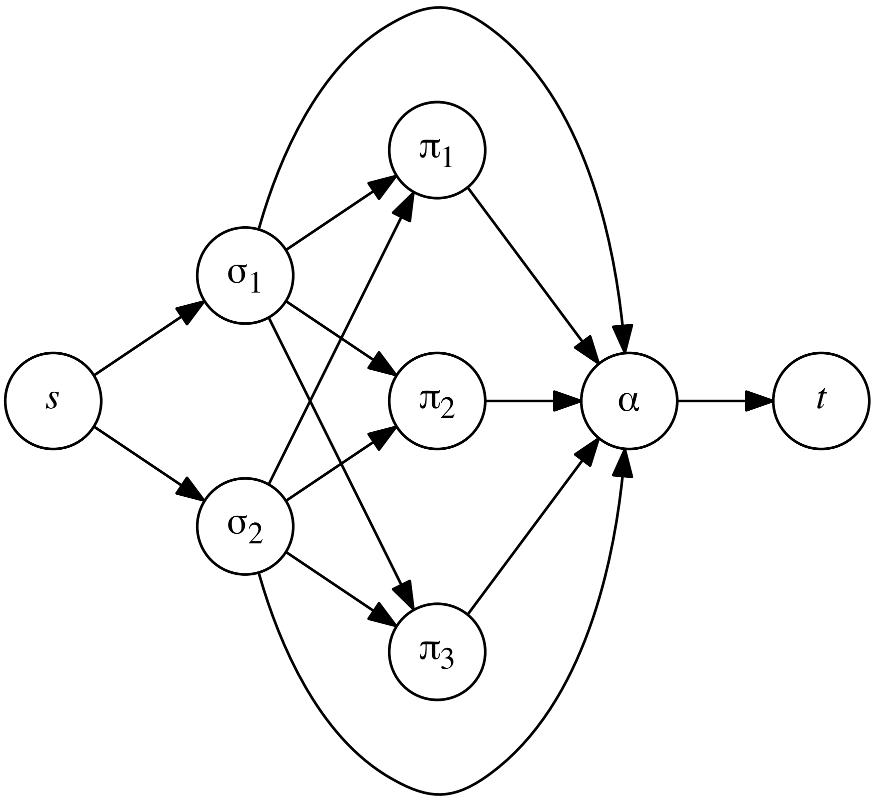

Latency and Throughput in Center versus Edge Stream Processing

A Case Study in the Transportation Domain

Master's thesis in Computer Science: Algorithms, Languages and Logic

Gregor Ulm

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

Latency and Throughput in Center versus Edge Stream Processing

A Case Study in the Transportation Domain

GREGOR ULM

Department of Computer Science and Engineering
Networks and Systems Division
Distributed Computing and Systems Research Group
Chalmers University of Technology
Gothenburg, Sweden 2016

Latency and Throughput in
Center versus Edge Stream Processing
A Case Study in the Transportation Domain
GREGOR ULM

Supervisor: Vincenzo Massimiliano Gulisano, Computer Science and Engineering
Examiner: Philippas Tsigas, Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Networks and Systems Division
Distributed Computing and Systems Research Group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The cover image is a graphical representation of the concrete topology that was executed in order to measure the performance of our implementation of a program that solves the accident notification problem. Operators are represented as nodes. In particular, this graph highlights the parallelization of the operators σ and π, which have two and three instances, respectively.

Typeset in LaTeX
Gothenburg, Sweden 2016

Latency and Throughput in
Center versus Edge Stream Processing
A Case Study in the Transportation Domain
GREGOR ULM
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The emerging Internet of Things (IoT) enables novel solutions. In this thesis report, we turn our attention to the problem of providing targeted accident notifications in near real-time. We use traffic data generated by Linear Road, a popular benchmark for stream processing engines, and simulate a possible real-world scenario in which connected cars continuously send position updates. We analyze this stream of position updates with the goal of identifying accidents, so that targeted accident notifications can be issued. This means that only cars within a certain distance of a known accident site will be notified.
In a real-world scenario, the required data analysis could be performed in different ways. We consider two possibilities. First, position reports are aggregated by road side units (RSUs) and forwarded to a central server. Afterwards, the results are sent back to the cars, again involving RSUs for transmission. We refer to this as *center stream processing.* Second, all data analysis is performed on RSUs. An RSU is less powerful than a server. However, RSUs are located much closer to the cars than a central server. We refer to this case as *edge stream processing.* Performing computations directly on RSUs has the benefit that the cost of the roundtrip time for data transmission from RSUs to the server and back will be avoided. We use a contemporary stream processing engine for data analysis, and compare latency and throughput of an implementation of our solution to the accident notification problem in both cases.

Keywords: big data, data analysis, data mining, distributed systems, edge computing, fog computing, Internet of Things, parallel-distributed systems, stream processing, real-time analytics.

