained locally and must be communicated over a network.

Evolved Packet Gateway (EPG) is a performance-critical distributed system responsible for processing mobile broadband data. An EPG system contains cards to process requests and can scale up to thousands of worker processes when running in production. The amount of data generated and transmitted to monitor these processes using traditional methods can overload the network cards in EPG use to communicate with one another and adversely affect the system's performance.

This thesis provides an overview of continuous distributed monitoring and evaluates continuous monitoring algorithms for distributed systems. The thesis presents EPGTOP, a monitoring service developed for continuous monitoring of EPG to asses communication-efficiency of monitoring algorithms. EPGTOP provides two modes of operation: *basic* and *approximate*. When running in the basic mode, monitoring data is periodically transmitted to a designed management node. To improve communication-efficiency of monitoring, the approximate mode allows an error threshold to be configured. The threshold is used to adjust the accuracy of system statistics continuously reported by the management node. Furthermore, the thesis discusses adjustments required to monitoring algorithms to integrate them into EPG and provides results for EPGTOP to compare and analyse trade-offs between accuracy and communication-efficiency when continuously monitoring distributed systems.

Our results demonstrate that continuous distributed monitoring algorithms are able to improve the efficiency of monitoring significantly by reducing communication costs. Additionally, utilizing larger error thresholds leads to far less monitoring data to be generated, albeit at the expense of accuracy.

Keywords: Computer, science, computer science, engineering, project, thesis.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Motivation . . . 2
1.3 Problem Description . . . 3
1.4 EPGTOP . . . 4
1.5 Report Structure . . . 5

2 Distributed Monitoring **7**
2.1 Approaches for distributed monitoring . . . 7
2.2 Simple algorithms for distributed monitoring . . . 8
2.3 Models for Distributed Monitoring . . . 8
2.3.1 The Continuous Distributed Monitoring Model . . . 8
2.3.2 The Data Stream Model . . . 11
2.3.3 Distributed Online Tracking . . . 13
2.4 Monitoring problems . . . 14
2.4.1 Forms of monitoring . . . 14
2.4.2 Functional monitoring problems . . . 15
2.5 Algorithms for Distributed Monitoring . . . 17
2.5.1 Countdown and frequency moments . . . 18
2.5.2 Basic counting . . . 19
2.5.3 Approximate counting, heavy hitters and quantiles . . . 21

3 Distributed Monitoring of EPG **23**
3.1 EPG Architecture . . . 23
3.2 Monitoring of EPG . . . 25
3.2.1 Definitions . . . 25
3.2.2 Measurements . . . 28
3.3 Implementation . . . 30
3.3.1 The system . . . 30
3.3.2 Communication . . . 31
3.3.3 Monitoring service and observers . . . 32
3.3.4 Coordinator . . . 36
3.4 Limitations . . . 39

4 Results **41**
4.1 Configuration . . . 41

4.2 Results . . . 42
4.2.1 Processor utilization . . . 42
4.2.2 Packet Processing Rate . . . 52
4.3 Discussion . . . 60

5 Conclusion **63**

Bibliography **65**

List of Figures **67**

List of Tables **69**

1
Introduction

1.1 Background

In recent years, the number of devices connected to the Internet has seen a steep rise. The Internet of Things (IoT) devices are expected to push this figure even further - in tens of billions according to industry projections. A large portion of these devices will use mobile broadband to connect to the Internet. Evolved Packet Core is the core network of Long-Term Evolution (LTE), the current mobile broadband standard, responsible for bridging the gap between cellular networks and the Internet [1]. Evolved Packet Gateway (*EPG*) sits in the middle of Ericsson's Evolved Packet Core and is responsible for processing all traffic that passes through a mobile network.

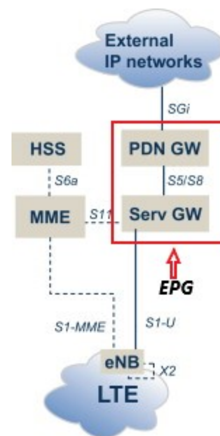

Figure 1.1: A simplified view of EPC architecture [2].

The architecture of EPG is inherently distributed. A typical EPG system can contain several nodes, referred as *cards*. These cards can be further categorized by the function that they are performing. Usually there is more than one card of the same kind in each EPG subsystem for fault tolerance. EPG is also divided into two distinct *planes* that do not interact directly with each other. These two planes are known as the *user plane* where most of subscriber data exists and the *control plane* that handles internal control data. **Figure 1.1** presents a simplified view of the standardized EPC architecture.

As networked systems grow in size, monitoring becomes necessary to assess the performance of a system and ensure that desired utilization levels are maintained. An effective monitoring system has to provide an accurate view of the whole system and its respective subsystems, while minimizing monitoring costs. A global view

can be crucial in allocating appropriate amount of resources to fulfill service level agreements and achieve high utilization. Additionally, monitoring data can be used for a variety of applications, ranging from computing usage statistics and detection of abnormal behaviour in the network to providing feedback on the design of the system.

A user plane card in EPG deploys worker processes to handle incoming traffic. Currently, monitoring in EPG is performed only at the card level, meaning statistics specific to each worker is maintained only on the card that worker is running on. In order to have a complete view of the whole system and its subsystems, monitoring data have to be gathered manually by connecting to each node through a secure shell (ssh). The aim of this thesis is to develop methods and a service - *EPGTOP* for continuously obtaining, storing and displaying measurements collected from different cards in an EPG system with considerations for communication-efficiency in order to monitor the real-time behaviour of the system.

1.2 Motivation

Monitoring has been an essential part of computer systems since the early origins of computers. A simple kernel-level packet filterer that displays statistics for dropped and allowed packets is an example for classic monitoring. When observations and retaining of monitoring statistics are performed only locally, the monitoring algorithms are generally straightforward to implement. As more nodes are added to a system, communication and coordination are required to perform monitoring at the system-level, thus monitoring algorithms need to be adjusted to reduce communication, and in certain systems, such as embedded, space and computation costs.

The architectural shift from high-performance computer systems utilizing small number of computationally powerful machines to systems utilizing large networks of components using commodity hardware is evident in current distributed systems design trends [3]. As systems grow in size, the complexity increases and necessities development of new algorithms and tools to operate on these systems and accommodate growth. Tracking system performance and health in these distributed systems is of paramount importance to ensure reliability in order to continuously adapt as components fail or under-perform over time. Such systems require rethinking of traditional monitoring algorithms originally designed to operate on a single machine or networks small in size.

Effective monitoring algorithms allow analyzing and identifying under-performing components, bottlenecks and anomalies in the system while ensuring costs associated with monitoring do not outweigh the benefits. A key design challenge for developing monitoring algorithms for distributed systems is to account for scalability. A distributed system consisting of large number of interconnected nodes is bound to have failures either due to hardware malfunction or software faults. A monitoring algorithm for such a system to track system health and performance needs to be designed in a manner to adapt to system growth since an increase in scale also leads to an increase in the complexity of interactions between components. Perfectly accurate monitoring is exceptionally costly for most distributed systems due to each observation detected requiring a monitoring message to be exchanged,

thus overloading the network with monitoring data.

1.3 Problem Description

EPG is a performance critical distributed system that can process millions of packets per second. Monitoring of a running EPG system involves monitoring individual cards and processes running on these cards. EPG is horizontally scalable with adding more cards to the system. An EPG system can contain thousands of processes running in parallel and cards use an internal network to communicate with one another. A monitoring system for EPG must ensure that extensions to existing processes to support monitoring do not disrupt the performance or behavior of these processes and is communication efficient to avoid overloading the internal network with monitoring data.

Processes such as *workers* running in EPG record their own statistics. Process specific statistics including associated processors' utilization, memory usage, the number of routed, processed, and failed requests are maintained and retained locally. Events that occur in an EPG system such as retrieval and processing of a packet cause these statistics to be updated. Correspondingly, these events and the affected values form an input to a monitoring system to track the behaviour of both the individual processes and cards they are running on. Considering there are hundreds of statistics that are of interest to be monitored in EPG, a monitoring system for EPG must be extendable to support adding more statistics and monitoring functions. EPG designates a certain group of cards as management cards and a monitoring system is required to transfer and make monitoring data gathered from processes available at these cards over a TCP network. The current approach to monitoring in EPG involves manually connecting to cards from the management cards and retrieving locally retained statistics and monitoring data. Such an approach is neither extendable nor communication-efficient since the monitoring data must be transferred regardless of the amount of changes in values of the monitored system parameters.

The most important metric that is of interest in EPG is **processor utilization**. Utilization allows detection of nodes and/or processes within nodes that are over or under utilized and allows identifying inconsistencies and anomalies in usage of the components. Additionally, the primary task of a worker is to process packets and **packet processing rate** is used as an indication of performance. In the most simplest form, a monitoring service for EPG must support monitoring the following system parameters:

1. *Processor utilization*
2. *Packet Processing Rate*

Due to the architecture of EPG, management cards control resources of a running system and a monitoring system is required to provide the management cards the values of these parameters for workers and averages for cards and the entire system continuously. These parameters are used to track the behaviour of an EPG system in real-time. Considering that the cards use the internal network to communicate, an efficiency of a monitoring system is mainly attributed to the number of monitoring messages it sends. To achieve better communication efficiency, a mon-

itoring system will attempt to reduce the number of messages sent, thus affecting the accuracy of the reported values of these parameters by the management cards. The monitoring system must guarantee that the error bounds on its output, in our case, the reported values of these parameters, is maintained all the time. In addition, due to the performance requirements of the system, monitoring must not have significant impact on the performance of processes running on cards. Neither the cards in EPG are globally time synchronized nor the changes to local monitoring statistics are time-stamped. Therefore, an eventual consistency must be provided by the monitoring system. The monitoring data from cards will be transferred over a TCP network, thus ordering of monitoring messages sent to the management cards is assumed to be handled by the internal network.

The main challenges in monitoring of EPG are its distributed nature and architecture specific requirements. Thousands of processes running in EPG can generate large amounts of monitoring data and a monitoring system must be able to optimize the amount of monitoring messages sent and adapt to system scale. Flooding the internal network with large amounts of monitoring data will directly affect the latency of processing of requests by the workers, thus the monitoring system must aim to minimize the number of messages transferred while ensuring the error guarantees are maintained continuously. Additionally, due to the implementation specific details of EPG and it components, any system added must be adapted to avoid re-engineering of these components to support monitoring. Monitoring algorithms for distributed systems make assumptions such as detection of events or transfer of messages being instantaneous to make analyzing and comparisons with other algorithms easier that do not have a direct implementation support either in EPG or in most distributed systems. These algorithms require adjustments to integrate them into EPG and effects of these changes on the efficiency and accuracy of monitoring need to be verified and will be the main focuses of our discussions. Additions to existing components must also be configurable to adjust accuracy and efficiency since monitoring of a system running in various settings such as production or testing can have different requirements.

1.4 EPGTOP

EPGTOP is a monitoring system developed to evaluate continuous monitoring algorithms for distributed systems and runs as part of EPG. EPGTOP executes monitoring services on cards in an EPG system to monitor worker processes and transfer monitoring data to a *coordinator* continuously that runs as part of a management card. Monitoring services execute monitoring algorithms designed for distributed systems that are modified to be integrated into EPG. The main objective of EPGTOP is to provide dynamic real-time monitoring statistics for the entire user plane, individual cards in the user plane and the workers running on these cards with a command-line tool while optimizing the number of monitoring messages exchanged. Evaluation of the algorithms that are executed as part of EPGTOP will be the main focus of our discussions regarding compromises in accuracy to achieve better communication efficiency when monitoring this distributed system in real-time.

The metrics described in the previous section for EPG can further be reduced

to monitoring counts and frequencies to represent events over a sliding interval of time. As an example, processing rate is merely the total the number of events over an interval of time where in our case, an event is handling of a packet received. A dedicated coordinator running on a management card acts as a sink node and subsequently uses the values streamed by monitoring services to continuously compute and report values of the monitoring functions such as average processor utilization of the system. Displaying real-time values of the monitored parameters corresponding to individual processes, cards and the entire system requires the coordinator to be able to continually receive and process updated counts and frequencies sent by monitoring services. EPGTOP supports running monitoring services either in a *basic* or *approximate* mode. The basic mode executes a simple polling based monitoring algorithm and the approximate mode executes an approximate algorithm where the level of accuracy needed can be configured to minimize the number of messages transmitted. In approximate mode, monitoring services that are running as part of EPGTOP must ensure that the coordinator's reported values are accurate up to a specified error threshold. As the following chapters will explore, continuously flooding the coordinator with updates as in the basic mode is communication intensive for most systems and would increase the latency of transmissions over the internal network, thus affecting the performance of the entire system.

Moreover, due to the performance oriented design of the system, we are aiming to analyze communication costs associated with different configurations of monitoring algorithms such as error thresholds for this system. Benefits of reducing the accuracy to minimize the number of monitoring messages transferred will be the main determining factor to establish if approximate mode is able to reduce the number of messages transmitted in comparison to the basic mode. The accuracy of the monitored values reported by EPGTOP running in approximate mode will be investigated to determine whether the monitoring services are capable of maintaining the error guarantees in real-time. Furthermore, we will be describing and reasoning about data structures and adjustments to theoretical algorithms needed to implement and integrate a monitoring system into EPG.

1.5 Report Structure

The structure of the report has been organized as follows. Section 2 presents an overview of distributed monitoring and the state of the art algorithms for solving common monitoring problems. Section 3 describes EPG architecture in more detail and presents our implementation of monitoring algorithms for this system. Our evaluation methodology and results are presented in section 4. Finally, section 5 concludes our findings.

2

Distributed Monitoring

Distributed monitoring is a generalization of classic monitoring that became relevant due to ever-increasing number of connected systems and networks. Deploying trivial solutions can have significant impact on system performance. Fundamentally, distributed monitoring algorithms achieve higher efficiency by tolerating a small loss of accuracy. Understanding trade-offs between accuracy and efficiency in the context of monitoring is crucial to designing algorithms for distributed systems. The main focus of this chapter will be *continuous distributed monitoring*, where a coordinator acts as a sink node and continually receives monitoring updates from nodes in a system to compute a monitoring function.

2.1 Approaches for distributed monitoring

Distributed monitoring systems for federating *clusters* operate on a massive scale to aggregate statistics from thousands of nodes to monitor system health and performance. Ganglia [4] is a cluster monitoring system that collects monitoring data in a distributed fashion within the network and displays a view of the clusters along with performance statistics for individual nodes and subsystems. In service-oriented large-scale architectures, *tracing* systems are employed to track the interactions between components to monitor system performance. Distributed monitoring systems such as Dapper [5] and Magpie [6] are deployed to collect information about request behavior for system tuning, capacity planning and performance debugging. An important feature of these systems is the ability to trace interactions and detect anomalies in the behavior of specific components in a system such as mis-configurations and software faults. These systems combine numerous algorithms and technologies to perform distributed monitoring effectively either for aggregate or local system metrics.

A particular form of widely-used monitoring is ***continuous*** monitoring in which an aggregate function is computed repeatedly over time and the results provide a view of the changes in the behaviour of a system for a certain time-frame. The input to the function is a multiset of observations detected at each node in a system. In this form of monitoring, we are more interested in the overall behaviour of the system rather than the behaviour of individual nodes or the interactions between them, such as computing the total number of packets processed in the entire system within the past minute. Certainly, sending monitoring data for each observation detected allows computing the result of an aggregate function trivially, however, this form of monitoring is very communication intensive for most distributed systems.

This chapter focuses on continuous distributed monitoring and we will use the term "distributed monitoring" for brevity in our discussions.

2.2 Simple algorithms for distributed monitoring

Monitoring in distributed systems requires deploying an observer at each site. Observers relay locally computed statistics to a coordinator, which acts as a sink node to aggregate and computes statistics for an entire system. A straightforward method of retrieving monitoring data from observers would be requiring each observer to inform the coordinator for each observation, such as an arrival of a packet, or periodically. These forms of monitoring in general are known as *flooding* and *polling*. However, these approaches have significant disadvantages. When observers communicate with a coordinator to inform about each observation separately, the number of packets transmitted for monitoring will be equivalent to the total number of observations and accuracy of periodic polling is dependent on the poling frequency, which can easily overload the network if narrow gaps are used and have reduced accuracy if larger gaps are used [7].

In this chapter, we are concentrating on continuous distributed monitoring of aggregate system metrics with particular attention to communication costs for the task. Our aim is to analyze distributed monitoring algorithms to develop a monitoring service for a performance-critical distributed system that minimizes communication costs by approximating a value of an aggregate function with a certain error margin rather than directly computing it by forwarding all locally detected observations. Sending a monitoring message for each detected observation will require a monitoring algorithm to produce linear number of messages to the number of detected observations, thus overloading the network with large amounts of monitoring data.

2.3 Models for Distributed Monitoring

Models for distributed monitoring formalize definitions and describe the system architecture a monitoring algorithm is developed for. The assumptions regarding calculating monitoring statistics such as definitions of an event, communication and processing delays are outlined for a model to enable formally defining the communication and space complexity of an algorithm. A correct monitoring algorithm must be able to provide guarantees defined for specific monitoring problems based on a selected model. This section describes widely used models for distributed monitoring and introduces definitions that form the basis of our discussions in the subsequent sections to formally define monitoring problems.

2.3.1 The Continuous Distributed Monitoring Model

The continuous distributed monitoring model describes a system where a coordinating node aims to continuously compute a function of all detected local observations combined to monitor the system. This section introduces the model and formally defines its requirements. A schematic view of the system architecture is provided

as well. Since the continuous distributed monitoring model is general enough to be applied to a wide range of scenarios, it forms the basis of our discussions throughout this chapter when discussing other models and monitoring problems.

Technical Background

In *the continuous distributed monitoring model*, each site (or node) acts as an observer in a system. Observers transmit information about locally detected events to a coordinator, which aggregates and computes the result of a monitoring function. Observers could be routers in a computer network, base stations in a cellular network or even services within a single system. Each individual observation is assumed to be a simple event, such as arrival of a packet. **Figure 2.1** provides an architectural view of the continuous distributed monitoring model. The communication channels between observers and the coordinator shown as arrows and are assumed to be bidirectional. In this model, the observers communicate only with the coordinator about local observations. The communications and detection of events are assumed to be instantaneous.

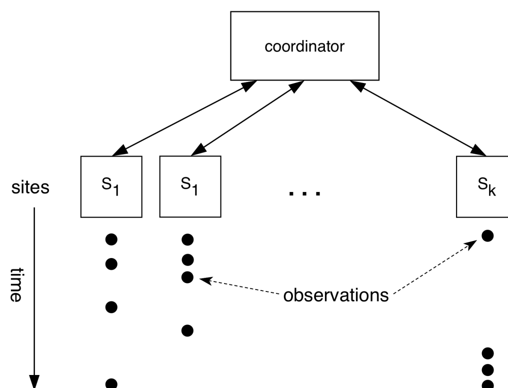

Figure 2.1: The Continuous Distributed Monitoring Model [7]

The straightforward methods of computing aggregate functions to perform distributed monitoring such as flooding and polling were presented in the previous sections. In the continuous distributed monitoring model, monitoring costs are reduced by approximating results of a monitoring function over an interval. An interval is defined either in terms of time units or events. Unless specified, we will generally refer to intervals as defined in terms of time units in our discussions for this model. Approximation allows computing monitoring statistics for a distributed system up to a certain error margin while aiming to minimize communication costs.

The Model

In the continuous distributed monitoring model, there are k remote sites and a single coordinator denoted as $S_1 \dots S_k$ and C respectively. Let $A = (a_1, \dots, a_m)$ be a sequence of items, where $a_i \in [n]$ was observed by exactly one site at time t_i and $t_1 < t_2 < \dots < t_m$. $[n]$ denotes the set of all possible items. $A(t)$ represents the

multiset of items received from all sites up until time t and we define the monitoring function as $f : [n]^m \to \mathbb{R}$. Let ϵ be an approximation parameter such that $0 < \epsilon < 1$. The aim of a monitoring algorithm defined based on this model is to approximate the value of $f(A(t))$ with a relative error of ϵ.

A connection is only allowed to be initiated by a remote site upon arrival of an item. Assume an item a_i arrives at t_i at site S_j, meaning an observation was made. S_j performs a local computation and chooses whether to inform C or not. All parties know ϵ and n in advance. Once C is informed about local results, C initiates connections to all remote sites and computes a global state, then shares it with all sites. Therefore, sites do not communicate with one another about local results. At any time t, C can calculate and approximate the value of $f(A(t))$. The cost of an algorithm is measured in the number of bits transmitted in total [8]. Even though we can define probabilistic protocols based on this model as well, we are focusing on deterministic protocols only in the following discussion.

