University of Oslo
Department of Informatics

Design, Implementation, and Evaluation of Network Monitoring Tasks for the Borealis Stream Processing Engine

Master's Thesis
30 Credits

Morten Lindeberg

May 23, 2007

Acknowledgements

I would definitely like to thank Jarle Søberg and Vera Goebel for outstanding guidance and support. Vera Goebel for presenting astonishingly clear thoughts and ideas. Jarle Søberg for always giving quick and wise advices, to numerous questions.

Standing on the shoulders of Jarle Søberg and Kjetil Hernes, by using their set of experiment tools and scripts, has helped me performing experiments otherwise not being possible within these 17 weeks.
Thank you.

Morten Lindeberg
University of Oslo
May, 2007

Abstract

Several data stream management systems (DSMSs), enabling their users to measure and analyze the behavior of a data stream on-line, are now available to the public. Network monitoring, as a challenging application domain, has shown to fit into the domain of DSMSs. With the use of the Borealis DSMS, we have implemented a set of network monitoring tasks. By performing these network monitoring tasks in a network on generated traffic, we have measured the loads that Borealis can handle. As we know the behavior of the generated network traffic, we have been able to investigate the correctness the monitoring results. Based on our evaluation, we have shown that Borealis can handle a 40 Mbit/s network load, with close to 99% accuracy performing measurements of average amount of received packets per second, as an example. The query language of Borealis has in addition enabled us to express complex network monitoring tasks. When increasing the complexity for the monitoring tasks, we see that the supported network load drops down to 10 to 30 Mbit/s.

Borealis has shown to handle significant higher traffic loads than expected, although we have found its load shedding mechanism for preventing overload situations as not efficient. We expect Borealis to handle higher loads than today, by altering parameters that are set in its source code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The number of people connected to the Internet increases each year [isc]. P2P and file sharing applications have in addition lead to a dramatic increase in network traffic volumes [SUKB06]. Measuring or monitoring the Internet traffic, consisting of streams of data sent between connected hosts, is increasingly of interest in many research fields. Network monitoring is also needed in order to detect and prevent congestion, equipment failures or malicious attacks.

As communication technology constantly is evolving, network monitors have to be able to monitor data transfers at increasing speeds and volumes. Since network traffic might extend to vast amount of terabytes within short periods of time, restrictions of what information to obtain, and how long to keep it, has to be set. In order to get a clear overview of what happens in a network, high volumes of measured data are often needed to be analyzed. This might lead to big challenges. Memory and storage resources and optimization issues regarding utilization of high traffic volumes, yield the need for complex and optimized solutions. Similar types of challenges have been of great interest within the recent scientific research field of Data Stream Management Systems (DSMS)[1], incorporating many technologies from the research field of Database Management Systems (DBMSs).

Throughout the 60s and 70s DBMSs enabled people to store huge amounts of data and provide them with easy access of the data, by introducing the declarative query language SQL. The field saw great interest in both the scientific world, where huge amounts of data of scientific interest could be stored for easy analysis and sharing, and in the business world storing infor-

[1] Also referred to as Stream Processing Engines (SPE) in Borealis

mation on customers, products and prices etc. The evolving technology of today has led to an increasing amount of data kept in the form of continuous data streams. Researchers have in the last years discovered the need of a new type of system; a system to incorporate the query language concepts from DBMSs on data streams. The need and usage of these systems are spread across many different application domains. Typical application areas are sensor networks, financial tickers and network monitoring [GO03].

A DSMS incorporates functionality for allowing its users to investigate the behavior of data streams. Posing queries on a stream of data, instead of data in a persistent database however requires that many challenges are solved. One important challenge is to restrict the consumption of memory, as the data volumes of streams grow large. Providing a query language fit to perform the stream operations is also important. Another notable challenge is to achieve low response times by processing the data as fast as possible, as it is received by the DSMS. As for measuring the Internet in real-time, several challenges are the same as those met by a DSMS. As many DSMSs are multipurpose, meaning they can be adapted to perform queries on almost any form of streaming data, many DSMSs are able to operate on the tasks of a network monitor. We need to investigate what network loads they can handle, and what types of network monitoring tasks they can solve.

1.2 Problem Description

Currently there exist several public domain DSMSs that have been implemented, as part of scientific research projects at different universities:

- TelegraphCQ [tel]
- STREAM [str07]
- Borealis [bor]
- TinyDB [MFHH05]

As most of these are developed for academic purposes and not for commercial use, they are not optimized in terms of performance, documentation, or graphical user interface. Their source code however might be available, giving a clear picture of how their stream processing techniques and functionality are implemented.

In this short Masters thesis we use the public domain system Borealis, which is a distributed Stream Processing Engine (SPE), to perform network

monitoring tasks, by looking at a stream of TCP/IP packet headers flowing on a network. We start by explaining the title of this thesis:

> *Design, Implementation and Evaluation of Network Monitoring Tasks with the Borealis Stream Processing Engine.*

By *designing*, we will investigate how a set of predefined network monitoring tasks can be designed and expressed in Borealis. As this thesis only has an duration of 17 weeks, a set of four tasks were identified prior to the semester start, as a minimal set of tasks to perform. The predefined tasks are:

Task-1: Verify Borealis load shedding mechanisms.

Task-2: Measure the average load of packets and network load per second over a one minute interval.

Task-3: How many packets have been sent to certain ports during the last five minutes?

Task-4: How many bytes have been exchanged on each connection during the last ten seconds?

We *implement* these tasks and deploy them on the Borealis query processor. The continuous queries are performed on continuous data streams of TCP/IP packet headers, generated in a network environment already set up in the DMMS lab [Søb06, Her06].

Following the implementation of the network monitoring tasks, we *evaluate* how good they can be expressed and identify what network load Borealis can handle, while performing the queries. The performance and consumption of system resources by Borealis will be thoroughly examined. We also investigate how tasks with different complexities affect the maximum supported network load. By using the same set up, and performing some of the same tasks as Søberg [Søb06] and Hernes [Her06], we can later compare the performance of Borealis with previous performance results logged from TelegraphCQ [Søb06] and STREAM [Her06].

We have most often designed several solutions for the tasks, in order to incorporate and test as much as possible of the continuous query language functionality that Borealis support. We have as an example managed to measure how different window sizes in sliding windows may affect memory consumption. We have also measured how drop_rate, as a parameter for load shedding, affects the network load Borealis can handle. In addition,

we include discussions on several parameters affecting the performance of Borealis, both set within the client applications, and within the source code of Borealis itself.

In addition to the predefined tasks, we have managed to design a task intended to show that intrusion detection can be designed and implemented, as a Borealis continuous query (*Task 5*). We have focused on detection of SYN Flood attacks, and have implemented two task solutions able to identify such a network event.

1.3 Outline

In this thesis, we start by covering the theoretical background for DSMSs. Chapter 2 describes a selection of DSMS application domains involving data streams. We discuss sensor networks, financial tickers and network traffic monitoring. Out of these domains, we focus mostly on network traffic monitoring, as we later perform network monitoring with the Borealis DSMS.

In Chapter 3, we introduce DSMSs in general. We look into the concepts and challenges of handling continuous queries on data streams, and techniques for solving these challenges discussed in the literature.

We summarize the concepts of Borealis in Chapter 4, and start by presenting its architecture, data model and query language. We also discuss how Borealis face the DSMS issues and concepts discussed in Chapter 3. A section where we discuss our own experiments after using Borealis is also presented.

Our main contribution in this thesis is to show how well Borealis is deployed for performing a set of predefined network monitoring tasks. In Chapter 5, we present our design for the network monitoring tasks, in the Borealis continuous query language. We start by discussing our stream definition for TCP/IP packet headers. Furthermore, we discuss each of the tasks, and present several attempts or versions, for solving each of them. In addition to the predefined tasks, we also introduce two task designs that are able of identifying possible SYN flood attacks.

Chapter 6 describes our experiment setup in detail. We discuss tools used for traffic generation and filtering, and scripts and parameters we use to execute the experiments. Even though our experiments are considered as black-box testing[2], we also include a section where we present some identified Borealis system parameters. These system parameters were found when investigating the Borealis source code, in order to identify load shedding

[2]In black-box testing, the external characteristics are the subject of testing. This means internal structures and functionalities are not considered

techniques. The section is included, as we believe that the parameters significantly affect the results of our experiments.

In Chapter 7, we evaluate the performance and results from each of the implemented network monitoring tasks. Consumption of system resources, such as CPU is identified, and we identify the network load each of the task versions can handle. In addition, we calculate the accuracy of the task results. Based on these calculations and measurements, we are able to compare task versions and identify how their query design affects the system performance.

Our conclusions are given in Chapter 8, summarizing the findings and results from this Master thesis. Finally, we present sections on critical assessment and future work.

In the appendixes, we include discussions, and figures not included in the main chapters: Appendix A includes figures showing the structure of both the TCP and the IP headers, as they are defined in RFC 793 [rfc81b] and RFC 791 [rfc81a]. We include a discussion on how to build and run the Borealis client applications, in Appendix B. As mentioned in Chapter 6, our experiment setup consists of several scripts that are created in order to help us perform the experiments. These are presented in Appendix C. In Appendix D, we present a discussion off how the value for the BATCH_SIZE parameter was chosen in the client applications. The discussion is included, as the BATCH_SIZE parameter has shown to have significant impact on our experiment results. At last, we include a DVD-ROM containing experiment data, in Appendix E.

Chapter 2

DSMS Application Domains

In many application domains data naturally occurs in the form of a sequence of data elements (data streams) [GO03]. Within the last years researchers have identified the need for new type of applications, enabling users to perform queries on these streams of data. By combining the query functionality from database management systems (DBMSs) with functionality for processing data streams, a new breed of systems is now emerging; data stream management systems (DSMSs).

In this chapter we present three DSMS application domains, and start by presenting the field of network monitoring. We present sensor networks in Section 2.2, and introduce the field of financial tickers in Section 2.3, concerning the collection of data from streams of stock prices and money exchange rates between markets.

2.1 Network Monitoring

The numbers of hosts connected to the Internet increase each year. As of July 2006, a survey from the Internet Systems Consortium stated that the total number of hosts on the Internet has reached 439,286,364 [isc]. Due to the increase growth and expansion of the Internet, the interest of measuring it has grown within the research community [Sie06].

In the design of the Internet Protocol (IP), the routers were not intended to retain information on ongoing transfers. Instead they were intended to make use of efficient packet switching [GR]. Because of this, the Internet has few built-in measurement mechanisms, and gaining the information mentioned above yields the need for network monitors. Traditionally, PERL scripts are often used to measure and analyze Internet traffic [PGB+04]. As the traffic continues to grow both in number and size, we need more power-

ful and adaptable tools. DSMSs incorporate many solutions to the several challenges met by a network monitor; hence network monitoring is as an emerging field of interest within the DSMS research field.

The purpose of network monitoring, is to track the state of the network [GR]. Measurements performed by a network monitor typically involve:

- Bandwidth utilization and packet roundtrip time (RTT)
- Overview of traffic flows
- Identifying weak links with possible congestions
- Identifying equipment failures
- Traffic analysis, e.g. what applications are causing traffic
- Anomaly traffic detection for security reasons, e.g., Denial of Service attacks (DoS)

Network monitors are not only needed in order to make sure a network works like it should. When performing *capasity planning* or changes to a network, network monitors are needed in order analyze the effect these changes might have. Even small changes in network set-ups may lead many routers to reconfigure their forwarding tables, leading to large changes in the network. The effect of even small changes can be complex to foresee [GR], but a network monitor can be used to gain a better understanding. Understanding the behavior of network traffic is also vital, when dealing with protocol improvements and development. As 90% of the Internet traffic today is carried with the TCP [MS05], significant research have been performed in order to locate possible factors in TCP, limiting utilization of bandwidth. Network monitoring and analysis of Internet traffic plays a vital role in these research fields.

2.1.1 Challenges

Monitoring and measuring the Internet, or parts of it, consists of several challenges. The Internet, as a selection of autonomous systems (ASs), consists of IP networks controlled by a variety of different operators. Within a single AS the means of communication are either wired or wireless, with a handful of different low-layered standardized ways of exchanging the bits. Measuring and monitoring ASs might involve collecting data from several corporations with different policies regarding privacy. Restrictions on what information

that can be gained and analyzed might further complicate the tasks of a network monitor. The use of Internet applications such as Peer to Peer (P2P) file sharing and streaming audio/video applications, has increased drastically the last years. This has lead to a drastic increase of Internet traffic volumes. Large traffic volumes will result in large amount of data to analyze, making analysis very demanding in terms of memory, CPU and disk storage.

2.1.2 Classifications

Network monitors can either perform measurements on-line, or off-line. In addition, the measurements are either performed passively or actively. In the following subsections, we explain each of these classifications.

On-line

On-line measurement and analysis is performed at the instance of time when the data is flowing through the network. Discarding analyzed data can save huge amounts of disk space, when dealing with large amounts of network traffic. Performing on-line analysis is crucial when performing measurements in time-critical environments. By time-critical, we mean environments that demand a short response, after identifying critical events.

Off-line

Off-line measurements are performed by logging traffic data into either flat files or a database. Further analyzes and measurements are performed on the persistent stored traffic data for complex analysis.

Passive Network Measurements

Passive measurements are used to gather data for network analysis, by measuring observed traffic on a host or router [Sie06]. The data to gather could for instance be the TCP/IP packet headers form all packets passing by a gateway or router on the network. The network monitoring tasks, introduced in Chapter 5, are based on on-line passive measurements of TCP/IP packets headers.

Passive monitoring only observes, hence it does not particularly increase network traffic. On the other hand, the size of the collected data can potentially be huge; at least in case off-line monitoring is performed. Hence, an important challenge when performing passive measurements is to restrict the total size of the collected data. In addition, there are also important

challenges regarding scalability. As Internet network traffic constantly is increasing, tools for measurements and analysis need to be scalable in order to meet the increasing amount of data to process.

We are now going to introduce some existing systems. An example of a hardware based passive network measurement tool is the *Endace DAG card*. The fastest card in production as of Spring 2007, is claimed to capture 100% of the packets in a 10 Gigabit Ethernet link [end]. However, these hardware cards are expensive. In addition, the platforms they use are considered to be primitive, and it is hard and challenging to implement monitoring applications on them [Der03].

There exist a variety of software based passive network utilities. *tcpdump* [tcp] is an example tool for off-line analysis of passive measurements. *tcpdump* can be used to produce statistics or graphs based on packet information dumped to disk. Systems like *tcpdump* although fails when network loads increase to 1 GBit/s [Der03].

For off-line analysis, there also exist several DBMS-based systems that enable users to store packets obtained from the network as relations. InTraBase is such an example [Sie06].

For on-line analysis, Gigascope, a proprietary solution from AT&T, is an example network monitor classified as a DSMS [JCS03]. It provides an SQL - like interface for processing streams of packets, and is said to handle network loads up to 1 Gbit/s. It includes traffic analysis, intrusion detection, router configuration analysis etc. In Chapter 5, we introduce Borealis as a network monitoring tool, performing on-line, passive network measurements.

2.1.3 Active Network Measurements

Active measurements are performed by actively sending probe packets into the network [Sie06]. These packets are then used as references to track the state of the network. Estimates of link capacities, and available bandwidth, are often the subject [Sie06].

As packets are sent to the network, active monitoring will increase network load. In contrast to passive network monitoring, only the probe packets are subject of the actual measurements. The number of probe packets should be significantly lower than the number of packets subject of the measurements in passive network monitoring. Hence, active monitoring often requires smaller disk space and memory resources, than passive monitoring.

An example of an active measurement tool are *traceroute*. It is used to trace the route to a host. *traceroute* obtains information from all gateways a packet needs to pass in order to get to a host. The information is obtained by the use of Internet Control Message Protocol (ICMP) packets. By using

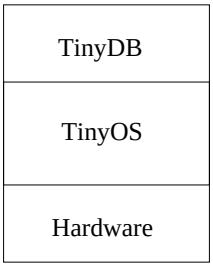

Figure 2.1: Schematic drawing of a TinyDB sensor mote.

the *timetolive* field in the IP header, latencies between the gateways are calculated.

2.2 Sensor Networks

Sensor networks have in the past few years matured to the point that they are now feasible to be deployed at large scale [MFHH05]. Each sensor within the network is a battery powered wireless device that can measure certain values, e.g., temperature, noise, light and seismographic data.

The sensors are capable of creating networks on their own, connecting to each other in an ad-hoc fashion, acting both as routers and clients. They communicate wirelessly; hence they do not require cables. By deploying many sensors, large geographical areas might be covered. Since they are energy efficient and low-end, their batteries can last for weeks or months at a time [MFHH05]. By time synchronizing with their neighbor sensors, they can save power by only staying awake when they need to transmit data.

Researchers have deployed sensor networks in many monitoring environments. Example deployments are monitoring of environmental properties for endangered birds during their burrows, and at a vineyard, and for earthquake monitoring [MFHH05].

TinyDB is an example system capable of processing distributed sensor data from wireless devices, also called motes [MFHH05]. These motes run their own operating system, called *TinyOS*. On top of *TinyOS*, each mote runs *TinyDB* incorporating distributed DSMS functionality. Figure 2.1, shows a schematic drawing of a TinyDB mote. Users can perform queries on the motes by the use of a Java based GUI, which can be run on a machine connected to the motes. We include a sample *TinyDB* query, performed on a set of sensors motes:

```
SELECT nodeid, temp, light, voltage, noise
```

```
FROM sensors
SAMPLE PERIOD 5
```

The query obtains the nodeid in addition to measurements of temperature, light, voltage and noise from all the sensors, every 5th seconds.

2.3 Financial Tickers

In the financial world, stock prices stream between sellers and buyers through stock exchange markets. Several of the challenges within the world of DSMS are met, when controlling these streams of data. Analysis of financial events typically involves discovering correlations, identifying trends, forecasting future values, etc. [GO03].

Traderbot is an example financial application, which incorporates DSMS functionality. It is a real-time financial search engine that provides a search-engine interface to a real-time data stream consisting of prices and financial news. *Traderbot* also incorporates an interface to a financial historical database [tra]. As an example query, it can for instance report stocks undergoing rapid increases or decreases in price on the exchange markets.

Chapter 3

Data Stream Management Systems

In this chapter, we describe data stream management systems (DSMSs)[1] in general.

The main requirements of a DSMS are presented in Section 3.1. As much of the concepts and functionality of a DSMS are inherited from database management systems (DBMS), we compare the two different types of systems in Section 3.2. In Section 3.3, we cover the DSMS stream data model.

DSMSs perform their queries with the use of continuous query language. We discuss concepts, and mention example languages, in Section 3.4. A number of data reduction techniques is presented in Section 3.5. In Section 3.6, we discuss unblocking operations when dealing with aggregation of values from infinite streams. Some example existing DSMSs, are presented at the end of this chapter, Section 3.7. Note that we thoroughly present Borealis, in Chapter 4.

[1] Also referred to as a stream processing engines (SPEs)

```
# FORMAT: timestamp, average packets count pr. second, average bits pr. second
13,5013,420156971
73,6084,511417531
133,6075,511232111
```

Figure 3.1: Result tuples from a query performed on TCP/IP headers.

3.1 System Requirements

DSMSs give its end users tools for extracting and analyzing information from data streams in real-time. As for selecting what information to retrieve, DSMSs perform continuous queries. The information retrieved from the stream is known as result tuples. Figure 3.1 shows example output from a query performed on a data stream consisting of TCP/IP packet headers.

We start with a short definition of a data stream, posed by Gölab et al. [GO03]:

> *A data stream is a real-time, continuous, ordered sequence of items.*

The conceptual behavior of data items arriving in *real-time*, is that they may arrive in an uncontrollable rate, and have to be computed during a restricted period of time. By *continuous*, we mean they might never stop arriving. By *ordered*, we mean that they either arrive in order implicitly by arrival time or explicitly by a timestamp [GO03]. We introduce the concept of timestamps in Section 3.1. The *items* in the stream, are the data elements that the stream consist of, these are also called *tuples*.

What mainly distinguishes data encapsulated in streams, with data residing persistently in a database, is that the data in streams is not available for random access from disk [BBD+02]. The streams possibility of unbound sizes yields that storing them completely is impossible. Latency issues might also make disk storage operations on the stream infeasible. Hence the tuple values in streams are only kept in memory during certain time intervals, and may be seen only once [SÇZ05]. We hereby state some important requirements for a DSMS. In addition to the stated requirements are also the requirements of correctness, predictability and stability as with most systems.

- Achieving a *minimum latency* is important in a DSMS, since low response time often is needed when reacting to online events. In addition, a DSMS should not stall the data stream, instead keep it moving [SÇZ05]. DSMSs operating on streams when rapid reactions are needed, should be able to perform online processing without costly disk storage operations [SÇZ05].

- In order to achieve low latency, incorporation of *approximation*, or *data reduction* techniques are also important parts of a DSMS [BBD+02]. This is because high-volume data streams can increase CPU and memory consumption. Data reduction is used to relief the systems, although

leading to approximate results. We cover *reduction techniques* in Section 3.5. The techniques can be successfully deployed where trends, rather than results of total accuracy are of interests.

- DSMSs are required to incorporate a *continuous query language*, in order to be able to present real-time computed analysis from the stream [SÇZ05]. We present *continuous query languages* in Section 3.4.

- *Load shedding* is a more drastic operation, performed to relief a DSMS with tuples to process. It is a common technique where selected tuples simply are discarded. The dropping of tuples, might certainly affect the correctness of query results. Chapter 4 includes a discussion on how Borealis deals with *load shedding.*

- *Adaptableness* is another key requirement of DSMSs [BBD+02]. Data streams are often strictly uncontrollable, hence there is often hard to foresee their behavior. Somehow, DSMSs have to deal with stream imperfections like delay, missing or out-of-order data, or bursty data rates [SÇZ05]. As mentioned, load shedding can be used to deal with high traffic rates, and is among techniques used to increase a DSMSs means of adaptiveness.

- Stream processing can either be *pull-based,* or *push-based*. The later form is a passive way of processing a stream, where the data continuously is pushed to the system. Network monitoring, as we perform it with Borealis described in Chapter 6, is an example of *push-based* stream processing. This is because the stream is pushed into the DSMS, without it being able to control the stream rate in which tuples arrives. In sensor networks, *pull-based* stream processing is performed, as the system can control in which rate each sensor should transmit their measured data.

- Finally, we mention a final DSMS requirement. This is the ability to *integrate* stored data with the streaming tuples [SÇZ05]. Most DSMSs are able to store data from streams in internal databases, and also compare streaming tuples with statically stored data.

3.2 Comparison of DBMSs and DSMSs

Obtaining data from data sources in general, is a task DSMS share with database management system (DBMS). We start by a definition of DBMSs given by in Elmasri et al. [EN07]:

> *The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications.*

We further compare these concepts with those shared with DSMSs. Note, that while DBMSs operate on databases, DSMSs mainly operate on data streams.

Defining a database in a DBMS involve specifying the data types, structures and constraints [EN07]. In DSMSs, stream definitions are used in order to specify the data elements/tuples, within the stream.

Construction of a database is something DBMSs do for storing data to a storage medium. A DSMS does not necessarily construct anything; it will rather connect to a data stream, than constructing it. Note that many DSMSs are able to store streams as relations to storage mediums as an additional feature.

Manipulating a database, in terms of a DBMS, involves queries either retrieving, updating or deleting specific data in the database. DSMSs also retrieve and update data in streams by posing queries. Since data in streams most often are seen as append-only, deletion of data in streams is often not supported, although selections of streams can be re-routed to a new stream elsewhere.

Sharing as an event in DBMSs, is the feature of allowing multiple users access to shared data in databases simultaneously. This concept of sharing is perhaps not something DSMSs are intended to support. Or at least sharing is a concept of less interest, since the nature of a data stream yields that it is easier to share the stream itself. Note, that several DSMSs support distributed queries, and cooperation in distributed environments, and that most DSMSs supports several real-time queries at the same time.

3.3 Stream Data Models

The *stream data model*, as part of a DSMS, is intended as a set of concepts used to describe the structure of the data streams to process. In a DBMS, data models describe the a set of concepts used to describe the structure of a database [EN07]. Most DSMSs see data streams as a sequential list of data elements. Each data element takes the form of relational tuples [GO03]. Borealis, introduced in Chapter 4, uses a data model where streams are seen as append-only sequences of tuples.

A notable behavior of data streams is that tuples can arrive un-ordered, although seen as an ordered sequence. If they later should be read in a

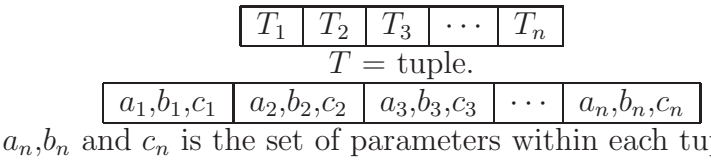

Figure 3.2: Schematic drawing of tuples in a stream.

certain order, they should contain a timestamp or a sequential number. This way the DSMSs can process un-ordered tuples, and order them on behalf of the timestamp or sequence number. Figure 3.2 shows schematic drawings, of tuples in a data stream. The figure shows two levels of hierarchies, in order to show the relation of tuples, and their parameters.