Acknowledgements

First and foremost I would like to thank Philippas Tsigas, Marina Papatriantafilou, and Vincenzo Gulisano of the Distributed Systems and Computing research group for their encouragement and support. They organized the supervision of Master's thesis projects in a highly efficient manner. In particular, I enjoyed the collaborative environment they fostered among their students. As the supervisor of this thesis project, Vincenzo provided helpful and timely feedback as well as guidance when needed. I also benefited from feedback by Philippas and Marina.
Drafts of this thesis report were read by Oskar Abrahamsson, Martin Helmersson, Barnabas Sapan, Lucas Wiman, and Niklas Logren.

Gregor Ulm, Gothenburg, August 2016

Contents

List of Figures

List of Tables

1 Introduction

This thesis project is concerned with latency and throughput in stream processing. Concretely, the focus is on the problem of providing near real-time accident notifications. In this introductory chapter we present the motivation behind our thesis project, highlighting its theoretical and practical relevance. This is followed by a problem description, which is presented in general terms. After providing an outline of our solution to the accident notification problem and a brief description of the evaluation, we conclude with an overview of the subsequent chapters of this thesis report.

1.1 Motivation

Big Data and the Emergence of the Internet of Things

Big Data is a hot topic both in computing research and industry. Indeed, the amount of data generated and processed has become substantial. Businesses and governments alike are increasingly pressured to make better use of the plethora of data at their disposal [5]. In a recent McKinsey study [20], for instance, it was estimated that efficient use of big data could lead to cost savings of $200 billion a year in the US healthcare sector, or €100 billion a year "in operational efficiency improvements" in European bureaucracies. As questionable as the predilection of management consultants for big round numbers may be, there is little doubt that there is value in data. That value may manifest itself after asking the right questions — Nate Silver's [27] highly accurate predictions of the outcome of the 2008 US presidential elections come to mind — or in processing uniform data quickly, with the goal of getting results sooner. The latter is particularly related to the current emergence of the Internet of Things (IoT), which consists of devices that are connected to the Internet. According to a 2015 OECD study covering the top 25 industrialized countries, there are around 300 million IoT devices. South Korea currently leads this development with close to 40 IoT devices per 100 inhabitants [22]. Yet, this is only the beginning.

Heterogeneous computing is common in the Internet of Things, considering that devices of vastly different computational power are in use. An IoT device possesses only a small fraction of the computational power of a dedicated server. Furthermore, parallel and distributed computing is becoming ever more important, for a variety of reasons. This relates to the *power wall* [18]. Even though we are able to build larger CPUs with greater numbers of transistors, there are physical limitations with regards to clock frequency, which makes it very difficult to efficiently cool chips that are running at high clock speeds. In addition, we are currently witnessing the end of Moore's law, which originally, in 1965, postulated that chip performance would double every year. Moore revised this prediction in 1975 in order to state that the number of transistors on a chip would double every two years [25]. In 2015, though, Intel stated that progress has been slowing to a doubling every two and a half years [6]. Thus, current engineering challenges are the effective utilization of multi-core CPUs and heterogeneous computing architectures. This is a problem in the small, for instance in ARM's *big.LITTLE* architecture [14], which pairs tiny, slow and low-powered CPU cores with larger, faster, and much less energy-efficient ones, utilizing whichever is more appropriate for the task at hand. It is also a problem in the large, for instance in distributed systems at web scale.

Benefits of the Internet of Things

The proliferation of IoT data sources has the potential to increase quality of life. There were early proposals of an arguably frivolous nature, though. For instance, recall the concept of the *intelligent fridge*, which is able to figure out that your milk has likely gone sour. While consumers considered innovation of such nature merely "moderately useful" [23], there have been other advances. One example is advanced metering via the *smart grid* [33], which promises to provide deeper insights in, for instance, electricity consumption patterns of households. Arguably, there are IoT applications that would increase human welfare to an even greater extent, such as modifying driver behavior in an attempt to benefit the environment, limit traffic congestion, and reduce the risk of accidents. In the last two decades, several jurisdictions [11, 9, 34, 24] have experimented with variable tolling, also referred to as congestion pricing, as a means of modifying the behavior of vehicle drivers, based on economic principles. Through variable pricing, drivers face positive or negative incentives, e. g. in order to resolve traffic congestions at certain hours, the transport authority could deliberately increase tolls for expressways.

Challenges

Devices that are connected to the Internet are capable of producing large amounts of data *ad infinitum*. In order to process these data efficiently, we need suitable approaches. Google's MapReduce programming paradigm [8] has proven highly suitable for distributed processing of batch data. Yet, in a connected world where hundreds of millions, and soon billions, of devices continuously produce data, MapReduce is not applicable, as it is restricted to processing finite batches of data. Instead, we need to be able to handle data that may not be transmitted as fixed-sized batches but instead in a continuous manner as *streams*, which are potentially unbounded sequences of data. This entails that data may come in variable loads, with peaks and droughts. Consequently, there is a need for being able to process such data efficiently, so that we are able to get continuous results, achieve high throughput, and low latency. Stream processing engines were designed to solve this problem.
The challenges surrounding big data in an IoT setting go beyond merely using the

right tools. It is also important to use one's tools well. This means that throughput and latency are affected by the chosen deployment strategy. Considering that data is generated by many edge devices, and potentially processed by a central server, there are several possibilities. One approach consists of sending all data to a central server, processing the data, and sending the results back to where they are needed. Alternatively, one could try to utilize all available resources and perform computations on edge devices as well as a central server. Lastly, one could perform all data analysis on edge devices, omitting a server altogether. This concept is commonly referred to as edge computing, or fog computing.