The definition of $A(t)$ above implies that all items received up to time t form an input for the monitoring function f. The result of a monitoring function f can also be computed based on a list of items received in a recent window of size w. A recent window of size w is defined either in terms of time units or events. We denote each item as a tuple of (a_i, t_i) which indicates item a_i was received at time t_i. Given a time t and $A(t) = ((a_1, t_1), ..., (a_m, t_m))$, the input to a monitoring function f in this case is defined as $A^{'}(t) = \{(a_i, t_i) \mid 0 \leq i \leq m \wedge t - t_i \leq w\}$. *Distributed streaming* is an analogous name for this variation of the model with an additional requirement that only sub-linear amount of memory can be used [9, 10].

Geometric approach

The geometric approach is based on the continuous distributed monitoring model. The algorithms for the geometric approach allow monitoring an arbitrary threshold function f over frequency of items. In the centralized version of this approach, we assume the system has the same architecture as shown in **Figure 2.1**.

Let $\vec{v_1}(t), \vec{v_1}(t), ..., \vec{v_k}(t)$ be d-dimensional real vectors, where d denotes the number of distinct items, i.e. the size of $[n]$. These vectors are known as *the local statistics vectors*. Let $w_1, w_2, ..., w_k$ be positive weights assigned to each stream belonging to a site. Usually, w_i corresponds to the number of data items in a local statistics vector. Weights are either defined for all items since initialization or a recent window of items and can change over time due to arriving events. Let

$$\vec{v}(t) = \frac{\sum_{i=1}^{k} w_i \vec{v_i}(t)}{\sum_{i=1}^{k} w_i} \tag{2.1}$$

denote the global statistics vector.

We define an arbitrary monitoring function $f : \mathbb{R}^d \to \mathbb{R}$ from the space of d-dimensional vectors to the reals. The estimated vector, denoted as $\vec{e}(t)$, is computed from the local statistics vectors. At a given time t, let $\vec{v_i^{'}}(t)$ denote the latest statistics vector received from a site i. Then, the estimate vector is computed as:

$$\vec{e}(t) = \frac{\sum_{i=1}^{k} w_i \vec{v_i^{'}}(t)}{\sum_{i=1}^{k} w_i} \tag{2.2}$$

Each node maintains a parameter known as *the statistics delta vector*, $\Delta\vec{v_i}(t)$, which is the difference between the current and the last shared statistics vector, i.e. $\Delta\vec{v_i}(t) = \vec{v_i}(t) - \vec{v_i'}(t)$. Each node also maintains a *drift vector* $\vec{u_i}(t)$ computed as

$$\vec{u_i}(t) = \vec{e}(t) + \Delta\vec{v_i}(t) + \frac{\vec{\delta_i}}{w_i} \tag{2.3}$$

where $\vec{\delta_i}$ denotes a *slack vector* assigned to each site by the coordinator and $\sum_{i=1}^{k}\vec{\delta_i} = 0$. At any given time, the estimate vector is known by all sites. In a centralized setting, this is the responsibility of the coordinator.

The main idea behind the geometric approach is to divide the monitoring task into a set of local constraints maintained at remote sites. It can be shown that the global statistics vector is in the convex hull of the drift vectors corresponding to each site. Therefore, a site does not initiate communication with a coordinator unless its local constraint is violated [11].

2.3.2 The Data Stream Model

The data stream model for distributed monitoring is adapted from the classic data streaming model. The key difference between the data streaming model and the continuous distributed monitoring model is that unlike in the continuous model, distributed observers in the data streaming model do not compute a function of all their inputs combined together, but instead keep a sublinear amount of information to approximate the result of a monitoring function. The continuous distributed monitoring model does not require each observer to use sublinear space; it treats space as a property of an algorithm used instead [7]. This section introduces a stream model for distributed monitoring, defines its requirements and provides a view of the system architecture.

Technical Background

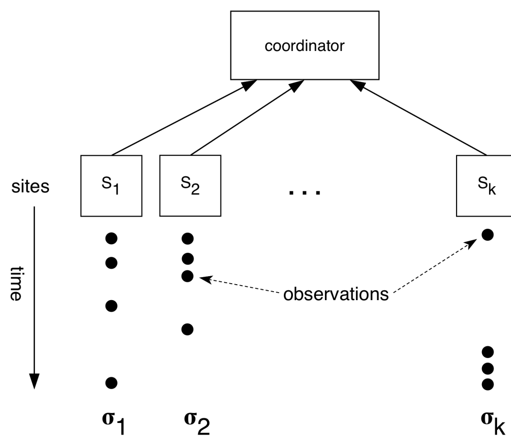

Figure 2.2: The Data Streaming Model [12]

The data stream model has a similar system architecture compared to the architecture of the continuous distributed monitoring model and is shown in **Figure 2.2**.

The major difference between the architectures is that the communications between observers and the coordinator are unidirectional and always from observers towards the coordinator instead of bidirectional. Therefore, the coordinator does not share the global state with observers and observers perform their local computations based on their own past history and decide when to inform the coordinator about local observations. It is important to note that this model for distributed monitoring matches closely to the classic data streaming model with the exception that we have $k \geq 1$ sites instead of 1. Furthermore, as discussed in the preceding section, *distributed streaming* in comparison to this model assumes that the communications are bidirectional as in the continuous distributed monitoring model, however bounds space requirements of an algorithm used in addition to communication costs [10].

The space complexity and error bound of an algorithm defined based on this model are crucial, since observers and the coordinator are assumed to not be able to hold all the received items in memory. Therefore, each site can only use sub-linear amount of space and maintains its statistics approximately. In comparison to streaming algorithms where space complexity is crucial, we are also interested in communication costs when streaming data to the coordinator in the context of distributed monitoring.

The Model

The data stream at each site is assumed to be a sequence of items from a totally ordered set U that defines the set of possible distinct items a data stream can contain. Each item in a sequence is associated with an arrival time-stamp based on a local clock. Each data stream is denoted as σ. We have $S_1 \ldots S_k$ remote sites (or observers) and a coordinator C. For a remote site S_i, the corresponding data stream is noted as σ_i. For a given stream σ, let $c_{j,\sigma}$ and c_σ be the count of item $j \in U$ and all items such that $c_j = \sum_\sigma c_{j,\sigma}$ and $c = \sum_\sigma c_\sigma$ denote the count of j and all items in all streams combined. The algorithms defined for this model typically attempt to approximate the count (or frequency) of specific items and all items to compute various statistics.

The statistics computed with an algorithm based on this model either applies to the whole stream or a recent window of size w. *Count-based sliding window* includes the last w items in each stream, and *time-based sliding window* includes items received in the last w time units. Unlike in the continuous distributed monitoring model, more than one item can be associated with a specific time t in the time-based sliding window. Sliding-window algorithms usually require adjustments to existing algorithms for the whole-stream case in regards to the space complexity due to sliding-windows converting monotonic functions to non-monotonic functions because of “deletions”. Assume we are interested in counting the frequency of a specific item. For the whole-stream case, this is achieved trivially by keeping a single counter. For the sliding-window case, this requires a space of $\Theta(\frac{1}{\varepsilon}log^2(\varepsilon w))$ bits if we allow a relative error of ε [13].

2.3.3 Distributed Online Tracking

The distributed online tracking describes a system with a general-tree structure. A coordinating node in this system aims to continuously compute a function of all local computed functions combined to monitor the system. This section introduces the model and formally defines its requirements. A schematic view of the system architecture for the model is provided as well. General-tree structure used in the distributed online tracking algorithms encapsulates the system described for the data streaming model in its simplest form. Our discussions in this section will focus on the most general form of a system - an arbitrary tree.

Technical Background

A system in distributed online tracking has a general-tree structure in which each leaf denotes a node (or an observer) and the root of the tree is the coordinator. Intermediate nodes in the tree are "relay" nodes that retrieve information from children nodes and transmit them to their parent nodes. Relay nodes do not observe their children, but are merely responsible to bridge the commutation between leaf nodes and a root. In practice, a relay node can be a router, a switch or a computing node solely responsible for transmission.

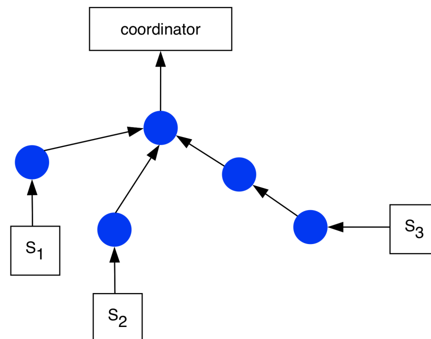

Figure 2.3: An example system for Distributed Online Tracking [14]

The communication channels between nodes are unidirectional and always from a child towards its parent. **Figure 2.3** provides a schematic view of an example system. Relay nodes in this system are shown as circles and S_i denotes a remote site i. It is important to note that any system that has a general-tree structure is valid in distributed online tracking. In the simplest case, all remote nodes are directly connected to the root and is similar to the system used in the data streaming model. An arbitrary number of relay nodes, leaf nodes and sub-trees can be inserted into the shown tree.

The Model

There are k remote sites (observers) denoted as $S_1 \dots S_k$ and a single coordinator C. Let f denote a monitoring function. The coordinator aims to compute $f(t)$ for a given time t continuously for all values of t. f is a function that takes a list of values such that each one was received from one observer for time t and computes an

aggregate result for t. Therefore, the coordinator is able to compute an aggregate function f to monitor the entire system since initialization. Due to computing $f(t)$ accurately requiring flooding the network with a monitoring message for each observation, the coordinator aims to approximate $f(t)$ by computing $g(t)$ such that $g(t) \in [f(t) - \Delta, f(t) + \Delta]$ for any $t \in [0, t_{now}]$ and some user-defined error threshold Δ [14].

Let f_i and $f_i(t)$ denote a local function at site S_i and its value at time t respectively. f at the coordinator is effectively a function that takes results of all local functions as an input for time t. Specifically, $f(t) = f(f_1(t), f_2(t), ..., f_k(t))$ for time t. Depending on the choice of a monitoring function, f and f_i can be either one-dimensional or multi-dimensional. As an example, consider an aggregate “max” function that determines the maximum value observed at time t since initialization. For each site, $f_i(t)$ corresponds to the local maximum value observed at time t, and $f(t)$ corresponds to the maximum value observed in the entire system since initialization at time t.

2.4 Monitoring problems

Monitoring problems are simplifications of common system metrics. The problems in this section are described with a reference to a general monitoring function f that takes a list of observations A and returns a value. Note that the definition of each item in this list depends on the model chosen and may be defined as a tuple of multiple parameters. We assume that the goal of a distributed monitoring algorithm is to either compute or approximate the result of $f(A)$.

2.4.1 Forms of monitoring

In general, distributed monitoring problems can be categorized into three main types: *threshold*, *value* and *set monitoring*. “Threshold monitoring” is the simplest form of monitoring and the objective is to be able to determine if $f(A) \geq \tau$ at a given time for a threshold τ . The goal of “value monitoring” is to estimate the actual value of a function at a given time. Consequently, the aim is to provide an estimate $\hat{f}(A)$ for $f(A)$ such that $\left|\hat{f}(A) - f(A)\right|$ is bounded [7]. Naturally, threshold and value monitoring are related problems. If the value of a function can be estimated, it is trivial to determine if it is greater than a certain threshold or not regardless of the algorithm used with respect to the error margins of the estimation. Similarly, running $O(\frac{1}{\varepsilon} log T)$ instances of a threshold monitoring algorithm in parallel solves value monitoring with a relative error of $1 + \varepsilon$ [8]. Finally, “set monitoring” aims to provide a set of all observations that satisfy some property. “Set monitoring” has many variations, and a popular example is to determine “top-k” largest values in a system, such as the most popular web documents across all servers or destinations that receive the most packets [15].

2.4.2 Functional monitoring problems

Sum and Countdown

The aim of a monitoring function that solves the *sum* problem is to determine if a number of observations has reached a certain threshold τ. *Countdown* is an interchangeable term for the same problem. The monitoring function must output "1" if the threshold has been reached; "0" otherwise. Since calculating the sum precisely is communication intensive, a monitoring algorithm will define an error margin on its results. Depending on the formal definition of this problem, the error margin and guarantees of a given monitoring algorithm can differ from other monitoring algorithms. In [8], Cormode et al. define the guarantees in a way that the monitoring algorithm always outputs "1" if the threshold has been reached, i.e. $f(A(t)) \geq \tau$ at time t. However, the algorithm might output "1" or "0" if the actual value is below than the threshold but within a certain error margin of it. When the value is bellow the threshold minus the safety margin, then the algorithm will always output "0". In [16], Cormode et al. define the problem slightly differently such that the algorithm always outputs "0" if the threshold has not been reached and may still output "0" even if the actual value reaches the threshold due to approximations. However, the approximated value is bounded and stays within a fixed error margin of the actual value, which allows the algorithm to output "1" subsequently. It is evident from the definition of the "sum" problem that algorithms solving it are instances of "threshold monitoring".

Basic and Approximate counting

In counting problems, a monitoring algorithm aims to approximate the actual count of an in item at a given time. Each observer's stream of observations can contain items of various interest. In *basic* counting, we are interested in the total count of all items. In *approximate* counting, we are interested in the count of a specific item or items. For instance, in a distributed computer network, basic counting can identify the total number of received packets at a given time since initialization or within a recent window, and approximate counting can determine the number of processed packets at that time since initialization or within that window. Distributed monitoring algorithms solving the counting problems are instances of "value monitoring". Depending on an algorithm, we are interested in the counts either at a given time t or within a recent window of size w defined in time units or events.

Let c and c_j denote the actual count of all items and a specific item j. In counting problems, a monitoring algorithm provides approximations for c and c_j with a certain relative error. Let $\hat{c}$ and $\hat{c_j}$ denote these approximations respectively. More formally, in *the data streaming model* (2.3.2), we define c_σ and $c_{j,\sigma}$ as the count of all items and item $j \in U$ whose time-stamps are in the current window of size w at a given time t for a stream σ. In other words, let t_i represent the arrival time-stamp of item $i \in U$. Then, for any item i in the current window of size w at time t, it holds that $t - t_i \leq w$. Note that, even though we define these counting problems for a setting in which we only consider the items in a recent window of size w at time t, these definitions can be extended to the whole-stream case (counting is

performed since initialization) by taking an infinitely large window of size W. For the data streaming model, we define c and c_j as [12]:

$$c = \sum_{\sigma} c_{\sigma} \text{ and } c_j = \sum_{\sigma} c_{j,\sigma} \tag{2.4}$$

The count of all items and specific items in *the continuous distributed monitoring model* (2.3.1) are defined with different terminologies but the guarantees that must be provided by an algorithm approximating these values are the same. More precisely, let a pair of (a_i, t_i) denote that an item $a_i \in [n]$ was received at time t_i. For a recent window of size w, let

$$c = \sum_{i=1}^{k} |\{1 \leq x \leq m,\ (a_x, t_x \in S_i) \mid t - t_x \leq w\}| \tag{2.5}$$

denote the count of all items. The count in this context represents the size of a set that contains all items received in a recent window of size w. Let

$$c_j = \sum_{i=1}^{k} |\{1 \leq x \leq m,\ (j_x, t_x \in S_i) \mid t - t_x \leq w\}| \tag{2.6}$$

denote the count of a specific item $j \in [n]$ in a recent window of size w. According to this definition, c_j represents the size of a set of all items received in a recent window of size w excluding items that are not j. Given an approximation parameter $0 < \varepsilon < 1$, the following are the formal definitions of counting problems [9, 12]:

- *Basic Counting*: return an estimate $\hat{c}$ such that $|\hat{c} - c| \leq c\varepsilon$ for all items in the current window.
- *Approximate Counting*: for an item j, return an estimate $\hat{c}_j$ such that $|\hat{c}_j - c_j| \leq c\varepsilon$ for all j in the current window.

Voting and Ranking

Voting (or *ranking*) is a form of non-monotonic counting. A site is allowed to make either a positive or negative contribution to the overall count. Similar to counting problems, we are interested in the current count at a given time t. Since the contributions can either increase or decrease the overall count, the monitoring algorithm must be able to support "deletion" operations to determine the voting margin. A simple approach to solve the voting problem would be utilizing two separate data streams for basic counting each with their own specific accuracy. However, computing the basic counts separately and taking their difference will not yield a relative error guarantee for the difference [17]. In general, voting problems are defined formally in the same way as counting problems with the exception that the assumption about all items being positive (or negative) is relaxed.

Frequency moments, heavy hitters and quantiles

Various other functional monitoring problems have also been studied in addition to the problems presented in the preceding sections. *Frequency moments* are frequently

referred and discussed in the literature [8, 17, 18, 19]. A frequency moment k is defined as $F_k = \sum_{i=1}^{n} f_i^k$ for a list of frequencies $f = (f_1, f_2, ..., f_n)$ of items. F_0 and F_1 map to counting the number of distinct items and all items correspondingly. F_2 is used to compute statistical properties of data and F_∞ is defined as the frequency of the most frequent item(s). In the context of monitoring, an item refers to an application-specific type of an observation detected at a remote site.

Frequent items (or *Heavy hitters*) is the problem of determining the most frequent item(s) whose frequencies are ϕ-heavy, meaning for an item i, $f_i \geq \phi \sum_{j=1}^{n} f_j$. This is a simplification of the problem of finding the maximum frequency element in all streams of observations, since it is impossible to approximate the maximum frequency in sub-linear space [20]. *Quantiles* is the problem of finding ϕ-quantile element of a multiset of observations, which is an element x, such that for a set of size l at most $\phi\, l$ elements are smaller than x, and at most $(1 - \phi)\, l$ elements are greater than x [21]. Given $0 < \phi < 1$, the following are the formal definitions of these problems [12]:

- *Frequent items*: return a set which includes all items j with $c_j \geq \phi c$ and possibly some items j' such that $(\phi - \varepsilon)c \leq c_{j'} \leq \phi c$.
- *Quantiles*: return an item whose rank is in $[(\phi - \varepsilon)c, (\phi + \varepsilon)c]$.

Frequent items and quantiles for distributed monitoring have been studied extensively [21, 22, 23, 24, 25, 26]. “Set monitoring” problems in certain cases can be reduced to data streaming problems. A popular data streaming problem - “top-k”, such as determining most frequent destinations in a computer network, can be solved using an existing algorithm for frequent items [25]. Furthermore, the exact property in “set monitoring” that members of a set need to satisfy is not defined generally. Thus, algorithms for “set monitoring” tend to be more problem specific.

The functional monitoring problems discussed in this section clearly indicate similarities to problems defined and studied for classic data streaming algorithms. In the context of monitoring, these problems map to metrics to compute statistics for in a system and we assume that there are more than one site. Additionally, these problems are generally studied in a setting where only items in a recent window are considered, so an algorithm’s space complexity is of importance.

2.5 Algorithms for Distributed Monitoring

Monitoring algorithms are defined for a specific model and a correct algorithm must be able to provide guarantees required by a monitoring problem. In the context of distributed monitoring, aside from correctness, we are also interested in the communication costs of an algorithm. If sites in the system are assumed to not be able to hold all items in memory, a monitoring algorithm must also achieve sub-linear space complexity. Previous sections described common monitoring models and problems, and this section provides an overview of renowned distributed monitoring algorithms.

2.5.1 Countdown and frequency moments

Recall that countdown is a “threshold monitoring” problem described in 2.4.2. Similarities exist among algorithms solving countdown and frequency moments problems, since frequency moments are generally studied in the context of “threshold monitoring”. An overview of simple algorithms in addition to a more advanced algorithm based on the geometric approach is provided in this section.

A simple algorithm

An algorithm for the countdown problem works like a software trigger switching from “0” to “1” whenever τ is reached. A simple algorithm for this problem defined based on the continuous distributed model works by distributing local thresholds to every observer [7]. The algorithm works by realizing that τ can not be reached unless the local count of at least one of the observers has reached τ/k. The coordinator keeps track of a slack, denoted as S, which is the difference between τ and the current total count. Therefore, the initial slack is set to τ. Whenever one of the local thresholds is reached, that site informs the coordinator of its current count, which in turn prompts the coordinator to ask each observer about its current local count. Then, the coordinator calculates a new slack as $S = \tau - n$ where n is the sum of all local counts. Finally, the coordinator distributes a new local threshold of S/k to each observer and all sites restart the counting process. This algorithm repeats until the slack is equal to k, at that point, observers inform the coordinator about every event. This algorithm has a communication cost of $O(k^2 log\frac{\tau}{k})$.

The algorithm can be improved by introducing a relative error of ε and splitting the counting process into $\log(1/\varepsilon)$ rounds [8]. At the start of round j, the coordinator sends the current total count n and a new local threshold $t_j = \frac{\tau - n}{2k}$ to each observer. Each site sends a bit whenever $\left\lfloor (n' - n)/t_j \right\rfloor$ increases by one, where n' is n plus the local increases in round j. Note that the initial threshold is half of that of the simple algorithm, since the coordinator ends a round whenever k bits in total have been received. At the end of a round, the coordinator retrieves the local increases from each observer, and shares the new total count and t_{j+1}. The algorithm returns “1” and terminates whenever the total count becomes larger than or equal to $(1 - \varepsilon/2)\tau$ for some error parameter $0 < \varepsilon < 1$. The approximate version has an upper bound of $O(k \log(1/\varepsilon))$ on the communication costs.