Tuples are seen as relational. The tuples T_1 and T_2 relate to each other when sharing the same key parameter(s). This often means having timestamp values within the same time interval. In Chapter 6, we include several queries where tuples are said to relate to each other when they have equal values for the tuple parameters *sourcePort* and *sourceIP*.

3.4 Continuous Query Languages

In order to retrieve information from the data streams, DSMSs perform queries expressed by the use of *continuous query languages*. Many of these languages are expressed with syntax and structure similar to SQL (Structured Query Language). Examples of SQL like languages are: CQL used in STREAM, GSQL used in Gigascope [JMSS05], AQuery and StreaQuel used in TelegraphCQ [GO03].

The SPE Aurora, and its successor, Borealis, gives its users ways of expressing continuous queries with a GUI containing conceptual arrows and operator boxes. The data flows are presented as arrows. The operator boxes can be seen as obstacles that by connecting to the streams, obtains result tuples from them. The results from the operations on the stream, performed by the operator box, are sent out from the box as separate stream(s). The parameters within each box declare what to retrieve from the incoming tuples, and whether or not to pass it to the output stream(s). By using the operator boxes, and stream definitions, users can specify the way the data flows within a query. Because of the way it gives users control of the data flow, the CQL used by Borealis is said to be a procedural language [GO03]. We describe the CQL in Borealis further in Chapter 4.

3.5 Data Reduction Techniques

Continuous queries that are run over data streams only look at data elements while they reside in memory. Since memory always is restricted, we need to set a size limit of maximum bytes or a maximum time interval to process data from, when dealing with data streams possibly of infinite size. One technique for setting these restrictions is to evaluate only samples of the data. This is called *sampling* [BBD+02]. When restricting the data to evaluate only to samples, some data will be thrown away, and not considered. In this case, result tuples will only present approximations, rather than accurate results. To restrict the use of resources, mainly memory, many other techniques have been prompted to create reasonable tradeoffs between approximated and accurate results.

The concept of *sliding windows* is a data reduction technique supported by most DSMSs. It is typically used when calculating aggregated values. Their results are somewhat only approximations of the total stream behavior, since only a portion of the stream (called a window) is evaluated, instead of the entire past history [BBD+02]. A restriction could be set, for instance only to evaluate tuples within an interval of five seconds. The restriction value that defines how many tuples to evaluate within a window is called the *window size*. *Window sizes* are either *time-based* or *tuple-based*. *Time-based* windows base their selection of tuples on their timestamps, and will only allow tuples from within a certain interval. *Tuple-based* windows base their selection of tuples upon a maximum count of tuples to process at the same time. The windows are called *sliding*, since the tuples they process are sliding through, as new ones arrives. Most DSMSs let the user control how the windows should advance or slide. In the Borealis continuous query language, the *advance* operator in aggregation boxes defines this value.

There exist a variety of window specifications in the DSMS literature, but there seem to be some disagreements on their definitions. The window specifications mainly distinguish themselves on how they advance, and how their advance value relates to their window sizes. We choose to classify four types of windows as presented by Krishnamurthy et al [KWF06]. The classifications are presented in Figure 3.3, and described in the list below:

- *Hopping windows* are windows where the advance value, is larger than the window size itself. This means that $A_d > W_s$, where A_d is the advance value, and W_s is the window size. Because of this, *hopping windows* do not overlap each other [KWF06]. Note that Gölab et al. [GO03] use the term non-overlapping *tumbling windows* for these. *Hopping windows* can be seen as a type of *sampling technique*, where win-

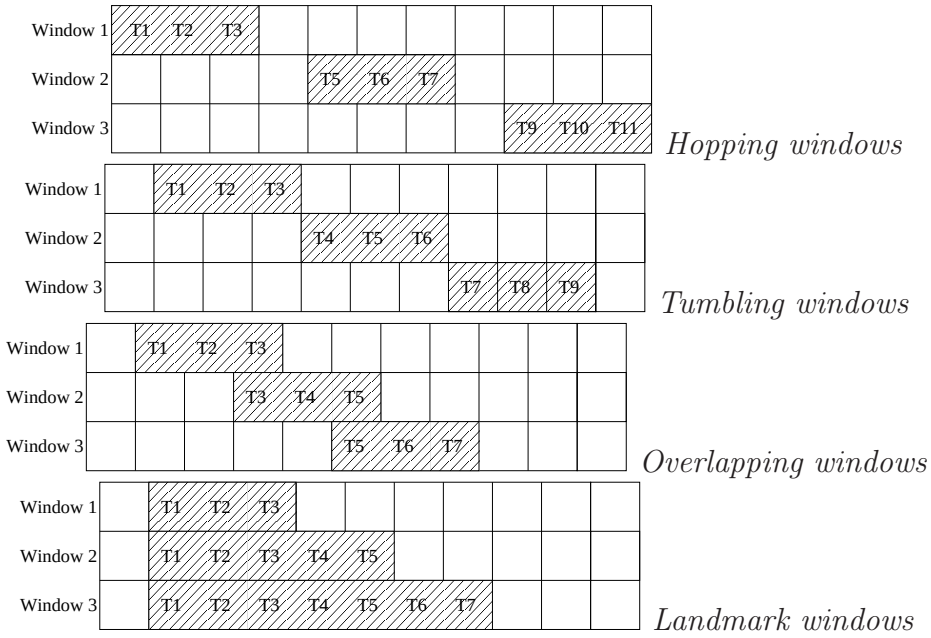

Figure 3.3: Schematic drawing of window techniques.

dows are *sampled* from the stream.

- *Tumbling windows* have equal window size and advance value. $A_d = W_s$. This means they will not overlap, and all values in the stream, will once, and only once, reside in a window. Note that Gölab et al. [GO03] use the term *jumping windows*, for these.

- *Overlapping windows* are windows where the advance values are smaller than the window sizes. $A_d < W_s$. *Overlapping windows* are used when one would wish to report aggregated values more often than for each window. The behavior of *overlapping windows*, is by many seen as the typical behavior of *sliding windows*.

- *Landmark windows* have a fixed point from where they move. Their size will increase while tuples are added. Using *landmarks windows* can result in *blocking operations*, since one will never will see the entire input in the window, when performing aggregations on data streams of infinite size.

There exist a variety of other reduction techniques. *Sketching*, *Histogram* and *Wavelets* [BBD+02] are all reduction techniques which involve summarization of estimates or representations.

3.6 Unblocking Operations

Aggregation operations are commonly used in order to summarize multiple values in a data set. Example aggregation operations are SUM, COUNT, MIN, MAX and AVG [BBD+02]. When using these aggregations on unbound data streams, a problem referred to as the blocking problem has to be taken into consideration [BBD+02]. These aggregations are known to be blocking operators. If a query processor derived from the DBMS world would be asked to compute an average value of an infinite continuous stream, it would never see its entire input. Thus it will be blocked in an infinite loop, computing average values forever, never able to report a final result. *Sliding windows* are often used as a solution for this problem. By only computing aggregations within windows, the DSMS sees the whole input to compute aggregations from. This way it is able to present final results, performing aggregation operations over infinite streams.

3.7 Overview of Existing Systems

In this section, we briefly present some existing DSMSs. We start with TelegraphCQ.

3.7.1 TelegraphCQ

TelegraphCQ, is a system developed at University of California, Berkeley. The initial system *Telegraph* was implemented with the following goal:

> *Develop an Adaptive Dataflow Architecture for supporting a variety of data intensive, network applications [CCD+03].*

Prototype extensions to early implementations of *Telegraph* are later built. These support stream processing, hence the name *TelegraphCQ*.

TelegraphCQ are written in the C language, and base its query processing functionality upon the open source DBMS *PostgreSQL* [Søb06]. Because of its relation to *PostgreSQL*, *TelegraphCQ* is supposed to store streams to disk that later can be retrieved as relations in PostgreSQL. *TelegraphCQ* uses *StreaQuel* query language, which is very similar to SQL. The language support windowing semantics like *sliding windows* and *tumbling windows* in order to prevent blocking operations. An in-depth description of *TelegraphCQ* can be found in [Søb06].

3.7.2 STREAM

STREAM is implemented at Stanford University. It is a general-purpose DSMS that support declarative continuous queries over continuous streams and traditional data sets [ABB+04].

> *It targets rapid and load varying streams, on systems where resources may be limited [ABB+04].*

STREAM uses CQL (Continuous Query Language), which is a relatively minor extension to SQL [ABB+04]. It supports aggregations over windows, that are either *time-based*, *tuple-based* or *partition based*. In other words, STREAM support *sliding windows*.

Streams are either defined as ISTREAM, which is tuples that are inserted, RSTREAM consisting of tuples that resides within a window, and DSTREAM tuples that are deleted from RSTREAM. Further descriptions and evaluation of *STREAM* can be found in [Her06].

3.7.3 Gigascope

Gigascope is a stream database for network applications. Gigascope supports traffic analysis, intrusion detection, router configuration analysis and monitoring, etc [JCS03]. It is developed at AT&T, and deployed at many sites within the AT&T network. It is supposed to be able of processing 1.2 million packets per second during peak periods, at a dual 2.4 GHz CPU server [JCS03].

The supported continuous query language used by Gigascope is *GSQL*. The language supports operations like *selections*, *joins*, *aggregations* and *merging*. GSQL are optimized for operation on network tasks, and support low-level optimizations in order to support high network loads. This means the query processing is performed at low-levels, such as within the network interface card (NIC) itself.

3.7.4 Aurora

Aurora is a general-purpose DSMS developed at Brandeis University, Brown University and M.I.T [ACC+03]. The supported stream processing operations include *filtering*, *mapping*, and *windowed aggregations and joins*. The query language of Aurora is expressed by the use of a Java GUI. The GUI supports dragging and dropping stream processing operators, and connecting them to data streams. In addition to the GUI, key components of Aurora include the *scheduler* reducing overhead and invoking batching techniques,

the *storage manager*, which among other things includes pull-based access to historical stored data. Finally, the *load shedder* is responsible for detecting and handling overload situations. Borealis, presented in Chapter 4, and Medusa presented in next section, inherits functionality from Aurora.

3.7.5 Medusa

Medusa is a distributed DSMS, based on the single-site functionality of Aurora [BBS04]. By distributing stream processing among several machines, many advantages are identified. This includes the ability to leverage query processing among several machines, and the ability to create replicas. The replicas enables that the task of a faulting machine can be handed over to another machine.

The concept of high availability (HA) is an important goal for the Medusa application. Complex functionality is implemented in order to achieve this. By achieving high availability, the overall system should for instance handle crashed machines, and broken communication lines, to some extent. Borealis, presented in Chapter 4, inherits its concepts and functionality regarding distribution of queries from Medusa.

Chapter 4

Borealis

In this chapter we describe *Borealis Spring version 2006*. Documents regarding the functionality of this version are what we base our descriptions on, at least throughout Section 4.5. In Section 4.1, we start with a general system description, and present the Borealis architecture in Section 4.2. We then present the data model used by Borealis to structure stream data elements in Section 4.3, and the query language used to pose continuous queries in Section 4.4. Additional query processing techniques are mentioned in Section 4.5. As we have thoroughly tested Borealis, we also present a section where we discuss our experiences of using it, in Section 4.6. Since Borealis, rather than being a full production commercial system, is made for academic purposes, parts of it have shown not to work as expected. These findings are among other things covered.

4.1 System Description

Borealis is a distributed stream processing engine (SPE)[1], developed at Brandeis University, Brown University and MIT [AAB+05]. It inherits functionality from two systems: Aurora and Medusa. From Aurora it inherits its core stream processing functionality. Functionalities regarding distribution of queries in networks, are inherited from Medusa [ABC+05].

Borealis, as part of a research project, has as purpose to identify and address shortcomings of current stream processing techniques [bor]. It is built up on several modules dealing with the different aspects of stream processing, stream query distribution, and tools to visualize and perform the processing operations. By looking at a set of input streams, it can aggregate, correlate and filter continuous streams of data to produce the output of

[1]Note that we use the terms DSMS and SPE interchangeably.

interest [Tea06]. Because of the way it is set up to handle I/O, Borealis can be seen as multipurpose DSMS, which can be deployed in many different stream processing application areas. These areas include sensor network and as we describe in Chapter 6, network monitoring. In a demonstration by Ahmad et al. [ABC+05], Borealis was even used to operate on a multi-player first person shooter network game, running continuous queries producing information on player positions in the game.

4.2 Architecture

In this section we present the different modules of Borealis. The modules are all part of the Borealis system, and deals with the different aspects of Borealis stream processing functionality. We will mainly describe their concepts.

4.2.1 Borealis Processing Node

A Borealis distribution is a collection of modules, although the Borealis server application, called the processing node, is the module that performs the actual stream processing. In order to operate Borealis on a distributed query network, each processing node runs a Borealis server instance [AAB+05]. It consist of the following components [Tea06]:

1. The *Query Processor* consists of the *Aurora Node* module, which is the actual stream processing engine within each Borealis processing node. Further more, it also keeps an administration interface for handling incoming requests. The module *Query Catalog* holds the query diagrams locally. In addition, the *Data Path* module routes tuples in and out from remote nodes. Within the *Query Processor*, there is also a module called the *Consistency Manager*, which deals with failure handling, and replicas in distributed query environments.

2. The *Availability Monitor* is a monitor that observes the state of neighbor nodes in distributed query networks. It is used by the *Consistency Manager*, which operates within the *Query Processor*.

3. The *Local Load Manager* improves load balance between nodes, by tracking the state of the node that it runs on. It also reads information from other *Local Load Managers*, if in a distributed query. By tracking the load on other nodes, the modules can perform load balancing. Note that there also exist a module called the *Global Load Manager*, as part of a module called *Meta Optimizer*, described Section 4.2.3.

4. The *Transport Independent RPC* is a layer module that handles control messages sent between components and nodes in the query network.

4.2.2 Distributed Catalog

As part of the collection of modules, Borealis also contains the *Distributed Catalog* module that keeps information about the overall system. Although the nodes perform the actual stream processing, the *Distributed Catalog* has overall deployment information, and description of query diagrams in a distributed query networ.

4.2.3 Meta Optimizer

The *Meta Optimizer* is a standalone application that monitors the Borealis processing nodes globally through the *Global Load Manager* module. It can also apply load shedding through the use of the *Global Load Shedder* module, as nodes get overloaded. The *Global Load Shedder* is responsible for detecting and handling overload information. It is supposed to act when nodes become overloaded due to increased data rates, or increased query workload [Tea06]. Although leading to approximated results, it will drop selected tuples. Borealis is supposed to contain complex computation for selecting which tuples to drop. It operates on three different levels: locally in the network, on nodes in the neighborhood and distributed all over the system.

4.2.4 Marshal Tool

The *Marshal Tool* is used for generating C++ helper functions regarding stream I/O. The functions are used by the Borealis Client Applications, described in Section 4.2.5. The *Marshal Tool* is a stand-alone module that takes a XML query diagram as input. By reading the query diagram, it creates well-suited C++ structures and functions on behalf of the streams and stream schemas defined in the XML. We describe Borealis query language, and XML in Section 4.4. The *Marshal Tool* is also well suited for validation of queries. Validation of the query XML is performed with the use of a document type description (DTD), and a XML parser.

4.2.5 Borealis Client Applications

A *Borealis client application* is responsible for sending and retrieving stream data to the Borealis processing node. By processing node, we mean the actual instance of the Borealis *Query Processor*. The actual sending and retrieving

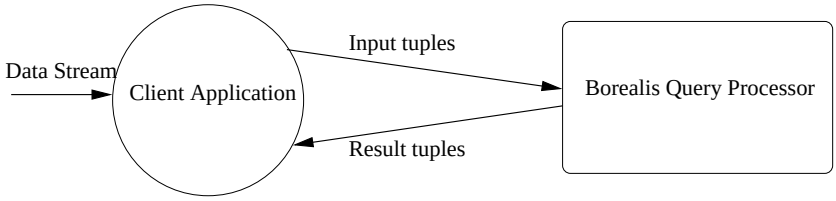

Figure 4.1: Drawing off the client application, and its relation to the Borealis query processor.

is made simple, by using the generated structures and functions created with the *Marshal Tool*. Structures for both input and output tuples are defined in a generated header file, as well as functions that take instances of these structures as arguments. Provided with the Borealis release is a number of executable *client applications* that show how to send and retrieve stream data with the use of the generated code. Figure 4.1 shows a drawing of the a client application, and its connections to both a data stream, and to the Borealis query processor.

4.3 Data Model

The Borealis stream data model is based on the one found in Aurora. This model defines streams as an append-only sequence of data items [Tea06]. The tuples take the form of $(k_1, \cdots, k_n, a_1, \cdots, a_m)$, where $k_1, \cdots, k_n$, consist of the key for the stream, and $a_1, \cdots, a_m$ provide the attribute values [ACC+03]. Each data item are called *tuples*.

Borealis will only recognize parameters within the tuples, if they appear as defined in the stream definition. We cover the concepts of stream definitions in next section.

4.4 Query Language

Borealis supports a continuous query language composed of conceptual operator boxes and stream arrows. Within the Borealis release, a GUI is included to build these queries. The queries eventually take the form as XML documents, that are uploaded to the *Query Processor* when queries are performed. Since XML is used, queries can also be written by hand, without using the Java GUI.

We hereby cover the main operation concepts, which are supported in order to pose queries on the streams:

- *Operator boxes*, are used to retrieve data from parameters within the tuples. There exist several types of operator boxes. We describe them later in this section.

- *Stream arrows* define the intended flow of the streams within the *Query Processor*. Note that in the actual XML documents, stream flows are expressed by using XML deploy diagrams, and implicitly by declaring their flow in and out from the *operator boxes*.

- *Stream definitions* are used in order to define the appearance of the tuples within the stream. By doing so, the *Query Processor* knows what parameters each of the tuples in a specific stream consist of. Borealis supports several field types that we present in Chapter 5.1. This chapter also describes how to express a stream definition for TCP/IP header fields.

- *Deployment diagram* are expressed in XML, and are optionally uploaded to the *Query Processor* during query execution. The diagrams can be used to connect *client applications* to streams traveling between *operator boxes* within the query. By default, only those streams that have not been identified as a input stream for any operator box, are sent back to the *client application*. By the use of *deployment diagrams*, users are able to better control the internal stream events, within the *Query Processor*.

Borealis support the following operator boxes [Tea06]:

- The *Map* operator box is used to transform input tuples, and map them into a output stream.

- The *Filter* operator box is used to apply predicates, and route tuples to output streams either matching the predicate or not.

- The *Aggregate* operator box is used to apply aggregate functions (e.g. AVG or SUM) over *sliding windows*. The size of each window is either given as a maximum time interval (time-based), or a maximum number of tuples to keep inside each window (tuple-based). The advance parameter tells how the window should slide.

- The *Join* and *AuroraJoin* operator boxes are used to pair tuples from two streams when both match a certain predicate. The result tuple can then be constructed by the use of values from both of the input tuples.

- The *BSort* operator box is used to sort tuples in a windows, based on a single integer parameters contained in each of the tuples. A restriction is set on the maximum number of tuples to be contained within each result set. The sorting algorithm used is *bubble sort*.
- The *Union* operator box is used to merge tuples from two streams with similar schemas into a stream containing all tuples from the streams in arbitrary order.
- The *Lock/Unlock* operator box is used for synchronization and concurrent access of tuples with an integer key field with a certain value.
- The *WaitFor* operator box is used for synchronization as well. It accepts two streams of tuples. They buffer tuples from stream s_1 until the tuple of stream s_2 matches a tuple in s_1 with a certain predicate. The tuple in s_2 is the one that is released to output stream.
- The *Random Drop* operator box can be used for load shedding. It drops tuples randomly with a probability of p_d, where p_d is the *drop_rate* parameter. These boxes can be deployed by the user itself, within queries, but are additionally deployed by the *Global Load Manager*, when overload occurs.
- The *Window Drop* operator box is also a operator for load shedding. Instead of dropping single tuples, it drops whole windows. The parameter *drop_rate* sets the probability p_d, of whether or not a window should be dropped. The actual window definition is defined in the same way as in *aggregate* operator boxes.

Borealis supports internal static tables that can be compared with input tuples. Four table operator boxes are supported:

- The *Select* table operator box passes tuples from static tables, as they match tuples from input streams given a certain predicate.
- The *Insert* table operator inserts tuples into a static table, as they are received from input stream.
- The *Delete* table operator box is used to delete tuples within a static table, as they are matched with tuples arriving at the input stream with a given predicate.
- The *Update* table operator box is used to update tuples within a static table, as they are matched with tuples arriving at the input steam with a given predicate.

4.5 Query Processing Techniques

In this section we present notable query processing techniques, not already mentioned.

- Dynamic Query Modification is implemented in order to be able to modify queries dynamically during operation.
- The High Availability module in Borealis makes the nodes monitor each other with the use of control messages. As nodes get overloaded, the tasks of faulting nodes can be handed over to others.
- The Statistics Manager provides an interface for fetching statistics from processing nodes. The statistics includes: stream rates, tuple latency and operator cost, selectivitys and queue lengths.

4.6 General Borealis User Experiences

Thoroughly testing Borealis gives us the impression that Borealis is a powerful stream processing engine. Since being subject of academic research rather than commercial production, it does not seem to operate flawlessly as described in its documentation. For instance, there were some challenges with respect to the *average* parameter. The documentation [Tea06] clearly states that it should be expressed as *average*. By coincidence, we tried typing *avg* instead, which seemed to be the solution. In addition, important client application functionality incorporated in the provided examples are not mentioned in the documentation. This includes *batching techniques* and parameters used to control the batching technique behavior. Chapter 6.3 includes a discussion of both the technique, and the parameters.

Installation of the system demands high knowledge of Linux system libraries, packages and general system setup as well as dependencies. We have not managed to compile Borealis on any Linux distribution other than Fedora Core 2. This distribution is supposedly the one Borealis was developed on. There still exist several modules that we have not been able to compile.

Performing queries and connecting to data streams both involve building a C++ application, and writing intricate XML queries. Several example applications are provided. By using these, building *client applications*, and creating XML queries, have proven to be not so demanding as we first feared, although basic C++ knowledge is needed. Expressing intricate queries however, has proven to be very time consuming. Intricate queries will soon lead to many lines of XML. Several lines of XML, when containing small

mistakes, e.g., typing errors, are very hard to debug. The provided XML parser used by the *Marshal Tool*, and during query uploads, help locating these to some extent, although the parser does certainly not always respond with understandable error messages. Although the learning curve of writing XML queries is steep, we have found that intricate queries can be performed with success. We have not been able to test the GUI application for writing queries, but based on our beliefs of the query concept, getting a high level graphical overview of complex queries should help overcome complexity of expressing them in XML.

We were not able to compile the *Global Load Shedder*, as part of the *Meta Optimizer* properly. The *Local Load Shedder*, which is supposed to operate on each processing node, does not seem to be working either. Further investigation of its source code shows lines that are commented out, that perhaps should not be commented out.

Performance wise, Borealis have proven to be more efficient of dealing with stream data than expected. We present a further discussion on parameters affecting system performance in Chapter 6.5. A performance evaluation of Borealis is given in Chapter 7.

Chapter 5

Network Monitoring Tasks

In this chapter we present the design of each network monitoring task. We start by presenting how TCP/IP header fields are defined within the stream definitions. As there are several ways of representing the fields, we include discussions explaining the field types we have chosen.

Following the explanation of the stream definition, we start explaining the task designs. Each task design often includes several versions, since there are several ways of solving them.

5.1 TCP/IP Stream Definition

The Borealis client applications receive a stream of TCP/IP header fields from a socket connection to *fyaf*[1]. Each header is represented as a comma-separated string containing the header fields[2]. In this section we discuss how to represent these values as a stream of tuples.

The TCP and IP headers are defined in RFC793 [rfc81b] and RFC 791 [rfc81a], presented in Appendix A. They consist of a variety of fields; many with different sizes. They range from one bit fields used for flags, to optional fields with a possible size of 81 bytes.

When defining a stream in the Borealis query language, six field types are supported [Tea06]. These are:

- int - A 32 bit signed integer
- long - A 64 bit signed integer

[1] We introduce *fyaf* in Chapter 6.2

[2] Note that for each task, not all fields are needed. But since we want an adaptive stream definition, we choose to present all the headers, even though some of them never will be used.

- single - A 32 bit IEEE floating point value
- double - A 64 bit IEEE floating point value
- string - A fixed length, zero filled sequence of bytes
- timestamp - A 32 bit time value

The lack of field types that perfectly match representations in the TCP/IP header, pose that the mapping will not be optimal in terms of size. A minimal size representation of fields in the input stream would possibly lead to a higher supported network load[3]. On the other hand, size optimal fields, in terms of choosing a minimum field representation, would in some cases lead to slower, or even impossible query executions. If we are only interested in obtaining field values through a map query, we can for instance represent the IP header field *totallength* in a *string* field of two bytes, as opposed to a 4 bytes *int*. This would spare us 2 bytes for each tuple. On the other hand, calling a Borealis aggregation operator like SUM or AVG over a sliding window, in order to calculate values from those field types, would not be possible. This is because operators like SUM and AVG only accepts *int*, *long*, *single* or *double* representations.

We have chosen to represent most numeric values in both the TCP and IP header fields as the field type *int*. In the following subsections, we discuss most of the field representations in the stream definition. Figure 5.1 shows the final stream definition in XML.