Parallel-distributed stream processing is particularly relevant in an IoT setting, where hardware of vastly different capabilities may be in use. There could be one single server farm with multiple many-core CPUs, housed at a remote location, in addition to several smaller multi-core servers in several larger cities, and millions of devices, spread all over the world, with a less powerful CPU. This leads to an interesting engineering challenge: if one wanted to analyze data that is generated by IoT devices, then sending all data to a central server farm would be a relatively expensive operation. On the other hand, processing data closer to their source might be a more promising approach as this eliminates the cost of transferring data to the server for processing, and results back to the point of origin, if this is where the results of the data analysis are needed. Thus, depending on the time it takes to transfer data, moving computations to the periphery and processing them on devices with much less computational power than a central server may be a worthwhile tradeoff.

A Concrete Example: Timely Accident Notifications

Assume a great number of *connected cars* that emit position reports in fixed intervals. Given a certain interpretation of these data, which we will discuss later, we are able to determine the position of accident sites. After detecting an accident, the goal is to notify cars in the vicinity of an accident site. It would be in the interest of the drivers of the affected cars to be quickly informed about the fact that an accident happened nearby, either in order to put the driver on high alert, or to prompt him to change his route. Conceptually, assume the car transfers a position report to a nearby monitoring station, a so-called road side unit, which eventually sends an accident notification to the car, irrespective of where data analysis was performed. Latency, i.e. the time it takes between issuing a position report and receiving an accident notification, should be as low as possible. On the other hand, throughput needs to be high enough to ensure that position reports are processed in a timely manner.

1.2 Problem Description

The focus of our study is latency and throughput of processing streaming data in two IoT scenarios, using synthetic traffic data. First, we perform all computations on a central server. This entails greater costs for data transmission, but the hardware is

more powerful. Second, we move all computations from the server to computationally bound edge devices. By running computations on such edge devices, which are located closer to data sources, we avoid the cost of transferring data to the server and back. A disadvantage, though, is that the hardware used for data analysis is comparatively less powerful.
Let us clarify the tradeoffs related to the computational power of a server and the limited computational power of edge devices, and the effect of their respective location. Data is generated close to edge devices. If we want to transfer it to a server, we incur a cost for data transmission, which may be substantial, depending on the distance between an edge device and the server. Figure 1.1 illustrates this. The issue is the distance between server and edge devices.

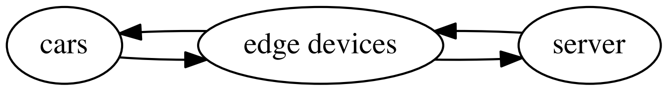

Figure 1.1: Data transfer from car to server and back

Available network technology, including protocols used, and the current network status all affect the duration of the transmission. The distance between the onboard unit of a connected car and a road side unit is assumed to be much shorter than the distance between a road side unit an a central server. Consequently, by using edge devices for computations, with the goal of processing data closer to their point of origin, we avoid the cost of transferring data from edge devices to the server, and the results back from the server to edge devices. If data is processed closer to their source, then the cost for transferring data to and from the server can be eliminated entirely. Thus, we are interested in using a concrete case study, namely timely notifying drivers of accidents, in order to determine how latency and throughput are affected in the aforementioned two scenarios.

1.3 Solution Overview

Using synthetic traffic data, we process a data stream in order to detect accidents. This entails the necessity of low latency and high throughput because drivers of vehicles need to know about accidents as soon as possible. Based on position reports that are provided by cars in regular intervals, our system detects accidents, monitors their status — accidents are not permanent, after all — and issues accident notifications for cars in the vicinity of accident sites. Every car that moves towards the site of an accident needs to receive a warning from the system.
A state-of-the-art stream processing engine is used for data analysis. The first task is to reliably detect accidents. Afterwards, information on accidents, while they are occurring and have not been resolved, needs to be maintained. With every position update a car sends, we need to check whether there is an accident site nearby,

and issue a response accordingly. We model two possible IoT scenarios. First, we perform data analysis on a monolithic server (center stream processing). Second, we perform all computations on a small number of computationally bound edge devices (edge stream processing).

1.4 Evaluation

Since the goal of our work is to measure the effect on latency and throughput in two scenarios, we need to find suitable measurements for them. Latency is measured as the time between emitting a position report and recording the result of a status request that notifies the driver of the presence or absence of an accident site nearby. Throughput is measured by keeping track of how many position reports per second the system is able to process, while maintaining a certain target for latency. Thus, we need to find the *saturation point* at which latency no longer meets our target, as the throughput is too high for the system to handle. At this saturation point we will have reached maximum throughput, given a particular value for latency.