This bound cannot be improved further without using randomization methods, but such algorithms are out of the scope of this thesis. Note that the original algorithm actually operates on frequency moments and is simplified in this discussion due to F_1 mapping directly to the total count of all items. Thus, a monitoring function to determine whether F_p for the p-th frequency moment has reached a threshold τ can be computed with the original version of the algorithm [8].

A geometric algorithm

There are two algorithms that derive from the geometric approach described in 2.3.1 [11]. The first algorithm is fully decentralized and relies on broadcasting. The second algorithm conforms to the continuous distributed monitoring model, where

sites do not communicate directly with one another but send their calculations to the coordinator. This section describes the centralized version of the algorithm.

Let $\vec{v_i}(t)$ denote the local statistics vector as defined previously. During the initialization phase, all sites send their local statistics vectors to the coordinator. The coordinator calculates the estimate vector $\vec{e}(t)$, which is distributed to all sites. Additionally, the coordinator sets the initial slack vector $\vec{\delta_i}$ to 0 during start-up. One of the advantages of the centralized algorithm over the decentralized one is that the coordinator uses the slack vector to balance out the local statistics vectors of the observers [11].

After the initialization, when a site receives new data, it calculates the statistics delta vector $\Delta\vec{v}_i\left(t\right) = \vec{v}_i\left(t\right) - \vec{v'}_i$, which is the difference between the current vector and the last statistic vector collected from the node. The site also calculates the drift vector defined as $\vec{u_i}\left(t\right) = \vec{e}\left(t\right) + \Delta\vec{v}_i\left(t\right) + \frac{\vec{\delta_i}}{w_i}$. After that, it checks to determine if its local constraint is not violated, i.e. $B\left(\vec{e}\left(t\right), \vec{u_i}\left(t\right)\right)$ remains monochromatic - all the vectors contained in a ball have the same color, where B is the ball on the site's drift vector, centered at $e + \frac{1}{2}\Delta v_i$ with a radius of $\frac{1}{2}\left\|\Delta v_i\right\|_2$. If the constraint does not hold anymore, it informs the coordinator. The coordinator then tries to create a balanced monochromatic vector from a group of sites in a way that the average of their vectors remain monochromatic:

$$\vec{b} = \frac{\sum_{p_i \in P'} w_i \vec{u_i}\left(t\right)}{\sum_{p_i \in P'} w_i} \tag{2.7}$$

Once the coordinator has selected the group of nodes, it sends them an updated slack vector which is calculated as: $\Delta\vec{\delta_i} = w_i\vec{b} - w_i\vec{u_i}\left(t\right)$. If the coordinator is unable to balance the nodes, it calculates a new estimate vector and forwards it to all the nodes, thus restarting the process. At all times, the sum of all slack vectors must be equal to 0 for the algorithm to function correctly.

2.5.2 Basic counting

The formal definitions of counting problems in the context of monitoring were provided in 2.4.2. A monitoring algorithm that provides an approximation for the total count of all items (basic counting) or specific items (approximate counting) at a given time t can define the count either based on the time since initialization or in a recent window of size w. In this section, we provide three different notable monitoring algorithms that solve the basic counting problem.

Thresholded Counting

Let A denote an algorithm defined based on the continuous distributed monitoring model that solves the countdown problem. The thresholded counting algorithm is performed by running $O(\frac{1}{\varepsilon} log T)$ instances of A in parallel for thresholds of $\tau = 1, (1+\varepsilon), (1+\varepsilon)^2, ..., T$ to approximately count all items with a relative error of $1+\varepsilon$ since initialization [8]. Let $r = 1, 1, 1, ...1, 0, 0, ...0$ denote the output of all running instances of A at some time t. Each output in this set corresponds to an output of instance of A for a specific threshold. The first output corresponds to A

for $\tau = 1$, the second output to $\tau = (1 + \varepsilon)$, and so forth. We can approximately determine the current count by checking the last "1" in this set, since the actual value will be between the corresponding threshold for the last "1" and the first "0" after it in the set. If the actual value was larger than the threshold for this "0", then the instances up to that value would have outputted "1" instead of "0". Therefore, running $O(\frac{1}{\varepsilon} log T)$ instances of A solves any "value" monitoring problem including "counting" given that the actual value is between 1 and T [8].

The forward/backward algorithm

The forward/backward algorithm solves the basic counting problem for a recent window of size w [9]. Assume that the time starts at 0 for simplification. The time axis can be divided into windows of fixed sizes: $[0..w)$, $[w..2w)$, ... A sliding-window at time t: $[t - w, t)$ can overlap with at most two fixed-size windows $[(i - 1)w, iw)$ and $[iw, (i+1)w)$ for some positive integer i. Essentially, our current sliding window is split into two windows known as "expiring" and "active" at any time t: $[t - w, iw)$ and $[iw, t)$. It is straightforward to realize that as the time passes, new items arrive in the active window and old items in the expiring window get removed. Therefore, approximation of the current count is performed by determining the respective counts in both of these windows.

In the forward problem, a site starts with an initial value at time t_0. Whenever this value increases by a $(1+\varepsilon)$ factor, the site sends a bit to the coordinator. Consequently, the coordinator can keep track of the count of arriving items at each site. In the backward problem, the simplest approach is to send information about arrivals at the end of each fixed window. Then, the coordinator can decrease the count for a specific site whenever an item in the expiring window gets outside of the current sliding-window. This necessities each site to maintain a history of past arrivals for the current fixed sized window. A more space efficient approach utilizes an *exponential histogram* described in [13]. An exponential histogram allows approximate counting of the number of items in the expiring window. A site initially informs the coordinator about the number of time-stamps stored in the local histogram. As items expire, the site sends a bit to the coordinator. From these, the coordinator is able to recreate the current approximate count of the expiring window for each site. At the end of each fixed-sized window, the current local active window becomes the next expiring window, and both algorithms are run again in parallel.

The forward/backward algorithm has a communication cost of $\Theta(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ bits where k is the number of sites and n is the number of observations in a window. Therefore, the communication costs depend on the number of items received in a window rather than the size of the window. Despite the fact that the original algorithm is defined based on *distributed streaming*, only one-way communication is required between the coordinator and sites. When an exponential histogram is used, the algorithm has a space cost of $O(\frac{1}{\varepsilon} log(\varepsilon n))$ [9].

The up/down algorithm

The up/down algorithm also focuses on a recent window of size w [12]. Let λ and γ be two constants set to $\frac{\varepsilon}{9}$ and 4λ respectively. Each site maintains a λ-approximate

data structure as in the backward problem. Therefore, at any time t, a site can compute $\hat{c_\sigma}(t)$. Let p denote the time $\hat{c_\sigma}(p)$ was sent to the coordinator and $p < t$. We define two events as follows:

- *Up*: $\hat{c_\sigma}(t) > (1+\gamma)\hat{c_\sigma}(p)$.
- *Down*: $\hat{c_\sigma}(t) < (1-\gamma)\hat{c_\sigma}(p)$.

A site informs the coordinator whenever one of the events occur at any time t. Therefore, the coordinator is able to compute the approximate count by summing all corresponding counts of each stream. In the most basic form, each event requires a message of size one word to be sent, since $\hat{c_\sigma}(t)$ is sent to inform about the current count. The communication costs are reduced by maintaining a "restricted" estimate. Assume we have a set of possible estimates denoted as $K = k_0, k_1, \ldots$. Consider a site that sent an estimate k_x the last time. Let a restricted estimate corresponding to a new event be k_y. The site only computes and sends the difference between these indices: $y - x$, which allows the coordinator to determine the corresponding restricted estimate. Therefore, a trade-off between accuracy and communication efficiency can be achieved by using different definitions for K. This algorithm has a communication complexity of $\Theta(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ [12].

2.5.3 Approximate counting, heavy hitters and quantiles

In this section, we focus on heavy hitters and quantiles problems for distributed monitoring. Both of these problems reduce to performing approximate counting of specific items. We initially describe approximate counting and explain the computations performed to extract heavy hitters and quantiles for each algorithm. As in basic counting, we focus on a recent window of size w.

The forward/backward algorithm

The forward/backward algorithm for heavy hitters (or frequent items) is an extension of the same algorithm for basic counting [9]. This extension allows approximating the count of every item j in addition to all items. Let A be equal to 1 for a site at time t_0. In the forward problem, each site maintains n_j and n which denote the count of j and all items. Whenever, n_j increases by A, i.e. $n_j \mod A$ is equal to 0 after an arrival of an item j, the site sends n_j to the coordinator. Additionally, the value of A is doubled whenever n becomes larger than $2\varepsilon^{-1}A$. At any time, the coordinator knows the count of any item j with an additive error of at most $A - 1$.

In the backward problem, the reverse of the forward algorithm is run in parallel. Let B denote the local error tolerance. The initial value of B is set to the value of A from the active window that concluded at the end of some fixed interval $[(i-1)w, iw)$ for a positive integer i. Initially, a site informs the coordinator about each j such that $n_j \geq B$. Then, as long as B is in the range $[0.5\varepsilon n, \varepsilon n]$, a site sends a bit to the coordinator whenever an expiration of an item causes n_j to decrease by B, i.e. the equation $n_j\ mod\ B = 0$ holds. Similar to the forward algorithm, B is divided by two whenever it goes out of the allowed range and the algorithm repeats. The termination is done at the end of a fixed size window. After that, the current active window becomes the expiring window, and both algorithms are executed in parallel

again.

As in the algorithm for basic counting, space requirements are reduced by approximating counts at each site instead of exactly computing them. From the coordinator's perspective, it is able to determine the approximate count of each item j at all sites at time t by considering active and expiring items. Therefore, the current c_j is estimated by adding all corresponding counts for each site. The coordinator can then apply the equation presented in 2.4.2 to extract the frequent items. The presented algorithm has a communication complexity of $\Theta(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ and a space complexity of $O(\frac{1}{\varepsilon} log(\varepsilon n))$ bits [9].

For the quantiles problem, the algorithm is more involved and makes use of a data structure presented in [27] that stores ε-approximate quantiles over a fixed sized sequence of items. It can be shown that an upper and lower bound of $O(\frac{k}{\varepsilon} log^2(\frac{1}{\varepsilon}) log\frac{n}{k})$ and $\Omega(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ can be achieved with a space complexity of $O(\frac{1}{\varepsilon} log^2 \frac{1}{\varepsilon} log\, n)$ to extract quantiles with this algorithm [9].

The up/down algorithm

Let λ be a constant set to $\frac{\varepsilon}{11}$. Each site maintains a λ-approximate data structures as described before to count c_σ and $c_{j,\sigma}$, the count of all items and item j. Let p denote the time $\hat{c_{j,\sigma}}(p)$ was sent to the coordinator and $p < t$. In the simplest form of the algorithm, we define only two events as follows:

- *Up*: $\hat{c_{j,\sigma}}(t) > \hat{c_{j,\sigma}}(p) + 9\lambda\hat{c_\sigma}(t)$.
- *Down*: $\hat{c_{j,\sigma}}(t) < \hat{c_{j,\sigma}}(p) - 9\lambda\hat{c_\sigma}(t)$.

A site informs the coordinator whenever one of the events occur at any given time t. Therefore, the coordinator is able to compute the approximate count of item j by summing all corresponding counts for each stream. Note that, in this simplified algorithm, the communication cost depends on the number of distinct items. The algorithm is extended further to reduce this dependency, and an upper bound of $O(\frac{k}{\varepsilon} log(\frac{n}{k}))$ and a lower bound of $\Theta(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ can be achieved [12]. To extract the frequent items, the sites and the coordinator perform both the basic and approximate counting algorithms for error parameters of $\frac{\varepsilon}{24}$ and $\frac{11\varepsilon}{24}$, and return all items that the equality $\hat{c_j} \geq (\phi - \frac{\varepsilon}{2})\hat{c}$ holds.

The quantiles problem with an up/down algorithm is solved by maintaining λ-approximate ϕ-quantiles for $\phi = 5\lambda, 10\lambda, ..., 1$. The coordinator is updated by each site to ensure that these approximations maintained at the coordinator and a site are the same. This algorithm achieves the same lower bound of $\Omega(\frac{k}{\varepsilon} log(\frac{\varepsilon n}{k}))$ for the communication costs as the forward/backward algorithm, but has an upper bound of $O(\frac{k}{\varepsilon^2} log(\frac{n}{k}))$ bits [12].

3

Distributed Monitoring of EPG

Evolved Packet Gateway (EPG) is a performance-critical distributed system that processes mobile broadband traffic. EPG in its virtual form is designed and optimized to scale and handle massive loads as mobile traffic grows aggressively as a result of increasing number of IoT devices. A monitoring service developed for such a system that processes millions of packets per second must adapt to traffic growth and incur minimal communication costs to avoid overburdening the internal network with monitoring data. Adjustments are required to theoretical solutions for distributed monitoring due to assumptions such as instantaneous communications having no equivalent practical implementation in EPG and such adaptations are the main focus of this chapter.

3.1 EPG Architecture

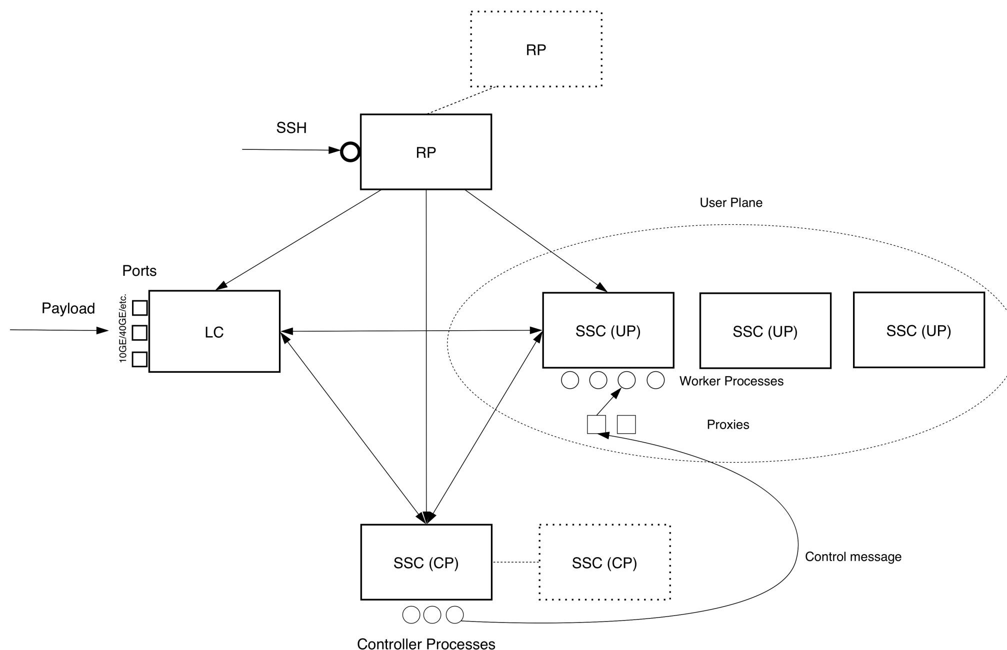

Figure 3.1: A simplified overview of EPG architecture.

An individual EPG system is a distributed system of interconnected components known as "cards". **Figure 3.1** provides a simplified overview of an architecture of an EPG system. Cards are either physical or virtual Linux machines that run on

custom or multi-processor architectures depending on their functionality. Payload is received through Ethernet ports of *Line Cards* (LC) and is routed within the network. A *Routing Processor* (RP) card is a management machine that provides administrative tools to control other cards in the system and is accessed via an SSH tunnel. *Smart Services Cards* (SSC) are the main worker nodes in the system and are in charge of processing the received payload. In an EPG system, functionality of a card is denoted as "role". An SSC can either run in "control plane" or "user plane" role in EPG jargon. Control Plane SSC's are utilized to manage User Plane cards and are supported with redundant cards to provide fault tolerance. An EPG system contains many SSC cards running in "user plane" role to process the received requests. Redundancies in the diagram are depicted with boxes with dashed borders. EPG is designed to tolerate one failure in each of its subsystems (N+1 redundancy), for RP and SSC in control plane role, this is achieved by having an additional card in "hot standby" mode. In the case of SSC user cards, N+1 is accomplished by reserving a portion of the resources and the use of *sharding* techniques such as routing requests based on identifiers.

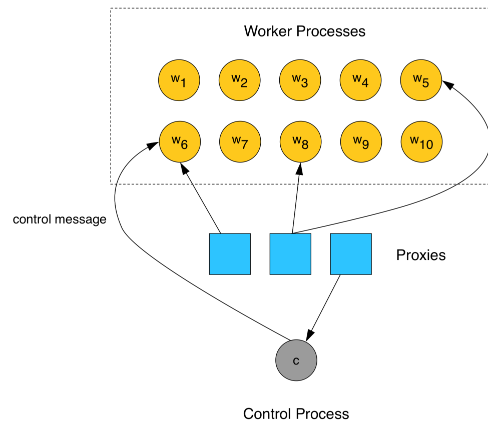

Figure 3.2: A simplified overview of User Plane SSC.

A User Plane SSC contains three types of processes: control (single), proxy (few) and worker (multiple). A control process of a card handles control traffic and proxy processes route the traffic within the card to worker processes and act as internal load balancers. **Figure 3.2** demonstrates the architecture of an example User Plane card. Circles with an orange and grey background represent worker and control processes respectively. Note that the number of proxies, worker, and control processes are specific to a card and in this example, there are ten worker processes and a single control process. Redundancies for subsystems within a card are omitted for simplification. Proxies are the entry endpoints to a card and receive all incoming traffic. The exact behaviour of proxies, internal routing mechanisms, and load-balancing logic are irrelevant to our discussions. We assume that the system is always operational and omit considering intricacies of fail-over mechanisms. Processor affinity is used for worker processes to bind each worker to a specific CPU. Inter-process communication is used to send messages and access shared data

structures between processes. A worker process runs in a busy loop always polling for incoming requests. Workers also maintain statistics about their operations such as the number of received packets. A control process is aware of all worker processes and has access to internal performance statistics. Workers are dedicated to process incoming traffic and any additional task is the responsibility of a control process.

A straightforward method of retrieving monitoring data from cards would be creating SSH tunnels to each card from an RP card and retrieving data for each observation or periodically. As explained in the preceding chapter, both approaches can overload the internal network with monitoring data and induce significant costs for monitoring. The inherent distributed nature of the architecture and its performance requirements necessitate development of monitoring methods that can aggregate and compute metrics for each card, subsystems and the entire system while ensuring the costs associated with monitoring are minimized and bounded. Additionally, standard Linux tools such as “top” can not be used to compute metrics such as utilization, since worker processes are running in busy loops and will still report 100% CPU utilization even if no packets are processed.

3.2 Monitoring of EPG

EPG is a complex system of many interconnected components. Continuous distributed monitoring of an EPG system can be performed on the entire system, subsystems such as RP, Control and User plane, individual cards and processes running on cards. Accordingly, we narrow down our focus to monitoring of worker processes running on SSCs in a “user plane” role only. Monitoring functions for an SSC are computed from monitoring data corresponding to workers running on that SSC and monitoring functions for the entire system are computed from monitoring data corresponding to all SSCs running in “user plane” role. Supporting “control plane” SSCs is merely an implementation detail and is not relevant to our discussions in this chapter of the monitoring algorithms implemented, since the monitoring services operate on a collection of counts and frequencies and there are implementation related differences in retrieving these values on a “control plane” card. We provide definitions to describe the system architecture and implementation of EPGTOP in subsequent sections, define metrics and briefly describe approaches to compute statistics for these metrics.

3.2.1 Definitions

Continuous distributed monitoring of an EPG system is performed with deploying a monitoring service on each card. The coordinator is a process running in RP. A monitoring service on a User Plane card is a POSIX thread running as part of the control process for that card. The coordinator has a bidirectional communication channel to each monitoring service. Cards do not communicate with one another to share information about local monitoring data. The coordinator uses the communication channels both to send control messages and retrieve updates from monitoring services.

On a User Plane card, a worker process always runs in a busy loop and updates its associated local performance statistics as it processes requests. A worker process's actions generate observations such as processing of packets and updating counters. Observations are monitored by the monitoring service on that card. The monitoring service utilizes local statistics corresponding to each worker to execute a monitoring algorithm on the behalf of that worker. Therefore, a worker does not communicate with the coordinator directly and the monitoring service acts as a relay maintaining and facilitating communications of worker processes with the coordinator. Even though a card only has a single process running as a monitoring service, from an algorithmic and the coordinator's perspective, each worker running on a card is treated as an observer due to monitoring data being associated with workers only. The main advantage of maintaining a single process for monitoring observations instead of a dedicated process for each worker is due to implementation specific details, since the workers should only be responsible for processing and not handle monitoring requests and a dedicated process to observe each worker would be resource intensive. Additionally, the monitoring service running on a card does not perform any aggregation other than packaging updates corresponding to multiple workers into a single message, meaning that the amount of monitoring updates sent is equivalent to allocating a dedicated monitoring service for each worker. In the following discussions, a worker refers to an observer from the systems described in the previous chapter. Conceptually, each worker in EPG performs its own observations by updating its internal counters. Considering each worker as a distinct observer allows the coordinator to compute and report monitoring statistics specific to that worker. Aggregating the monitoring data of all workers running on a card would have removed the association of the monitoring data with individual workers and the coordinator would only be able to report aggregate monitoring statistics for a card.