5.1.1 Timestamp

In order to compute aggregations over *time-based* sliding windows, timestamps are used to distinguish new tuples from old tuples. Timestamps can also be used for ordering the out-of-order arrived tuples. The timestamp we use is created and added to the tuple as it is inserted into the batches within the client applications. We have used the Borealis type *int* to represent timestamps, although there even exist a field type called *timestamp* that we have not tested. *int* representations were chosen to easily set time values from client applications, to timestamps in the streaming tuples.

[3]The concept of supported network load are further described in Chapter 7.1. We have found one important bottleneck to be a restriction for a buffer in the Borealis data handler, with a maximum size set in the generated marshalling code.

5.1.2 Representation of IP Addresses

We choose to represent the IP addresses as a Borealis *string* with the size of 16 characters. This results in a usage of $16 * 8 = 128$ bits, which is quite large in contrast to the original 32 bit representation.

Another possible representation would be to store the IP addresses as four *int* fields, resulting in a $4 * 32 = 128$ bit representation also. Test runs have shown this to result in a decrease of what network load Borealis can handle. The decrease of supported network load was identified during test runs with *Task 1*, introduced in next section. One of the reasons why the four *int* field representations is slower, is possibly due to the increased numbers of tuple fields to process. In terms of query design, representation of IP addresses as *string* field types, as opposed to four *int* fields, require fewer statements when for instance comparing them in predicates.

5.1.3 Representation of Option Fields

Option fields are optional fields kept in both the TCP header and the IP header. Their sizes can be maximum 81 characters. Hence, we represent them in *string* fields with the size 81.

5.1.4 Sequence and Acknowledgement numbers

Out from the 32 bit representation of TCP/IP sequence and acknowledgment numbers, it should be sufficient to represent these in the 32 bit *int* fields. Results during our implementation however show that the *int* type cannot handle these numbers when they grow large. This limit is caused by the fact that *int* types are *signed* in Borealis, enabling a range of $[-n, +n]$ where $n = 2,147,483,648$. The sequence and acknowledgement numbers in the IP protocol are *unsigned*, meaning they can only be positive, and have a bigger range among positive numbers than signed representations ($[0, 2n]$). In order to properly support *int* numbers larger than n, we need to use *long*, that within Borealis occupies 64 bits. We represent all numbers among the header fiels, that occupies maximum 16 bits in the TCP/IP headers as the Borealis field type *int*.

5.1.5 Representation of Flags

We choose to represent the URG, ACK, PSH, RST, SYN and FIN flags, as the type *int*, even though they in the IP header are represented as single bits. *int* representation of these flags are certainly not optimal in terms of size.

A *string* with the size of 1, should be sufficient, saving $32 - 8 = 24$ bits for each flag. This would totally spare us $24 * 6 = 144$ bits, or 18 bytes for each tuple.

During the design phase, experiments where performed with the flags represented as single character *string* types. These experiments showed no immediate increase in performance in terms of what network load Borealis could handle. Most significantly, using the flags in an expression with a *filter box*, showed not work due to difficulties when comparing single character *string* values with expected flag values within the query. As we are filtering packets with the ACK and SYN flags set in *Task 5*, we choose to represent the flags as *int* values.

Note that if Borealis had supported *boolean* flags as field types, we could save $\lceil \frac{31bits*6}{8bytes} \rceil = 24$ bytes per tuple.

5.2 Task Design and Implementation

In this section we present our designs for each of the predefined network monitoring task. In addition we have designed a fifth conceptual task, dealing with intrusion detection. Since we have restricted time and equipment resources, we have chosen to design the network monitoring tasks for single node queries only. This mainly because our system experiment setup are based on the one used by [Søb06, Her06], which is not built for distribution. We regard designing distributed queries for the network monitoring tasks as future work.

Following the design and implementation of these tasks, we present the experiment setup in Chapter 6. The experiment setup is used when we run the tasks at several network loads. Evaluation of the experiments is presented in Chapter 7.

5.2.1 Task 1: Load Shedding

Verify Borealis load shedding mechanisms

The bursty nature of traffic load on a network could result in higher loads than a DSMS can handle. We discuss the general need for load shedding mechanisms in Chapter 3.5. In order not to overload the DSMS, load should be shedded in order to relief the DSMS with tuples to process. When running single nodded queries, load shedding means Borealis should drop a certain amount of tuples when overload occurs.

We have not been able to verify a working load shedding mechanism. As mentioned in Chapter 4, load shedding in Borealis involves both the *Global*

```
<schema name="PacketTuple">
     <field  name="timestamp"        type="int" />

     <!--               IP                        -->
     <field  name="version"           type="int" />
     <field  name="ipheaderlength"    type="int" />
     <field  name="tos"               type="int" />
     <field  name="totallength"       type="int" />
     <field  name="id"                type="int" />
     <field  name="flags"             type="int" />
     <field  name="fragoffset"        type="int" />
     <field  name="ttl"               type="int" />
     <field  name="protocol"          type="int" />
     <field  name="headchksum"        type="int" />
     <field  name="sourceip"    type="String"  size="16" />
     <field  name="destip"      type="String"  size="16" />
     <field  name="ipoptions"   type="String"  size="81" />
     <!--            END OF IP                    -->

     <!--               TCP                       -->
     <field  name="sourceport"        type="int" />
     <field  name="destport"          type="int" />
     <field  name="seqnum"            type="long" />
     <field  name="acknum"            type="long" />
     <field  name="tcpheaderlength"   type="int" />
     <field  name="reserved"          type="int" />
     <field  name="urg"               type="int" />
     <field  name="ack"               type="int" />
     <field  name="psh"               type="int" />
     <field  name="rst"               type="int" />
     <field  name="syn"               type="int" />
     <field  name="fin"               type="int" />
     <field  name="windowsize"        type="int" />
     <field  name="chksum"            type="int" />
     <field  name="urgtr"             type="int" />
     <field  name="tcpoptions"  type="String"  size="81" />
     <!--           END OF TCP HEADER                    -->

 </schema>
```

Figure 5.1: The TCP/IP stream definition used within all our Borealis queries.

Load Shedder module, and the *Local Load Shedder* module. The *Global Load Shedder* did not compile during installation. Neither did its parent component, *Meta Optimizer*. Since the *Local Load Shedder module* is passive, and only performs load shedding when asked to by the *Global Load Shedder*, Borealis at our setup will not perform automatic load shedding when overload occurs.

The errors during compilation of the *Local Load Shedder* and *Meta Optimizer* modules are probably caused by lack of some external libraries. By altering several make files we have managed to partly compile the *Meta Optimizer*, as well as the *Load Shedder* module. Although they do not seem to react when Borealis gets overloaded.

Even though the *Global Load Shedder* module does not work properly, Borealis client applications support some way of dealing with overload. For each packet that is batched, there is a test in the generated marshal code that checks if the Borealis input stream buffer has space available. If there is not enough space for another tuple, the client application will sleep for a while, trying one more time afterwards. This way of dealing with overload has proven not to be efficient when dealing with high stable network loads. Since load is never decreasing, resending tuples when out of space only seems to lead to congestion.

A complete load shedding solution, in our opinion, should both involve identifying overload situations, and deal with it by dropping tuples. By running simple queries, by using the *map* operator box at high network loads, we could not identify any signs of a load shedding mechanism. Hence, we cannot verify any meaningful load shedding mechanism in the current Borealis distribution. However, as mentioned in Chapter 4, applying load shedding can be performed by the use of the Borealis operator box *random_drop*. From what we understand from the source code of the *Global Load Shedder*, *drop_boxes* is the mechanism used when shedding load. Hence, we move the focus for this task away from the *LoadShedding* module, and will instead measure the effect of the *random_drop* boxes.

We have designed a set of tasks that perform a simple map operation on all the fields in the stream. This by using a *map* operator box. We present the map box in Figure 5.3 and 5.4. In addition to the *map* boxes, we deploy *random_drop* boxes, in order to investigate their effect. Figure 5.2 shows one of the boxes used, with a drop_rate of 0,8. Schematic drawings of the task design with and without the *random_drop* box deployed are presented in Figure 5.5 and 5.6. For our evaluation in Chapter 7, we will include several test runs with different drop rates, in order to analyze the effect of them.

```
<box name="box1" type="random_drop">
   <in   stream="Packet" />
   <out  stream="Shedded" />

   <parameter name="drop_rate" value="0.8" />
   <parameter name="max_batch_size" value="1000" />

</box>
```

Figure 5.2: Task 1: The random_drop box used to shed load

```
<box name="box2" type="map">
    <in   stream="Shedded" />
    <out  stream="Map"/>
    <parameter name="expression.0" value="timestamp"  />
    <parameter name="output-field-name.0" value="timestamp"  />
    <!--               IP HEADER                                  -->
    <parameter name="expression.1" value="version"  />
    <parameter name="output-field-name.1" value="version"  />
    <parameter name="expression.2" value="ipheaderlength"  />
    <parameter name="output-field-name.2" value="ipheaderlength"  />
    <parameter name="expression.3" value="tos"  />
    <parameter name="output-field-name.3" value="tos"  />
    <parameter name="expression.4" value="totallength"  />
    <parameter name="output-field-name.4" value="totallength"  />
    <parameter name="expression.5" value="id"  />
    <parameter name="output-field-name.5" value="id"  />
    <parameter name="expression.6" value="flags"  />
    <parameter name="output-field-name.6" value="flags"  />
    <parameter name="expression.7" value="fragoffset"  />
    <parameter name="output-field-name.7" value="fragoffset"  />
    <parameter name="expression.8" value="ttl"  />
    <parameter name="output-field-name.8" value="ttl"  />
    <parameter name="expression.9" value="protocol"  />
    <parameter name="output-field-name.9" value="protocol"  />
    <parameter name="expression.10" value="headchksum"  />
    <parameter name="output-field-name.10" value="headchksum"  />
    <parameter name="expression.11" value="sourceip"  />
    <parameter name="output-field-name.11" value="sourceip"  />
    <parameter name="expression.12" value="destip"  />
    <parameter name="output-field-name.12" value="destip"  />
    <parameter name="expression.13" value="ipoptions"  />
    <parameter name="output-field-name.13" value="ipoptions"  />
    <!--         END OFF IP HEADER                               -->
```

Figure 5.3: Task 1: The first half of the map box

```
    <!--            TCP HEADER                                  -->
    <parameter name="expression.14" value="sourceport"  />
    <parameter name="output-field-name.14" value="sourceport"  />
    <parameter name="expression.15" value="destport"  />
    <parameter name="output-field-name.15" value="destport"  />
    <parameter name="expression.16" value="seqnum"  />
    <parameter name="output-field-name.16" value="seqnum"  />
    <parameter name="expression.17" value="acknum"  />
    <parameter name="output-field-name.17" value="acknum"  />
    <parameter name="expression.18" value="tcpheaderlength"  />
    <parameter name="output-field-name.18" value="tcpheaderlength"  />
    <parameter name="expression.19" value="reserved"  />
    <parameter name="output-field-name.19" value="reserved"  />
    <parameter name="expression.20" value="urg"  />
    <parameter name="output-field-name.20" value="urg"  />
    <parameter name="expression.21" value="ack"  />
    <parameter name="output-field-name.21" value="ack"  />
    <parameter name="expression.22" value="psh"  />
    <parameter name="output-field-name.22" value="psh"  />
    <parameter name="expression.23" value="rst"  />
    <parameter name="output-field-name.23" value="rst"  />
    <parameter name="expression.24" value="syn"  />
    <parameter name="output-field-name.24" value="syn"  />
    <parameter name="expression.25" value="fin"  />
    <parameter name="output-field-name.25" value="fin"  />
    <parameter name="expression.26" value="windowsize"  />
    <parameter name="output-field-name.26" value="windowsize"  />
    <parameter name="expression.27" value="chksum"  />
    <parameter name="output-field-name.27" value="chksum"  />
    <parameter name="expression.28" value="urgtr"  />
    <parameter name="output-field-name.28" value="urgtr"  />
    <parameter name="expression.29" value="tcpoptions"  />
    <parameter name="output-field-name.29" value="tcpoptions"  />
    <!--           END OF TCP HEADER                    -->
</box>
```

Figure 5.4: Task 1: The second half of the map box

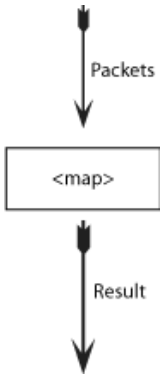

Figure 5.5: Task 1: Schematic drawing of initial version 1

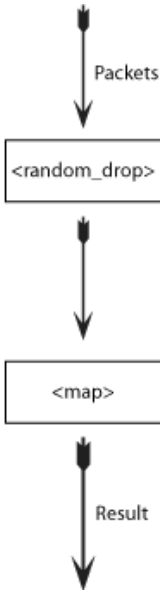

Figure 5.6: Task 1: Schematic drawing of initial version 2.

5.2.2 Task 2: Average Load

Measure the average load of packets and network load per second over a one minute interval.

In this task, we display both the numbers of packets per second, and their average load, during a one minute interval. Load is calculated by using the *totallength* field in the IP header.

In the initial solution shown in Figure 5.7, we use two aggregate boxes. The first aggregate box uses a sliding window over one second. This box deliver tuples each second. Each of these tuples keeps the size and packet count per second. The output result tuples are sent through a second aggregate box that calculates the average values per minute. We include a schematic drawing of the version in Figure 5.8.

In the second initial version, shown in Figure 5.9, we use only one aggregation box. It calculates the sum of packets and their load per minute in one operation. Since we want the results as averages rather than sums, a map box is used to divide the results with 60. We present a schematic drawing of the version in Figure 5.10

One of the notable differences between the two versions is that the version 1 will have fewer tuples being processed in window at the same time. This since the window size is set to one second, instead of one minute. The second notable difference is that the second version has one, instead of two

```
<box name="box1"   type="aggregate" >
        <in   stream="Packet"    />
        <out  stream="AggPerSecond" />

        <parameter   name="aggregate-function.0"  value="count()"  />
        <parameter   name="aggregate-function-output-name.0"
                     value="count"  />
        <parameter   name="aggregate-function.1"  value="sum(totallength)"  />
        <parameter   name="aggregate-function-output-name.1"
                     value="avgsize"  />
        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="1"        />
        <parameter   name="advance"               value="1"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp" />
    </box>

    <box name="box2"   type="aggregate" >
        <in   stream="AggPerSecond"    />
        <out  stream="AggPerMinute" />

        <parameter   name="aggregate-function.0"  value="avg(count)"  />
        <parameter   name="aggregate-function-output-name.0"
                     value="count"  />
        <parameter   name="aggregate-function.1"  value="avg(avgsize*8)"  />
        <parameter   name="aggregate-function-output-name.1"
                     value="avgsize"  />
        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="60"        />
        <parameter   name="advance"               value="60"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp" />
    </box>
```

Figure 5.7: Task 2: Initial version 1

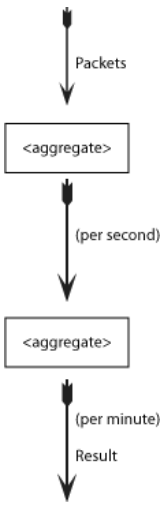

Figure 5.8: Task 2: Schematic drawing of initial version 1.

```
<box name="box1"   type="aggregate" >
     <in   stream="Packet"    />
     <out  stream="AggPerMinute" />

     <parameter   name="aggregate-function.0"  value="count()"  />
     <parameter   name="aggregate-function-output-name.0"
                  value="count"  />
     <parameter   name="aggregate-function.1"  value="sum(totallength)"  />
     <parameter   name="aggregate-function-output-name.1"
                  value="avgsize"  />
     <parameter   name="window-size-by"        value="VALUES"   />
     <parameter   name="window-size"           value="60"        />
     <parameter   name="advance"               value="60"        />
     <parameter   name="order-by"              value="FIELD"    />
     <parameter   name="order-on-field"        value="timestamp" />
 </box>

 <box name="box2" type="map" >
   <in  stream="AggPerMinute" />
   <out stream="Map"         />

   <parameter name="expression.0"           value="timestamp" />
   <parameter  name="output-field-name.0"   value="timestamp" />
   <parameter name="expression.1"           value="count / 60" />
   <parameter  name="output-field-name.1"   value="count" />
   <parameter name="expression.2"           value="avgsize *8 / 60" />
   <parameter  name="output-field-name.2"   value="avgsize" />

 </box>
```

Figure 5.9: Task 2: Initial version 2

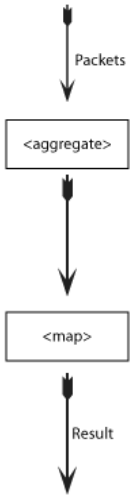

Figure 5.10: Task 2: Schematic drawing of initial version 2

aggregation boxes[4].

During initial testing, we found there was a big difference on what network load the two solutions could operate on. Initial version 1, presented in Figure 5.7. seems to handle significant higher network load than initial version 2, presented in Figure 5.9. We expect this behavior is caused by the different window sizes used to calculate the aggregate values.

By using the same schema as in Figure 5.9, we can try several window sizes. This by changing the window size parameter in the *aggregate* box, and what we divide our result upon in the *map* box. By doing so, we can identify how the window size affects Borealis ability to handle high network loads. The task evaluation in Chapter 7 includes several test runs with several window sizes.

5.2.3 Task 3: Destination Ports

> *How many packets have been sent to certain ports during the last five minutes?*

In this task, we count packet occurrences designated to certain ports. The count of the designated packets tells us something about the types of applications used on the network. Although this applies for most types of applications, some P2P applications tend to communicate with the use of traditional HTTP port numbers, in order to hide or make sure to get through firewalls [SUKB06]. With our task P2P applications using traditional HTTP port numbers will not be identified.

When choosing what destination ports to count occurrences of, the simplest solution in terms of query design would be to filter packets destined for ports defined in a predicate, e.g. *destport = 22* or *destport = 80*. By filtering only those packets of interest to a stream, counting occurrences of them is possible by using an *aggregate* box. This solution will lead to a high number of XML lines, as the number of ports to compare increase. This is because every single port would be required to be mentioned. We present a solution with the *filter* and *aggregate* box in Figure 5.11, where we only filter those packets destined for port 60010, used by the traffic generator that we use. A schematic drawing is presented in 5.12.

We propose a second solutions in Figure 5.13. In addition, we have included a schematic drawing of the query in Figure 5.14. As mentioned, Borealis supports the use of static tables. In solution number two, we send two streams into Borealis; S_1 and S_2. The stream S_1 represents the TCP/IP

[4]Note that a map box is needed in order to divide the results with 60.

```
<box name="box1" type="filter">
   <in    stream="Packet">
   <out  stream="Filtered">

   <parameter name="expression.0"            value="destport = 60010" />
   <parameter name="pass-on-false-port"      value="0" />
</box>

<box name="box2"   type="aggregate" >
       <in   stream="Filtered"    />
       <out  stream="Matched" />

       <parameter   name="aggregate-function.0"  value="count()"  />
       <parameter   name="aggregate-function-output-name.0"
                    value="count"  />
       <parameter   name="window-size-by"        value="TUPLES"    />
       <parameter   name="window-size"           value="999999999" />
       <parameter   name="advance"               value="999999999" />
       <parameter   name="order-by"              value="TUPLENUM"  />
       <parameter   name="group-by"              value="destport"  />
       <parameter   name="timeout"               value="300"        />
   </box>
```

Figure 5.11: Task 3: Version 1

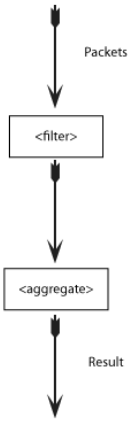

Figure 5.12: Task 3: Schematic drawing of version 1

```
<query name="tableQuery">
      <table name="testTable"  schema="TableTuple" >
          <key       field="destport"/>
          <index     field="destport" />
          <parameter  name="truncate"  value="0" />
          <parameter  name="create"    value="1" />
      </table>
   </query>

   <query name="writeQuery">
       <box name="insertBox"  type="insert" >
           <in      stream="Insert" />
           <access  table="testTable"/>
           <parameter name="sql"
               value="insert into testTable
                           values (destport, description)"/>
       </box>
   </query>

   <query name="readQuery">

       <box name="selectBox"  type="select" >
           <in      stream="Packet" />
           <out     stream="Select" />

           <access  table="testTable"/>
           <parameter name="sql"
                      value="select *
                             from testTable
                             where testTable.destport == input.destport"/>
       </box>

       <box name="box1"   type="aggregate" >
          <in   stream="Select"    />
          <out  stream="Matched" />

          <parameter   name="aggregate-function.0"  value="count()"  />
          <parameter   name="aggregate-function-output-name.0"
                       value="count"  />
          <parameter   name="window-size-by"        value="TUPLES"    />
          <parameter   name="window-size"           value="999999999" />
          <parameter   name="advance"               value="999999999" />
          <parameter   name="order-by"              value="TUPLENUM"  />
          <parameter   name="group-by"              value="destport,description"  />
          <parameter   name="timeout"               value="300"        />
      </box>

   </query>
```

Figure 5.13: Task 3: Version 2

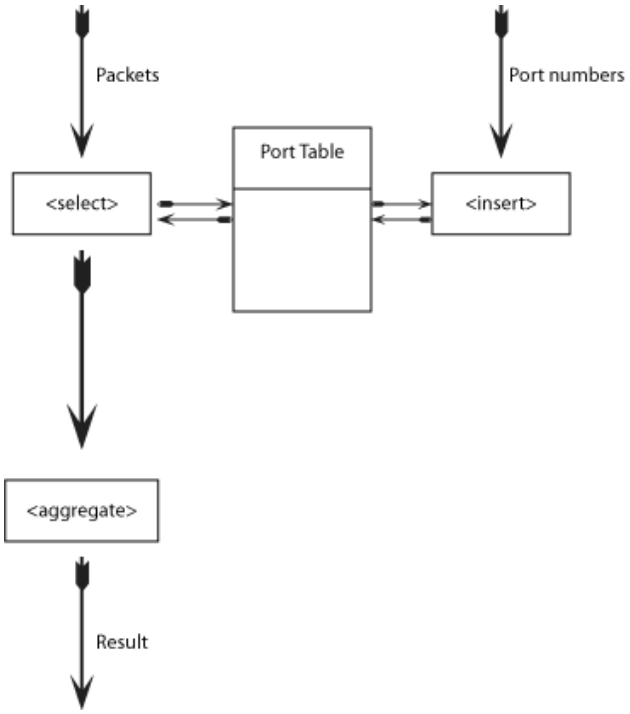

Figure 5.14: Task 3: Schematic drawing of version 2

packet headers. The other stream S_2 represents port numbers with descriptions, which will be stored in a static table. In order to store the data in a static table, we have to use the *insert* operator box. This box stores the tuples in S_2 in a *BerkeleyDB* table. This way, we can easily identify a high number of ports to count occurrences of. In our implementation, we read port numbers and their descriptions from a from a text file, and send them to Borealis from the client applicaiton. We base the text file on data retrieved from IANA (Internet Assigned Numbers Authority) [ian]. During our experiment, several thousands ports are sent to the table. In addition, we have also included one port number (60010), which is used as destination for all the generated packets.

Solution number two is definitely more adaptive in terms of selecting the ports to count packets destined for. The ports are simply read from a file at runtime. Adding additional ports, or changing existing ones, are also possible within the database. A notable difference between the two solutions is that solution number one does not have to upload the port numbers to the tables.

```
<box name="box1"   type="aggregate" >
     <in   stream="Packet"    />
     <out  stream="Aggregate" />

     <parameter   name="aggregate-function.0"  value="sum(totallength -
                          (ipheaderlength*4) - (tcpheaderlength*4) )" />
     <parameter   name="aggregate-function-output-name.0"  value="bytes" />
     <parameter   name="aggregate-function.1"              value="count()" />
     <parameter   name="aggregate-function-output-name.1"  value="count" />
     <parameter   name="window-size-by"                    value="VALUES" />
     <parameter   name="window-size"                       value="10" />
     <parameter   name="advance"                           value="10" />
     <parameter   name="order-by"                          value="FIELD" />
     <parameter   name="order-on-field"                    value="timestamp" />
     <parameter   name="group-by"                          value="sourceip,sourceport,
                                                                  destip,destport" />
 </box>
```

Figure 5.15: Task 4: Version 1

5.2.4 Task 4: Connection Sizes

How many bytes have been exchanged on each connection during the last ten seconds?

In order to design a solution for this task, we start by defining a connection. A TCP/IP connection is a connection between two hosts in a network. The source- and destination address, and the source- and destination port number of a packet can be used to define its connection. In order to establish a connection, a *three-way handshake* is performed between the two hosts [rfc81b]. In the beginning of a handshake, the client sends a packet with the SYN flag set to the server. The server responds a packet with both the ACK and SYN flag set. In order to synchronize the sequence and acknowledgement numbers between the client and the server, the acknowledgement number in the packet is set to $x + 1$, where x is the sequence number of the packet originally sent by the client. The final event in the *three-way handshake*, is when the client respond to the server with an ACK packet, with sequence number set to $x + 1$, and acknowledgement number set to $y + 1$, where y is the sequence number received from the server. There exists several ways of terminating a connection. Many connections are often simply timed out, although TCP support controlled terminations of connections that involve the header flag FIN.