1.5 Thesis Organization

The rest of this thesis report is organized as follows. Chapter 2 provides the necessary background for our work, covering streams, the Linear Road benchmark, stream processing engines, and vehicular networks. Building on this foundation, we present the problem description in much greater detail in Chapter 3. Chapter 4 discusses related work on Linear Road and stream processing in general. Chapter 5 presents the main part of our work: a discussion of the accident notification problem, covering theoretical as well as practical aspects of our solution. In Chapter 6 we present our results, i.e. measurements of latency and throughput of both center and edge stream processing, when performing data analysis required for accident notifications based on synthetic Linear Road traffic data. Chapter 7 contains a brief conclusion and an outlook towards potential future work.
Several appendices provide relevant supplementary information and material, so that our work can be more easily evaluated, but also replicated. Appendix A includes information on how to run the Linear Road data generator on a modern Linux installation. Appendix B provides the entire source code related to our data analysis with Apache Storm. Lastly, Appendix C lists our empirical measurements of latency and throughput values.

2
Background

This chapter provides relevant background information. We start with a presentation of streams and common operators. After introducing the Linear Road benchmark, which was the first benchmark for stream processing engines, we turn to stream processing engines in general. Lastly, we connect the simulations performed by the Linear Road data generator with current engineering efforts towards vehicular networks. Those may eventually make it possible to move beyond mere simulations and enable the implementation of targeted near real-time accident detections in the real world.

2.1 Streams

In data streaming, a *stream* is defined as an unbounded sequence of data. More concretely, streams are commonly understood as infinite sequences of uniform tuples, where a *tuple* is a fixed-length sequence of data. While tuples are expected to be uniform, they can nonetheless hold different data types. For instance, a tuple may contain the surname, first name, age, and marital status of a person, which combines two strings, an integer, and a boolean value in one tuple, e.g. $(\texttt{Doe}, \texttt{Joe}, \texttt{50}, \texttt{true})$.
There are two categories of operators on streams: they can either be stateless or stateful. Stateless operators do not need to maintain state related to the tuples they process. Commonly used stateless operators are *map*, *filter*, and *union*. The operator *map* applies a function to each incoming tuple, so that a value x is turned into $f(x)$. The *filter* operator discards tuples that do not satisfy a given predicate, and keeps all those that do. Lastly, the *union* operator takes n streams as its input, and produces one stream as its output. Stateless operators are easily parallelizable, because the computations they perform are atomic. Thus, due to the absence of interdependencies, the data they operate on can be processed in an arbitrary order.
On the other hand, stateful operators can be more complicated. Examples are window aggregations, where state consists of a predefined time window, or data classification tasks via machine learning, where state consists of the current edge weights of a neural network. For the sake of simplicity, though, let us ignore machine learning and instead illustrate window aggregation instead. For instance, if a website owner wanted to keep track of the most popular pages of his site, one approach would be to create a stream consisting of one tuple for each incoming HTTP request as well as a time stamp. Using a suitable granularity for the measured time, the state would keep track of the number of request per page in the given interval. The sliding window purges old records, though. For instance, a choice of 60 minutes

as the size of the sliding window, and a granularity of one minute, entails that every minute the records of the current first minute in this interval are dropped. Sliding windows are also relevant in an IoT setting. One example is the k nearest neighbors problem [40], where the k nearest neighbors may constantly change. Note that stateful operators can be difficult to parallelize because the order in which they are applied to data may matter. In fact, the parallelization of stateful operators is a subject of ongoing research. This field is of particular relevance since stateful operators are often bottlenecks in data analysis [38].

2.2 The Linear Road Benchmark

Linear Road [3] is an established benchmark for stream processing engines (SPEs). It originally appeared in 2004. Despite its age, it is still relevant, not in the least due to stream processing engines having gained a foothold in industry in recent years. The core idea behind this synthetic benchmark is the simulation of vehicular traffic in a virtual city. Concretely, it models a complex toll system that adjusts pricing based on constantly changing traffic conditions. This is referred to as *variable tolling* or congestion pricing, which has become an important tool in public policy [4]. In addition, Linear Road measures performance of historical queries, which plays towards the strengths of traditional relational databases as opposed to SPEs. Thus, they are not of any interest to us. Our focus is exclusively on a central part of variable tolling, namely accident detection and notification. In Linear Road, the presence of an accident is intended to trigger an increase in toll fees, with the goal of minimizing or rerouting traffic.
In the following, we present the specification and requirements of Linear Road, as stated by Arasu et al. [3], in condensed form. For precise values as well as an exact specification, we urge the reader to refer to their paper, as our focus is on a high-level description, which would be insufficient for guiding an implementation. Linear Road uses data generated by the traffic simulator MITSIM [39], which ensures *semantic validity* of data. This means that all data is internally consistent so that the location of all cars is plausible at all times. Traffic takes place on a simplified road layout consisting of parallel expressways with a length of 100 miles. Each expressway is subdivided into 100 one-mile segments, and consists of six one-directional lanes, of which one half are westbound and the other half eastbound. Of those three lanes each, one is for traveling, one is the entrance lane, and the remaining one is the exit lane.
The Linear Road benchmark uses a stream consisting of four kinds of tuples: position reports, and three different historical query requests, which we ignore. Position reports are issued in an interval of 30 seconds by every vehicle on an expressway. A position report is represented as a tuple consisting of the following values: `Type`, `Time`, `VID`, `Spd`, `XWay`, `Lane`, `Dir`, `Seg`, and `Pos`. A `Type` of value 0 indicates that the tuple is a position report, while `Time` is a timestamp of the traffic simulation that specifies when a position report was sent. `VID` is short for vehicle identifier, and `Spd` reflects the speed of the vehicle in miles per hour. `XWay` is the expressway the vehicle is on, while `Lane` specifies the particular lane of a particular expressway. The encoding of `Lane` differentiates between entry, exit, and travel lanes. `Dir` is