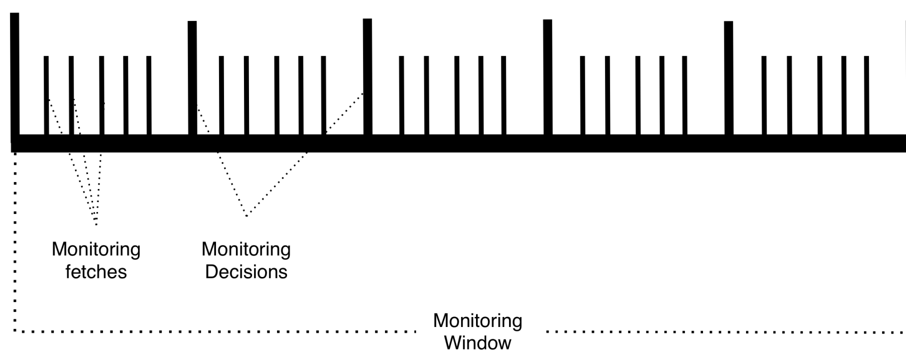

Figure 3.3: Periods for fetching and making monitoring decisions.

Albeit each processed packet by a worker generating an observation by definition, the observations are only reviewed periodically on a card. An SSC may process hundreds of thousands of packets per second, thus monitoring computations if performed per packet basis would have induced significant computation costs. EPG is a performance-oriented system and additions such as monitoring services to a card must ensure that the workers and services running on a card are neither disrupted

nor affected by these additions significantly to allow the system to maintain the same service-level agreements such as response time. **Figure 3.3** provides an overview of the time-line seen by a monitoring service. A *monitoring period* refers to a period which a monitoring service executes a monitoring algorithm for each worker at the end of this period such as updating internal data structures and making a decision regarding whether to send an update to the coordinator or not. The decision is specific to each monitored value for each worker. The *polling frequency* refers to the number of times a monitoring service queries internal data structures to retrieve statistics about observations made during a monitoring period for all workers. Both the monitoring period and the polling frequency are implementation details configured by the coordinator and can be adjusted to increase and decrease the number of local calls to retrieve monitoring data. A *monitoring window* refers to the window the monitoring statistics are kept for and is a sliding window of fixed size defined in time units. Each worker is associated with its own monitoring window for each monitored value. As shown in the figure, a monitoring window is divided into monitoring periods and each monitoring period is divided into polling periods. The size of a polling period is equal to the size of a monitoring period divided by the polling frequency. The polling period is basically the amount of time a monitoring service must wait before performing the next local fetches for all workers. At the end of each monitoring period, the monitoring service checks local statistics corresponding to each worker process and performs monitoring computations. A monitoring algorithm dictates whether the monitoring service should inform the coordinator about the monitoring data for each worker or not. Unlike sending a message over a network to the coordinator, local polling is vastly faster and induces minimal costs due to inter-process communication. The aim of these parameters is to configure the frequency of local fetches and execution of the monitoring algorithms since due to performance reasons, we cannot perform monitoring calculations for each arriving packet. Naturally, these parameters can be configured to achieve much better accuracy on reported values due to more frequent data fetches. However, in our analysis, these parameters are set to fixed values, since our discussions of accuracy are more concerned with determining whether a monitoring service is capable of maintaining the error bounds it is configured to guarantee on its reported values. The monitoring service in a card maintains both its own data structures and data structures corresponding to each worker. An important detail to mention is that the actions of the monitoring service, a control process or proxies in a card do not generate observations. Therefore, worker processes are the only entities observed in a card.

The system architecture of EPGTOP resembles similarities to the architectures discussed in the previous chapter. In particular, as in the continuous distributed monitoring (2.3.1), the communication channels are bidirectional and the observers do not communicate with one another. Sites map to worker processes in this system and we have a tree architecture as described in 2.3.3 where a monitoring service is a relay node. The coordinator can communicate with each monitoring service to share control information, such as error parameters, frequencies and periods, or state (restart, terminate, etc.). Contrary to the theoretical models, monitoring calculations are not performed per packet basis due to resource and performance

implications outlined in this section.

3.2.2 Measurements

EPGTOP aims to compute performance figures such as processor utilization and packet processing rate for the whole system which contains the entire User Plane, subsystems (SSCs running in "user plane" role) and worker processes (running on User Plane SSCs) to accurately and efficiently display real-time information about the system performance at any time continuously. Processor utilization is the main metric to consider since each worker is assigned to and runs on a dedicated processor and its affinity does not change throughout the execution. The rest of this section discusses all the metrics used by EPGTOP.

Monitoring Events

Most system metrics such as packet processing rate or CPU utilization can be represented as a combination of counters and frequencies. The functional monitoring algorithms enable tracking counters and frequencies for a distributed system in a communication and space efficient manner. We treat observations occurring in this system as events and track the corresponding counts of these events for a fixed-sized sliding monitoring window. A sliding monitoring window provides an overview of the recent behaviour of the system and its size is configured by the coordinator. An important implication of tracking counts and frequencies is that the following system metrics are subsequently represented as a set of monotonic functions that are computed from these counts and frequencies.

Processor Utilization

The principal metric to monitor system utilization in EPG is to track CPU utilization. Section 3.1 described technical details about worker processes. An important property of this system is processor affinity, meaning each worker process is associated with a particular dedicated processor and its affinity is fixed throughout the execution. Therefore, utilization of a worker process can be defined as utilization of its corresponding processor.

Existing kernel methods only provide information about utilization of a specific worker either for the last second or one microsecond which is computed from the number of executed CPU cycles over time. We are interested in tracking the aggregate processor utilization of the entire system and individual cards in addition to worker processes for a sliding window of any size specified by the coordinator. Processor utilization is a non-monotonic value and can change rapidly over short periods of time. Therefore, our monitoring algorithm to track processor utilization works on a recent window of items defined in time units to compute an average value. A monitoring service keeps track of the frequencies of processor utilization from 0 to 100 for each worker process, meaning there are 100 entries per worker for each monitoring period. The local fetches performed in a monitoring period update the corresponding frequencies of the current period and a sliding window is maintained per worker to keep track of all the frequencies for a recent window.

The coordinator computes the average utilization for a recent window corresponding to a worker process by multiplying each individual frequency f_i with its respective weight and dividing them by F_1, the first frequency moment. Correspondingly, for a card, the average utilization is computed from the utilization of its workers, and for a system, the average utilization is computed from the utilization of all cards running with a “user plane” role.

The average processor utilization computed by the coordinator will be the focus of our analysis since we are more interested in the efficiency of transferring these frequencies and counts to the coordinator than the monitoring function computed by the coordinator. An advantage of not simply relaying only locally computed averages to the coordinator is that with these sets of frequencies, the coordinator can be extended to compute more monitoring functions such as min, max, mode and standard deviation for processor utilization without modifying the algorithms executed by the monitoring service. Essentially, a monitoring service is tasked only with retrieving local monitoring data and executing a monitoring algorithm, and the coordinator is tasked with processing the monitoring updates and displaying results of computed monitoring functions. Subsequently, since a monitoring algorithm is only executed by the monitoring services, additions of new algorithms or adjustments to current algorithms do not require modifying the coordinator.

Packet processing rate

Since the primary function of EPG is to process packets, we also monitor packet processing rate for the User Plane, individual cards running on this plane, and worker processes. Packet processing rate is defined as the count of processed packets over an interval. We compute the packet processing rate for a recent sliding window of items which contains the total number of processed packets in this window, but this definition can be extended to the whole-stream case by taking an infinitely large window.

Computing the rate requires counting the total number of processed items over a window defined in time units. For a single worker process, this calculation is straightforward to implement due to workers maintaining their own statistics. However, aggregate counts must be calculated to determine the rate for a card and the entire system. Note that, the packet processing rate is calculated by efficiently tracking the value of a counter. The coordinator at any given time is able to provide counts for workers, cards and the entire User Plane. Therefore, the coordinator can support calculating any rate that can be represented as a set of counts by utilizing the same algorithm to transfer these counts. The packet processing rate is the main focus of our analysis to discuss efficiency of tracking counters.

Extensions

The metrics described above are monitored with sets of frequencies and counts as explained in each corresponding section. An efficient functional monitoring algorithm aims to allow the coordinator to continuously track these values and provide approximations at any given time. Accordingly, EPGTOP can support monitoring of more metrics if these metrics can be reduced to be represented as a set of fre-

quencies and counts. Given that our focus is to mainly evaluate communication efficiency of monitoring, we focus on the amount of updates sent to the coordinator for the metrics defined in this section only.

3.3 Implementation

Due to architecture-specific details of EPG, both the coordinator and monitoring services are single-threaded applications. An event loop is a set of steps executed either by the coordinator or monitoring services to compute statistics based on a monitoring algorithm, update internal data structures and transfer monitoring data if required. The details of the event loops are described in this section.

3.3.1 The system

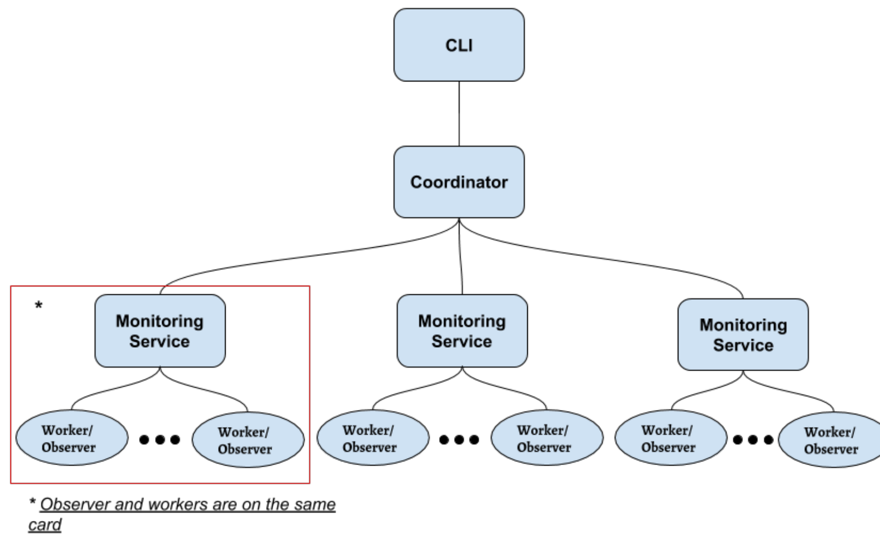

Figure 3.4: EPGTOP Architecture.

EPGTOP is a monitoring service for a distributed system that contains three main entities: the coordinator, the monitoring services and workers/observers. EPGTOP facilities the local monitoring corresponding to each worker to be transferred to the coordinator and displayed continuously in real-time. The coordinator is a dedicated process initialized first and runs as part of RP. The coordinator's responsibility is to process the updates received from monitoring services and continuously display system statistics. At any given time, the coordinator can be queried to retrieve statistics for metrics described in 3.2.2. Each monitoring service is run as

part of a card's control process and is responsible for reviewing internal statistics corresponding to workers, making monitoring computations and deciding for each worker whether to send updates to the coordinator.

Figure 3.4 provides an overview of the architecture. A monitoring service can be configured to run in two different modes: *basic* and *approximate*. The basic mode uses a simple periodic polling algorithm which requires each monitoring service to send updates at the end of all monitoring periods. The approximate mode requires each monitoring service to maintain error thresholds for monitoring statistics corresponding to each worker and a monitoring service sends updates when the error threshold cannot be maintained due to recent local fetches updating the tracked counters and frequencies. Therefore, the coordinator is able to provide approximations for the counts and frequencies of events corresponding to any worker and these approximations are only valid for the recent window whose size is configured by the coordinator when it initializes and shared with monitoring services during handshakes. The approximate mode reduces the amount of monitoring data that must be sent to the coordinator by trading off accuracy for efficiency. The error parameter is also shared during a handshake which will be explained later in this section.

3.3.2 Communication

Figure 3.5: EPGTOP message.

In order to facilitate the communication between the monitoring services and the coordinator, we devised a simple communication protocol with the goal of minimizing the overhead for transferring monitoring data. Generally, there exists two categories of messages: monitoring messages which are sent from monitoring services to a coordinator and control messages that are mostly sent in the opposite direction. **Figure 3.5** describes the structure of a message. All messages start with a message type which dictates in what manner the message should be decoded. The payload of a message may contain more messages. Broadly, there exists two types of monitoring messages: counts and frequencies. An update for a count has the form $\langle \text{tag}, \text{value} \rangle$ where the tag identities the worker this update corresponds to. Frequency updates have the form ⟨tag, item, value⟩ where the item number identifies the specific event. Frequency updates have significantly larger overheads than count updates due to this simple scheme of identifying specific events from updates. All message are sent with established TCP connections.

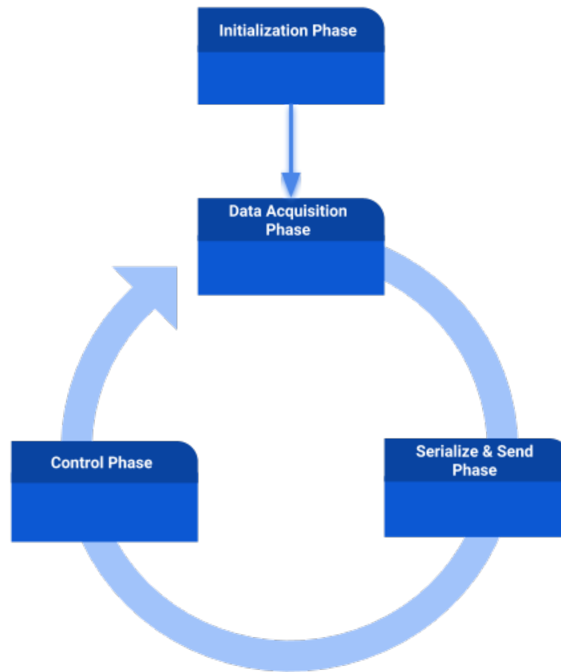

Figure 3.6: A monitoring service's event loop.

3.3.3 Monitoring service and observers

As mentioned above, an instance of our implementation of a monitoring service runs on each one of the User Plane cards. Specifically, a monitoring service is a single-threaded process that is launched and executed by a dedicated control process in each card and is tasked with handling monitoring data for workers. **Figure 3.6** shows a visualization of the event loop of our monitoring service. In this subsection, we discuss the implementation in more detail, explain the initialization phase, the data acquisition process, the manner the service handles control data and mechanisms that were used to send updates to a coordinator.

Initialization

The monitoring service is launched automatically when a card boots by the control process. The address of the coordinator is retrieved with core system functions. Once the service is launched, it attempts to set up a connection with the coordinator and acquire the parameters of the monitoring algorithm it is executing. In order to establish the connection, the monitoring service sends a discover message to the coordinator which is always listening at a predefined port. The coordinator responds with an acknowledgement message that contains algorithm parameters such as the length of the monitoring window, the error parameters, etc. The handshake can be seen in **Figure 3.7**. The handshake is done over an established TCP connection. The monitoring service is configured to retry initiating a connection if the coordinator is not reachable. As soon as the the connection has been established, the monitoring service goes through an initialization phase, sets up the required data structures based on the mode it is running in for each worker and starts continuous

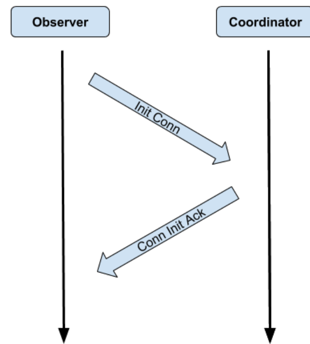

Figure 3.7: Initialization Handshake.

monitoring.

The event loop

Once the initialization process has been completed, the monitoring service enters its event loop which can be seen in **Figure 3.6**. The loop consists of three phases. In the data acquisition phase, the monitoring service retrieves monitoring data corresponding to each worker and processes the data. In the serialization and send phase, the internal data structures are reviewed to make monitoring decisions and if necessary, updates are sent to the coordinator. Lastly, in the control data phase, the service receives control data (if any) from the coordinator and applies the requested changes. The following subsections provide more details about each of the aforementioned phases.

Data acquisition phase

EPGTOP considers each worker process as a standalone observer with its own stream of updates and monitoring statistics. Due to the architecture of EPG, a worker process does not interact with the coordinator directly. A monitoring service that runs as part of the control process for a card acts on the behalf of each worker on that card to execute a monitoring algorithm. In order to collect monitoring data corresponding to each worker, a monitoring service loops over the list of worker processes and polls them with system functions to retrieve the data.

In section 3.2.1, we described the time-line seen by a monitoring service. A monitoring window consists of monitoring periods and each period consists of polling periods. Monitoring windows are sliding windows defined in time units and are maintained to report the recent value of a monitored statistics such as processor utilization. The number of times a monitoring service retrieves and updates its internally stored monitoring data for each worker is determined by the configured

polling frequency. A monitoring service sleeps after polling of all workers is complete and is awaken by the control process. A monitoring period ends when the number of polls in that period is equal to the polling frequency. Once a monitoring period ends, the monitoring service slides the monitoring window and executes a monitoring algorithm. The use of more complex methods to retrieve data on a card was considered but deemed as not necessary, since the control process can access the data structures in shared memory to acquire the data with minimal costs. This adjustment does not affect the monitoring algorithms that are executed since the amount of updates sent for all workers on a card in total is equivalent to having a dedicated monitoring service for each worker. Dedicated monitoring services would have consumed significantly more resources and executing them as part of the workers would have affected the performance of workers. The performance of workers is critical in this system and having an external process to perform monitoring is to not disrupt the processing of packets by the workers.

In section 3.3.1, we briefly described two modes a monitoring service can be run in: *basic* and *approximate*. Acquiring of local data is the same for both modes but the manner monitoring windows are maintained and the coordinator is updated is different. Additionally, all monitoring related actions such as execution of monitoring algorithms and maintaining the sliding windows is performed by a monitoring service. More precisely, the coordinator is only aware of the mode a monitoring service is configured to run and does not know the details of the algorithm it is executing. The data structure utilized by the coordinator to store the aggregate counts and frequencies corresponding to the current sliding window for each worker is the same for both modes. Division of responsibilities allows supporting monitoring algorithms that do not require a global state to be shared, thus making modifications to or replacement of an algorithm a service is executing to be performed without affecting the implementation of a coordinator.

Basic Mode In basic mode, a monitoring service maintains a sliding window for each monitored statistics such as processor utilization for a worker which corresponds to the following data structure:

Listing 3.1: The data structure for current window.

```
struct data {
    uint64_t packet_count;
    uint64_t cpu_util[100];
    // internal fields not shown ...
}
```

In 3.2.2, we described the mapping of frequencies and counts in this data structure to metrics. Consider that the monitoring window size is five and the current recent window contains items $\{4, 20, 34, 10, 2\}$ which correspond to the number of packets processed in each monitoring period. Accordingly, the value of *packet_count* in the data structure will be $4 + 20 + 34 + 10 + 2$. As explained in the preceding section, the last poll in a monitoring period ends that period and the monitoring service slides the window by dropping the first item, 4 in this case, and adding the arriving item. A coordinator will also maintain this exact data structure for each

worker, but its data structure is only modified based on updates received from a monitoring service.

It is evident that the method described above requires storing all items for the current window to be able to drop and add items. The internal data structures to store each unit in a window is not shown for simplification. Consequently, maintaining all items in a recent window of size n requires $O(n)$ words of memory, where n is the number of items. However, a single item in EPGTOP can contain several fields such as counts and frequencies.

When a monitoring window slides, the monitoring service compares the values of the fields in the data structure above to the values sent last time. Whenever a difference is found, the field is flagged. This is a small optimization over sending all items at the end of all monitoring periods. In "serialize and send" phase, the service packages all flagged items into a monitoring message and tags each value to enable the coordinator to distinguish updates corresponding to specific counts, frequencies and workers.

Approximate Mode In the approximate mode, we do not store all items in a recent window. Instead, we use an *exponential histogram* described in [13] to approximate the current counts and frequencies for the window. The histogram is more space efficient and requires $O(\frac{1}{\epsilon}(log\, n + log\, r)(log\ n))$ bits of memory where r is the range of elements of an item such as a count or frequency. The histogram's efficiency is evident when large window sizes are used. The approximation parameter is retrieved from the coordinator during the handshake. In both the basic and approximate mode, we use the same data structure in 3.1 to store the aggregate data for the current window for each worker. As in basic mode, the coordinator maintains the same data structure for each worker when monitoring services are configured to run the approximate mode.