Initially, our solutions are not based upon identifying three-way handshakes. Instead we simplify the task by using only *IP* and *portnumber* fields. This is based on a simple heuristic proposed by Plagemann et al. [PGB+04], where all packets with the same destination and source IP address and port number, belong to the same connection during a one minute window.

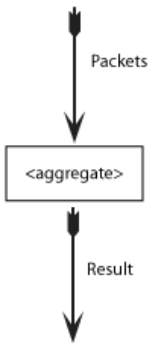

Figure 5.16: Task 4: Schematic drawing of version 1

The task of counting bytes sent only in a one-way connection can be solved by simply using an *aggregate* box, that groups by *sourceIP*, *destIP*, *sourcePort* and *destPort*. By using the *sum* operator, we can easily obtain the sum of bytes that the client has attempted to send to the server. We have designed such a solution shown Figure 5.15. A schematic drawing is presented in Figure 5.16. To summarize the actual bytes sent within the connection, we only want the size of the packet payloads. Because of this, we do not want to include the two fields *ipheaderlength* and *tcpheaderlength*. We calculate the payload with the following equation: $Payload = totallength - (ipheaderlenght * 4) - (tcpheaderlenght * 4)$. Note that we multiply the header length values with 4 to get the byte value, since their values represent the number of 32 bit lines they occupy, padding included, in the TCP/IP packet.

In a TCP connection, bytes are most often exchanged in a two-way fashion, which yields we have to join the tuples sent from host a to host b, with the tuples sent from b to a. The query language of Borealis supports two boxes for joining tuples. These are called *Join*, and *AuroraJoin*. We tend to use *Join*, since this operator is better explained in the Borealis documentation [Tea06]. (*AuroraJoin* seems to require the use of parameters that is not even mentioned in the documentation.) A straightforward join of tuples from the streams in the two directions, is not feasible. There is no guarantee that the number of packets sent from a to b, is the same as the number of packets sent from b to a. If the number of packets from a to b outnumbers packets from b to a, the outnumbered amount of packets from a to b will not be joined. A result of this is that we need to aggregate tuples sent from a to b into a tuple T_1, and join it with another aggregate tuple T_2, consisting of packets sent from b to a.

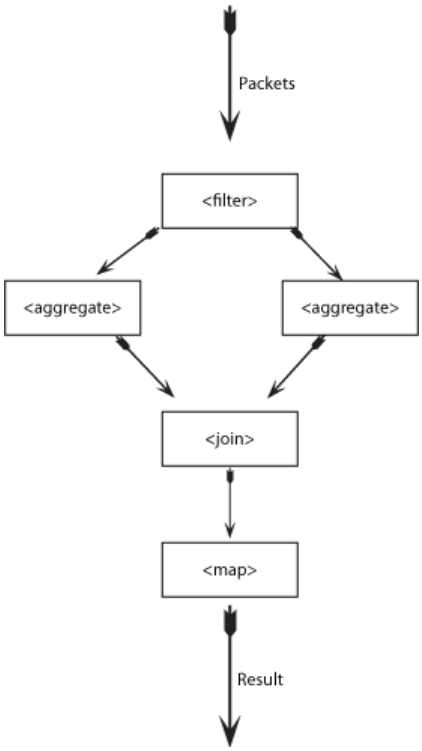

Figure 5.17: Task 4: Schematic drawing of version 2.

```
<query name="Query" >
  <box name="filterboxleft" type="filter">

      <in  stream="Packet"      />
      <out stream="Left"  />
      <out stream="Right"  />
      <parameter name="expression.0"              value="seqnum &gt; acknum" />
      <parameter name="pass-on-false-port"        value="1" />
   </box>

   <box name="box1"   type="aggregate" >
        <in   stream="Left"    />
        <out  stream="LeftAggregate" />

        <parameter   name="aggregate-function.0"  value="sum(totallength -
                                  (ipheaderlength*4) - (tcpheaderlength*4) )"  />
        <parameter   name="aggregate-function-output-name.0"  value="bytes"  />
        <parameter   name="aggregate-function.1"  value="count()"  />
        <parameter   name="aggregate-function-output-name.1"  value="count"  />
        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="10"        />
        <parameter   name="advance"               value="10"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp"    />
        <parameter   name="group-by"              value="sourceip,sourceport,
                                                      destip,destport" />
    </box>

    <box name="box2"   type="aggregate" >
        <in   stream="Right"    />
        <out  stream="RightAggregate" />

        <parameter   name="aggregate-function.0"  value="sum(totallength -
                                  (ipheaderlength*4) - (tcpheaderlength*4) )"  />
        <parameter   name="aggregate-function-output-name.0"  value="bytes"  />
        <parameter   name="aggregate-function.1"  value="count()"  />
        <parameter   name="aggregate-function-output-name.1"  value="count"  />
        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="10"        />
        <parameter   name="advance"               value="10"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp"    />
        <parameter   name="group-by"              value="sourceip,sourceport,
                                                      destip,destport" />
    </box>
```

Figure 5.18: Task 4: Version 2, first half

```
<box name="joinbox"  type="join" >
     <in   stream="LeftAggregate "    />
     <in   stream="RightAggregate " />
     <out  stream="Join" />

     <parameter name="predicate"  value="left.sourceip == right.destip and
                                         left.sourceport == right.destport and
                                         left.destip == right.sourceip and
                                         left.destport == right.sourceport and
                                         left.timestamp == right.timestamp" />

     <parameter name="left-buffer-size"  value="1" />
     <parameter name="left-order-by"  value="VALUES" />
     <parameter name="left-order-on-field"  value="timestamp" />
     <parameter name="right-buffer-size"  value="1" />
     <parameter name="right-order-by"  value="VALUES" />
     <parameter name="right-order-on-field"  value="timestamp" />
     <parameter name="out-field-name.0" value="leftsourceip"  />
     <parameter name="out-field.0" value="left.sourceip"  />
     <parameter name="out-field-name.1" value="leftsourceport"  />
     <parameter name="out-field.1" value="left.sourceport"  />
     <parameter name="out-field-name.2" value="leftbytes"  />
     <parameter name="out-field.2" value="left.bytes"  />
     <parameter name="out-field-name.3" value="rightsourceip"  />
     <parameter name="out-field.3" value="right.sourceip"  />
     <parameter name="out-field-name.4" value="rightsourceport"  />
     <parameter name="out-field.4" value="right.sourceport"  />
     <parameter name="out-field-name.5" value="rightbytes"  />
     <parameter name="out-field.5" value="right.bytes"  />
     <parameter name="out-field-name.6" value="timestamp"  />
     <parameter name="out-field.6" value="right.timestamp"  />
</box>


<box name="resultbox"   type="map" >
     <in   stream="Join"    />
     <out  stream="Result" />

    <parameter name="expression.0" value="leftsourceip"  />
    <parameter name="output-field-name.0" value="leftsourceip"  />
    <parameter name="expression.1" value="leftsourceport"  />
    <parameter name="output-field-name.1" value="leftsourceport"  />
    <parameter name="expression.2" value="rightsourceip"  />
    <parameter name="output-field-name.2" value="rightsourceip"  />
    <parameter name="expression.3" value="rightsourceport"  />
    <parameter name="output-field-name.3" value="rightsourceport"  />
    <parameter name="expression.4" value="timestamp"  />
    <parameter name="output-field-name.4" value="timestamp"  />
    <parameter name="expression.5" value="leftbytes + rightbytes"  />
    <parameter name="output-field-name.5" value="bytes"  />

</box>
```

Figure 5.19: Task 4: Version 2, second half

We design a solution where we split the stream of packets into two separate streams S_1 and S_2. S_1 consists of all packets that have a greater acknowledgement than sequence number. S_2 consists of packets with sequence number greater or equal to its acknowledgement number. Each of the streams are sent into aggregate operator boxes, with window sizes of 10 seconds. By joining their result tuples, and adding the exchanged bytes for each join, we get a single result for each connection per 10th second. The solution is presented in Figure 5.18 and Figure 5.19. In addition, a schematic drawing of the task is presented in Figure 5.17

The results from a 10 seconds window may not be correct in some situations. Borealis will drop the aggregated tuples from S_1 when they do not match any tuples from S_2. Since we operate within a 10 seconds window, tuples that have not been acknowledged within this interval will not be considered. We see a solution to this problem, as fairly intricate since there is not possible to fetch unmatched tuples directly from a *join* box. On networks like those we are performing our experiments on, we do not expect 10 second ACK delays, hence it should not affect our results.

An optimal solution to this task would be to identify each connection by their *three-way handshakes*, and as well only compute packets that are acknowledged within the TCP timeout interval. In addition to identify when connections are terminated, either by TCP timeout, or by the use of FIN flags. We have not managed to pose such a query by the use of the Borealis query language. We have managed to identify *three-way handshakes* with the use of *Filter*, *Join* and *WaitFor* boxes, although their disability of releasing un-matched tuples seems to make it hard for us to build queries which could deliver accurate results.

5.2.5 Task 5: Intrusion Detection

Identify Possible TCP SYN Flood attacks

TCP SYN Flood attacks are a type of a Denial-of-Service (DOS) attack, where the attacker initiates several half-open TCP connections to a server. Eventually the number of half-open connections becomes higher than the server can handle. This can then lead to the server denying services to legitimate hosts. The specific part of the TCP protocol exploited is the *three-way handshake*. By flooding a server with SYN packets from several fake sources, the server will initiate half-open connections to each of the faked sources, resulting in it reaching a maximum value of available connections, denying all hosts trying to connect.

```
<box name="synfilter"   type="filter" >
     <in   stream="Packet"    />
     <out  stream="Syn" />
     <out  stream="Normal" />

     <parameter    name="expression.0"           value="syn == 1"/>
     <parameter    name="pass-on-false-port"     value="1" />
 </box>

<box name="Normalcount"   type="aggregate" >
     <in   stream="Normal" />
     <out  stream="Aggregatenormal" />

     <parameter   name="aggregate-function.0"  value="count()"  />
     <parameter   name="aggregate-function-output-name.0" value="count"  />
     <parameter   name="window-size-by"        value="VALUES"   />
     <parameter   name="window-size"           value="1"        />
     <parameter   name="advance"               value="1"        />
     <parameter   name="order-by"              value="FIELD"    />
     <parameter   name="order-on-field"        value="timestamp" />
 </box>

 <box name="Syncount"   type="aggregate" >
     <in   stream="Syn" />
     <out  stream="Aggregatesyn" />

     <parameter   name="aggregate-function.0"  value="count()"  />
     <parameter   name="aggregate-function-output-name.0" value="count"  />
     <parameter   name="window-size-by"        value="VALUES"   />
     <parameter   name="window-size"           value="1"        />
     <parameter   name="advance"               value="1"        />
     <parameter   name="order-by"              value="FIELD"    />
     <parameter   name="order-on-field"        value="timestamp" />
 </box>

 <box name="SynfloodJoin" type="join" >
     <in  stream="AggregateNormal" />
     <in  stream="AggregateSyn" />

     <out stream="Result" />

     <parameter   name="predicate"            value = "left.count * 2 &lt; right.count" />
     <parameter   name="left-buffer-size"     value = "1" />
     <parameter   name="left-order-by"        value = "VALUES" />
     <parameter   name="left-order-on-field"  value = "timestamp" />
     <parameter   name="right-buffer-size"    value = "1" />
     <parameter   name="right-order-by"       value = "VALUES" />
     <parameter   name="right-order-on-field" value = "timestamp" />
     <parameter   name="out-field-name.0"     value="timestamp" />
     <parameter   name="out-field.0"          value="left.timestamp" />
     <parameter   name="out-field-name.1"     value="ratio" />
     <parameter   name="out-field.1"          value="right.count / left.count" />
     <parameter   name="out-field-name.2"     value="syn" />
     <parameter   name="out-field.2"          value="right.count" />
     <parameter   name="out-field-name.3"     value="normal" />
     <parameter   name="out-field.3"          value="left.count" />

</box>
```

Figure 5.20: Task 5: Version 1

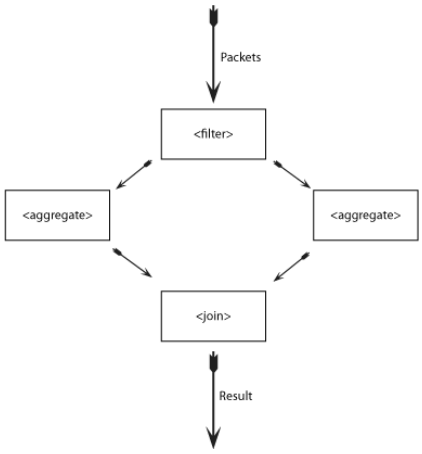

Figure 5.21: Task 5: Schematic drawing of version 1

A principal solution proposed by Cisco [cis], intended for their routers, is to alarm when the number of SYN packets n_s on a network exceeds the number of ordinary packets n_n with a certain multiple, for instance $n_s \geq 2 * n_n$. With Borealis this solution can be implemented by the use of a *filter* box, dividing the packet stream into a stream of SYN packets, and a stream of ordinary packets. By sending each of the streams through *aggregate* boxes, we can count their occurrences. This way we can join the tuples if a predicate is met; a predicate that demands that the number of SYN packets n_s is higher than $2 * n_n$, where n_n is the number of ordinary packets. Such a joined tuple can be seen as a warning of a possible attack. Figure 5.20 shows the query. In addition we have included a schematic drawing of the query in Figure 5.21

Johnson et al. [JMSS05] propose a principal solution, that is perhaps more accurate. The correlation between SYN packets and their matching ACK packets are identified during time intervals. As too many SYN packets are unmatched, a possible SYN Flood attack is identified.

Based on the solution proposed by Johnson et al. [JMSS05], we have designed a task version where we filter all SYN packets not having the ACK bit set, to a stream S_{syn}. We then send S_{syn} into a *WaitFor* box, connected directly to the packet stream. This *WaitFor* box keeps the S_{syn} packets, until a matching SYN/ACK packets arrives. A matching tuple must have source IP and source port number equal to destination IP and destination port number for the waiting tuple. And in addition, a match would require both the ACK and the SYN bit set, and also acknowledge number equal to $n+1$, where n is the sequential number from the waiting tuple. All matching tuples are sent to a stream S_{synack}, representing all SYN packets that have been acknowledged. We then send the two streams S_{syn} and S_{synack} into

```
<query name="query">
   <box name="SynFilter" type="filter">
       <in  stream="Packet"        />
       <out stream="Handshake1"    />

       <parameter name="expression.0"       value="syn == 1 and
                                                   ack == 0 and
                                                   fin == 0 " />
       <parameter name="pass-on-false-port" value="0" />
   </box>
   <box name="SynAckWait" type="waitfor">
      <in  stream="Packet" />
      <in  stream="Handshake1"    />
      <out stream="Handshake2"    />

      <parameter name="predicate" value="buffered.syn == 1 and buffered.ack == 1 and
                                         enabler.sourceip == buffered.destip and
                                         enabler.sourceport == buffered.destport and
                                         enabler.seqnum+1 == buffered.acknum"/>
      <parameter name="timeout"   value="10" />
   </box>

   <box name="syncount"   type="aggregate" >
        <in   stream="Handshake1" />
        <out  stream="Aggregatehs1" />

        <parameter   name="aggregate-function.0"  value="count()"  />
        <parameter   name="aggregate-function-output-name.0" value="count"  />

        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="1"        />
        <parameter   name="advance"               value="1"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp" />
        <parameter   name="group-by"              value="destip,sourceip" />
    </box>
```

Figure 5.22: Task 5: Version 2, first half

aggregate boxes, in order to count their occurrences. The aggregated tuples are then joined, if the count of S_{syn} packets are twice or more, the count of tuple tuples from S_{synack}. ($S_{syn} \geq 2 * S_{synack}$). Setting twice the count as a warning threshold is only done as an example. A predicate should also check that their timestamp and source/destination match. At the event of a joined tuple, a possible SYN Flood attack is identified. The solution is shown in Figure 5.22 and 5.23.

```
<box name="Synackcount"   type="aggregate" >
        <in   stream="Handshake2" />
        <out  stream="Aggregatehs2" />

        <parameter   name="aggregate-function.0"  value="count()"  />
        <parameter   name="aggregate-function-output-name.0" value="count"  />
        <parameter   name="window-size-by"        value="VALUES"   />
        <parameter   name="window-size"           value="1"        />
        <parameter   name="advance"               value="1"        />
        <parameter   name="order-by"              value="FIELD"    />
        <parameter   name="order-on-field"        value="timestamp" />
        <parameter   name="group-by"              value="destip,sourceip" />
    </box>

  <box name="SynfloodJoin" type="join" >
        <in  stream="Aggregatehs1" />
        <in  stream="Aggregatehs2" />

        <out stream="Result" />

        <parameter   name="predicate"            value = "left.count > 0 and right.count > 0 and
                                                          left.count &gt; right.count * 2 and
                                                          left.destip == right.sourceip" />
        <parameter   name="left-buffer-size"     value = "1" />
        <parameter   name="left-order-by"        value = "VALUES" />
        <parameter   name="left-order-on-field"  value = "timestamp" />
        <parameter   name="right-buffer-size"    value = "1" />
        <parameter   name="right-order-by"       value = "VALUES" />
        <parameter   name="right-order-on-field" value = "timestamp" />
        <parameter   name="out-field-name.0"     value="timestamp" />
        <parameter   name="out-field.0"          value="left.timestamp" />
        <parameter   name="out-field-name.1"     value="ratio" />
        <parameter   name="out-field.1"          value="left.count / right.count" />
        <parameter   name="out-field-name.2"     value="syn" />
        <parameter   name="out-field.2"          value="left.count" />
        <parameter   name="out-field-name.3"     value="synack" />
        <parameter   name="out-field.3"          value="right.count" />
        <parameter   name="out-field-name.4"     value="destip" />
        <parameter   name="out-field.4"          value="left.destip" />
  </box>
</query>
```

Figure 5.23: Task 5: Version 2, second half

Chapter 6

Experiment Environment

In this chapter we describe the experiments, where we have performed the tasks presented in Chapter 5. We start presenting the goals of our experiments, and describe our experiment setup in general in Section 6.2. This section also includes descriptions of the tools we have used.

In Section 6.3 we cover the system parameters of our experiments, and discusses alternative setups in Section 6.4. In addition, we also present additional system parameters found when performing *white box testing*[1] of the Borealis load shedding mechanism in Section 6.5. We present these parameters, since we believe they significantly affect our experiments.

6.1 Experiment Goals

By performing the experiment, we initially start with four main goals:

Goal-I: Verify the Borealis load shedding mechanism.

Goal-II: Show how good network monitor tasks can be expressed and performed by using Borealis and its continuous query language.

Goal-III: Find the accuracy of the task results.

Goal-IV: Find the maximum network load the tasks can handle.

We start by describing *Goal-I*: We try to verify the Borealis load shedding mechanism, by stressing Borealis with high network loads, performing a simple map operation of the packet headers. By increasing the traffic volume

[1] White box testing, in contrast to black box testing, is testing based on the internal structure of the program. In our case, we tried altering the source code in order to provoke certain events to happen.

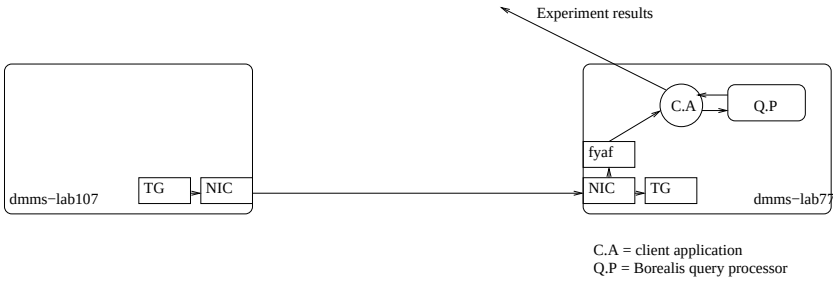

Figure 6.1: Schematic drawing of data flow in the experiment system

until Borealis reaches its limits, in terms of how much traffic it can handle, we expected to see the load shedding module (*Load Shedder*) start dropping tuples. Test runs early in the design phase, showed no traces of the *Load Shedder* module acting on overload situations. Instead we measure the effect off the *random_drop* boxes than can be used for load shedding.

For evaluation of *Goal-II*, we design and implement a set of predefined network monitoring tasks for Borealis. We want to evaluate whether or not they can be expressed, and how good.

To evaluate the accuracy of the task results, as part of *Goal-III*, we compare them with estimated values calculated by us. The calculations for the estimated results are based on our knowledge of the generated the network traffic data.

To determine how much network load Borealis can handle, *Goal-IV*, we perform several experiments with our setup, with increasing network load. We will consider the accuracy, in terms of correctness for the results, for each task at the different network loads. We will also determine how many packets that are dropped by *fyaf*[2] due to overload of the Borealis query processor.

6.2 Experiment System Description

In order to perform network monitoring in real-time, we have performed our experiments by looking at artificially generated packets on the network at the Distributed Multimedia Systems (DMMS) lab. We investigate only packets received on the NIC (network interface card) of the machine *dmms-lab77*, more specifically only those traversing the link between *dmms-lab77* and the machine *dmms-lab107*. We have chosen only to look at these packets, in order to be able to control the traffic behavior. The actual network traffic is generated with the use of the traffic generator TG [tg], run on both machines.

[2]fyaf is explained in Section 6.2.

dmms-lab77 runs the Borealis processing nodes, and also the Borealis client applications. By running the packet filtering tool *fyaf* [Søb06] [Her06], the selected TCP/IP headers are sent to the Borealis client application through a TCP connection. The traffic generator and the NIC packet header filtering with *fyaf*, are further described in Section 6.2.1 and Section 6.2.2. We present system parameters affecting the performance of our experiments in Section 6.3.

6.2.1 NIC Packet Filtering with fyaf

The actual transformation of packet headers is performed with an application called *fyaf* [Søb06] [Her06]. It is a filtering tool that transforms raw data from network packets, into a comma separated format, accepted by most DSMSs. The packets are retrieved with the use of the *pcap* library interface, used to dump packets directly from the NIC [Søb06]. After the retrieval of the packet headers, they are transformed and sent to a receiver through a TCP socket connection. The receiver, in our setup, will be the Borealis client application.

In order to achieve accurate amounts of network traffic load in our experiments, *fyaf* could be set to only filter packets traversing a certain link. This is how we can filter only the packets traversing between *dmms-lab107*, and *dmms-lab77*.

fyaf also calculates the number of packets that the DSMS due to slow stream processing time, has not managed to retrieve. The feature of counting these packets is heavily used throughout our experiments, as it is of significant interest. The number of lost packets is a good way of identifying what network loads Borealis can handle. We store *fyafs* output on lost packets for each experiment.

6.2.2 Generating Traffic with TG

In order to generate accurate network loads, we have used *TG 2.0* [tg], which is a traffic generator tool developed at SRI International (SRI) and Sciences Institute of the University of Southern California (USC-ISI). *TG* generates traffic from a client, to a server that is acting like a sink node. By sink node, we mean that it receives the network flow, rather than being the source of it. The behavior of the generated network traffic can be expressed in the provided specification language. This enables us to address the remote host to act as the receiving sink, and set parameters for the behavior of the traffic.

Both the UDP and TCP traffic can be generated. There exist a variety of Quality of Service (QoS) parameters that can be set in order to generate

Parameter	Value	Software/Hardware	Description/may affect
CPU dmms-lab77	3 GHz	H	Overall performance
RAM dmms-lab77	1 GB	H	Overall performance
OS dmms-lab77	Linux (2.6), FC 2	S	Overall performance
CPU dmms-lab107	3 GHz	H	Overall performance
RAM dmms-lab107	1 GB	H	Overall performance
OS dmms-lab107	Linux (2.6), FC 5	S	Overall performance
MAX_BW (Borealis)	1000000	S	Bandwidth within Borealis
MAX_UTILIZATION(Borealis)	0.99	S	Max CPU load
MAX_BUFFER(Client App.)	64000	S	Max size of input buffer
SLEEP_TIME (Client App.)	1 ms	S	Sleep time between batches
BATCH_SIZE (Client App.)	45	S	Numbers of tuples per batch

Table 6.1: System Parameters

traffic with certain behavior. At our setup, the execution scripts presented in Appendix C set these parameters. The parameters are mainly used to control the network workload, and the number of generated connections. We present the workload factors further in Chapter 7.1.1.

6.3 Experiment System Parameters

In this section we present important system parameters affecting performance during our experiments. We have chosen to separate all parameters affecting the experiments into system parameters and workload parameters as in [Jai91]. System parameters are constant parameters, within our system. In contrast, the workload parameters are factors we set in order to measure the performance of the system. We present the workload parameters in Chapter 7.1. The system parameters are presented in Table 6.1.

We now include a deeper presentation of the system parameters SLEEP_TIME and BATCH_SIZE, since they significantly affect our experiment results. We choose to see them as system parameters, as with all parameters set in the Borealis client applications. The parameter values will not vary among the experiments.

The SLEEP_TIME and BATCH_SIZE parameters are declared within each Borealis client application. They concern a DSMS technique known as *batch processing* [BBD+02]. This technique is used to speed up query execution, by buffering data elements as they arrive within a certain time interval. In addition to speeding up query execution, batching can to some extent compensate for bursty load. As opposed to *sampling*, *batch processing* does not cause inaccurate answers, timeliness is instead sacrificed [BBD+02]. Figure 6.2 shows a schematic drawing of tuples in a batch.