the direction of the vehicle, which is either eastbound or westbound. `Seg` is the segment of the expressway the vehicle is on. Lastly, `Pos` is the horizontal position of the vehicle. Note that encoding the vertical position would be redundant, since expressways are horizontally aligned.

2.3 Stream Processing Engines

Stream Processing Engines (SPEs) emerged as a response to the shortcomings of traditional database management systems (DBMSs) with regards to querying data streams. This was highlighted in the original Linear Road paper [3], which showed that Aurora [1], one of the first SPEs, greatly outperformed a state-of-the-art commercial database system, anonymized as "System X". It was pointed out that System X struggles with the latency requirements of Linear Road, with results that are up to several orders of magnitude worse than the performance of Aurora. By design, a DBMS first stores data and only afterwards processes it. Stonebraker et al. [29] succinctly summarize this by stating that "DBMSs do not keep the data moving". In contrast, SPEs are designed for continuous queries, i.e. real-time processing of data. Thus, data is analyzed on-the-fly by an SPE, without necessarily storing it. However, an SPE could of course be supplemented by a traditional DBMS. Part of the Linear Road benchmark, for instance, measures performance when processing historical data.
For our research project, we rely on Apache Storm [32], an SPE that was originally developed by Nathan Marz at BackType in 2010. After Twitter acquired BackType in 2011, Storm was released as an open-source project, and quickly adopted in industry. In 2013, Storm was accepted to Apache Incubator, and eventually promoted to an Apache top-level project in 2014. Conceptually, an Apache Storm job is modeled as a *topology*, which is a directed acyclic graph (DAG). Nodes are either *spouts* or *bolts*. The former are source nodes that emit tuples for further processing. The latter are operators that process tuples. In a Storm topology, streams of tuples are visualized as edges that connect operators. In its entirety, a topology is a data transformation pipeline of arbitrary complexity.

2.4 Vehicular Networks

Vehicular networks, while having been hypothesized about in academia for some time, for instance in an early paper by El Zarki et al. [41] in 2002, are slowly becoming a reality. With the advent of the connected car [12], i.e. cars with access to the Internet and wireless connectivity, vehicular networks are indeed becoming feasible. In fact, standardization has already begun. The IEEE standard 802.11 is the basis for products marketed with the WiFi logo. An extension to this standard is IEEE 802.11p [17], published in 2008, which covers wireless access in vehicular environments (WAVE). This enables the real-world scenario we model in our research project. IEEE 802.11p relies on the existence of onboard units (OBUs) in cars, as well as roadside units (RSUs) that are placed close to streets and highways. RSUs have the primary goal of increasing the coverage of a vehicular network, and are understood to be more powerful than OBUs. RSUs are normally understood to be separate entities, even though a proposal in which cars serve as RSUs has been made [31]. With connected cars increasingly being demanded by consumers, a scenario like the one modeled in our case study may become reality in the foreseeable future.

3

Detailed Problem Description

In this chapter we present a detailed problem description which assumes that the reader is familiar with the preceding chapter on background information. Our goal is to analyze how latency and throughput are affected by moving computations from a central server to computationally bound edge devices. In order to highlight the relevance of this problem, we first describe the real-world problem we base our case study on. This is followed by a description of our model, which approximates that real-world scenario.

3.1 Accident Detection in a Future IoT Setting

The real-world problem that provides the motivation for our project is an anticipation of a future IoT scenario. Consider a vehicular network, in which many connected cars continuously send data via on-board units (OBUs), such as position updates. Devices embedded in such cars are computationally bound. The same is true of Road Side Units (RSUs) that monitor vehicle traffic. RSUs are placed along a road, covering its entire length. A central location houses powerful servers. In the resulting hardware scenario cars communicate with RSUs that themselves communicate with central servers. Those servers perform analysis of the data they receive, and afterwards send their results back to RSUs. In turn, RSUs transmit information to cars. The roundtrip cost for transmitting data to the server and back may be negligible when RSUs are closely located to a server, but when the server is located far away from the RSUs it receives data from, the time penalty may be substantial.