To reduce the number of updates sent, the approximate mode runs the "up/down algorithm" described in 2.5. An error threshold is maintained for each monitored count and frequency. When a monitoring period ends, the monitoring service inserts the data corresponding to that period into associated histograms. Histograms internally maintain a sliding window and report the current value of a monitored count or frequencies for that window. The monitoring service uses the inequalities from the algorithm to compare the current values with the values sent to the coordinator last time to determine whether omitting to send an update for a count or frequency to the coordinator will result in the corresponding reported value by the coordinator having an error exceeding the threshold. Such counts and frequencies are flagged to be sent in the "serialize and send" phase. Hence, in the approximate mode, the service can skip sending updates for some of the counts and frequencies to reduce the number of monitoring updates sent. An important detail to consider is that the error threshold is set by the coordinator and the monitoring services must guarantee that this threshold is maintained. When presenting our results, accuracy of the reported values by the coordinator will be considered to determine if monitoring services are able to maintain the error thresholds. The thresholds dictate the correctness of this mode, since without the guarantees, the monitoring services would be able to avoid sending updates altogether, thus providing better communication efficiency,

but resulting in values by reported by the coordinator being severely outdated.

Serialize and Send Phase

During the serialize and send phase, a monitoring service checks the data structure shown in 3.1 for each worker to find flagged fields. A monitoring message is created that contains updated values of the flagged items. The updated values are tagged to associate each update with a specific count, frequency or worker. The monitoring service packages updates of all workers into a single message to avoid TCP overhead of sending multiple messages for each worker.

Control Data Phase

During the control data phase, the monitoring service checks to see if there has been any control messages received from the coordinator. The control messages dictate the actions must be executed. These actions can be state changes such as terminate, restart etc. or updates to the algorithm parameters such as the monitoring window size, polling frequency etc.

3.3.4 Coordinator

A coordinator is a stand-alone process running in RP. The aim of the coordinator is to process updates from monitoring services and display monitoring statistics for the entire system, cards and workers continuously. The coordinator is initiated with setting up internal data structures to handle connections and binding to a predefined port. The coordinator is also run in either basic or approximate mode. There is no difference in handling of requests for two modes, but the mode determines the parameters the coordinator needs to provide during the handshake, since the coordinator does not execute any monitoring algorithm and merely processes the updates received. The coordinator requires all monitoring services to run in the same mode, thus rejecting connections from services started with a different configuration.

Initialization

A coordinator is initiated with a configuration object that defines the parameters of the mode the monitoring services must be run in. The coordinator is always listening at a predefined port and its address is discovered with system functions since the process is part of RP. **Figure 3.7** depicts the manner a handshake is performed. After the coordinator binds to the port, a monitoring service can initiate a handshake. The coordinator will include parameters of the mode the monitoring services are configured to run in the acknowledgement message. After the handshake, internal data structures are constructed and the service is added a list the coordinator uses to detect new messages with epoll. The coordinator enters its event loop after its initialization is complete.

The event loop

As mentioned in the previous sections, the coordinator is also a single-threaded service. After the initialization, the coordinator runs the event loop shown in **Figure 3.8**. The events are detected by checking an *epoll* instance. An event can either be a new connection, an update from a monitoring service or a command received from the public interface. The public interface allows the coordinator to be configured dynamically with a command line tool.

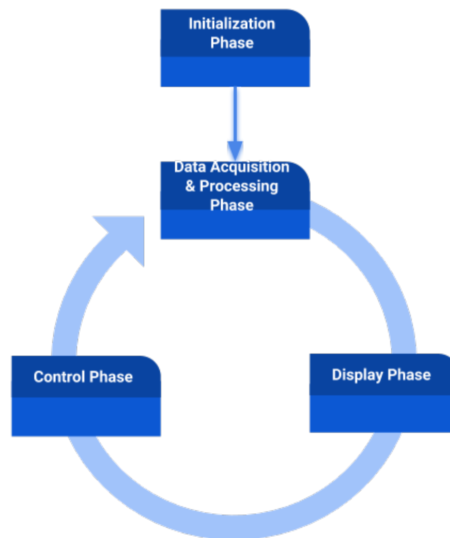

Figure 3.8: The Coordinator's Event Loop.

Data Acquisition and Processing Phase

A coordinator in this phase processes the messages that have been sent to its socket related to monitoring. A message can either be a connection message from a monitoring service or an update from an already connected monitoring service. A connection message retrieved from a monitoring service is handled with running the initialization step for that service. After initiation, the service can choose to send updates to the coordinator at any time. The coordinator is responsible to maintain its own data structures for each worker in the entire system consistent with the same data structure stored in the corresponding monitoring service. An eventual consistency is achieved with processing updates received from the monitoring services.

In addition to connection messages, monitoring messages are handled in this phase as well. Regardless of the mode a monitoring service is running in, each monitoring message is decoded by the coordinator and the updates in the message are applied to the corresponding internal data structures in the same manner. The

coordinator neither executes a monitoring algorithm nor maintains a sliding window for workers, since all algorithm related operations are performed by the monitoring services. The data retained by the coordinator for each worker corresponds to the aggregate data for the current sliding window for that worker. In fact, with some implementation related differences, the coordinator maintains the same data structure shown in 3.1 for each worker. Therefore, the monitoring services are responsible to send monitoring messages to update the counts and frequencies maintained by the coordinator.

The parameters of the mode the monitoring services are configured to run dictates the frequency of the monitoring messages that will be received but the coordinator never interferes with the method a monitoring service is using to send monitoring messages. In the basic mode, monitoring updates will be retrieved from each monitoring service for each modified count and frequency at the end of a monitoring period. In the approximate mode, some services may choose to not send monitoring messages even if the locally maintained counts and frequencies were modified at the end of that period. The error parameter supplied to the coordinator as part of its configuration affects the accuracy of results and frequencies of updates sent by monitoring services running in this mode.

Control Phase

A command to configure the coordinator can be received from a command-line tool only. During the data acquisition and processing phase, if a command is retrieved, then the coordinator enters the control phase, where it encodes and sends a control message to all monitoring services in the system. The coordinator does not wait to receive acknowledgements regarding control messages. Any error detected during the event loop forces the coordinator to terminate connection to the corresponding monitoring service and destroy all the associated internal data structures.

Display Phase

The display phase actually does not have a direct mapping in our implementation. At any point in time, the coordinator has access to data structures containing counts and frequencies of events corresponding to each worker. Therefore, the coordinator can be configured to display monitoring statistics in any manner. For each connected monitoring service, the coordinator keeps specific data structures to be able to dynamically extract monitoring statistics. As an example, average CPU utilization for a worker, a card or the system is retrieved through this data structure. Updates to a count or frequency forces all the associated internal data handlers to also flush and recalculate monitoring statistics that depend on that count or frequency. Section 3.2.2 explained possible extensions to support more measurements. These extensions are supported with this exact data structure that is responsible for computing monitoring statistics from sets of counts and frequencies. It is important to mention that, if the monitoring services are running in the approximate mode, then the values returned by handlers are merely approximations of the actual system statistics constrained to be accurate up to the configured error threshold. The following code contains examples of simplified interface methods to retrieve processor utilization:

Listing 3.2: Example methods in the interface to extract monitoring statistics.

```
double averageProcessorUtilization(int card, int worker);
double averageProcessorUtilizationCard(int card);
double averageProcessorUtilizationSystem();
```

3.4 Limitations

The implementation's limitations are mainly attributed to real-time and complex behavior of distributed systems. Unlike in the theoretical models, we can not assume that the connections and transfer of messages are instantaneous. Network delays are difficult to predict and out of the scope of this thesis project to account for. Therefore, the reported values by the coordinator are eventually consistent. We assume that monitoring messages sent to the coordinator at the end of a monitoring period are processed by the coordinator before the next monitoring period ends since the coordinator will report a stale value until a monitoring update is processed. Inserting time-stamps into monitoring messages to allow the coordinator handle messages arriving late due to network delays would have required globally syncing all monitoring services and is complex to implement for monitoring purposes.

The theoretical algorithms assume that observers are able to act instantly when an event is generated. Although possible from an implementation perspective, this would have significantly increased the computation costs of monitoring, making it linear to the number of packets received. The exponential histogram we have implemented has a substantial cost for insertions when compared to the simple sliding window maintained in the basic mode. Executing a monitoring algorithm, inserting items to histograms, and evaluating inequalities for each processed packet would have had significant effects on the performance of workers if monitoring was performed per packet basis. To maintain the same service level agreements of an EPG system after addition of a monitoring service such as response time, either the monitoring service would have needed to be run on dedicated processors or the number of workers increased if monitoring was incorporated into worker processes. Therefore, we relaxed the requirement and allowed the monitoring service to poll workers to retrieve and aggregate the generated observations occurred since the last poll. As explained previously, both the polling frequency and the length of a monitoring period are configurable implementation parameters. Also, neither the implementation nor our analysis take into account delays in processing of the data. Both serialization and de-serialization operations can induce significant delays in processing of large amounts of data. Our approach to alleviate computation related processing delays is to use a monitoring period large enough to make these costs insignificant but also small enough not to significantly reduce the accuracy of reported values by the coordinator. Consequently, with larger polling frequencies, more monitoring data can be retrieved but monitoring will require more computations. With smaller monitoring periods in size, the monitoring services make more monitoring decisions and are bound to send more monitoring messages. Accordingly, in our analysis of monitoring services, accuracy will simply correspond to the level of accuracy guaranteed by the monitoring services and whether these guarantees are maintained

throughout the execution. Parameters such as polling frequency will be set to fixed values for both modes for evaluation purposes.

Additionally, the system architecture of EPGTOP is based on the architectures discussed in chapter 2. Both EPGTOP and systems described in that chapter assume that a single coordinator as a sink node is capable of processing monitoring messages received from all observers. Such an assumption simplifies the analysis of the algorithms, since the number of coordinators affects neither the number of observations nor the number of monitoring messages transmitted. An EPG system running in production can have thousands of workers and hundreds of other system parameters may need to be monitored for such a system. A single-threaded coordinator will not have the capacity to handle monitoring data corresponding to all these workers. Given that the emphasis of our discussions is on the efficiency of monitoring algorithms by considering the number of monitoring messages sent, more robust implementations such as running a set of coordinators on dedicated processors to handle the monitoring data was considered as too complicated for the purpose of our evaluations due to those implementations requiring us to modify core components of RP. Moreover, running more complex configurations for the coordinator would not have affected the number of messages transmitted but the accuracy of results due to processing delays, since monitoring algorithms are only executed by monitoring services and number of workers and maintained statistics are the only contributors to the total number of messages sent.

Finally, EPGTOP itself is not fault tolerant and cannot handle a failed coordinator. However, fault tolerance of any component is already provided in EPG. The coordinator is able to detect basic communication errors and monitoring services that are sending messages with incorrect headers and it is configured to terminate communication channels to such monitoring services and perform clean-up of its own data structures. Nevertheless, EPGTOP cannot recover from system failures and a monitoring service on a failed card reconnecting after a reboot will be considered as a different monitoring service.

4 Results

EPGTOP makes use of core functions provided by EPG to retrieve local system statistics that are subsequently used to execute monitoring algorithms to make monitoring decisions and send updates to the coordinator. Evaluations and assessments are necessary to determine whether monitoring services are capable of maintaining the configured error bounds on reported values and the amount of efficiency achieved with various configurations. The chapter starts with a description of an EPG system and monitoring configurations used to generate the results and subsequently provides graphs and our discussions of the results.

4.1 Configuration

The empirical evaluation of the algorithms incorporated into EPGTOP were performed on an EPG system configured with two cards containing 146 workers in total. Accordingly, the system has a single coordinator and 146 workers/observers generating observations. A stability test provided by the team working on EPG was used to generate and inject packets into the system. The test generates actual packets and consists of two main stages: ***increasing load*** and ***stable load***. In the increasing load stage, the number of packets injected into the system is increased over time until a 20-minute time mark. After the 20-minute time mark, the stable load stage starts executing. In this stage, the test stabilizes and the number of packets injected remains the same. The total number of packets sent to the system with this test is deterministic but behaviour of sub-components of a running EPG system to route, assign and handle packets is not. The implication of this is that during multiple runs, any set of selected workers can be assigned different work-loads due to internal and inter-card load balancing logic. Therefore, we are primarily focusing on evaluating and comparing aggregate system monitoring functions for various configurations since only the aggregate behaviour of the entire system will remain about the same between runs.

Due to variances in this system's behaviour during multiple runs, monitoring services were extended to run both the basic and approximate mode concurrently. The coordinator utilizes a message tag added to each monitoring message to differentiate updates corresponding to each mode and maintains monitoring statistics separately. In essence, the extension allows inspection of the system behavior to directly compare results of a single configuration for the approximate mode to the basic mode for that run. A fixed window size of ten seconds was used for all configurations for evaluation purposes. Due to the approximate mode sending updates depending

on the amount of change to each monitored parameter, larger window sizes have an effect that the approximate mode sends far less messages than the basic mode, particularly in the stable load stage, since as the window size grows, the amount of change required to necessitate sending of a monitoring message also increases. The approximate mode was run with three different error thresholds: $\varepsilon = 5\%$, $\varepsilon = 10\%$ and $\varepsilon = 20\%$. Each simulation was run for 35 minutes in total and the timings are normalized in our graphs to a fixed time point starting at "00:00" to make comparisons easier. 35 minute simulations contain the increasing load stage which takes approximately 20 minutes to execute and the first 15 minutes of the stable load stage. Simulations longer than 35 minutes were considered and run during testing but given that the behaviour of the entire system in the stable load phase remains the same, the results corresponding to monitoring data after the 35-minute mark do not provide any new findings.

During booting of the cards when the monitoring services have still not connected and when the script is initializing, the coordinator reports zeroes for all counts and frequencies. These data points were removed from the generated data for our evaluation of all configurations. The observers were configured with a monitoring period of 1 second and polling frequency of 1000. These parameters indicate that a monitoring service makes 1000 local fetches for each worker per second and makes a monitoring decision after the last fetch. We used a polling frequency of 1000 since the system functions used for local fetches have a granularity of 1ms. Therefore, we are able to record monitoring statistics for the entire duration of the monitoring periods for evaluation purposes.

In order to compare results corresponding to basic and approximate mode, we considered two main system statistics: processor utilization and packet processing rate. The main goal of these simulations is to determine whether monitoring services running in approximate mode are more communication efficient than the services running in basic mode while maintaining the error bounds on reported values. Additionally, we are interested in determining the amount of difference in the number of monitoring messages transmitted when larger error thresholds are used. Note that due to the coordinator computing the average processor utilization and packet processing rate from a set of counts and frequencies, we do not provide graphs for additional monitoring functions such as min, max, and standard deviation since the monitoring function is only computed and reported by the coordinator and the choice of a monitoring function does not have an effect on the number of monitoring messages sent for a particular system parameter such as processor utilization.

4.2 Results

4.2.1 Processor utilization

Average processor utilization is computed from a set of frequencies as explained in 3.2.2. **Figure 4.1, Figure 4.2** and **Figure 4.3** present the average system processor utilization reported by the coordinator for both of the modes for various configurations. In each run of the simulation, monitoring services execute computations and algorithms for the basic mode and a single configuration of the approximate mode.

The approximate mode was run with three different error thresholds: 5%, 10% and 20%. All simulations were run with a monitoring window of ten seconds and the reported utilization is the average utilization computed over this window. The graphs show that all configurations of the approximate mode have similar trends as the basic mode in terms of increase in processor utilization over time. The processor utilization of the system for all configurations does not demonstrate any rapid increases nor decreases throughout the runs and remains approximately the same in the stable load stage.

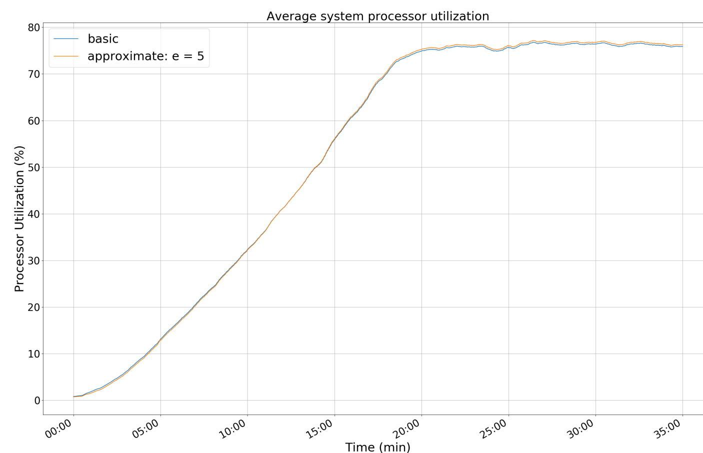

Figure 4.1: Average system processor utilization: $\varepsilon = 5\%$.

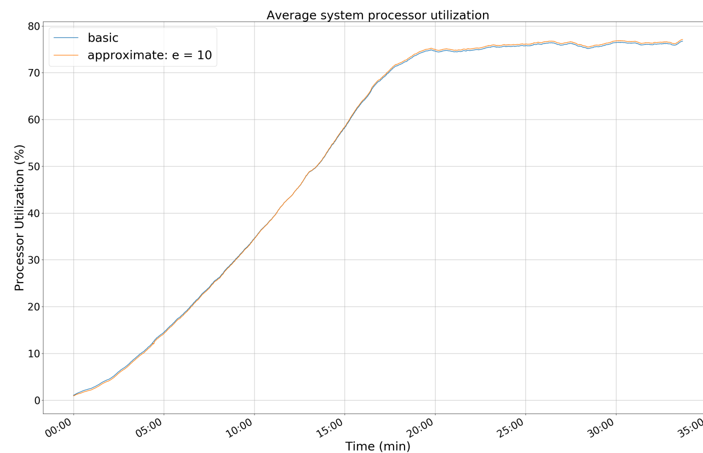

Figure 4.2: Average system processor utilization: $\varepsilon = 10\%$.

Evidently, graphs indicate that the difference in reported values corresponding to the basic and the approximate mode do not differ significantly when larger ε values

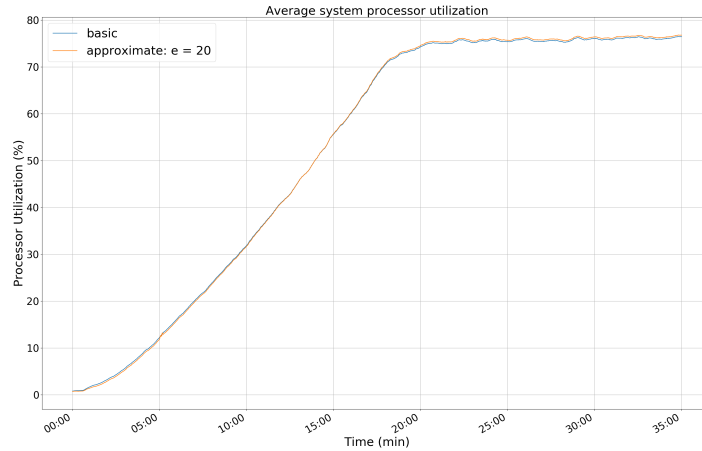

Figure 4.3: Average system processor utilization: $\varepsilon = 20\%$.

are used. During the increasing load stage, the changes are significant enough to require all configurations regardless of the error threshold used to send monitoring updates to the coordinator. A limitation of monitoring services is that processor utilization is represented as a set of frequencies to enable executing continuous monitoring algorithms for monotonic functions to track both the system and worker processor utilization. Therefore, changes to each frequency is tracked individually and requires monitoring updates to be sent even in the stable load stage irrespective of these changes' effects on the computed utilization for the system, which is performed by the coordinator. Consequently, a monitoring service when tracking a large set of frequencies for an individual worker will under-perform in terms of communication-efficiency compared to tracking a single count.

Figure 4.5 shows the running average relative error for system processor utilization computed over the last 5 minutes. As the graphs demonstrate, the configured error guarantee is maintained throughout the runs for $\varepsilon = 10$ and $\varepsilon = 20$ configurations, but the relative error exceeds the threshold for $\varepsilon = 5$ configuration at few time points in the first few minutes of the corresponding run. Even though calculations are performed with floating-point numbers when executing the algorithms, conversions to integers are done when sending monitoring updates to the coordinator, which results in rounding errors. These data points are difficult to interpret from the previous graphs due to the absolute error being less than 0.4 in the first few minutes. Additionally, the difference in the relative error corresponding to different configurations at many time points is quite small in terms of value. Maintaining the processor utilization using frequencies is requiring the monitoring services to send monitoring messages frequently even in the stable load stage, thus making the configurations demonstrate similar behaviour throughout the runs. Next, we need to consider the amount of monitoring messages sent to track processor utilization for each configuration to compare for communication-efficiency.