After sending a batch of tuples from a client application to a processing node, a certain sleep interval is required in order to process the tuples. This

a_1,b_1,c_1
a_2,b_2,c_2
a_3,b_3,c_3
...
a_n,b_n,c_n

Table 6.2: Showing a batch consisting of n tuples

requirement manifests itself through a required function argument within the generated marshal code. In the example Borealis client applications provided with the release, the BATCH_SIZE and SLEEP_TIME parameter are set in order to control the send rate of the tuples. Instead we try to set these values so that a highest possible rate of tuples can be sent to the query processor. Sleeping as little as possible will result in higher accuracy, as tuples do not have to wait to be processed. Thus, we have given SLEEP_TIME the value of 1, which is the smallest value allowed.

For the BATCH_SIZE parameter, we have run several tests, in order to find a suitable value. Setting the BATCH_SIZE to high has resulted in Borealis loosing a high number of tuples, no matter what network load it operates on. This loss of tuples is caused by a limitation within the source code, restricting the number of tuples being processed at the same time. We present the variable, defining this limitation in Section 6.5. High values of BATCH_SIZE have also shown to give inaccurate query results at low network loads, as tuples are not sent through the query before the batch is filled up. Setting it to low, we are restricting the maximum workload, as only a certain number of tuples are allowed each millisecond[3].

We have chosen to set BATCH_SIZE = 45. Experiments that show this as a suitable value are presented in Appendix D.

6.4 Alternative Setups

A possible alternative setup that in future experiments should be tested, is to perform the actual filtering of the packets within the Borealis client application. This would mean to read the headers directly from the *pcap* library interface without using *fyaf*. Incorporating *fyaf*s functionality in Borealis client applications, should be possible. Doing so should lead to less load on the machine performing the queries. Both the read and write operations through the TCP sockets to *fyaf* would not be needed, neither the transformation of the TCP/IP headers between the *fyaf* representation, and the

[3]For example a SLEEP_TIME of 1 ms and a BATCH_SIZE of 5 given an average packet size of 796 bytes, would result in $\frac{5*1.000s*796bytes*8bits}{1.000.000bytes} = 31,8Mbit/s$ of maximum possible relative throughput.

Borealis preferred structure types. A drawback would be the loss of accurate comparison of experiment results, with the results from the experiments performed by Hernes [Her06] and Søberg [Søb06].

6.5 Beyond the Black Box Testing

In order to provoke the *Global Load Shedder* module to perform load shedding, we have tried to alter specific variables within the source code[4] of Borealis. During our exploration of the source code, we identified several important variables set to limit resource utilization. By limiting utilization, the challenges of handling an overloaded system are simply met by avoiding it. A drawback is that migration of such a system onto more powerful machines necessarily does not increase the possible system performance. We cannot know in what degree the parameters are limiting the utilization at our setup. At least not without performing further testing. Measurements and analysis of performance with other values than set in the release, might be subject in further studies. We include a list of important variables:

- MAX_BUFFER is set in the generated part of the client application. It restricts the number of tuples waiting to be processed. In our setup the restriction value is set to 64,000. When overload occurs, Borealis often complains with an error message that the restriction is violated.
- TUPLE_PROCESSING_LIMIT is set to 100, and limits the number of tuples being processed within at certain period of time. After mail exchanges with the Borealis developers, we have been told that setting it lower, will provoke the *Load Shedder*.
- MAX_UTILIZATION is set to 0.99. The definition within the source code states that it limits the maximum CPU utilization for the node.
- MAX_BW is set to 1,000,000. We have performed test where BUFFER_SIZE within client applications have been set immensely high. Even though this has led us to achieved higher tuple processing rates, it has shown to result in a new error message. We believe this error message is caused by a violation of the MAX_BW parameter.
- MAX_SOCKET_BUFFER is set to 64,000. We believe it affects the maximum number of tuples, held in the socket buffer queue between the client application and the query processor.

[4]The variables can be found in the file src/common/common.h, within the Borealis source code.

Chapter 7

Performance Evaluation of Borealis

In this chapter we evaluate the performance of Borealis performing the network monitoring tasks, presented in Chapter 5. During each experiment run, task results, and measurements of CPU utilization and memory consumption are written to files. Based on these data, we will evaluate the performance of Borealis. We start with a general description of our evaluation in Section 7.1, and define workload factors and evaluation metrics. The results from the evaluation are presented throughout Section 7.2. Section 7.3 summarizes our findings.

7.1 Description of Evaluation

In order to perform our evaluation, we start by identifying the set of workload factors that are used in our experiments. We explain each of the factors, and present the actual values set in our experiments. We further introduce a set of metrics, used to measure our results. During our evaluation we will present measured results in a collection of different types of graphs and tables. We explain these graphs and tables in general, at the end of this section.

Note, that throughout the evaluation we use 1 Mbit, for the value $1,000,000$ bits. Others may refer to the value as 2^{20}, although we consider such a reference as not correct [meg]. After performing the experiments, we have found that 1 Mbit is referred to as 2^{20} bit, within the experiment scripts. Hence in the generated traffic, we expect some inaccuracy when measuring the network load.

7.1.1 Factors

Time and storage consumption limits the number of runs we are performing for each experiment. For each task, we often include several versions, and each of them is performed several times. The experiments are run with different network loads, each with a duration of several minutes. For each of them, several system and result values are written to disk during execution. With restriction of time and storage consumption in mind, we have specified a duration time for each of the tasks to run. We have also specified a set of network loads that the tasks will be tested with. We describe each of these workload factors, and present the actual values used for our experiments in Table 7.1.

- Duration time D_t (minutes) is a factor for how long each of the tasks is running at each *stage* of the experiment. By *stage* we mean a single version of a task, running on a single network load during a duration period D_t.

- Network Load N_l (Mbit/s) is a factor for how much workload in terms of generated bits per second (Mbit/s) that Borealis receives. During our experiments, we have chosen to generate stable network loads, as opposed to bursty network loads. For instance, on a 5 Mbit/s Internet connection, the connection line is seldom fully utilized in terms of bandwidth. If Borealis can handle 5 Mbit/s performing a query, during D_t, we claim that it can be deployed as a network monitor, on that Internet connection. The network load represents a worst-case value, in which we try to identify whether or not Borealis can handle in.

- Packet size P_s (Bytes) is a metric for the total size of each packet. The value resides in the *totallength* field in the IP header. The actual packet size we have set TG to generate is 796 Bytes. This value was chosen for our experiment setup, since the same value was used in the experiments of Søberg [Søb06] and Hernes [Her06]. By using the same packet size, we can later compare the experiment results. Note, that the packet size received from *fyaf* has shown to be different from what we set TG to generate. We believe this is caused by the TCP implementation used in the experiment setup, which has placed the payload delivered from TG in segment sizes that differ from what TG is set to send. We expect this behavior is caused by a built in mechanism in TCP. The mechanism is supposed to optimize packet segments sizes delivered to the IP layer. The mechanism we refer to, is based on solving the small-packet problem described in RFC0896 [Nag84], also referred to

Parameter	Task 1	Task 2	Task 3	Task 4	Task 5
D_t (s)	600	600	905	600	600
N_l (Mbit/s)	1,5,10,20,$\cdots$,80	1,5,10,20,$\cdots$,60	1,5,10,20,$\cdots$,60	1,5,10,20,$\cdots$,60	1,5,10,20,$\cdots$,60
N_r (count)	5	5	5	5	5
N_c (count)	1	1	1	10	20
P_s[1] (bytes)	796	796	796	796	796

Table 7.1: Workload factors

as *Nagles algorithm*. The idea is to reduce the number of packets that need to be sent through the network, in order to decrease network load caused by overhead of the TCP packet headers.

- Number of test runs N_r is a factor for how many times each stage of an experiment is run. This means the number of runs for a single task version, on a specific network load N_l.

- Number of clients N_c is a factor for how many client connections TG is set to generate. N_c is most often set to 1, since it for most of the tasks should not affect the experiment results. Task 4 and Task 5 will on the other hand be affected by the number of clients N_c, hence we have set them higher. (See Table 7.1.)

7.1.2 Metrics

Our evaluation of the network monitoring tasks is based on output results from the continuous queries, and system resources logged during the experiments. In the following subsections, we discuss the metrics we use in our evaluation.

Supported network load

By supported network load, we mean the network load N_l that we measure that Borealis can handle. In order for Borealis to handle a certain network load, we expect *fyaf* to report a packet loss of 0%. We introduce a metric for lost packets L_p. L_p is the average percent of lost packets that *fyaf* reports for each experiment run, for each specific network load. L_p is calculated by looking at the total number of packets that *fyaf* has sent, and the total number of packets that *fyaf* has reported to be lost. We include a subsection on *Dropped packets reported by fyaf*, for all task evaluations, where we identify the supported network load.

Accuracy of the results

Task results are evaluated by looking at the output from the queries. Most often we present the arithmetic mean values, for the task results. Equation 7.1 shows the formula used to compute the arithmetic mean value $\bar{T_r}$, for a measured result. x_i is the measurement for the single experiment run. Ad-hoc scripts, used to read the task results, will perform the actual computation.

$$\bar{T_r} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{7.1}$$

For each task version, we will identify the *accuracy* by looking at the output from the tasks, and compare it with our calculated expected values. By looking at the *fyaf* output and knowing the behavior of the generated traffic, we are most often able to calculate the expected values. We introduce a metric for accuracy A_c, which is measured in percent. In order to show how A_c is calculated, we start with defining our metric for error. The error E_r is calculated for each measured result $\bar{T_r}$ in Equation 7.2. C_v is the calculated estimate.

$$E_r = \frac{C_v - \bar{R_r}}{C_v} * 100 \tag{7.2}$$

Based the Equation 7.2, we define our metric for accuracy A_c. Equation 7.3 shows how it is calculated.

$$A_c = 100 - |E_r| \tag{7.3}$$

Consumption of system resources

We present *CPU utilization* and *memory consumption* in order to identify possible overload situations and overall system utilization. We will look at values for the Borealis process only. These values are measured each second during the experiments, with the use of the Unix application *top*. For each task, we include both a subsection describing the CPU utilization, and a subsection describing the memory consumption. We define metrics for both of them:

- CPU utilization C_{pu} (percentage) is a metric for how much CPU is utilized during our experiments. We use *top* to investigate the CPU usage during our experiments.
- Memory consumption M_c (percentage) is a metric for the memory resources used by the system.

7.1.3 Presentation of Results

Throughout the evaluation in Section 7.2, we introduce a set of graphs and tables in order to present our measured metrics for all runs of the different task version experiments.

- *Lost packet graphs* is used to present the metric L_p. The plotted values are averages from the five experiment runs, for each network load the experiments have run on.

- *Task result graphs and tables* are used to present the query results, as well as the accuracy A_c, from each of the task versions. The presented values are averages of the output from the five experiment runs, at each network load.

- *CPU average graphs* are used to present the average CPU utilization C_{pu} during the experiments. One graph presents values for each of the task versions, at all network loads.

- *CPU maximum value graphs* are used to present the maximum CPU utilization for each of the task versions. We choose to look at maximum values, especially where we want to identify overload situations.

- *Memory average graphs* are used to present the amount of memory M_c consummed by Borealis during the experiments. The values are averages from the five runs, presented for each task version, at each of the network loads. As with CPU, we also present maximum memory graphs, since we often are interested in the maximum values.

- *CPU comparison graphs* are used to present single experiment runs, when comparing different task versions.

- *Memory comparison graphs* are used to present memory consumption for single experiment runs, and compare them for different task versions.

Note, that when we present CPU utilization and memory consumption during the experiments, we only present from measurements the Borealis process. The Borealis process is the process that performs the query processing. Investigations of client application resource utilization and consumption are not presented, since we are more interested in the performance of the query processor. Investigations however show that the client applications consume fewer resources than the query processor during the experiments, as expected.

Version	Drop rate
dr: 0	0
dr: 0.2	0.2
dr: 0.4	0.4
dr: 0.6	0.6
dr: 0.8	0.8

Table 7.2: Different drop rates used in the Task 1 versions

7.2 Evaluation of Tasks

In this section we present the results from the network monitoring tasks, five in total. A subsection is included for each of them, showing measured results from the different task versions.

7.2.1 Task 1: Load Shedding

This task was originally intended for verifying the load shedding mechanism in Borealis. A complete load shedder, should perform two important operations:

- Identify overload situation
- Shed load by dropping tuples, when overload occurs

As discussed in Chapter 5, due to compilation problems, we cannot verify the module that is supposed to identify overload situation. But we have managed to successfully deploy the module that covers the second functionality of a load shedder; to shed load. This functionality is covered by the *random_drop* operator box. The operator box sheds load by dropping a certain percentage of tuples, randomly. Figure 7.2 shows the drop rates we have tested. By running several task versions with different drop rates, we can evaluate the effect of the supported load shedding mechanism.

Dropped packets reported by fyaf

In order to identify what load Borealis can handle, we start by looking at Figure 7.1. The figure shows the average number of dropped packets at each network load, reported by *fyaf*. We expect that *fyaf* will report significant packet losses, as we the experiments reach higher network loads than Borealis can handle.

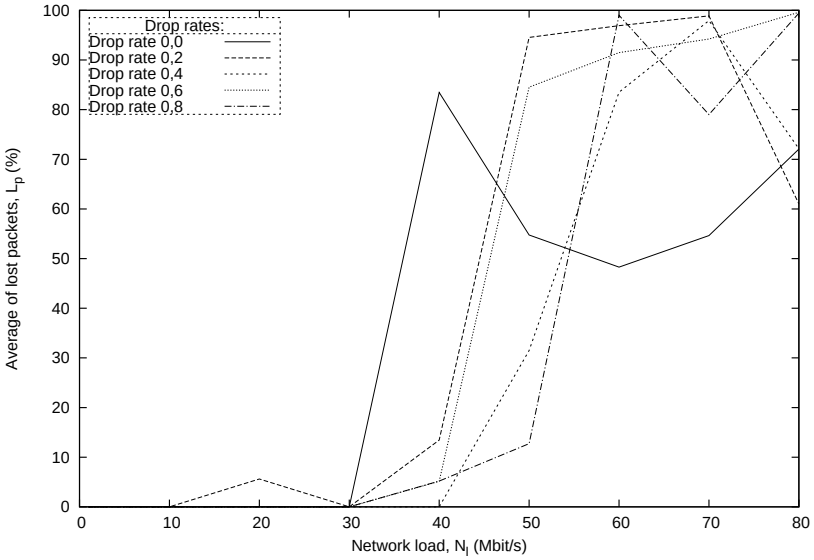

Figure 7.1: Task 1: Lost packets

At a network load of 20 Mbit/s, the plot shows that *Task 1 dr: 0.2* reports an average packet loss of 5.6%. Further investigations show that packet losses only are reported during experiment run 4. We suspect this may be caused by an unknown overload situation in our experiment setup. At a network load of 30 Mbit/s, no task version reports any significant packet loss. At 40 Mbit/s however, all but *Task1 dr: 0.4*, report packet losses ranging from approximately 5% for *Task1 dr: 0.6 and dr: 0.8*, to 13% for *Task1 dr: 0.2* and 80% for *Task1 dr: 0*. At a N_l of 50 Mbit/s, we see that all versions drop a significant amount of packets. *Task1 dr: 0.8* is the one that drops the least amount of packets, approximately 12%. *Task1 dr: 0.4* drops 31%, and *Task1 dr: 0,6 and dr: 0.2* both drop more than 80% of the packets. However we see an interesting decrease of L_p for *Task1 dr: 0* at N_l of 50 Mbit/s. This is probably caused by a crash in our experiment setup, and that the number of total packets received from *fyaf* has been reported lower than it should have been. We believe that the cause of the decreasing L_p for the other versions are the same.

The highest supported N_l is 40 Mbit/s, for *Task1 dr: 0.4*. This is 30% more network load than *Task1 dr: 0, dr: 0.2, dr: 0.6 and dr: 0.8* is supporting, hence the *random_drop* boxes have only shown to increase the supported network load at a drop rate of 0.4. In other words, *random_drop* boxes does not dramatically increase the supported network load.

N_l	v. 0	v 0.2	v. 0.4	v. 0.6	v. 0.8
1 Mbit/s	100%	80.1%	60.1%	40%	20%
5 Mbit/s	100%	80%	60%	40%	20%
10 Mbit/s	100%	80%	60%	40%	20%
20 Mbit/s	100%	80%	60%	40%	20%
30 Mbit/s	100%	80%	60%	40%	20%
40 Mbit/s	-	-	60%	-	-

Table 7.3: Task 1: Accuracy

Accuracy of the results

The output from Task 1, consist of all the packet headers sent during each of the experiments. We have not stored all the original *fyaf* data, and base our calculations for the accuracy A_c of the count of packets present in the result files. These are compared to the number of packets sent by *fyaf*. We expect the accuracy to decrease, as we are increasing the drop_rate. We present the results in Table 7.3. The results are as expected, and shows that the amount of dropped packets with the use of the *random_drop* box, is consistent with the achieved accuracy.

CPU Utilization

In order to present CPU utilization, we have chosen to present three plots. We start with a discussion of Figure 7.2 showing the average CPU utilization for the task versions, at the different tested network load. The figure shows average C_{pu} values of approximately 2 - 3% for all versions, at 1 Mbit/s. We see that the version with the highest CPU utilization is *Task1 dr: 0.2*. Further investigation of the measured results shows that it utilizes 0.12% more CPU than *Task1 dr: 0*. This can be explained with the fact that *Task1 dr: 0* does not contain any *random_drop* box. At a network load of 5 Mbit/s, we see that *Task1 dr: 0.2* utilizes 0.93% more CPU than *Task1 dr: 0*. We claim that deploying a random_drop box with only a drop rate of 0.2, might lead to even higher CPU utilization. At a network load of 20 Mbit/s *Task1 dr: 0* shows higher CPU utilization than *Task1 dr: 0.2*, but we expect this is only because *Task1 dr: 0.2* has suffered the packet losses shown in Figure 7.1.

As long as the drop rate is set to 0.4, or more, the plot is showing that increasing drop rates decrease CPU utilization. At 5 Mbit/s, we see that the CPU utilization measurements are evenly distributed between 11 - 14%. The measured results show 14% for *Task1 dr: 0*, 13% for *Task1 dr: 0.4*, 11% for

Task1 dr: 0.6 and 8.6% for *Task1 dr: 0.8*. By looking at the plots, we see a similar trend for network loads up until 30 Mbit/s. At this point, we see sudden drops for most versions, as they are reporting to drop packets. This does not apply for *Task1 dr: 0.4*, as it did not start dropping packets until 40 Mbit/s. The reason for the sudden increase of CPU utilization for *Task1 dr: 0*, from 40 Mbit/s to 60 Mbit/s is unknown.

In overload situations, maximum values are often of interest, hence we include Figure 7.3. The figure shows the measured average of maximum CPU utilization, for each task version, at each network load. We see that the plots are very similar to the ones in Figure 7.2, but there is a clear distinction between the plots at 30 Mbit/s of network load. All plots in Figure 7.2, except *Task1 dr: 0.4*, are rapidly decreasing, but in Figure 7.3 they are in fact increasing. *Task1 dr: 0* reaches a maximum of 75% at 40 Mbit/s. All plots show a decrease in maximum values for CPU utilization from 40 Mbit/s and towards 60 Mbit/s. From this, we can tell that overload, leading to lost packets, occurs before CPU utilization reaches its maximum. The cause of overload is probably not that the system reaches a maximum of CPU utilization.

The last figure for CPU utilization we introduce is Figure 7.4. It shows a CPU utilization comparison between *Task1 dr: 0* and *Task1 dr: 0.8*. Each of the plots show a single experiment run, at a network load of 30 Mbit/s. The plots clearly show that the *Task1 dr: 0* utilizes more CPU than *Task1 dr: 0.8*. More precisely, *Task1 dr: 0* consumes approximately 15% more CPU than *Task1 dr: 0.8*.

Memory Consumption

By looking at the Borealis memory consumption, we are interested in identifying the effect *random_drop boxes* have on relieving the Borealis with memory. We include two figures, and start by describing Figure 7.5. The figure shows the average maximum measured memory consumption values, for the different task versions, at the different drop rates. We are interested in the maximum measured values, as we want to identify possible overload situations. For *Task1 dr: 0, dr: 0.2, dr: 0.4, dr: 0.6 and dr: 0.8*, we see stable maximum values at 3.1% up until 30 Mbit/s of network load. In other words, it seems *random_drop boxes*, has no effect on memory consumption. At 40 Mbit/s, we see that *Task1 dr: 0.0*, shows an increase up to 3.42%. This corresponds to a increase of 3.2 MBytes. From Figure 7.1, we know that *Task1 dr: 0.0* has dropped packets at 40 Mbit/s, hence we have an overload situation. We believe the increase is caused by the overload situation. *Task1 dr: 0.2*, and *Task1 dr: 0.4*, shows increases at 50 Mbit/s, while *Task1 dr: 0.0*

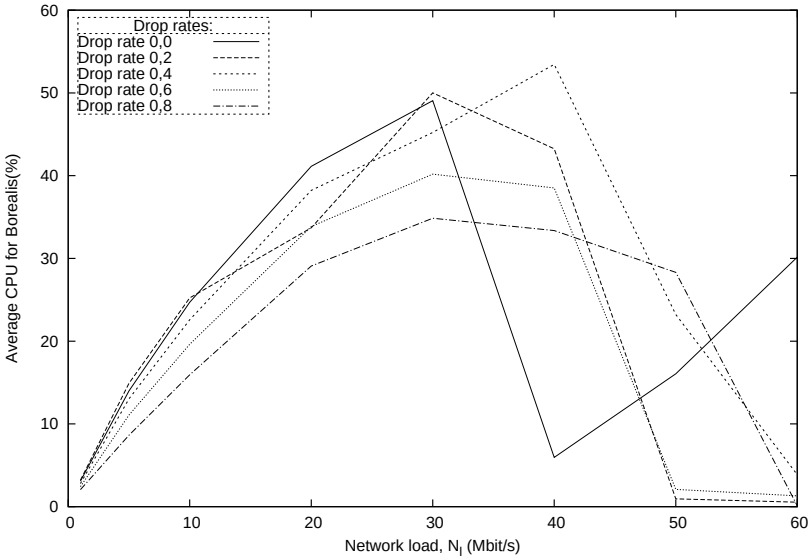

Figure 7.2: Task 1: Average load for the Borealis process

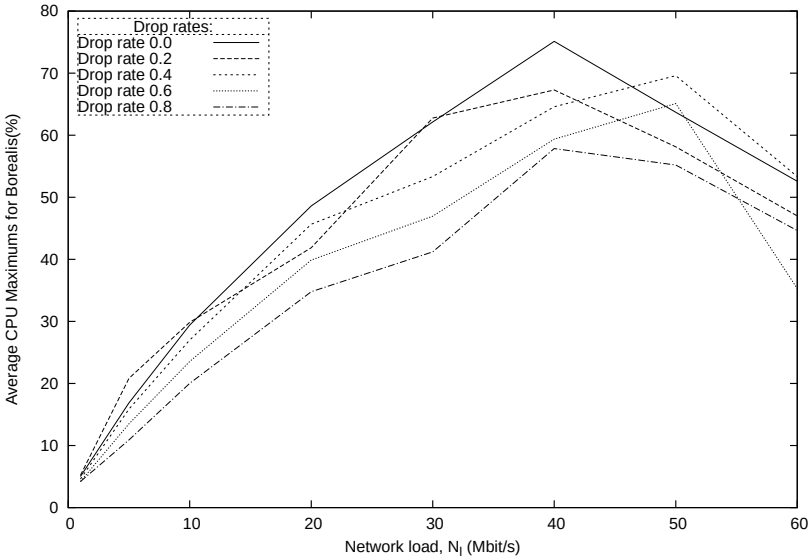

Figure 7.3: Task 1: Average maximum CPU values for the Borealis process

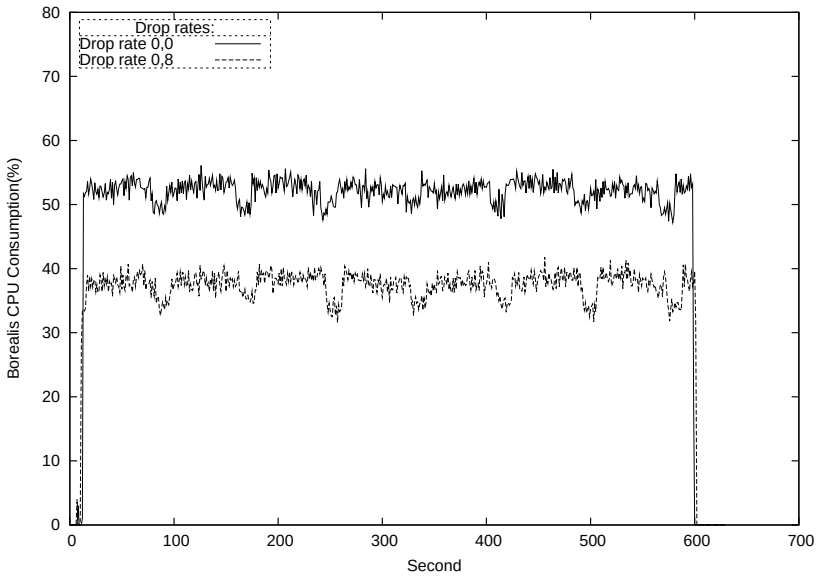

Figure 7.4: Task 1: Single run comparison of CPU utilization at 30 Mbit/s for *Task 4 v. 1 and v. 5*

shows a decrease. We believe inconsistent behavior of the measured results is caused by the fact that Borealis and the experiment systems behave unstable at such loads.