We intend to model two scenarios: center stream processing and edge stream processing. In the first scenario, RSUs send position reports to a central server. After detecting an accident, based on data analysis, the server sends data to RSUs. In turn, those RSUs send accident notifications to cars if they come near an accident site. In the second scenario, all of the data analysis is performed on RSUs, which avoids the overhead associated with data transmission to a central server and back. An empirical question is whether there are benefits of *not* transferring data to the server and back. The trade-off is that there is no cost for data transmission between RSUs and the central server, since it is omitted. On the other hand, an RSU is a much less powerful device than a server, which may affect the amount of data that can be processed without jeopardizing meeting certain latency goals.

One possible benefit of connected cars sending position data, and being able to receive data via the network, are targeted near real-time accident notifications. Nowadays, untargeted accident notifications are transmitted via radio. This has the potential downside that such information reaches drivers who have no need for it, since their route is not even remotely close to the accident site. In a pathological case there may not be a single car near the site of an accident, yet an accident notification is sent out to all cars within a certain radius. In the case of connected cars, such information could be sent only to cars whose drivers it is relevant to. Furthermore, one may assume that automated accident notifications are much faster than the contemporary radio broadcast method, considering that the detection of accidents is automated, and the middle man, i.e. the radio station, is cut out.

3.2 Modeling Accident Detection Experimentally

Since we do not have access to a real vehicular network, we use synthetic data. Our data source is traffic data generated by Linear Road. This data contains traffic information for idealized expressways. Our data is for one expressway consisting of 100 one-mile segments. The data itself consists of position reports, which every single car emits in 30-second intervals. The Linear Road data generator ensures that the data itself is consistent, meaning that position reports mirror realistic traffic flow. Using this data, we model accident detection and notification. The data stream consisting of position reports is processed by the stream processing engine Apache Storm. Accidents are identified based on position reports, requiring two cars to be stopped at the same location for four consecutive position reports. Whenever an accident is detected in highway segment s, an accident notification is sent to all cars that issue a position report in segments $s, s-1, s-2$, and $s-3$.
The data itself consists of a stream of tuples. We measure throughput by the number of position reports per second we are able to process, with the goal of identifying the saturation point at which Apache Storm reaches its limit, given the hardware it is used on. Latency is measured as the difference of the time between a car sending a position report, and the system recording a notification that there has been an accident, or a notification that there has been none. The comparison of latencies, though, has to take the chosen scenario, edge or center stream processing, into account, as the measured latency only refers to the *internal* latency of the hardware the computations are performed on.
In one of the real world scenarios we consider, data is emitted by cars and sent to servers via RSUs. In the other, the server is omitted and all data is processed on RSUs. In Apache Storm we model a similar scenario. However, due to limitations in terms of equipment, we cannot model a scenario in which dozens or even hundreds of connected RSUs perform the data analysis. The first scenario is modeled by performing all computations on a server, on which we process a stream of position reports. For the second scenario we perform the entire data analysis on a computationally bound edge device, which we understand to approximate the computational power of an RSU, and extrapolate our results to using a greater number of RSUs. We measure latency and throughput in both scenarios, which may provide insights into how the location of computations affects those two metrics. In a real-world scenario, the number of RSUs to cover a highway of the length that formed the basis of our simulator data is rather high. Recall that in Linear Road, an expressway is 100 miles long, which roughly equals 160 kilometers. However, in real-life an RSU may

not be able to cover more than just a few hundred *meters*. For instance, in a 2010 paper, Lee et al. [19] describe a placement scheme for RSUs in Jeju City in South Korea. They achieved 72.5% connectivity with 1000 RSUs that have a transmission range of 300 meters each. With such a limited range, around 530 edge devices would be needed to cover one Linear Road highway.

4
Related Work

The main part of our research project is an analysis of the effect on latency and throughput in the context of stream processing when moving computations from a server to computationally-bound edge devices. As a case study, we use part of the Linear Road benchmark for stream processing engines. In the following, we therefore briefly discuss work related to the Linear Road stream processing benchmark as well as relevant research on stream processing itself.

4.1 Linear Road

An experiment quite similar to our own was part of a study undertaken by Costache et al. [7] who likewise model a vehicular network with connected computationally bound edge devices, and execute a custom implementation of Linear Road on it. The main difference is that they use Apache Flink and Apache Spark as opposed to Apache Storm.
In a study on "massive scale-out of expensive continuous queries," Zeitler et al. [42] introduce operators that split data streams into parallel substreams. Their approach led to a improvement of the rate of processed streams, adhering to the full Linear Road specification, of an order of a magnitude. The idea of splitting data streams into parallel substreams has been used in our work as well. In our case, this allowed us to increase throughput by distributing some computations on multiple instances of particular operators. This increased capacity limits. Instead of splitting streams into parallel substreams, Viel et al. [35] present a partitioning strategy, which they demonstrate via using Linear Road. The key point of their work is the identification of temporal approximate dependencies, which can be used to minimize communication costs.
Sheykh Esmaili et al. [26] use parts of the Linear Road benchmark for studying modifications of continuous queries. Concretely, they focus on accident notifications, analyzing the effect of query modification when strict latency requirements need to be maintained. We similarly focus on accident notifications, with the difference that we increase throughput in fixed intervals in order to determine the saturation point at which latency starts to deteriorate.
Surdu et al. [30] introduce window sizes for continuous queries, with the goal of reducing resource strain, both in terms of processing cost and memory consumption. In their paper they describe how to compute optimal window sizes, using Linear Road traffic data as input. A different approach to a related problem was presented by Gedik [13] who worked on adaptive compression of data streams, subject to