Monitoring services package multiple updates corresponding to monitored pa-

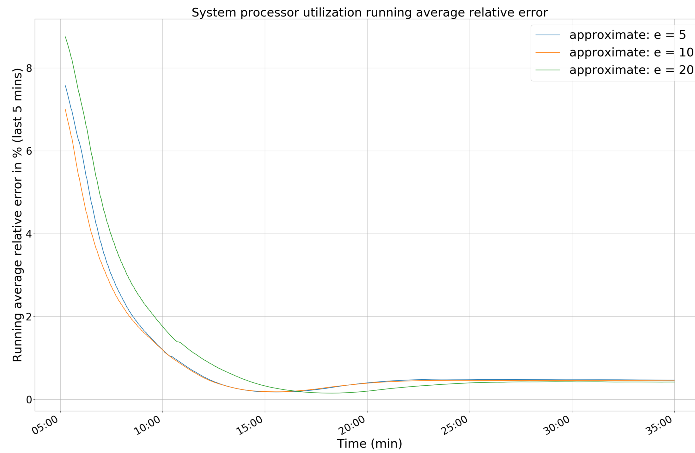

Figure 4.4: Running average relative error for system processor utilization.

rameters for each worker into a single message to avoid TCP overhead of transmitting multiple messages. The structure of monitoring messages was explained in 3.3.2. Therefore, a single message may contain from few to hundreds of updates. The number of updates in a message directly determines the size of the message. EPGTOP represents monitoring updates for counts and frequencies differently. Nonetheless, the size of a monitoring update is fixed and depends on its type.

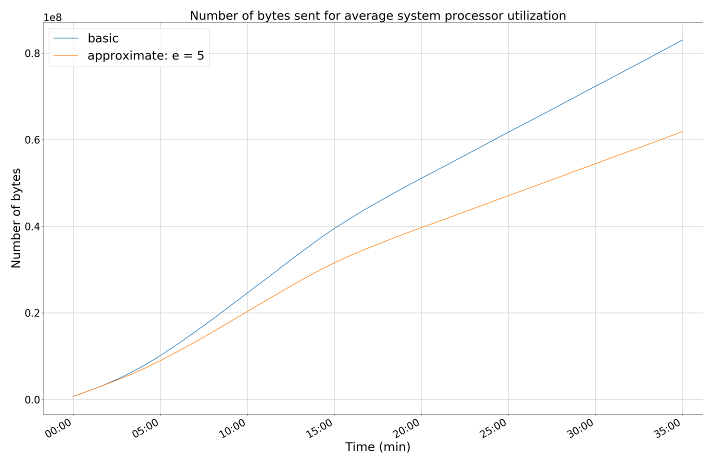

Figure 4.5: Amount of monitoring data for processor utilization: $\varepsilon = 5\%$.

Figure 4.5, Figure 4.6 and **Figure 4.7** present the total number of bytes of data sent for various configurations. The total number of bytes sent corresponds to the communication-efficiency of a specific configuration. In our implementation, each monitoring update for processor utilization is a tuple of three items and has

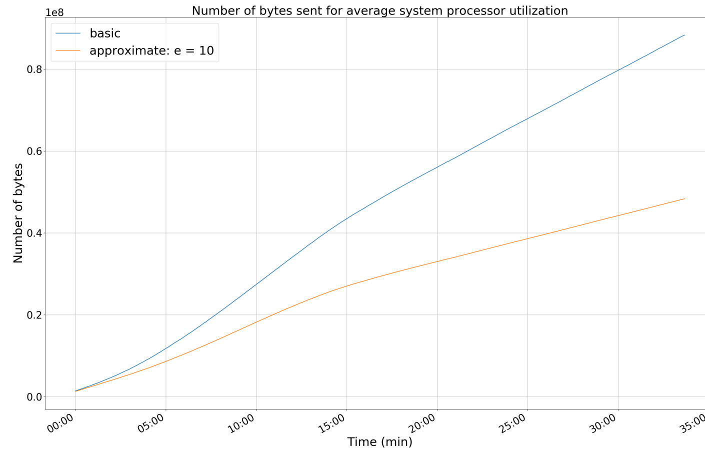

Figure 4.6: Amount of monitoring data processor utilization: $\varepsilon = 10\%$.

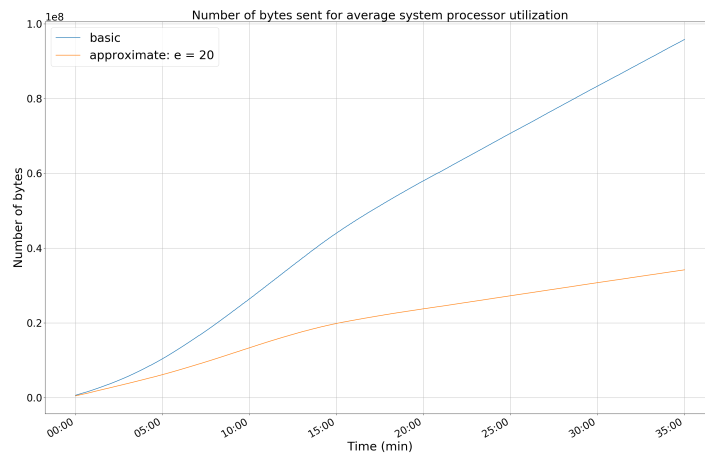

Figure 4.7: Amount of monitoring data for processor utilization: $\varepsilon = 20\%$.

a fixed size. Seemingly, the approximate mode sends far less messages than the basic mode in all of the runs. An important consideration from graphs is that there exists a small variance in the total number of bytes sent when executing the basic mode during multiple runs. Due to the complex nature of EPG, controlling load assignment and handling of packets for evaluation purposes is not possible without significant changes to EPG. Nevertheless, the results showcase that the total number of bytes sent significantly decreases as larger ε are used. In terms of communication-efficiency, $\varepsilon = 20\%$-configuration performs the best and sends the least amount of messages.

In all simulations, the total number of bytes sent when executing the basic mode increases almost linearly and the slope does not change significantly over time. In

comparison, after the 15-min mark, the slopes corresponding to the configurations of the approximate mode starts to become smaller as larger ε are used. The stable load stage of the simulations starts after the 20-min mark and the approximate mode is expected to be less affected by changes in this stage. The graphs demonstrate that the amount of data sent per monitoring period decreases gradually over time as the system stabilizes when running in the approximate mode.

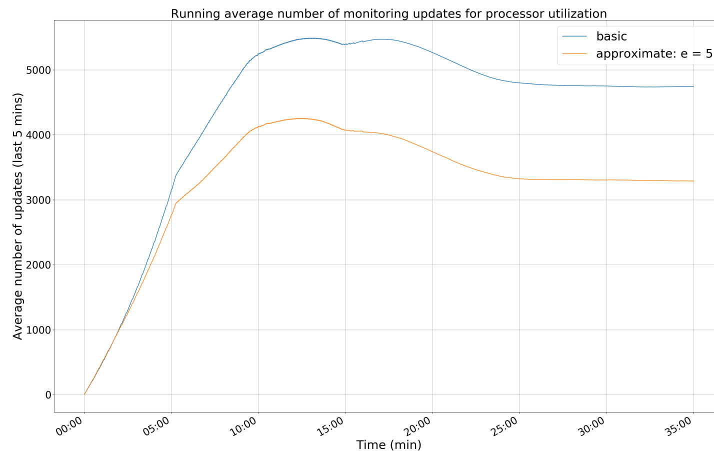

Figure 4.8: Processor utilization, average update count: $\varepsilon = 5\%$.

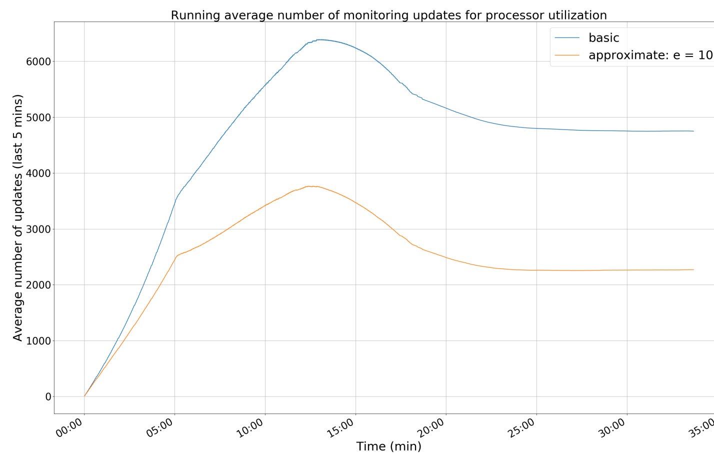

Figure 4.9: Processor utilization, average update count: $\varepsilon = 10\%$.

Figure 4.8, Figure 4.9 and **Figure 4.10** present the running average number of monitoring updates sent for each configuration for the last 5 minutes. The graphs demonstrate that the rate of the number of monitoring updates sent in the increasing load stage reduces as larger ε are used. When the system load increases over

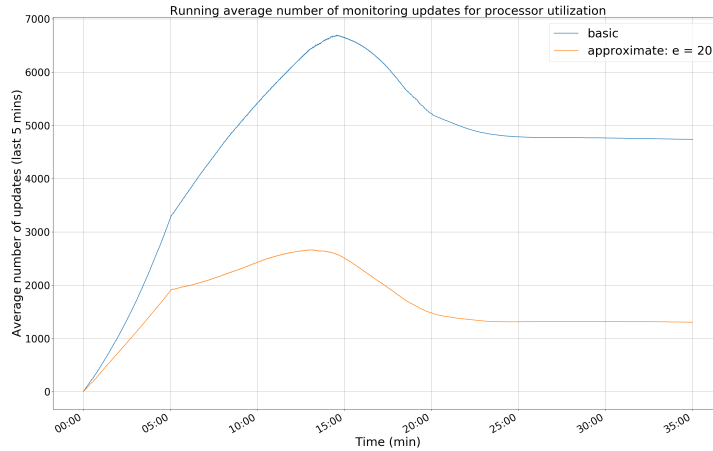

Figure 4.10: Processor utilization, average update count: $\varepsilon = 20\%$.

time in this stage, the processor utilization of individual workers will correspondingly increase, thus resulting in monitoring services sending monitoring updates for all workers. Larger ε-thresholds require larger changes to monitored parameters to necessitate an update to be sent, hence the running averages decrease as larger thresholds are used. All graphs demonstrate that after the stable load stage starts, the running averages for all configurations remains roughly the same. The previous graphs for the average system processor utilization showed that the system's utilization even in this stage varies slightly over time, indicating that monitoring updates will still need to be sent in this stage. Consider **Figure 4.11** and **Figure 4.12**, which provide histograms to view the frequency of the number of updates sent per monitoring period in the last 15 minutes of the simulation for the basic and $\varepsilon = 5\%$-configuration.

The frequencies in the histograms indicate that in the stable load stage, the basic mode sends monitoring messages that contain far more updates than the approximate mode. Due to the fact that even when the load in the system remains the same, routing and assigning of packets to workers can cause small fluctuations in the processor utilization of workers. The small differences in the fetched local values causes monitoring updates to be sent in the basic mode regardless of the amount of change. In comparison, the approximate mode with a $\varepsilon = 5\%$-configuration is less affected by small changes to utilization values and at a given monitoring period, the differences in local frequencies become large enough to send monitoring messages for only a small set of workers, accordingly resulting in monitoring messages smaller in size to be sent.

Finally, **Figure 4.13** and **Figure 4.14** provide plots to analyze the mean and standard deviation of the number of updates per monitoring period for processor utilization for the entire duration of the simulations. The yellow lines in these box plots are the mean values and the vertical lines in the bar plots show the standard deviation. Graphs for all configurations show high variability in the number of

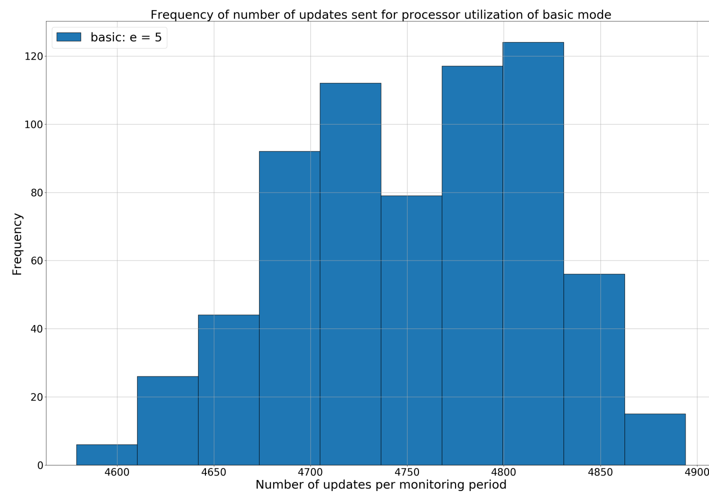

Figure 4.11: Processor utilization, frequency of updates (last 15 mins).

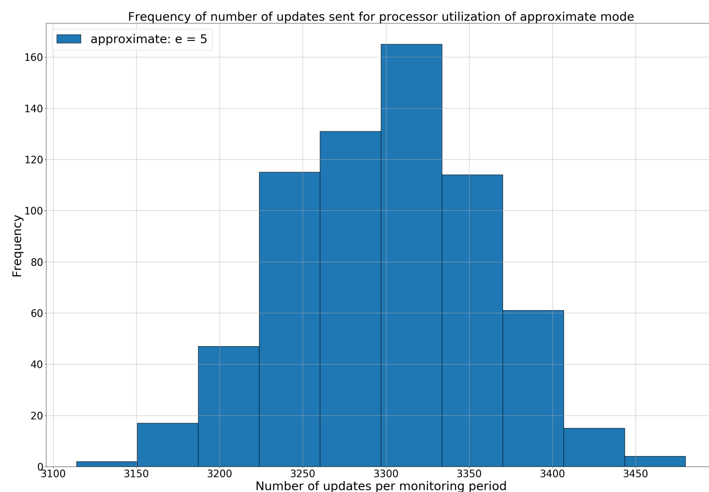

Figure 4.12: Processor utilization, frequency of updates (last 15 mins).

updates sent per monitoring period and this is mainly attributed to the increasing load stage of the test when the system load is gradually increasing and forcing all configurations to send updates. In spite of the variance in these graphs for the basic mode during multiple runs, the standard deviation of the basic mode is much larger than the standard deviation of the corresponding approximate mode. The box plots also contain many data points that are far away from the mean for the basic mode. In contrast, the behaviour of the approximate mode is much more consistent, since the approximate mode is more resilient to small differences in reported local values of frequencies for processor utilization.

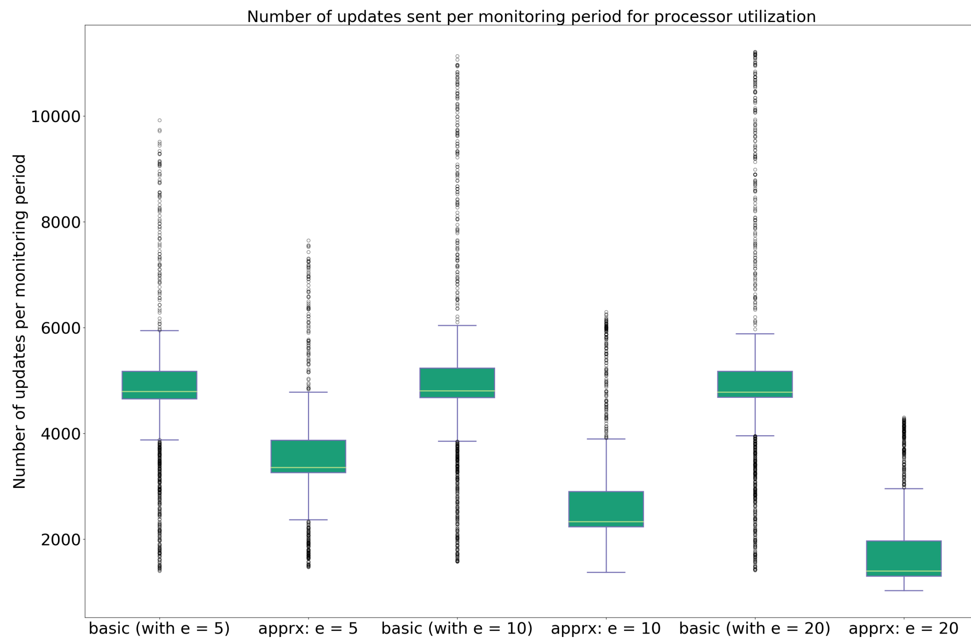

Figure 4.13: Processor utilization, update count per period: w = 10 sec.

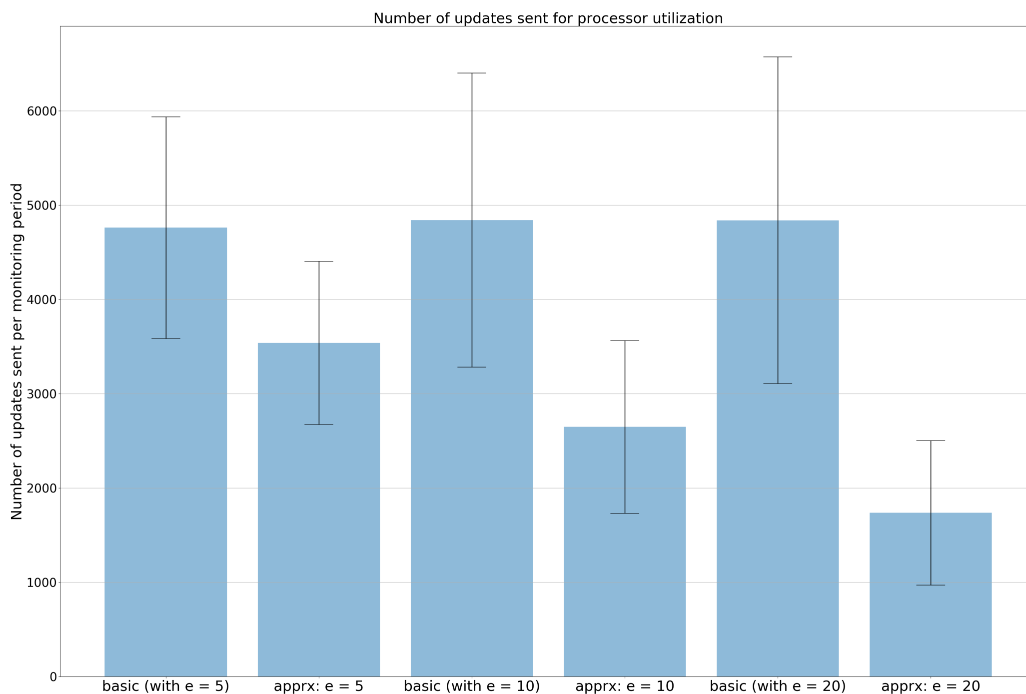

Figure 4.14: Processor utilization, update count per period: w = 10 sec.

Worker Processor utilization

The coordinator calculates the average system processor utilization continuously from the utilization of individual workers. EPG internally uses complex load-balancing logic to route traffic to workers. Therefore, it is not possible to compare processor utilization of workers in a set of runs, since the same worker may get assigned different amount of load during multiple runs. To evaluate both modes of

EPGTOP for individual workers, we consider the fastest and slowest workers determined by summing per period processor utilization in addition to the mean processor utilization of all workers. Note that the mean processor utilization is equivalent to the system processor utilization.

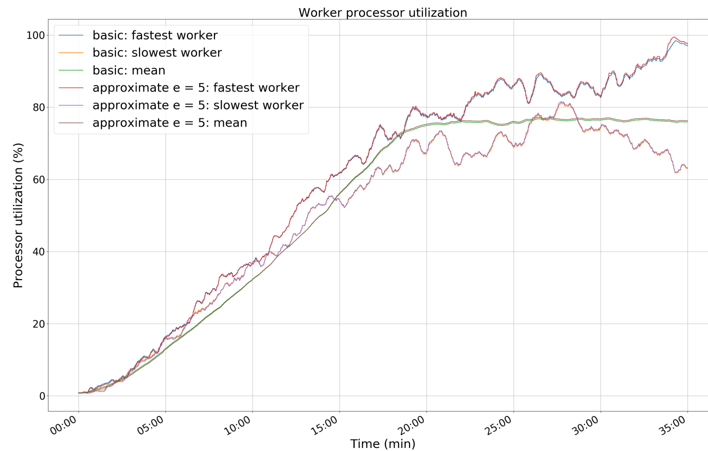

Figure 4.15: Processor utilization of workers.