We also include single experiment comparison for average memory consumption between *Task1 dr: 0.0* and *Task1 dr: 0.8*. The comparison is shown in Figure 7.6. The experiments were run at a network load of 30 Mbit/s, during 600 seconds. The plots show that the two versions consume almost exactly the same amount of memory during the experiment. A notable behavior of the plots, is that we see an increase for the average memory consumption during the first seconds for both of the task versions. This is probably caused by the fact that the experiment was just about started, and that the network traffic had not reached full speed.

7.2.2 Task 2: Average Load

In Task 2, we measure the average amount of packets per second, and the network load per second, in a one minute interval. We introduce two metrics for the query results. Packet rate P_r (packets/s) is a metric for the rate of packets received by Borealis each second. In addition to P_r, measured network load M_l (Mbit/s) is a measured metric for how much network load the packets represent in terms of traffic that has been retrieved from the NIC.

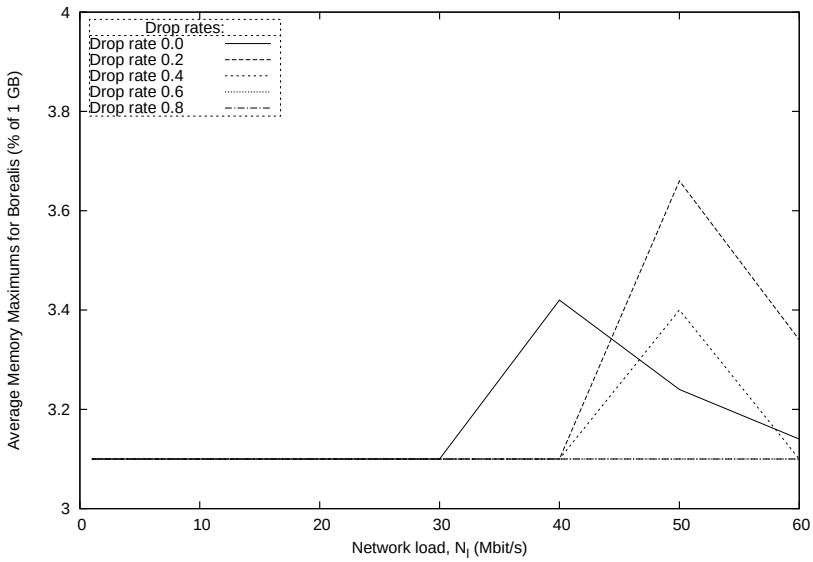

Figure 7.5: Task 1: Average maximum memory values for the Borealis process.

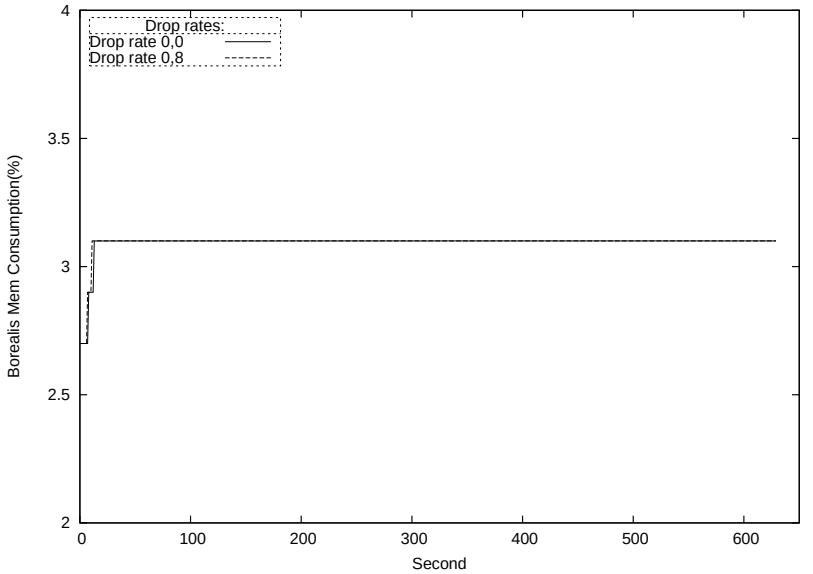

Figure 7.6: Task 1: Comparison of memory consumption at 30 Mbit/s network load for *Task1 dr: 0.0* and *Task1 dr: 0.8*

Version	Window size (seconds)	Based on
ws: 1	1	Initial version 1
ws: 20	20	Initial version 2
ws: 40	40	Initial version 2
ws: 60	60	Initial version 2

Table 7.4: Different window sizes used in the different versions

We expect $M_l \approx N_l$. Equation 7.4 shows how M_l is calculated, given a fixed packet size.

$$M_l = P_r * P_s \tag{7.4}$$

From what we have seen in test runs early in the task design phase, we suspected that the window sizes used in the *aggregate boxes*, has significant effect for the supported network load. We included several versions with different window sizes, in order to identify a possible trend. Table 7.4, shows the task versions and their window sizes. Note that we have not tested bigger window sizes because this would not fit into the predefined task description very well.

Dropped packets reported by fyaf

The number of lost packets is presented in Figure 7.7. The plots shows that *Task 2 ws: 1, ws: 20 and ws 40*, did not suffered from any significant packet loss up until a network load of 40 Mbit/s. *Task 2 ws: 60*, however, suffered a 4.8% packet loss at 20 Mbit/s and a 5.1% packet loss at 30 Mbit/s. At both 20 Mbit/s and 30 Mbit/s, only one out of five experiment run reported packet losses, hence the cause might be experiment inaccuracy, etc.

At a network load of 40 Mbit/s, *Task 2 ws: 1, ws: 20 , ws: 40 and ws: 60*, reports close to 0% packet losses, but at 50 Mbit/s they all loose packets. From here, the packet losses ranges from 100% for *Task 2 ws: 40*, approximately 15% for *Task 2 ws: 20 and ws: 30*, and approximately 5% for *Task 2 ws: 60*. Based on this, we cannot conclude any notable effect off increasing the window size, in terms of lost packets, other than during overload situations, where measurements show different values for the different window sizes. These variations might be caused by the random behavior of a overloaded system.

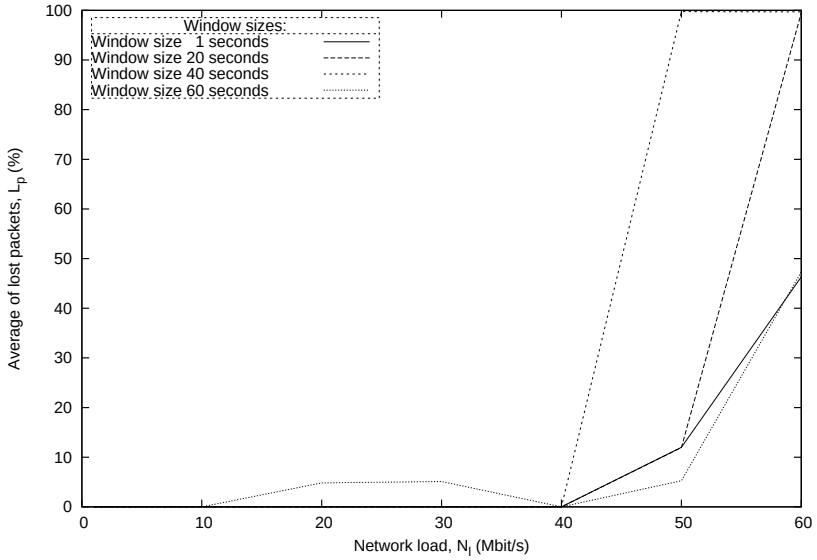

Figure 7.7: Task 2: Lost packets

Accuracy of the results

We start by presenting the average network load per second. Figure 7.8 shows the average results of measured network load M_l over a one minute interval, reported each second. The graph shows the results at the different network loads, for each task version. For these results, we expect $N_l \approx M_l$, hence the plot of M_l should reassemble a plot of the function $f(x) = x$.

The plot in the graph in Figure 7.8 show that the measures of M_l for all Task 2 versions, are a bit higher than N_l. We believe this is caused by the fact that the generated traffic are based on the consumption that 1 Mbit equals 2^{20} bits, not 1,000,000.

We present the average accuracy A_c for the versions in Table 7.5. The table show that all measured values have an accuracy A_c of 90% or better. As the network load increases, so does the accuracy. We see no clear trends that window sizes affect the accuracy.

Task 2, also measures the average number of packets per second. For this, we have introduced the metric P_r. We startby presenting a plot of P_r in Figure 7.9. The figure show average values for the experiment runs, for each version, at the different network loads. The plot is not straight, as the packet sizes varies among the different network loads. Because of this, we have not been able to estimate the exact expected values, only based on what we know from the behavior of the generated traffic. Initially, we set the generated traffic to use packets with the total size of 796 bytes. Investigation

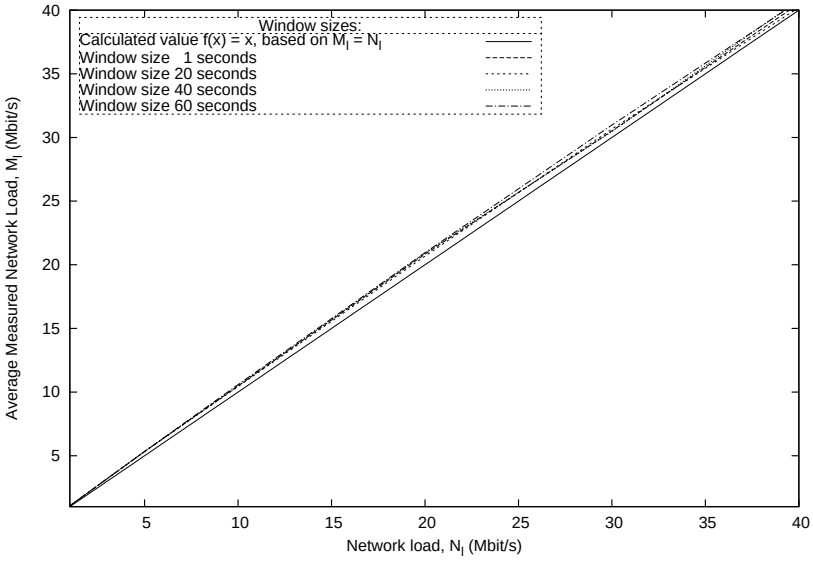

Figure 7.8: Task 2: Average measured network load

N_l	Task 2 ws: 1	Task 2 ws: 20	Task 2 ws: 40	Task 2 ws: 60
1	93.2%	93.8%	94.0%	92.5%
5	93.3%	93.3%	93.4%	93.3%
10	95.1%	95.7%	95.2%	94.1%
20	95.8%	96.6%	98.4%	95.2%
30	98.3%	97.7%	98.5%	96.7%
40	98.7%	99.5%	98.0%	98.2%

Table 7.5: Task 2: Average accuracy A_c of measured network load M_l

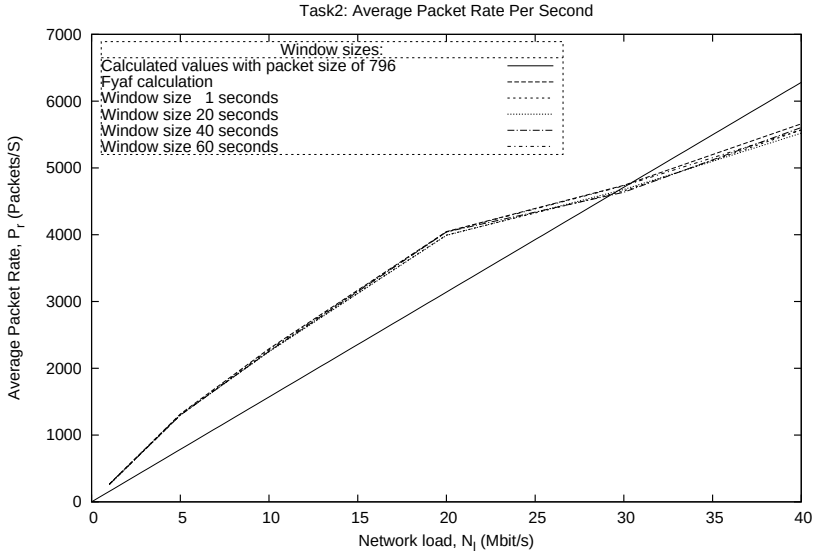

Figure 7.9: Task 2: Packets per second

of the packet sizes, as their packet headers were captured by *fyaf*, shows that the TCP layer has chosen a different packet size, for the segmented packets. With a stable packet size of 796 bytes, it would have been possible to calculate the packet rate with the following equation:

$$P_r \approx \frac{N_l * 1000000bits}{796bytes * 8bits} \tag{7.5}$$

We include this equation in the Figure 7.9, since it can be seen as an approximation of the accurate result. A better way of calculating the expected packet rate P_r is by looking at the *fyaf* result files, for each of the experiment runs. Since we know the total amount of packets P_{tot} that *fyaf* has sent to the client application, and as well the experiment duration T_d, we can pose the following equation:

$$P_r \approx \frac{P_{tot}}{T_d} \tag{7.6}$$

Based on the calculation of P_r, we show the accuracy A_c of the measured packet rate P_r for each of Task 2 versions in Table 7.6. The Table shows that the measured results for the versions are approximately 98% accurate or more, except from *Task 2 ws: 60* at the network loads 20 Mbit/s and 30 Mbit/s. This inaccuracy is probably caused by the dropped packets, discussed earlier.

N_l	Task 2 ws: 1	Task 2 ws: 20	Task 2 ws: 40	Task 2 ws: 60
1	98.4%	98.0%	97.7%	99.2%
5	98.9%	98.9%	98.8%	98.8%
10	98.5%	98.0%	98.4%	99.4%
20	98.7%	97.9%	99.2%	76.0%
30	98.2%	98.8%	98.1%	75.3%
40	98.4%	98.6%	98.9%	98.8%

Table 7.6: Task 2: Average accuracy A_c for measured packet rate P_r

CPU Utilization

Figure 7.10 presents the average maximum CPU utilization measurements, among the five experiment runs, for each of the task versions. We see that the plot from *Task 2 ws: 60*, has lower values than the other versions, at the network loads 20 Mbit/s and 30 Mbit/s. This is probably caused by the packet losses, already explained. By looking at the graph, we see no consistent trend that shows that increasing window sizes in aggregate operator boxes should decrease CPU. At 40 Mbit/s, we see that the CPU maximums for *Task 2 ws: 40* is 10% higher than for *Task 2 ws: 1 and ws: 20*. The plot also shows that *Task 2 ws: 20* and *Task 2 ws: 40* most often utilizes approximately 0.5% more CPU than *Task 2 ws: 1*. We regard 0.5% of CPU utilization to represent a so small percentage, that it does not necessarily mean anything in terms of how window sizes affect CPU utilization.

Memory Consumption

Figure 7.11 shows the average maximum memory consumption values for the experiment runs. We see that all versions, except *Task 2 ws: 60*, shows stable maximum values at 3.1%. *Task 2 ws: 60*, drops down to 3.08% at 20 Mbit/s and 30 Mbit/s. The dropped packets, discussed earlier, probably cause this. Other than that, we see that there is no variation for the maximum consumption values for the different window sizes, and conclude that window sizes up to 60 seconds do not affect memory consumption.

When increase the window sizes, we expected to see results of increased memory consumptions. Increasing window sizes should lead to more tuples being held in memory. As we see no increase, we believe memory consumption is mainly restricted by the system parameters set in the Borealis code, presented in Chapter 6.5.

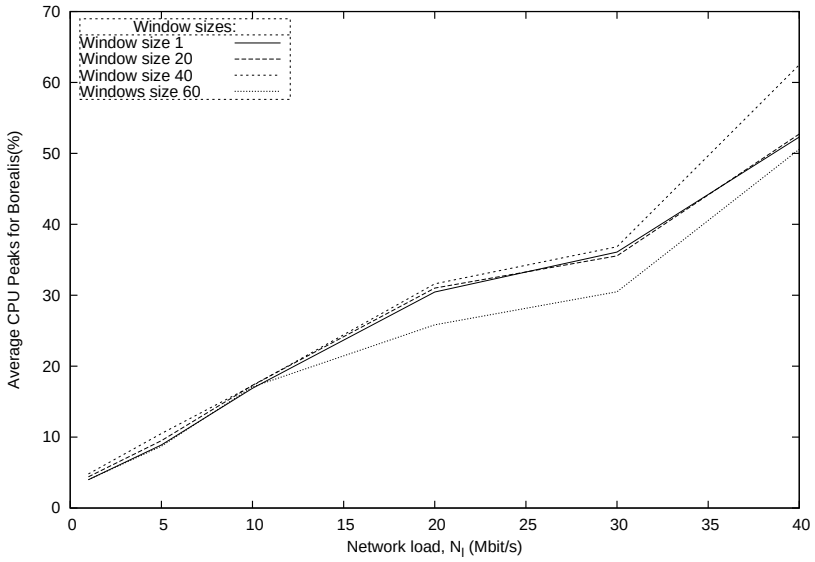

Figure 7.10: Task 2: Average maximum CPU utilization values for the Borealis process

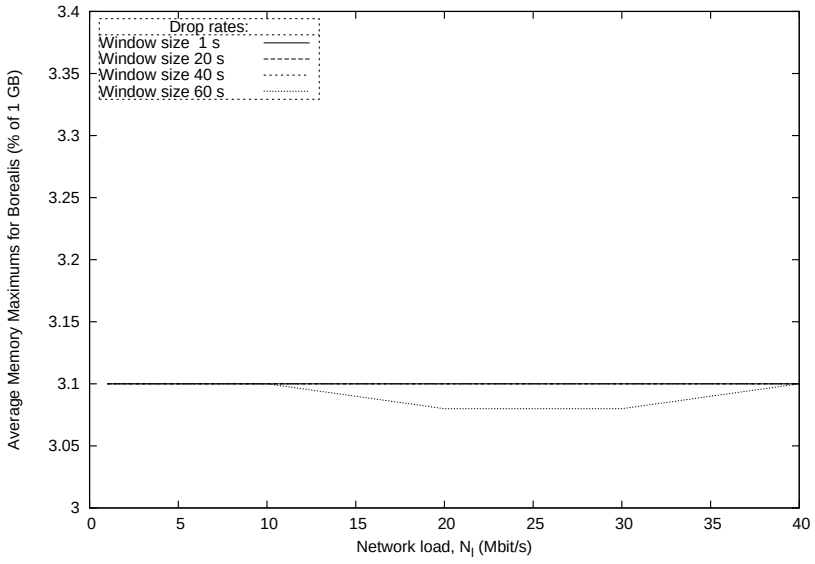

Figure 7.11: Task 2: Average maximum memory consumption for the Borealis process

7.2.3 Task 3: Destination Ports

In this Task 3, we are counting the occurrences of packets with certain destination ports, during a five-minute interval. In our experiment, we have chosen to count occurrences of packets with destination port 60010. This is the destination port used by the packets in the TG generated traffic. We have included two versions. *Task 3 v. 1*, which uses a *filter* operator box, and *Task 3 v. 2*, which uses internal static tables, in order to decide which port to count occurrences of. In order to test the static table function, we have included several thousands of ports from IANA [ian], in a stream that is stored within the static tables. We expect this to have a significant effect on the supported network load for this version.

Dropped packets reported by fyaf

We start by looking at the number of dropped packets in Figure 7.12. We see that *Task 3 v. 1* does not report any packet loss before reaching a network load of 40 Mbit/s, meaning it can handle 30 Mbit/s. At 40Mbit/s it suffers an average packet loss of 3.5%, which increases to 16.8% at 50 Mbit/s. *Task 3 v. 2* has an average packet loss of 0.9% at a network load of 5 Mbit/s. At 10 Mbit/s however, it reports a 0% packets loss. We regard the loss at 5 Mbit/s as so small that it can be caused by random inaccuracy in our experiment setup. At 20 Mbit/s, *Task 3 v. 2* reports an average packet loss of 7.2%.

As expected *Task 3 v. 1* seems to handle a significant higher network load than *Task 3 v. 2*.

Accuracy of the results

Figure 7.13 shows the average count of packets destined for the TG port 60010, each fifth minute. We have decided to not include any results from *Task 3 v. 2*, above 10 Mbit/s of network load, because of the high number of lost packets. We see a close relation between the curve of *Task 3 v. 1* (Figure 7.13), and the curves from Task 2 packet rates (Figure 7.9).

In order to be able to calculate the expected results, we have expressed an equation based on a very simple assumption. We assume that half of the packets exchanged between the two machines in the experiment network, are destined for the port 60010. With this assumption, we can express Equation 7.7 for the expected results for the count of port occurrences P_o. P_r represents sample values from the packet rates measured at the different network loads in Task 2.

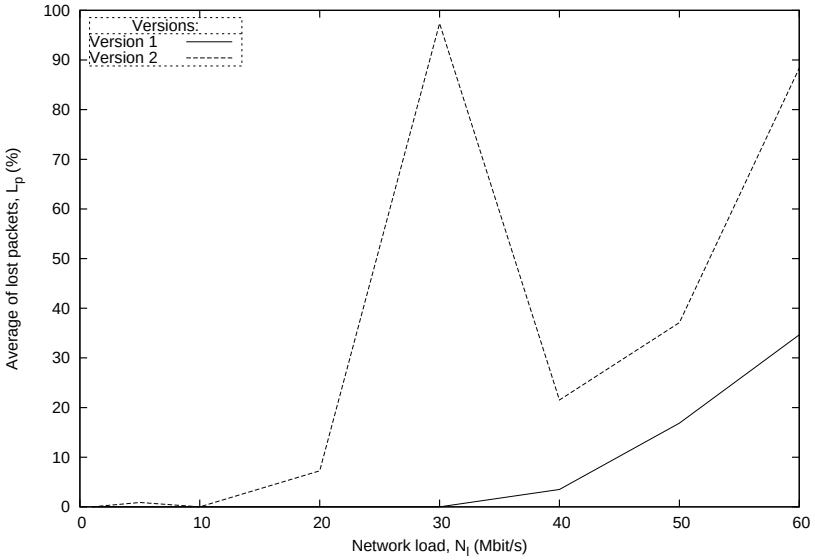

Figure 7.12: Task 3: Lost packets

N_l	P_o	Expected value based on P_r
1	45638	39300
5	225480	195150
10	399839	338550
20	761588	599250
30	947414	683400

Table 7.7: Task 3 v. 1: Measured values P_o, and estimates based on P_r from Task 2

$$P_o \approx \frac{P_r * 300 seconds}{2} \tag{7.7}$$

We include the calculated values in Figure 7.7 and Figure 7.13. Note that this calculation is fairly inaccurate and only included in order to compare the trends. The trends however, by looking at the graph in Figure 7.13, seems to be consistent with the expected calculated estimates. However we see that the gap between the two plots increases as the network loads get higher. We believe the correctness of our assumption that half of the packets are destined for 60010, decrease as network load increases.

As our calculated values are based upon an inaccurate assumption, we choose to not present any accuracy table for the task results. The average measured values from *Task 3 v. 1*, are shown together with the estimated

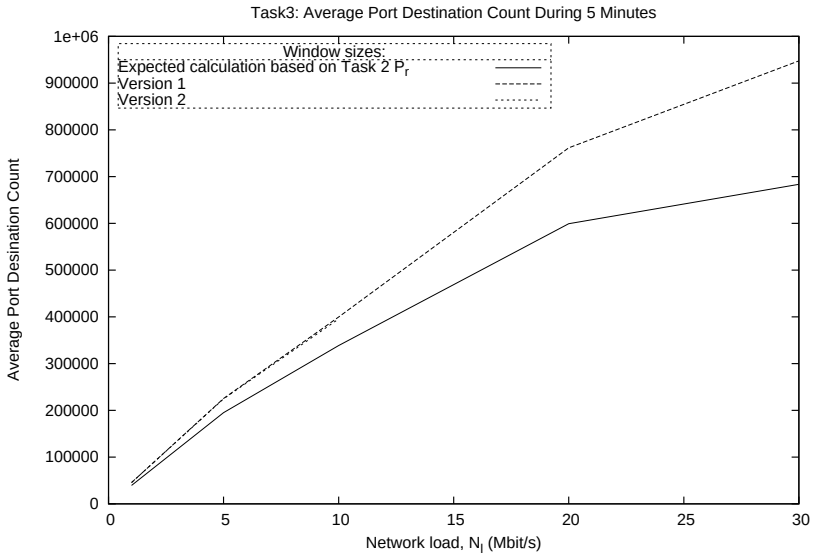

Figure 7.13: Task 3: Count of destination port occurrences

values in Table 7.7

CPU Utilization

Figure 7.14 shows the average maximum CPU utilization values for the Borealis process, for *Task 3 v. 1 and v. 2*. As expected, we see that *Task 3 v. 2* has higher CPU utilization maximums, than *Task 3 v. 1*, until *Task 3 v. 2* drops a significant amount of packets at a network load of 20 Mbit/s. *Task 3 v. 2* then reach a maximum CPU utilization of 37%, and decreases to 19% at 30 Mbit/s, as the system gets unstable because of overload. *Task 3 v. 1* has an average maximum value of 34.3%, before dropping packets at 40 Mbit/s.