available CPU resources, with the goal of reducing the amount of bandwidth used. In our work, we limit processing cost by using an efficient data structure that allows amortized access time of $O(1)$, though. We furthermore limit memory consumption not by using window sizes but by pruning stored entries when it is appropriate. As a consequence, the total amount of memory we consume is bounded.

4.2 Stream Processing

In Digital Signal Processing (DSP), input data are streams as well. However, in DSP data is processed in chunks, which are fixed-size sequences of bits. Srivastava et al. [28] present heuristics as well as algorithms for designing workflows that minimize latency and maximize throughput. Their reasoning is done on a much lower level, involving bit-level operators. Their work influenced operator placement in our research project, which was possible since high-level data analysis tasks are conceptually related to low-level optimization in their DSP setting.
Vydyanathan et al. [36, 37] studied how latency and throughput in application workflows on clusters can be optimized. They develop heuristics for mapping, which is a stateless and therefore fully parallelizable operation, as well as scheduling. They achieve high throughput through parallelism. Identified key causes for low latency are task parallelism in addition to reducing communication between processes. We were able to confirm these general insights with the results of our experiments.
Work on operator migration is related to our research project as well. Ottenwälder et al. [21] study the effect of operator placement on efficient stream processing. They present an algorithm that considers both program state and streaming size in order to determine an optimal sequence of migrations, with the goal of maximizing network utilization. While their work is of a more general nature, Aniello et al. [2] focussed on a similar task using Apache Storm, describing two schedulers for deploying a Storm topology to available hardware, one that analyzes a given Storm topology before deployment, and another that monitors a topology that is being executed, with the goal of changing operator assignment in order to improve performance.
While we have been focussing on an application of stream processing in the transportation domain, there are other promising applications of this paradigm in Internet of Things settings. One prominent example is work related to advanced metering infrastructure, often referred to as the *smart grid*. The key idea behind smart metering is that connected devices communicate with an utility company such as an energy provider. Gulisano et al. [15] were arguably the first to use stream processing in the context of advanced metering. They measure latency and throughput of various batch sizes when performing data validation tasks, and show that stream processing, using a powerful server, is suitable for this domain. In a related later study, Gulisano et al. [16] explore issues of differential privacy [10] as, for instance, highly detailed energy consumption reports make it possible to profile users. For their evaluation, they use computationally bound edge devices. Their results are not directly comparable to ours, though, as we specifically measure the number of processed traffic reports instead of the total number of tuples processed in a topology. The latter is potentially a large multiple of the former since one tuple traversing n edges is processed $n + 1$ times.

5
Case Study: Accident Notifications

5.1 Overview

Our study focuses on targeted accident notifications, which is a problem within the transportation domain. To briefly summarize it: our data consists of position reports of a large number of vehicles. Whenever a vehicle issues such a position report, we want to send a timely response, tailored to that particular vehicle, indicating whether there is an accident ahead or not.

The accident detection problem can be divided into several separate parts, and modeled as network flow in a directed acyclic graph (DAG) or as a *topology*, to use Apache Storm terminology. As a first step, we detect whether a vehicle has stopped. The convention is that a vehicle is interpreted as having stopped if it occupies the exact same position for four consecutive position reports. Since position reports are issued every 30 seconds, this timespan amounts to two minutes. Of course, a single stopped car does not constitute an accident. Once the data shows that two cars have stopped at the same time in the same location, we interpret this as an accident. This information is relevant for cars that come near that accident site.

Accident notifications have to be provided in a timely manner in order to be effective. After all, the driver of a vehicle needs to receive an accident warning well before reaching the site of an accident in order to be able to react appropriately. Thus, throughput can only be increased up to a certain level. Concretely, latency needs to be kept below a certain threshold as otherwise an accident notification may, in an extreme case, be issued after a car that issues position reports has reached an accident site.

5.2 The Abstract Topology

To start with, we present a simplification of our topology, which omits some details. It also does not exhibit any kind of operator parallelism. This means that there is exactly one instance of each node. This simplified topology could be executed as it is. However, it may not exhibit optimal performance in the presence of multiple CPU cores as it assigns only one *executor* (thread) to each node. Nonetheless, this simplified abstract topology can be used to illustrate the various computations and how they relate to each other. The topology that was deployed on hardware is presented later on.

Three main tasks need to be performed: identifying stopped cars, identifying accidents, and providing accident notifications as a response to position reports sent by cars. An accident notification is a boolean value that states whether there is an accident ahead, relative to the position of a car that has sent a position report. Figure 5.1 presents a high-level view on how data is processed in order to perform the tasks just mentioned. Recall figure 1.1 in this context.