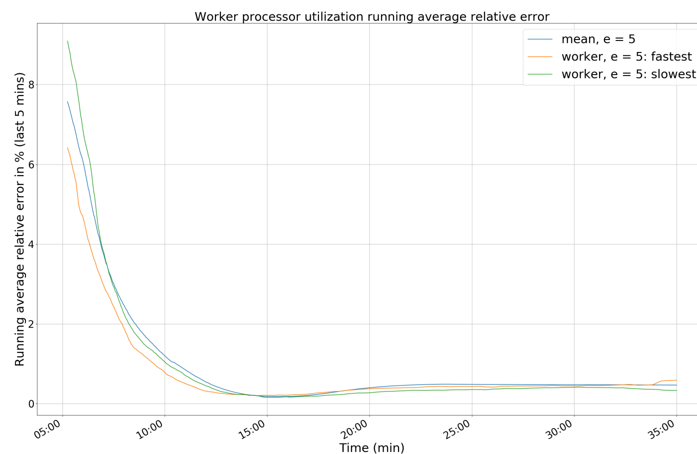

Figure 4.16: Running average relative error for worker processor utilization.

Figure 4.15 and **Figure 4.16** show the processor utilization of the fastest and slowest workers in the entire system for the basic and approximate mode and the corresponding running average relative error. The graphs indicate that there exists a significant difference in the processor utilization of the slowest and fastest workers. Additionally, unlike the mean system processor utilization, utilization of individual workers contain far more frequent spikes. The graphs also show that at many time

points, when the load of the fastest worker is increasing, the load of the slowest worker is decreasing. Due to the complexity of EPG, without explicitly pinning traffic to specific workers, it is not possible to determine whether this behaviour is indeed related to the load-balancing logic. The running average relative error for the workers is very similar to the relative error of the system processor utilization. Except the few time points in the first few minutes of the corresponding run, the error threshold is maintained for $\varepsilon = 5\%$ configuration for the most of the duration for both of the workers.

4.2.2 Packet Processing Rate

Packet processing rate is determined with tracking a counter continuously as explained in 3.2.2. **Figure 4.17, Figure 4.18** and **Figure 4.19** present the system packet processing rate reported by the coordinator for both of the modes for various configurations. In each run of the simulation, monitoring services execute computations and algorithms for the basic mode and a single configuration of the approximate mode. The approximate mode was run with three different error thresholds: 5%, 10% and 20%. All simulations were run with a monitoring window of ten seconds and the reported processing rate is the number of packets processed over this window.

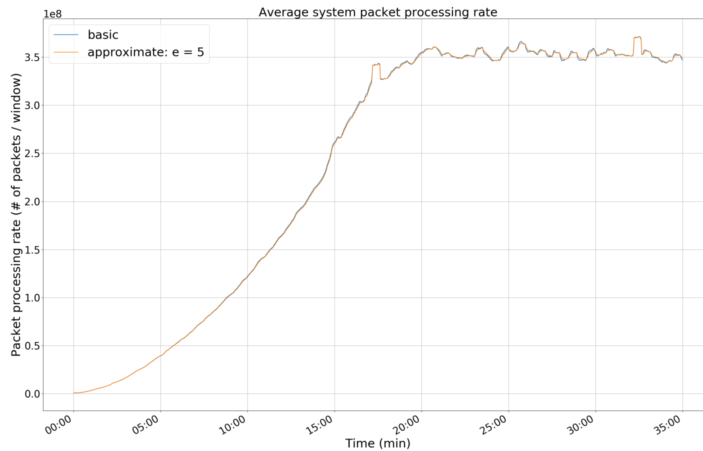

Figure 4.17: System packet processing rate: $\varepsilon = 5\%$.

The graphs for the basic mode depict that unlike the processor utilization, the packet processing rate varies over time in the stable load stage with frequent "ups" and "downs". Regardless of the amount of traffic sent to the system in this stage, EPG contains numerous components in addition to workers and assignment of packets to workers is not predictable. Monitoring packet processing rate with larger ε values results in the approximate mode being more forgiving to changes when compared to the processor utilization. Considering that there exists a single counter per worker to track the rate, the reported corresponding processing rate by the coordina-

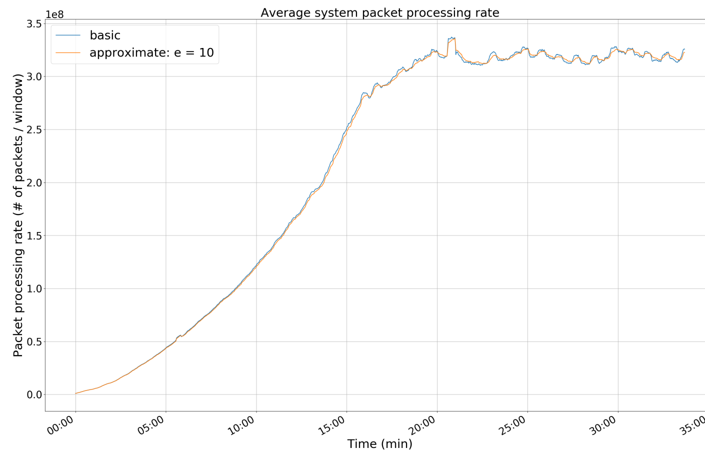

Figure 4.18: System packet processing rate: $\varepsilon = 10\%$.

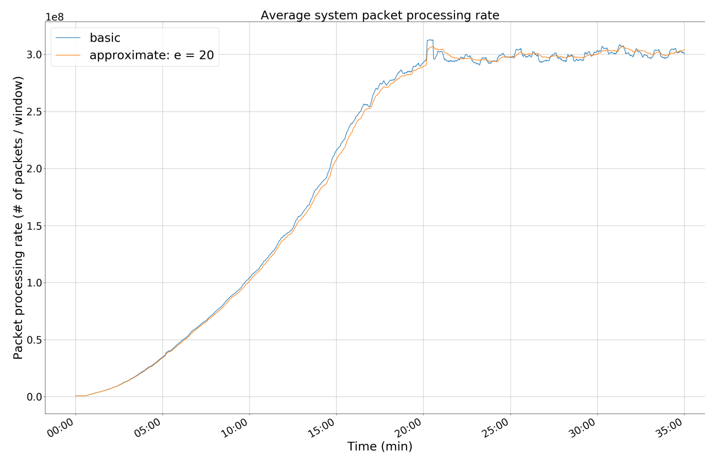

Figure 4.19: System packet processing rate: $\varepsilon = 20\%$.

tor is only affected by this counter and as the graphs indicate, $\varepsilon = 20\%$-configuration deviates the most from the results of the basic mode. Additionally, all configurations exhibit similar trends in the increasing load stage in terms of increase in the reported packet processing rate. Due to the number of packets injected into the system increasing consistently in this stage, monitoring services for all configurations will be required to transmit monitoring messages to update the coordinator.

Figure 4.20 shows the running average relative error for system packet processing rate computed over the last 5 minutes. As the graphs demonstrate, the configured error guarantee is maintained throughout the runs for all configurations of the approximate mode. In comparison to the processor utilization, the difference in the relative error corresponding to various configurations is much larger in value

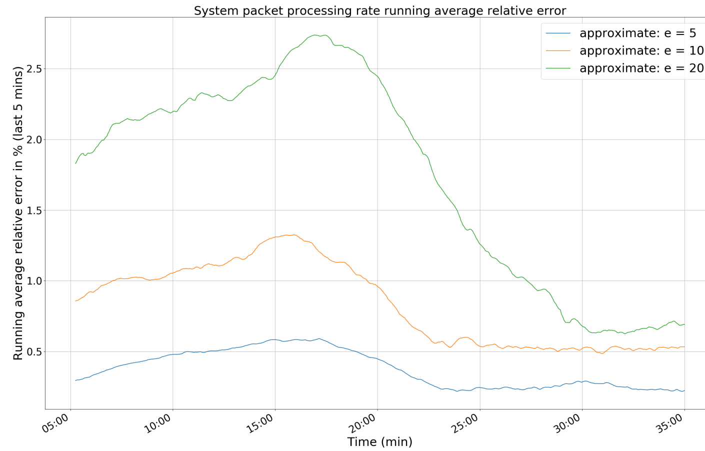

Figure 4.20: Running average relative error for system packet processing rate.

at most time points. Due to the packet processing rate for a worker being computed from a single counter, a monitoring decision to send an update is only affected by changes to this counter. Therefore, the system packet processing rate is affected by a much smaller set of counts than the system processor utilization and changes to individual worker packet processing rates are tolerated by monitoring services more effectively.

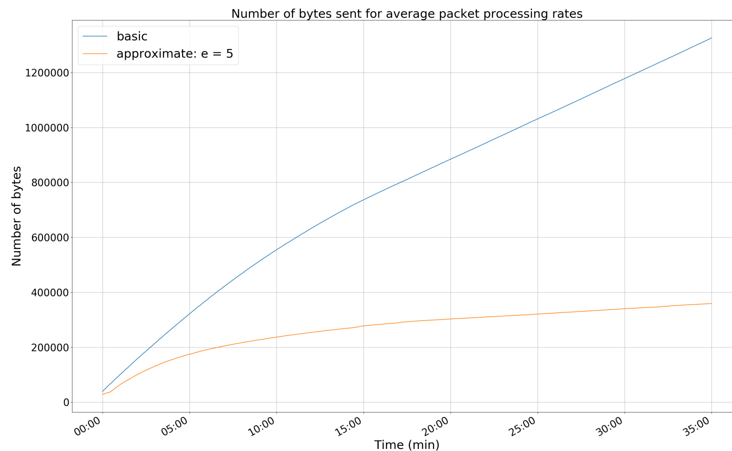

Figure 4.21: Amount of monitoring data for packet processing: $\varepsilon = 5\%$.

Figure 4.21, Figure 4.22 and **Figure 4.23** present the total number of bytes of data sent for various configurations. A monitoring update corresponding to packet processing rate is of fixed size and similar to processor utilization, updates are packaged into a single monitoring message before being sent to the coordinator.

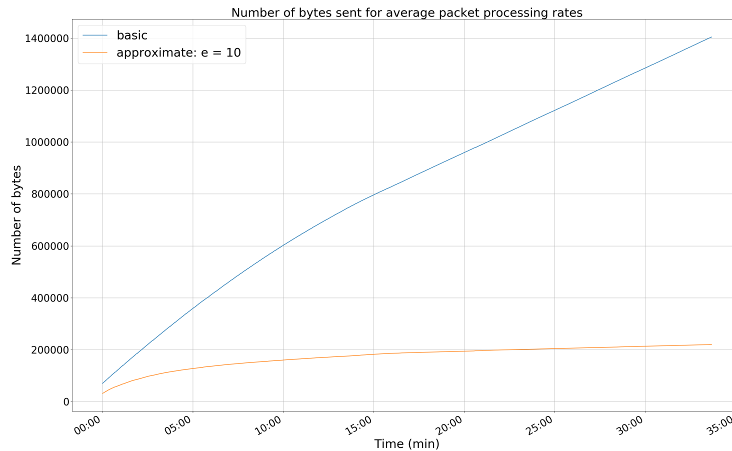

Figure 4.22: Amount of monitoring data packet processing: $\varepsilon = 10\%$.

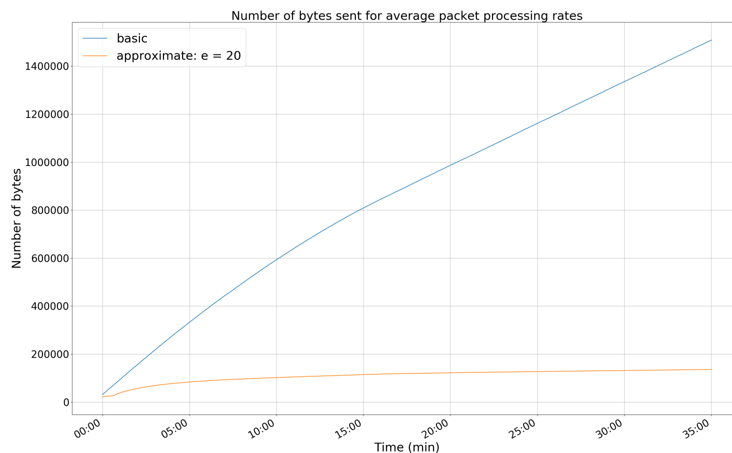

Figure 4.23: Amount of monitoring data for packet processing: $\varepsilon = 20\%$.

Regardless of the slight variance in the number of bytes sent for the basic mode during multiple runs, the approximate mode considerably reduces the amount of monitoring data that must be transmitted. The graphs for the approximate mode have much smaller slopes and unlike the basic mode, the total number of bytes sent does not increase consistently over time. Configuring the approximate mode with larger ε values mainly affect the rate in the increasing load stage, in particular, $\varepsilon = 10\%$ and $\varepsilon = 20\%$ configurations display minimal increase in the number of bytes sent in the stable load stage. Similar to the processor utilization, monitoring services in the stable load stage react far less frequently to changes as larger ε are used, since the amount of change needed to send a monitoring update becomes larger as the value of ε increases.

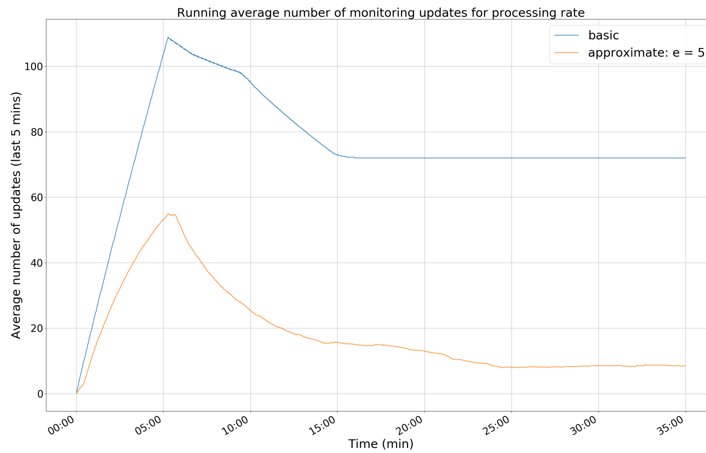

Figure 4.24: Packet Processing rate, average update count: $\varepsilon = 5\%$.

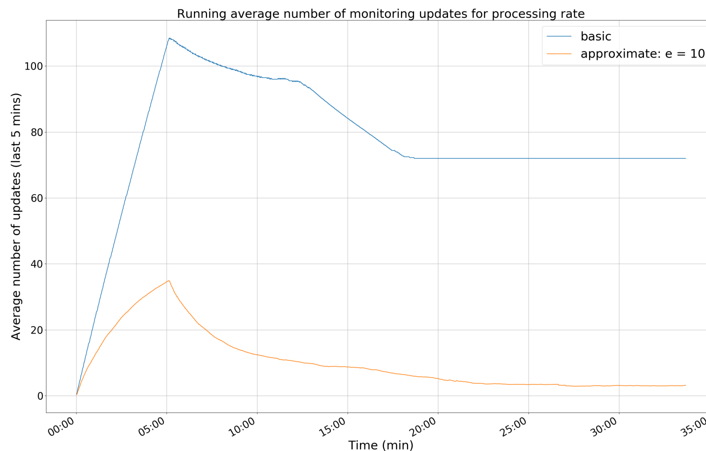

Figure 4.25: Packet processing rate, average update count: $\varepsilon = 10\%$.

Figure 4.24, Figure 4.25 and **Figure 4.26** present the running average number of monitoring updates sent for each configuration for the last 5 minutes. The graphs demonstrate that the rate of the number of monitoring updates sent in the increasing load stage increases rapidly in the first 5 minutes and subsequently, starts to slowly decrease over time. In the stable load stage, the running averages remain roughly the same for all configurations with larger ε-configurations approaching near zero values. Larger ε-thresholds mainly affect the running average in the increasing load stage. Gradual decrease in running averages over time is most likely attributed to the fact that as the system stabilizes over time, the difference in the number of assigned packets to a worker per monitoring period compared to the previous monitoring period will decline over time, thus requiring less monitoring messages to be

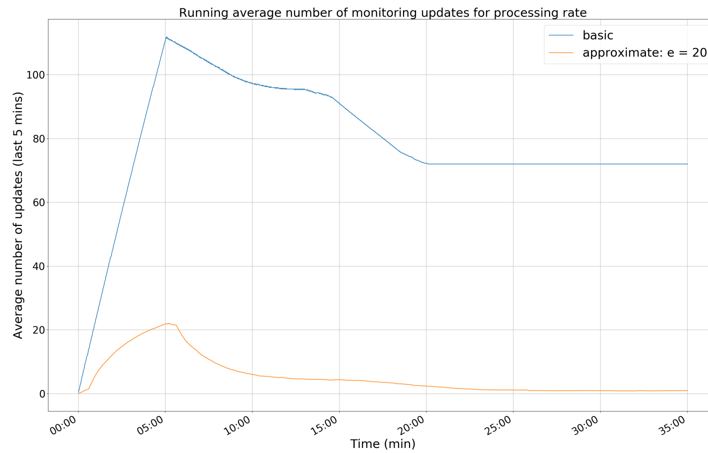

Figure 4.26: Packet processing rate, average update count: $\varepsilon = 20\%$.

sent. Consider **Figure 4.27** and **Figure 4.28**, which provide histograms to view the frequency of the number of updates sent per monitoring period in the last 15 minutes of the simulation for the basic and $\varepsilon = 5\%$-configuration.

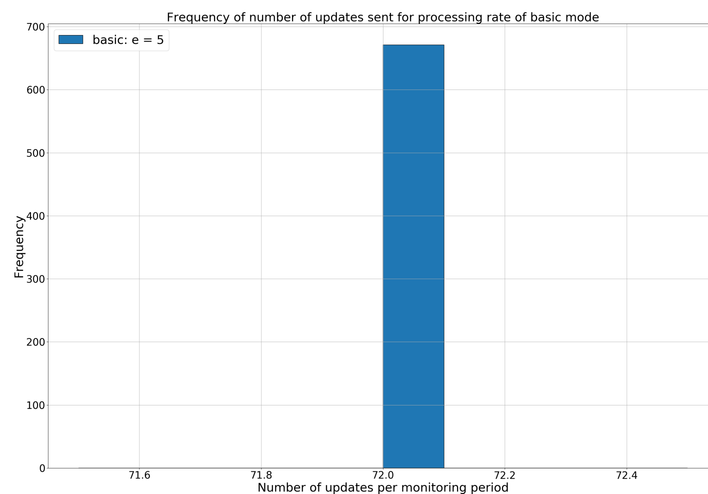

Figure 4.27: Packet processing rate, frequency of updates (last 15 mins).

The histogram for the basic mode is consistent with the graph for the running average, since in the last 15 minutes of the simulation, the running average for the basic mode remains constant. The histogram contains a single frequency which is close to the number of workers on a single card. The coordinator is able to process updates from a single monitoring service at a time, thus the histogram demonstrates that the basic mode sends updates for almost half of the workers in the system per monitoring period in the stable load stage. The histogram for the approximate

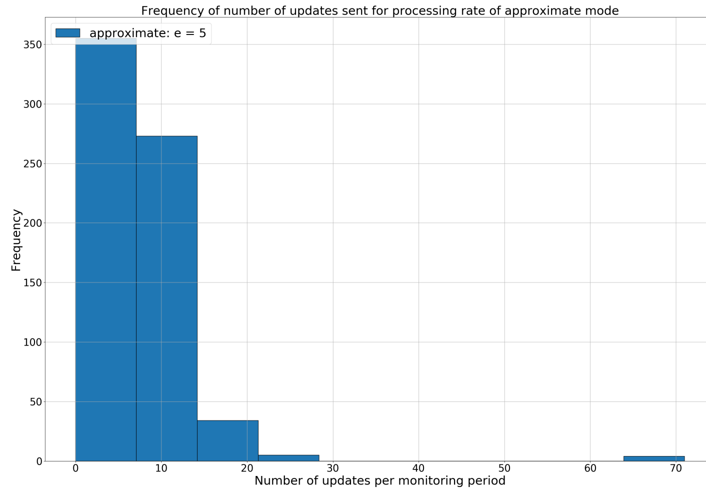

Figure 4.28: Packet processing rate, frequency of updates (last 15 mins).

mode contains more data points on the left side, where the number of updates reaches almost zero. Therefore, in the stable load stage, the approximate mode is near unaffected by small variances in the processing rate of individual workers and transmits far less monitoring updates.