The figure shows, as expected, that *Task 3 v. 2* utilize more CPU than *Task 3 v. 1* at the supported network loads. We expected this since *Task 3 v. 2* compare thousands of port numbers for each packet, in contrast to *Task 3 v. 1* were only on port number is included in the comparison.

Memory Consumption

Figure 7.15 shows the average maximum measured memory consumption values for *Task 3 v. 1 and v2*. For the first time in this evaluation, we see maximum values above 3.1%, without any reported packet loss. *Task 3 v. 2*, shows 3.3% consumption at the network loads 1 Mbit/s, 5 Mbit/s and 10

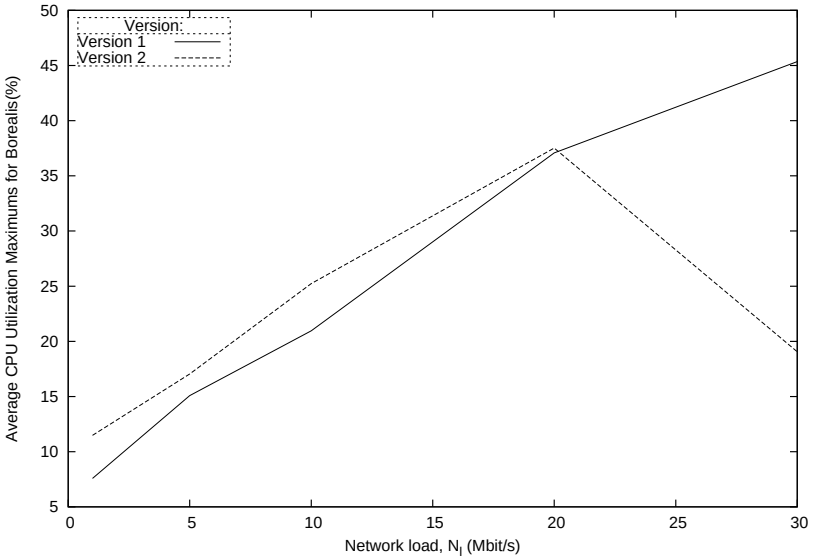

Figure 7.14: Task 3: Average maximum CPU consumption for the Borealis process

Mbit/s. At 20 Mbit/s the maximum values of memory consumption decrease for *Task 3 v. 2*. We consider this to be caused by the high percentages of dropped packets, as seen in Figure 7.12. *Task 3 v. 1*, however consumes a steady percentage of 3.1%.

We believe that the increase in memory consumption for *Task 3 v. 2*. is caused by the use of the internal static tables.

7.2.4 Task 4: Connection Sizes

Task 4 investigates the number of exchanged bytes for each connection, during 10 seconds. Initially, we designed two solutions. Their most significant difference is whether or not they join the results form both sides in the connection. *Task 4 v. 1* will return two result tuples for each connection per time interval, in contrast to *Task 4 v. 2*, where these result tuples are joined. Initial testing of *Task 4 v. 2*, showed correct results. During our experiment however, we have increased the number of generated TCP connections to 10. As we also increased the network load, result begun to show that we had a problem in the *join* operator. The measured results for *Task 4 v. 2* were significant lower than what we expected. Further investigations showed that several connections often were not identified. A new task version, *Task 4 v. 3*, was implemented, where we increased the window sizes in the *join* operator, hoping to identify all connections. However, this has shown to result in ever

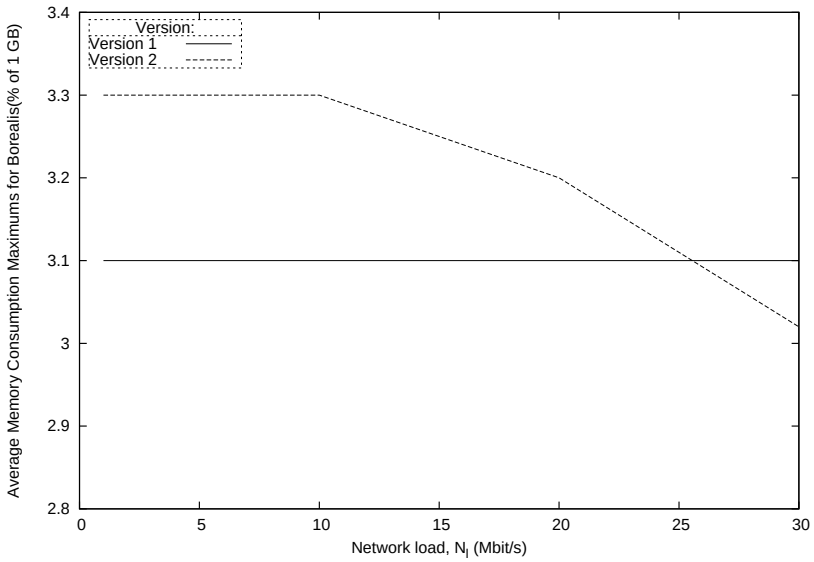

Figure 7.15: Task 3: Average maximum memory consumption for the Borealis process

fewer identified connections.

Dropped packets reported by fyaf

Figure 7.16 shows the lost packets L_p for *Task 4 v. 1, v. 2 and v. 3*. We see that all version show a significant amount of dropped packets at a network load of 30 Mbit/s. *Task 4 v. 1* shows 58% loss, *Task 4 v. 2 and v. 3* shows losses of 87% and 89%. In general, all versions have shown to handle up to 20 Mbit/s of network load, in terms of packet losses.

Accuracy of the results

Figure 7.17 shows the task results for the three versions. The results are again average values from five runs, at several network loads. We present the total sum of exchanged bytes, for the 10 connections, since each of them represent an even share off the total exchanged number of bytes. We have also included a plot that represents the expected values. These estimates are based on Equation 7.8. In Equation 7.8, E_b is the total number of exchanged bytes. P_{factor} is an approximation of the factor that we have found that the packet rate P_r increases with in relation to the network load N_l. H_s is the total size of the IP and TCP header. We have calculated an approximate values for H_s to 52 bytes, and 150 for P_{factor}. These approximations are

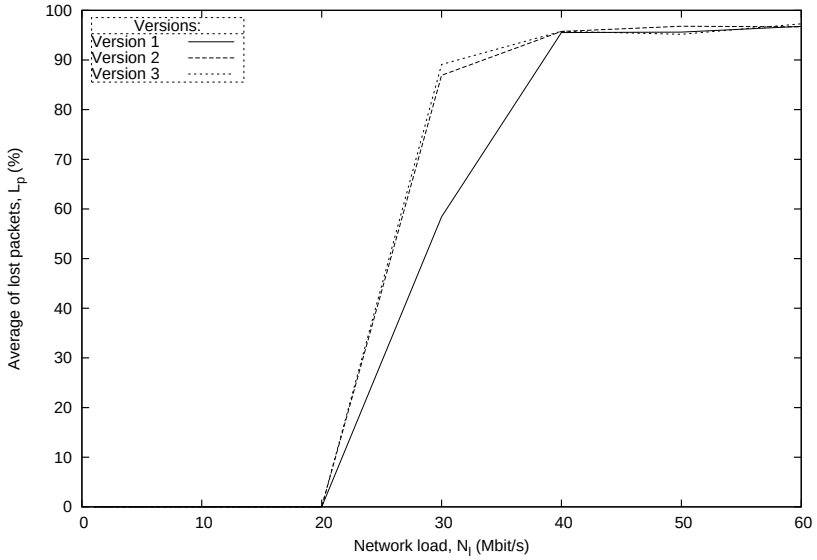

Figure 7.16: Task 4: Lost packets.

based on the Task 2 results.

$$E_b = \frac{N_l * 10 connections * 1000000 Mbytes}{8bit} - (P_{factor} * N_l * H_s) \qquad (7.8)$$

By looking at the Figure 7.17, we see that *Task 4 v. 2 and v. 3* shows increasingly lower results than *Task 4 v. 1*, as network load increases. We believe the reason for the lower result values, are the unidentified connections. *Task 4 v. 1*, however seems to follow the plot of the calculated values, although the plot show that the precision is decreasing, as network load increases. We have included an accuracy table for the task version, shown in Table 7.8, based on the estimates calculated with Equation 7.8. Again, we see that the precision of the estimates and the measured values are decreasing, as network load is increasing.

We present task results from the five experiment runs at 30 Mbit/s, in the tables 7.9, 7.10 and 7.11. The tables show result measured for each of the connections. Note that we in the graph in Figure 7.17 have shown the sum of all connections, in contrast to the results presented in the tables, showing the specific exchanged bytes on each of the connections. We see that several connections have not been identified for *Task 4 v. 2 and v. 3*, where no results are presented.

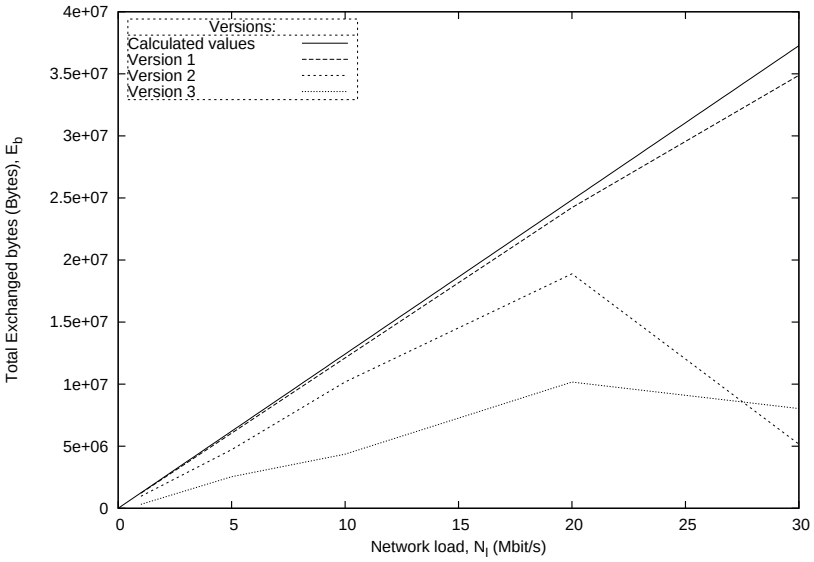

Figure 7.17: Task 4: Total exchanged bytes for the 10 connections

N_l	Task 4 v. 1	Task 4 v. 1	Task 4 v. 3
1	97.7%	77.9%	25.1%
5	97.3%	76.0%	41.0%
10	97.5%	81.9%	35.1%
20	97.5%	76.0%	41.0%
30	93.6%	31.1%	21.6%

Table 7.8: Task 4: Average of accuracy A_c for the measured results of total exchanged bytes

Run	Con. 1	Con. 2	Con. 3	con. 4	Con. 5	Con. 6	Con. 7	Con. 8	Con. 9	Con. 10
0	607348	603368	603368	604816	607348	603368	604816	606552	604816	603368
1	604960	604020	604164	604816	620880	604020	604960	604816	604816	604816
2	605756	605756	605756	605756	603368	604164	605756	603224	605756	605756
3	602572	602716	601920	604164	602716	604164	602716	602716	623268	604164
4	601632	601776	601776	601776	601776	601920	601776	601776	601776	601776

Table 7.9: Task 4, *Task 4 v. 1*: Sums of exchanged bytes on the 10 connections, at 5 Mbit/s

Run	Con. 1	Con. 2	Con. 3	con. 4	Con. 5	Con. 6	Con. 7	Con. 8	Con. 9	Con. 10
0	603368	604960	605900	604960	605756	604960	604960	605900	604816	604816
1	605104	612776	605104	606552	612776	604960	606552	605104	606552	-
2	-	-	-	-	-	-	-	-	-	-
3	604164	604816	604960	604960	604816	604960	604960	604960	604164	604960
4	603368	604816	603512	604960	604960	604816	603368	604960	604816	604816

Table 7.10: Task 4, *Task 4 v. 2*: Sums of exchanged bytes on the 10 connections, at 5 Mbit/s

Run	Con. 1	Con. 2	Con. 3	con. 4	Con. 5	Con. 6	Con. 7	Con. 8	Con. 9	Con. 10
0	606552	605104	606552	606552	606408	605104	605104	605104	605104	605104
1	604816	607348	607348	604164	607348	605900	604960	607348	604960	604960
2	-	-	-	-	-	-	-	-	-	-
3	606552	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-	-

Table 7.11: *Task 4 v. 3*: Sums of exchanged bytes on the 10 connections, at 5 Mbit/s

CPU Utilization

Figure 7.18 shows the CPU utilization maximums for the Borealis process. We present plots for the different versions, at the different network loads. As expected we see that *Task 4 v. 2 and v. 3* utilizes more CPU than *Task 4 v. 1*. This is probably caused by the fact that *Task 4 v. 2 and v. 3*, involves a higher number of operator boxes than *Task 4 v. 1*, and that *Task 4 v. 2 and v. 3* both include *joins*.

Memory Consumption

Figure 7.19 shows the memory consumption maximum values for the Task 4 versions. The plot shows that all the versions have equal maximum values at the different measured network loads. The shared maximum value is 3.1%, a value that has been measured for most tasks up until now.

7.2.5 Task 5: Intrusion Detection

Our intention for Task 5 has been to identify the possibility of expressing network monitoring tasks dealing with intrusion detection. We have focused on detection of SYN Flood Attacks. Our task design described in Chapter 6, has not as purpose to design a complete and trustworthy intrusion detection task. Doing so would perhaps deserve a thesis alone. Throughout this subsection, we present the CPU utilization and memory consumption for the two task versions. This in order to identify what kind of loads these kinds of queries support. Note, that we have not been able to identify the actual

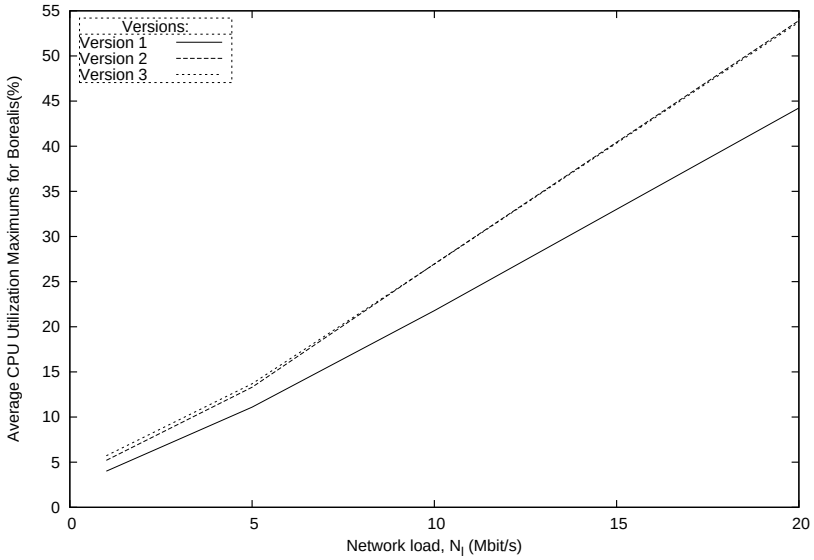

Figure 7.18: Task 4: Average CPU utilization peaks for the Borealis process

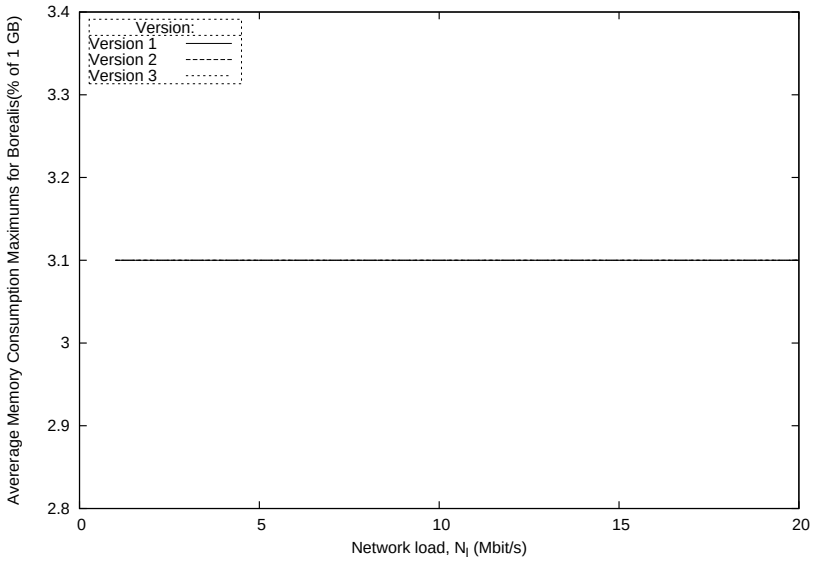

Figure 7.19: Task 4: Average memory consumption peaks for the Borealis process

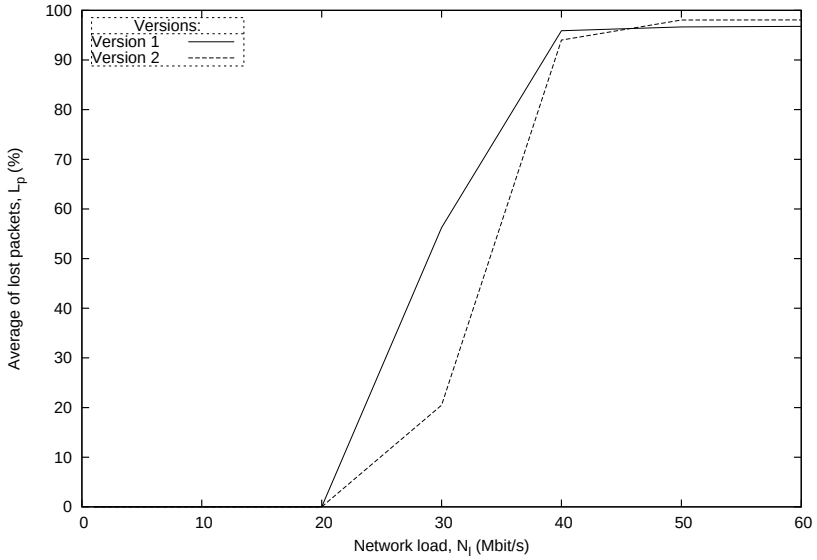

Figure 7.20: Task 5: Lost packets

load that is supported during a real malicious attack, since we do not have a packet trace form such an event.

Dropped packets reported by fyaf

Figure 7.20 shows the lost packets L_p, for Task 5. We see that both versions drop packets at a network load of 30 Mbit/s. *Task 5 v. 1* reports a P_l of 56%, and *Task 5 v. 2* reports 20%. At 40 Mbit/s, *Task 5 v. 1 and v. 2* both report packet losses above 90%. It is unexpected that *Task 5 v. 2*, reports fewer lost packets than *Task 5 v. 2* at 30 Mbit/s. We believe that the cause can be explained by the random behavior of the experiment system, as it gets overloaded. In general, both Task 5 versions have shown to handle a maximum network load of 20 Mbit/s.

Accuracy of the results

We have not been able to get hold off traffic sample data from SYN Flood attacks, but have tested solutions on faked traffic intended to simulate it. We do not claim that these solutions are fulfilled, but claim that creating such queries is possible. Because of this, we choose not to present any results, or their accuracy.

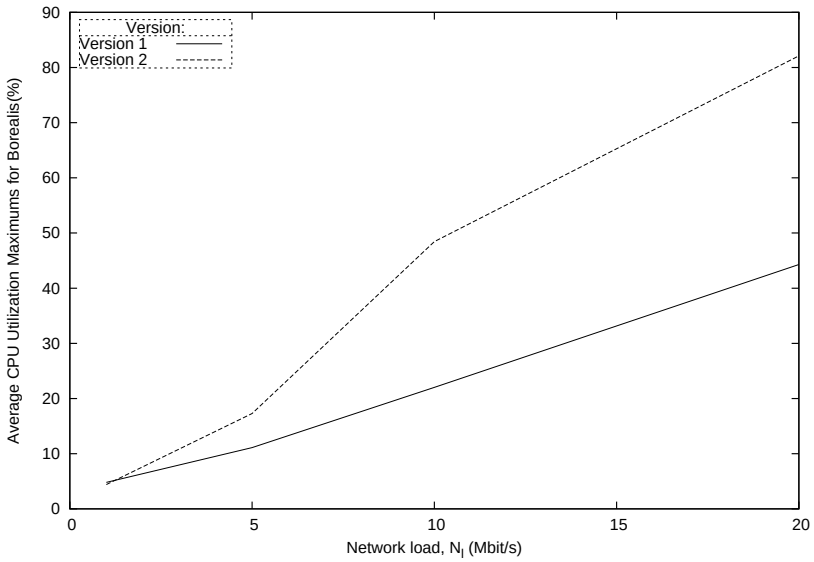

Figure 7.21: Task 5: Average CPU utilization peaks for the Borealis process

CPU Utilization

Figure 7.21 shows that *Task 5 v. 2* have higher CPU peaks than *Task 5 v. 1*. This is somewhat expected, because of the complexity of the two queries. The variation at 10 Mbit/s is measured to 26%, and increases even further at 20 Mbit/s.

Memory Consumption

Figure 7.22 shows stable maximum values of 3% memory consumption for *Task 5 v. 1* at all network loads. *Task 5 v. 2*, however, show significantly higher values. At 10 Mbit/s, we see an average maximum of 4.8%, and 5.3% at 20 Mbit/s. This is the highest measured maximum average for all tasks. *Task 5 v. 1* distinguishes it selves from the other tasks, by deploying a *WaitFor* operator box. The *WaitFor* box causes tuples to be buffered, until one of them matches a certain predicate. We suspect that the *WaitFor* box is what is causing the high memory consumption values for *Task 5 v. 2*. The increase of memory consumption from 3 to 5.3%, on our system is equivalent to 23 MByte.

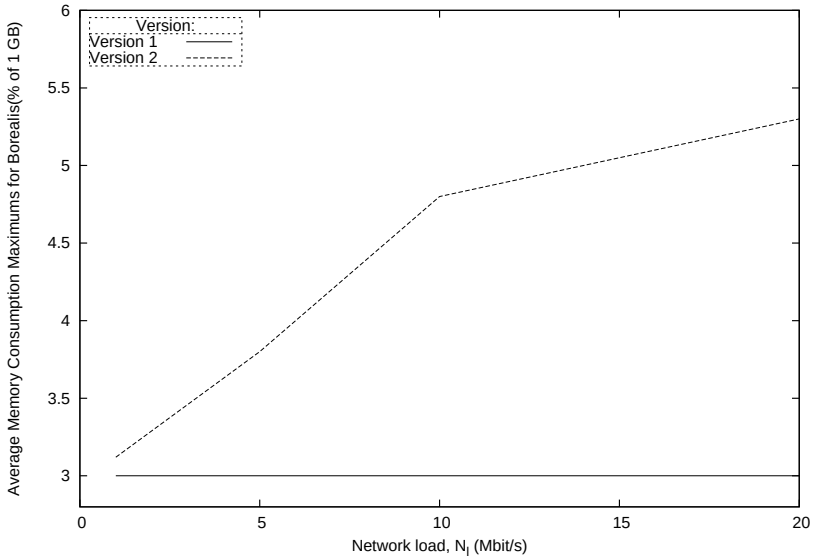

Figure 7.22: Task 5: Average memory consumption peaks for the Borealis process

7.3 Summary of the Performance Evaluation

We have shown that Borealis can handle ten-minute bursts of a generated network load of 40 Mbit/s, while performing simple network monitoring queries. By simple network monitoring queries, we mean queries similar to those of Task 2, and Task 3 v. 1. For more advanced queries, Borealis only seems to handle 20 Mbit/s. When incorporating a static table, and comparing the input tuples with thousands of statically stored tuples, 10 Mbit/s is supported. The generated traffic consists of packets sent between two hosts on a network, where the packets with payloads are approximately 796 bytes.

We have shown that the Borealis load shedding technique of deploying random_drop operator boxes does not increase performance much, in terms of what network load Borealis can handle. With a drop rate of 0.4, we only see a 30% increase. Higher or lower drop rates have not shown to increase the supported network load at all. By looking at the CPU utilization, we see a small decrease when deploying the *random_drop* boxes, but not for memory consumption.

By comparing query results with calculated expected values, we have shown that Borealis is accurate. We have for most task versions achieved accuracies A_c of 93% or more, and believe that the estimates often are what is causing the inaccuracy. The accuracy does not apply for our results when using the *join* operator, where we get inaccurate results measured to A_c of

80% or less. The cause is still unknown, but the window size set for the two input streams in the join operator might affect it. The cause of the inaccuracy should be pursued further in future work.

Based on the measurements of memory consumption during the experiments, we see that Borealis consumes significantly low percentages of the available memory. As memory challenges will always restrict a DSMS, as it performs aggregated queries over data streams in main memory, we would have expected that memory consumption would increase constantly at increasing data rates. Based on the fact that Borealis only consumes approximately 3% of the available main memory resources, a value that does not increase at higher network loads, we believe inner system parameters are what is really restricting the overall performance, in terms of what network load Borealis can handle. As we base our evaluation on black box testing, we have not pursued further investigations of this claim, although we have included a discussion on these parameters in Chapter 6.5.