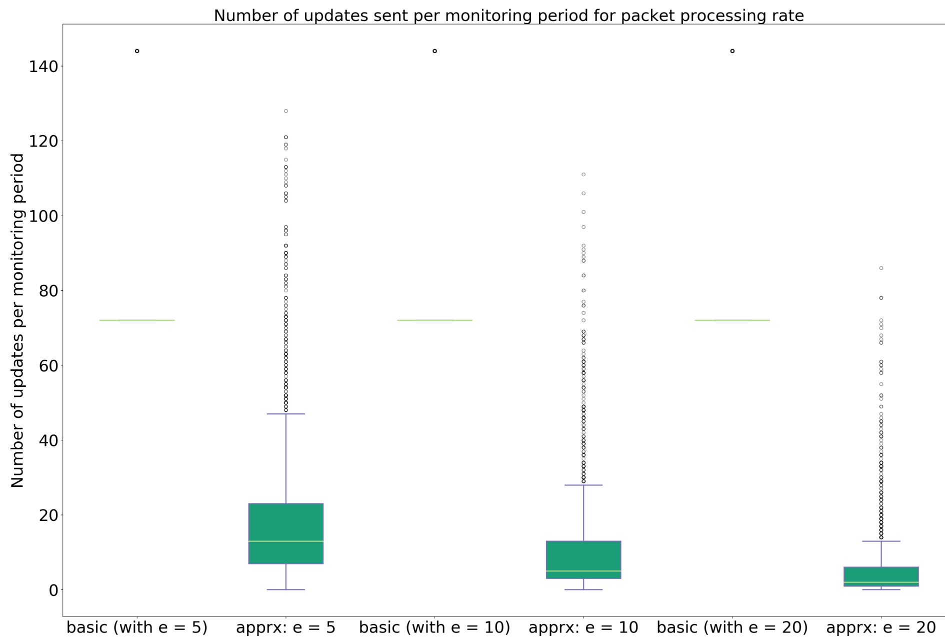

Figure 4.29: Packet processing rate, update count per period.

Finally, **Figure 4.29** and **Figure 4.30** provide plots to analyze the mean and standard deviation of the number of updates per monitoring period for packet processing rate for the entire duration of the simulations. The yellow lines in these box plots are the mean values and the vertical lines in the bar plots show the standard

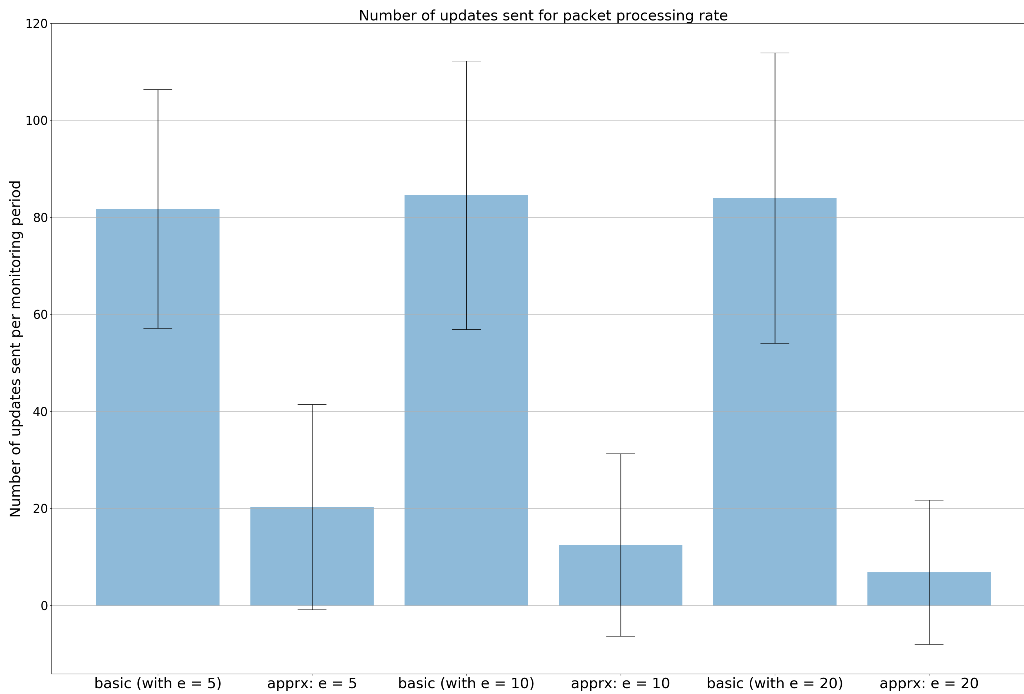

Figure 4.30: Packet processing rate, update count per period.

deviation. Unlike the processor utilization, the graphs for the basic mode demonstrate that there exists marginal variance in the number of monitoring updates sent per monitoring period. Consequently, when monitoring services are running in the basic mode, monitoring updates for almost all of the workers must be sent. However, the approximate mode demonstrates significant variance due to the number of monitoring updates sent in the stable load stage becoming considerably small. The standard deviation plots also display the large variance for the configurations of the approximate mode. Nevertheless, larger ε-values results in smaller means when compared to the basic mode.

Worker Packet Processing Rate

Packet processing rate of workers cannot be compared among multiple runs due to the load assigned to workers being in-deterministic. Therefore, we consider the fastest and slowest workers determined by summing per period packet processing rates for basic mode and the approximate mode with an error threshold of $\varepsilon = 5\%$. Note that the system's packet processing rate is not included in the graphs to make them easier to analyze, since this rate is significantly larger than the individual worker rates.

Figure 4.31 and **Figure 4.32** demonstrate the packet processing rate of the fastest and slowest workers in the entire system for the basic and approximate mode. Due to the graph for the entire simulation containing many data points, **Figure 4.32** is included to depict the behaviour in the last 15 minutes only. Similar to the processor utilization, the graphs indicate that there exists a significant difference in the packet processing rate of the slowest and fastest workers. Additionally, the graphs corresponding to the approximate mode contain many flat lines. These lines

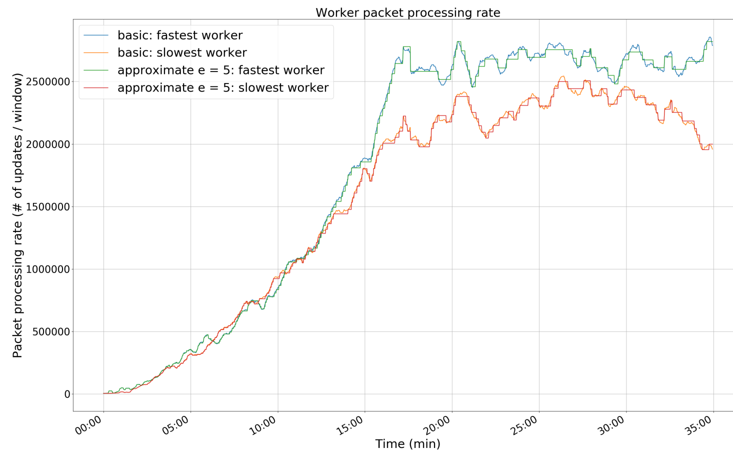

Figure 4.31: Packet processing rate of workers.

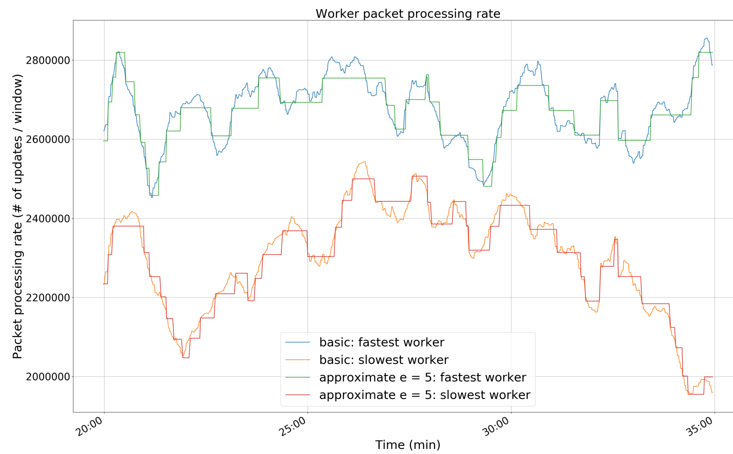

Figure 4.32: Packet processing rate of workers (last 15 mins).

indicate time points which monitoring services do not transmit monitoring updates due to the error guarantees still being valid. As **Figure 4.33** demonstrate, the approximate mode is capable of maintaining the error guarantees for these workers for $\varepsilon = 5\%$ configuration in addition to the system packet processing rate.

4.3 Discussion

EPGTOP when running in approximate mode, provides significantly better communication efficiency for continuously monitoring distributed systems compared to the basic mode. The error threshold is adjustable and set by the coordinator and

Figure 4.33: Running average relative error for worker packet processing rate.

the graphs in the previous sections demonstrated that larger error thresholds considerably improve the efficiency of monitoring over the basic mode and provide a trade-off between efficiency and accuracy. Most importantly, monitoring services are capable of maintaining the error thresholds configured for individual and aggregate system metrics. The graphs in the previous sections demonstrated that the difference between these two modes becomes significant when the load in the system remains stable. Small fluctuations to monitored parameters require the basic mode to send monitoring updates regardless of the difference. The approximate mode is more resilient to changes to monitored parameters and its configured error threshold can enhance this resilience, albeit at the cost of reduced accuracy.

System parameters such as packet processing rate which involves monitoring a single counter can be tracked far more efficiently than parameters such as processor utilization due to the monitoring function being affected by changes to a single parameter only. Processor utilization is represented as a set of frequencies to enable continuous monitoring algorithms to be used, since individual frequencies are monotonic parameters. An advantage of this conversion is that monitoring services are capable of transmitting the entire monitoring data to the coordinator and monitoring functions such as average, min, max, etc. can be computed on-demand without making changes to monitoring services. The analysis of the approximate mode showcased that the approximate mode is capable of transmitting frequencies to the coordinator far more efficiently than the basic mode. However, a limitation of the approach is that as the number of frequencies to monitor increase to track a particular system parameter, the number of monitoring updates to send will potentially increase as well. Therefore, for a small set of monitoring functions, making monitoring calculations such as average processor utilization for each worker locally and transmitting the results of these monitoring functions continuously with flooding may produce better efficiency than tracking individual frequencies with the approximate mode. Nevertheless, if the number of monitoring functions to compute

	5%	10%	20%
Processor utilization average savings	0.75254	0.54917	0.36613
Processor utilization average error	1.73851	1.62584	1.98326
Packet processing rate average savings	0.23464	0.12592	0.06757
Packet processing rate average error (%)	0.35383	0.81516	1.67170

Table 4.1: Comparison of configurations.

is large and performed on-demand, the approximate mode will be able to provide the entire monitoring data efficiently and accurately with respect to the configured error threshold.

Table 4.1 presents a table that compares average savings and average relative errors for system processor utilization and packet processing rates. Average savings are computed from averaging per period savings by dividing the number of bytes sent for the approximate mode with the basic mode. As the table shows, increasing the error threshold provides significantly better communication efficiency for packet processing rate than processor utilization. Evidently, the choice of a ε is more important for the rate which affects both the average savings and the relative error. The effectiveness of a choice of an error threshold is also dependent on the behaviour of a monitored system parameter. In essence, if changes to a monitored parameter in each monitoring period are large enough to require even configurations with large ε to send monitoring updates, adjusting the threshold to smaller values will not significantly affect the efficiency.

The results gathered from runs with multiple configurations indicate that EPG-TOP's approximate mode is more communication efficient than the basic mode when tracking counters and frequencies while ensuring that its configured error thresholds are maintained. The efficiency is particularly prevalent when monitoring moderately stable system parameters, since the basic mode that sends updates periodically is affected by even minor changes to monitored parameters, thus transmitting numerous monitoring updates when monitoring these parameters.

5
Conclusion

Continuous distributed monitoring has been studied extensively in the context of various distributed systems and settings. EPGTOP is based on a combination of system models and is designed to be integrated into an EPG system to monitor worker processes, cards and the entire system. The simple monitoring algorithms such as polling and flooding are too communication-intensive for most distributed systems. To efficiently monitor distributed systems, the number of monitoring messages sent is reduced by introducing an error parameter and sending monitoring messages only when error thresholds are exceeded. EPGTOP is incorporated with two modes: basic and approximate. The basic mode sends updates to the coordinator at the end of each monitoring period for any system parameter that was modified. In contrast, the approximate mode is more resilient to fluctuations in values of monitored parameters since this mode can be configured to tolerate up to a certain amount of error. Consequently, the approximate mode sends far less monitoring messages and its communication efficiency is dependent on the choice of the error threshold.

The theoretical models and algorithms make assumptions such as absence of communication and processing delays which do not have a direct implementation support in EPG. An EPG system running in production is a performance critical distributed system and making monitoring calculations upon every single event happening in the system is significantly computation intensive for monitoring purposes. EPGTOP provides techniques and data structures to alleviate these differences by supporting configuration parameters that can be tuned to alter the frequency of data retrievals and behaviour of the monitoring algorithms. EPGTOP consists of two main components: monitoring services and a coordinator. Monitoring is provided in the form of monitoring services that run on cards and these services collect the local monitoring data and execute monitoring algorithms. Therefore, more algorithms can be supported by extending the monitoring services. Correspondingly, the coordinator contains data handlers to compute monitoring functions on-demand and display the results continuously. Additional monitoring functions can be added to the coordinator without requiring changes to monitoring services.

The results presented in the previous chapter demonstrated that monitoring costs can be reduced significantly by utilizing approximate monitoring algorithms. EPGTOP represents system statistics such as processor utilization and processing rate as a set of counts and frequencies and provides methods to track and monitor these values. EPGTOP's approximate mode allows monitoring system parameters such as packet processing rate represented as counters significantly more efficiently than simple approaches described above. Representing parameters with a set of frequen-

cies reduces the effectiveness of EPGTOP due to more than a single value affecting monitoring decisions. Nevertheless, EPGTOP's approximate mode is capable of maintaining its configured error thresholds for both the individual worker statistics and the aggregate system statistics while reducing the amount of the monitoring data transmitted.

EPGTOP and its system architecture demonstrate methods to implement and integrate efficient monitoring algorithms into a production distributed system and provide system statistics for inspection and monitoring continuously over time. The analysis of EPGTOP establishes that communication efficiency of monitoring algorithms for distributed systems can be improved considerably with continuous monitoring algorithms. Effects of adjustments required to integrate these algorithms into EPG do not substantially impact the efficiency and reductions in the number of monitoring data transmitted makes EPGTOP and the incorporated algorithms viable for monitoring this system with minimal impact on performance.

Bibliography

[1] M. Olsson and C. Mulligan, *EPC and 4G packet networks: driving the mobile broadband revolution*. Academic Press, 2012.

[2] 3GPP, “Network architecture,” Technical Specification (TS) 23.002, 3rd Generation Partnership Project (3GPP), 04 1999. Version 14.1.0.

[3] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[4] D. E. C. Matthew L. Massiea, Brent N. Chunb, “The ganglia distributed monitoring system: design, implementation, and experience,” *Parallel Computing*, vol. 30, pp. 817–840, 2004.

[5] M. B. P. S. M. P. D. B. S. J. C. S. Benjamin H. Sigelman, Luiz Andre Barroso, “Dapper, a large-scale distributed systems tracing infrastructure,” *Google Research*, 2010.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems.,” in *HotOS*, pp. 85–90, 2003.

[7] G. Cormode, “The continuous distributed monitoring model,” *ACM SIGMOD Record*, vol. 42, no. 1, pp. 5–14, 2013.

[8] K. Y. Graham Cormode, S. Muthukrishnan, “Algorithms for distributed functional monitoring,” *ACM Transactions on Algorithms (TALG)*, vol. 7, 2011.

[9] K. Y. Graham Cormode, “Tracking distributed aggregates over time-based sliding windows,” *PODC ’11 Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 213–214, 2011.

[10] K. Y. Q. Z. Graham Cormode, S. Muthukrishnan, “Optimal sampling from distributed streams,” *PODS ’10 Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 77–86, 2010.

[11] D. K. Izchak Sharfman, Assaf Schuster, “A geometric approach to monitoring threshold functions over distributed data streams,” *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 23, 2007.

[12] L.-K. L. H.-F. T. Ho-Leung Chan, Tak-Wah Lam, “Continuous monitoring of distributed data streams over a time-based sliding window,” *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, vol. 62, 2012.

[13] P. I. R. M. Mayur Datar, Aristides Gionis, “Maintaining stream statistics over sliding windows,” *SIAM Journal on Computing*, vol. 31, pp. 1794–1813, 2002.

[14] Y. T. Mingwang Tang, Feifei Li, "Distributed online tracking," *SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, vol. 42, pp. 2047–2061, 2015.

[15] C. O. Brian Babcock, "Distributed top-k monitoring," *SIGMOD '03 Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 28–39, 2003.

[16] J. R. Ram Keralapura, Graham Cormode, "Communication-efficient distributed monitoring of thresholded counts," *SIGMOD '06 Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 289–300, 2006.

[17] M. V. Zhenming Liu, Bozidar Radunović, "Continuous distributed counting for non-monotonic streams," *PODS '12 Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, vol. 42, pp. 307–3018, 2012.

[18] D. W. Piotr Indyk, "Optimal approximations of the frequency moments of data streams," *STOC '05 Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pp. 202–208, 2005.

[19] A. C. Chrisil Arackaparambil, Joshua Brody, "Functional monitoring without monotonicity," *International Colloquium on Automata, Languages, and Programming*, pp. 95–106, 2009.

[20] M. S. Noga Alon, Yossi Matias, "The space complexity of approximating the frequency moments," *Journal of Computer and System Sciences*, vol. 58, pp. 137–147, 1999.

[21] Q. Z. Ke Yi, "Optimal tracking of distributed heavy hitters and quantiles," *Algorithmica*, vol. 65, p. 206–223, 2013.

[22] M. H. Graham Cormode, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, pp. 1530–1541, 2008.

[23] W. S. L. Joong Hyuk Chang, "Finding recent frequent itemsets adaptively over online data streams," *KDD '03 Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 487–492, 2003.

[24] K. D. C. O. A. Manjhi, V. Shkapenyuk, "Finding (recently) frequent items in distributed data streams," *ICDE 2005 21st International Conference on Data Engineering*, 2005.

[25] A. E. A. Ahmed Metwally, Divyakant Agrawal, "An integrated efficient solution for computing frequent and top-k elements in data streams," *ACM Transactions on Database Systems (TODS)*, vol. 31, pp. 1095–1133, 2006.

[26] S. M. R. R. Graham Cormode, Minos Garofalakis, "Holistic aggregates in a networked world: distributed tracking of approximate quantiles," *SIGMOD '05 Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 25–36, 2005.

[27] G. S. M. Arvind Arasu, "Approximate counts and quantiles over sliding windows," *PODS '04 Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 286–296, 2004.

List of Figures

1.1 A simplified view of EPC architecture [2]. 1

2.1 The Continuous Distributed Monitoring Model [7] 9
2.2 The Data Streaming Model [12] 11
2.3 An example system for Distributed Online Tracking [14] 13

3.1 A simplified overview of EPG architecture. 23
3.2 A simplified overview of User Plane SSC. 24
3.3 Periods for fetching and making monitoring decisions. 26
3.4 EPGTOP Architecture. 30
3.5 EPGTOP message. 31
3.6 A monitoring service's event loop. 32
3.7 Initialization Handshake. 33
3.8 The Coordinator's Event Loop. 37

4.1 Average system processor utilization: $\varepsilon = 5\%$. 43
4.2 Average system processor utilization: $\varepsilon = 10\%$. 43
4.3 Average system processor utilization: $\varepsilon = 20\%$. 44
4.4 Running average relative error for system processor utilization. . . . 45
4.5 Amount of monitoring data for processor utilization: $\varepsilon = 5\%$. 45
4.6 Amount of monitoring data processor utilization: $\varepsilon = 10\%$. 46
4.7 Amount of monitoring data for processor utilization: $\varepsilon = 20\%$. . . . 46
4.8 Processor utilization, average update count: $\varepsilon = 5\%$. 47
4.9 Processor utilization, average update count: $\varepsilon = 10\%$. 47
4.10 Processor utilization, average update count: $\varepsilon = 20\%$. 48
4.11 Processor utilization, frequency of updates (last 15 mins). 49
4.12 Processor utilization, frequency of updates (last 15 mins). 49
4.13 Processor utilization, update count per period: w = 10 sec. 50
4.14 Processor utilization, update count per period: w = 10 sec. 50
4.15 Processor utilization of workers. 51
4.16 Running average relative error for worker processor utilization. . . . 51
4.17 System packet processing rate: $\varepsilon = 5\%$. 52
4.18 System packet processing rate: $\varepsilon = 10\%$. 53
4.19 System packet processing rate: $\varepsilon = 20\%$. 53
4.20 Running average relative error for system packet processing rate. . . 54
4.21 Amount of monitoring data for packet processing: $\varepsilon = 5\%$. 54
4.22 Amount of monitoring data packet processing: $\varepsilon = 10\%$. 55

4.23 Amount of monitoring data for packet processing: $\varepsilon = 20\%$. 55
4.24 Packet Processing rate, average update count: $\varepsilon = 5\%$. 56
4.25 Packet processing rate, average update count: $\varepsilon = 10\%$. 56
4.26 Packet processing rate, average update count: $\varepsilon = 20\%$. 57
4.27 Packet processing rate, frequency of updates (last 15 mins). 57
4.28 Packet processing rate, frequency of updates (last 15 mins). 58
4.29 Packet processing rate, update count per period. 58
4.30 Packet processing rate, update count per period. 59
4.31 Packet processing rate of workers. 60
4.32 Packet processing rate of workers (last 15 mins). 60
4.33 Running average relative error for worker packet processing rate. . . 61

List of Tables

4.1 Comparison of configurations. 62