Chapter 8

Conclusion

Throughout the stages of writing this Master thesis, we have designed, implemented and evaluated network monitoring tasks, for the Borealis stream processing engine. In this chapter, it is time to conclude our work. We start with presenting and summarizing our contributions in Section 8.1. Section 8.2 describes our critical assessment, things we would have done differently, if writing this thesis all over again. Last, in Section 8.3 we present future work.

8.1 Contributions

In this section, we summarize our contributions. We have chosen to divide the section in two. The first section summarizes the contributions from the design and implementation phases. In the second subsection, we summarize the contributions from the evaluation of the network monitoring tasks.

8.1.1 Design and Implementation

In Borealis, posing continuous queries on a stream of data is a somewhat complex task. It involves building a client application, expressing a stream definition, and designing the query itself. When this is said, examples provided with the release have given a solid foundation on how to do this.

We see great possibilities derived from the way of using custom created client applications, in order to connect streams to Borealis, and retrieve their query results. Building these applications however demands moderate C++ knowledge. But on the other hand, it enables users to build their own application front-end, and control both the input stream and the query results. We believe this feature affects the ability to apply network monitoring tasks

in a positive way, since so much functionality can be implemented in the client applications.

Borealis supports a number of field types that can be used to represent parameters within the streaming tuples. We managed to represent all TCP/IP header fields in the supported field types. The TCP/IP representation was not optimal in terms of size. A single bit field type, able of representing Boolean, or flag values, is something we definitely have missed.

The Borealis query language made up of conceptual boxes and stream arrows, has shown to give great possibilities when expressing continuous queries. It is a new way of thinking, in contrast to the SQL based languages offered by other DSMSs. At least for complex queries, we see several advantages with the conceptual boxes and arrows solution. By combining several operator boxes, one could split a complex query into smaller units. The concept of creating a graphical overview will in addition make it easier to understand what to express. Simple queries however are perhaps easier expressed in SQL like languages, as fewer lines of code are required. XML is used in the Borealis query language, although a Java GUI can create it for you. Posing a simple query at least includes two XML entities each consisting of several lines of code. Thus, several more lines of code are needed in a Borealis query expression, than would be needed in a simple SQL-like expression.

Although queries often may consist of many lines of code, we believe the choice of expressing the Borealis query language in XML is a good choice. XML is a well-known language that many users are familiar with. The XML parsers used to validate the queries before query execution have however shown to give imprecise or inconsistent error messages.

After understanding the language concepts, we have managed to express all the predefined tasks, almost as we wanted them. The only design we really had to abandon was a complete solution for Task 4. Not only did it concern joining packets from both sides in a connection, we also wanted it to identify connections by looking at their three-way handshakes. We believe the main reason for why we had to abandon this full solution for Task 4, is that the *join* operator does not support any feature allowing an output stream consisting of the tuples that are not joined. By not being able to identify these, they will eventually be lost, and not considered.

We would finally like to point out a feature of Borealis that we consider being especially good when designing complex queries. This feature is the functionality regarding the deployment diagrams. The deployment diagrams enable us to monitor the streams as they travel between the operator boxes, within the query. When designing complex queries, this feature has really shown to ease debugging and better understand how the flowing tuples within

the query processor look like.

8.1.2 Evaluation

The performance evaluation of Borealis shows results measured from Borealis while performing network monitoring tasks, at a real network with generated traffic. By controlling the generated traffic, we have identified both the accuracy of the expressed tasks, and found the network load Borealis can handle, while performing them.

We have shown that Borealis, in our setup, can handle ten-minute bursts of 10 to 40 Mbit/s, depending on the complexity of the tasks. Exceeding these network loads will result in dropped packets. The supported network loads are definitely higher than expected. STREAM, in similar experiments performed by Hernes [Her06], has shown to handle up to 30 Mbit/s performing simple queries, but only 3 Mbit/s for complex queries. In experiments performed by Søberg [Søb06], TelegraphCQ only showed to handle 2.5 Mbit/s.

Although Borealis can handle a network load of 40 Mbit/s, overload situations occurring at higher network loads are not something Borealis has shown to handle gracefully. We have not managed to compile or verify any fully utilized load shedding mechanism. The load shedding technique we have managed to measure, have shown to result in a 30% increase for supported network load, although this would cause 40% of the packets to not be considered.

As for accuracy, we have shown that Borealis is able to present task query results that are 99% accurate at 40 Mbit/s. For all our measured tasks, we have most often achieved accuracies of 93% or higher. We often believe our calculated estimates are what is causing the inaccuracy, rather than the measured result itself. We have however had problems with the *join* operator. The problems have caused inaccurate results for *Task 4 v. 2 and 3*, as several connections were not identified. We do not know the cause of the unidentified connections, but regard further investigation of this as future work.

System performance measurements performed on the machine Borealis was running on, showed that the system resources consumed by Borealis were far from fully utilized. Even though Borealis was just about to loose packets caused by a filled up buffer, we seldom saw memory consumption greater than 3.1%, and we seldom saw CPU utilization above 80%. We believe the cause of these resources not being fully utilized are parameters set in the source code. We describe these parameters further in Chapter 6.5, but regard further investigation as future work.

8.2 Critical Assessment

In case we were to redo the whole process of writing this thesis, there are a couple of things we would have done differently. In this section, we will briefly discuss these things.

The last 17 weeks have been intense. The restriction of available time has always been over us. We spent at least a week, trying to verify a load shedding mechanism in Borealis. This included altering several source code files, in order to get rid of compilation errors. We have also spent several days trying to install a new Borealis beta version that we due to our system setup had to abandon. We believe the time consumed on trying to verify the load shedder, and testing out the beta version, better could be consumed on other things. This includes further investigation and understanding of the generated network traffic, and the implementation of TCP in our experiment setup. Gaining a better understanding of the traffic would have enabled us to calculate more accurate estimated results, for the network tasks.

Initially, we started out designing Task 5, in order to design a network monitoring task for intrusion detection. We have managed to design such a task, even in two versions. The task versions are based on our understandings of how a SYN flood attack looks like. As we lack packet traces from any legitimate SYN Flood attack, we have not been able to verify the accuracy of this task. In addition, have we only shown that *Task 5 v. 1 and 2* can handle network loads up to 20 Mbit/s, when traffic is normal. We have not measured the network load that can be dealt with during a SYN flood attack. The time we spent running the experiments and evaluating the data of *Task 5* could be used elsewhere, among the subjects of this thesis.

8.3 Future Work

This short Master thesis has been carried out during 17 weeks. With the restriction of time in mind, we have not been able to pursue several ideas of improvements, or challenges meet during this somewhat short period of time:

- Borealis support advanced functionality regarding *distribution of queries*. By distributing queries to several machines we believe that a higher supported network load would have been possible to achieve. This by balancing the load for the query processor, on several machines.
- We believe that by incorporating the functionality of *fyaf* into the client application, the system would benefit from not needing to obtain a TCP

socket between *fyaf* and the client application. This should increase the overall load on the system. In addition, by further developing more functionality into the client application, load shedding could directly be implemented, possibly leading to a higher supported network load.

- During evaluation of the results in Chapter 7, we encountered a problem we refer to as the *join* problem, in *Task 4 v. 2*. Several connections were not identified in the *join* operator. We do not know whether this is caused by an error made by us, or an error in Borealis.

- Early in the design phase, the parameters *SLEEP_TIME* and *BATCH_SIZE*, were identified as parameters significantly affecting the network load Borealis can handle. The parameters are set within the client applications. We decided to choose a *SLEEP_TIME* of 1 ms, in order to make Borealis sleep as little as possible. Appendix D, shows how the *BATCH_SIZE* was chosen. There is of course a possibility that by increasing the *SLEEP_TIME* parameter value, we could achieve higher supported network load. We have only performed testing with *SLEEP_TIME* set to 1 ms.

- In Chapter 6.5, we present a set of system parameters, we believe affecting the overall performance of Borealis significantly, as mentioned. These parameters include:

  - TUPLE_PROCESSING_LIMIT
  - MAX_UTILIZATION,
  - MAX_BW
  - MAX_SOCKET_BUFFER

  We believe that by changing and optimizing these values, Borealis should be able to handle higher network loads, than as of today.

- As of spring 2007, a new Borealis version has been released. We have partly started compilation of it, but did not pursue this, because we were afraid of consuming to much time on it. We have been told that the spring version is supposed to have improved functionality regarding load shedding.

- Finally, we would like to further investigate and compare the results obtained from both TelegraphCQ [Søb06] and STREAM [Her06] with our results from Borealis.

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinskaur, Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, 2005.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Report, Department of Computer Science, Stanford University, 2004.

[ABC+05] Yanif Ahmad, Bradley Berg, Ugur Cetintemeland, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran Anurag Maskey, Olga Papaemmanouil, Alex Rasin Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the borealis stream processing engine. *ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, 2005.

[ACC+03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. *SIGMOD 2003, June 9-12, 2003, San Diego, CA. Copyright 2003 ACM 1-58113-634-X/03/06*, 2003.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2002.

[BBS04] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load managament and high availability in the medusa distributed stream processing system. *SIGMOD 2004 June 13-18. 2004 Paris France*, 2004.

[bor] The borealis project web page. `http://www.cs.brown.edu/research/borealis/public/`.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong*, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman**, Fred Reiss, , and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. *Proceedings of the 2003 CIDR Conference*, 2003.

[cis] Characterizing and tracing packet floods using cisco routers.

[Der03] Luca Deri. Passively monitoring networks at gigabit speeds using commodity hardware and open passively monitoring networks at gigabit speeds using commodity hardware and open source software. *NETikos S.p.A.*, 2003.

[EN07] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Greg Tobin, 2007.

[end] Endace web page. `http://www.endace.com/`.

[GO03] Lukasz Golab and M. Tamer Ozsu. Issues in data stream management. *SIGMOD Rec., Volume 32, No. 2*, 2003.

[GR] Matthias Grossglauser and Jennifer Rexford. Passive traffic measurement for ip operations.

[Her06] Kjetil Hernes. Design, implementation, and evaluation of network monitoring tasks with the stream data stream management system. Master's thesis, University of Oslo, Department of Informatics, 2006.

[ian] Iana web page. `www.iana.org`.

[isc] Isc internet domain survey. `http://www.isc.org/ds`.

[Jai91] Raj Jain. *The Art Of Computer Systems Perfomrance Analysis; Techniques for Experimental Design, Measurement, Simulation nd Modeling.* John Wiley & Sons, Inc, 1991.

[JCS03] Theodore Johnson, Charles D. Cranor, and Oliver Spatscheck. Gigascope: A stream database for network application. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, June 9-12*, 2003.

[JMSS05] Theodore Johnson, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Streams, security and scalability. *19th IFIP WG11.3 Working Conference on Data and Application Security*, 2005.

[KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. *SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.*, 2006.

[meg] Wikipedia articla on megabit. `http://en.wikipedia.org/wiki/Megabit`.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems, Vol. 30, No. 1, March 2005.*, 2005.

[MS05] E. W. Biersack T. EnNajjary M. Siekkinen G. UrvoyKeller. Root cause analysis for longlived tcp connections. *CoNEXT'05*, 2005.

[Nag84] John Nagle. Congestion control in ip/tcp internetworks, January 1984. `http://www.ietf.org/rfc/rfc896.txt`.

[PGB+04] Thomas Plagemann, Vera Goebel, Andrea Bergamini, Giacomo Tolu, Guillaume Urvoy-Keller, and Ernst W. Biersack1. Using data stream management systems for traffic analysis – a case study –. *Lecture notes in computer science (Lect. notes comput. sci.) ISSN 0302-9743*, 2004.

[rfc81a] Internet protocol (rfc 0791), 9 1981. `http://www.ietf.org/rfc/rfc0791.txt`.

[rfc81b] Transmission control protocol (rfc 0793), 9 1981. `http://www.ietf.org/rfc/rfc0793.txt`.

[SÇZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record 34, 4 (Dec 2005)*, 2005.

[Sie06] Matti Siekkinen. *Root Cause of TCP Throughput: Methodology, Techniques, and Applications*. Ph.d. thesis, Universite De Nice-Sophia Antipolis - UFR Sciences, 2006.

[Søb06] Jarle Søberg. Design, implementation, and evaluation of network monitoring tasks with the telegraphcq data stream management system. Master's thesis, University of Oslo, Department of Informatics, 2006.

[str07] Stream web page, 15.04.2007. `http://infolab.stanford.edu/stream/`.

[SUKB06] Matti Siekkinen, Guillaume Urvoy-Keller, and Ernst W. Biersack. On the interaction between internet applications and tcp. Report, Institut Eurecom, 2006.

[tcp] Tcpdump web page. `http://www.tcpdump.org/`.

[Tea06] Borealis Team. *Borealis Developer's Guide*, May 2006.

[tel] Telegraphcq web page. `http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/`.

[tg] Tg (traffic generator) web page. `http://www.postel.org/tg/`.

[tra] Traderbot web page. `http://www.traderbot.com`.

Appendix A

TCP/IP

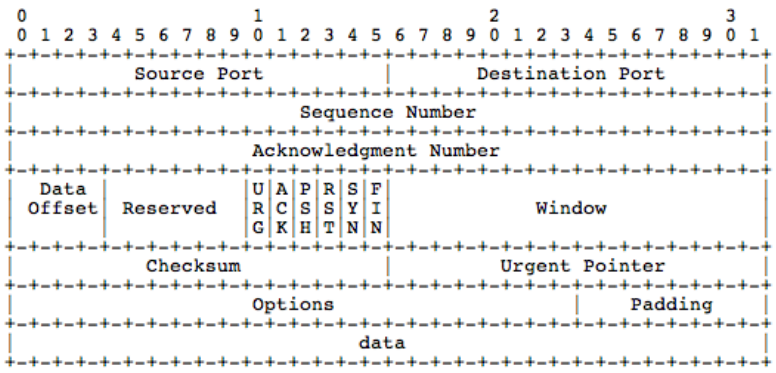

Figure A.1: RFC793 - Transmission Control Protocol

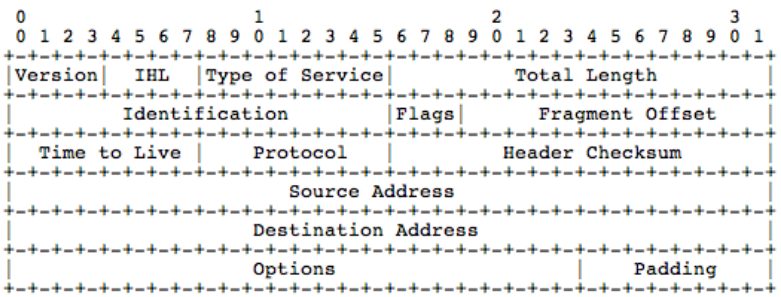

Figure A.2: RFC791 - Internet Protocol

Appendix B

Building and Running a Borealis Continuous Query

In this chapter we explain how to build and run a Borealis continuous query. We focus a little more on the technical side, than we have done in the previous chapters. We start by describe the concepts of the actual query implementations. Secondly we explain how to start the query processor. Last, we discuss how to use the *Marshal Tool* in order to build a client application.

B.1 Starting a Borealis Node

The Borealis node is the actual instance of the query processor. Borealis can simply be started from a *Unix terminal*, provided that the borealis distribution is compiled and working. Borealis will then listen on the following ports [1]:

- 15000 will accept TCP/RPC connections
- 15001 will accept XML/RPC connections for dynamic query modification.
- 15002 will accept data connections

The Borealis node instance will automatically respond to events from the client application. By these events, we mean the event of loading the query into the query processor, and to connect the query processor to the input and output stream(s).

[1] Note that these ports are set as a default setting, other ports can be chosen as default at compile time.

B.2 Implementation of Continuous Queries

Chapter 5 should give a good overview of how to implement the queries in XML. An important part of the query is the stream definitions. These stream definitions, and the defined input and output streams, are what is used to generate code for the client applications, described in next section. Note, that the XML has to follow a document type description (DTD), provided with the Borealis distribution.

Deployment of the query diagrams is normally performed by the client application. *Big Giant Head*, as a standalone module, can also upload queries to the query processor, although we have not tested this.

B.3 Client Application

The Borealis distribution provides a tool for simplifying the programming of the Borealis client applications. The name of the tool is *Marshal Tool*. It is used to generate C++ code consisting of functions and header files. It is executed with the command 'marshal [queryname.xml]', and will then generate code based on the query in the file [queryname.xml]. The generated code will be places in the files 'ApplicationnameMarshal.cc and .h'. The files then include generated functions that can be used to open a connection, and send /receive batches of tuples.

Skeletons of source code found in the test directory of the Borealis distribution, helps implementing a executable program that uses the generated code in order to connect the streams to the query processor. When running a successfully implementation of such a program, the query will be uploaded to the Borealis node, and the data stream will begin to flow into the query processor.

Note, that in order to compile the needed code for running a Borealis client application, the tools *automake* and *autoconf* are used. The provided test applications includes Makefiles and filepaths for the needed libraries, in order to properly compile a client application.

Appendix C

Experiment Scripts

The execution scripts help us perform the experiments several times with out starting each of them manually. In this appendix, we briefly present a list of the scripts. Developed as part of [Søb06] and [Her06], they are used to execute the following stages of an experiment:

- Execute the tools for generating traffic
- Execute the tools for logging consumption and utilization of system resources
- Start and stop the DSMS
- Execute the queries, and store the output results to files

Since the experiments are running at two machines, two scripts are needed in order to start a series of experiments. The machine *dmms-lab107* acts like the experiment server, and *dmms-lab77* acts like the experiment client. This machine is also the one to execute the queries. We start with describing the scripts on the machine *dmms-lab77*:

- sscripts2.pl is called from the terminal. It is used to define the tasks to run, how many times, and at what network workload. It will further invoke other helper scripts.
- supers_script3.pl is called from sscript2.pl. It starts and stops the DSMS, and the single task runs.
- experiment_client.pl is called by super_script3.pl, and starts *fyaf* and the system monitors. In addition, it communicates with experiment_server.pl on the other machine, and transmits variables for defining the network traffic behavior.

- create_servers.pl is called by the experiment_client.pl. It defines the generated network traffic duration time, for a single experiment.
- tg_clients_run.pl is called by experiment_client.pl, and starts the TG instance.

The machine *dmms-lab107* contain the following scripts:

- experiment_server.pl is called from the terminal. It takes only one key argument. This argument defines the count of single experiment phases, included in the series of experiments about to be run.
- change_template.pl is called from experiment_server.pl, and helps setting the traffic behavior on behalf of arguments retrieved from the connection to *dmms-lab77*.
- create_client.pl is called from change_template.pl, and creates the TG files that controls the behavior of the network traffic.
- tg_clients_run.pl is called by experiment_server.pl, in order to start transmitting the actual traffic data.

Appendix D

BATCH_SIZE

In this appendix, we explain how we selected the value for the BATCH_SIZE variable, set in the client applications. The value was chosen after running a series of experiments, where we tried to identify the BATCH_SIZE leading to a highest possible supported network load. In the experiment, we ran *Task 2 v. 1*, at high network loads, with varying BATCH_SIZEs. We include a set of figures, showing plots of the total number of packets lost, at different BATCH_SIZE values.

Figure D.1 shows how different values of BATCH_SIZEs affect the number of lost packets. The experiment was run on BATCH_SIZE values ranging from 1 to 60. The figure shows no results with values of 50 or 60, since the experiments did not finish, possible due to Borealis crashing. Hence we see them as inappropriate values. Out from Figure D.1, it seems BATCH_SIZEs around 40 leads to the least amount of lost packets

Figure D.2 shows the measured average network load per second performed with *Task 2* described in Chapter 5.2.2, at varying BATCH_SIZE. Out from the tasks, *Task 2* was chosen because it reports the measured network load N_l, which is a easily comparable result value. We see that low values of BATCH_SIZE leads to inaccurate results. Again we see a trend that BATCH_SIZEs around 40 gives satisfying results at workloads up to nearly 50 Mbit/s of generated bandwidth.

Figure D.3 shows reported lost packets from 20 test runs at 45 Mbit/s. Each run had duration of 10 minutes. At 45 Mbit/s we know that Borealis will loose tuples due to overload with a high probability. By running several experiments, we hope to see which BATCH_SIZE that leads to the fewest lost tuples. Within the experiment we have tested sizes of 30, 35, 40 and 45. We choose not to test sizes of 50 and above, as the experiments behind Figure D.1 showed indications of those values resulting in system crashes. By looking at Figure D.3, we see that the BATCH_SIZE of 45, resulted in

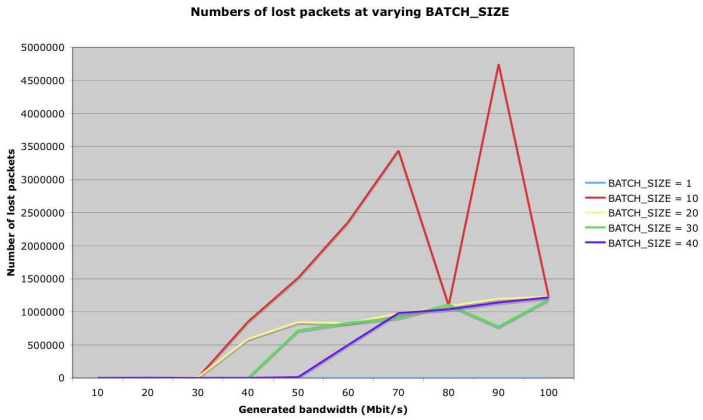

Figure D.1: Number of lost packets with varying BATCH_SIZE and network load

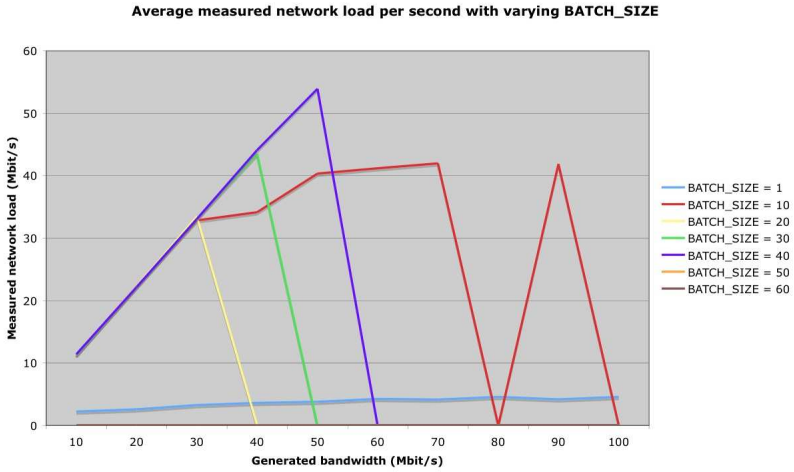

Figure D.2: Measured network load with varying BATCH_SIZE and network load

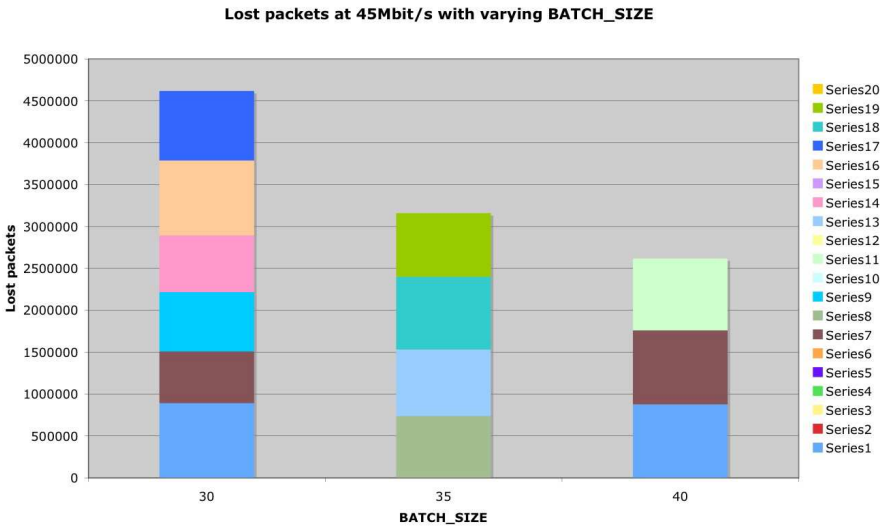

Figure D.3: Lost packets at 45Mbit/s bandwidth at varying BATCH_SIZE

the fewest lost packets. Hence this is the value we have chosen throughout our task experiments. The experiments upon where we conclude this are perhaps not as documented and deeply tested as one would wish. This since we see them as somewhat not part of the actual evaluation of Borealis. We consider a deeper evaluation of BATCH_SIZE and SLEEP_TIME parameters as future work.

Appendix E

DVD-ROM

The tasks, experiment results, and the scripts used to calculate result values from them, are included in the DVD-ROM. At the root of the DVD, we have included three directories:

- *experiments* - includes a directory for each of the tasks versions, including the experiment results and system resource consumption logs.
- *scripts* - includes a number of scripts used to simplify the presentation of the experiment scripts. We have not included any documentation for these scripts.
- *tasks* - includes the task versions, both C++ code, and the XML queries.

Note, that the content of the DVD is compressed.