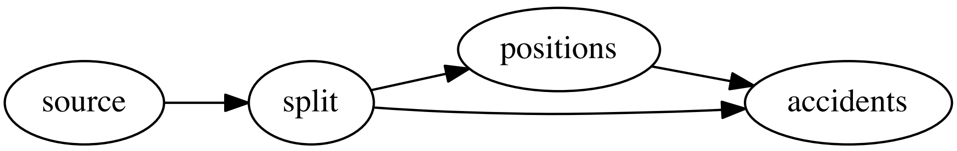

Figure 5.1: Simplified abstract accident detection topology

We stated above that a topology is modeled as a DAG. Indeed, the graph in figure 5.1 contains no cycles, and all its edges are directed. Using terminology from graph theory, `source` is the *source* of the network flow, while `accidents` is the *sink*. The source emits a potentially infinite stream of tuples. The simplified abstract topology consists of the four operators `source`, `split`, `positions`, and `accidents`. They are described below.

`source`	The source node `source` ingests data generated by Linear Road and outputs tuples that represent a position report.
`split`	The operator `split` takes a position report and transmits it to the operators `positions` and `accidents`. It transforms one input tuple into two different output tuples, which are then passed on to different nodes in the graph.
`positions`	The operator `positions` receives position reports. Its purpose is to identify stopped cars in the data it receives. It also detects when a previously stopped car is no longer stopped.
`accidents`	Using tuples that indicate stopped cars as inputs, as well as updates when a formerly stopped car is moving again, the operator `accidents` maintains information on every currently stopped car in its state. Once two stopped cars are detected that share the same location, this information is interpreted as an accident. Furthermore, incoming position report from a car are interpreted as status requests regarding accident sites, and processed.

After discussing the operators of the topology in figure 5.1 in detail, we will now describe how they interoperate. Position reports are emitted from `source`. Each position report constitutes one tuple. The recipient of this tuple is the operator `split`, which sends tuples to the operators `positions` and `accidents`. The operator `positions` analyzes incoming tuples. When it detects a stopped car, it sends a notification regarding that car to the operator `accidents`. Once that car is no longer stopped, another notification is sent to the operator `accidents`, prompting it to update the information it keeps on stopped cars. The operator `accidents` performs two actions. First, based on tuples sent from `positions`, its internal state records where cars have stopped. If there is a location with two stopped cars, this is interpreted as an accident. The internal state of this operator is also updated whenever a car is no longer stopped. Furthermore, whenever a position report from a vehicle, via `split`, reaches `accidents`, the latter determines whether there is an accident on the road ahead for that vehicle.

5.3 Extending the Abstract Topology

While the topology described in the previous section solves the accident detection problem in the context of Linear Road, one further addition is needed for bookkeeping, namely logging, in order to measure latency and throughput. The extended abstract topology is shown in figure 5.2.

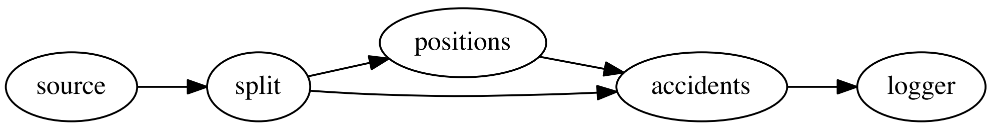

Figure 5.2: Extended abstract accident detection topology

Latency is the total time between emitting a position report as a tuple in `source` and recording the result of an accident notification request by the operator `logger`, which is the *sink* node of the extended abstract topology. The sole purpose of `logger` is to record statistics. In addition to determining the internal latency of processing position reports, it also determines throughput. Tuples sent from `accidents` constitute the input. All tuples pass through this operator, i.e. each position report triggers an accident notification request. Thus, a response indicating whether there is an accident site ahead is associated with each fully processed position report.

5.4 Implementation Details

In this section, we describe the implementation of the most relevant computations performed by the operators of the abstract topology we just discussed. The source code of our entire implementation in the Java programming language is provided in appendix B. In the following, we modify the terminology and refer to the operators by using terms associated with Apache Storm. A data source is a *spout*, while all other operators are *bolts*. Edges in the abstract topology represent *streams*, which consist of uniform tuples.

Creating a Stream from Finite Data

Considering that input/output operations are costly, we work with in-memory data. As part of the initialization of the spout `source`, a file of roughly 90 megabyte, consisting of one hour's worth of Linear Road position reports, is stored in an array. This array is used as a circular buffer of which each entry contains an unprocessed string representing a Linear Road position report. A global counter variable keeps track of the number of traversals through the buffer. The spout with its context is shown in figure 5.3.

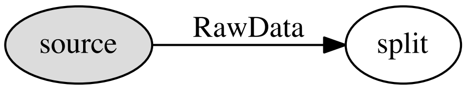

Figure 5.3: Data source

In our Apache Storm topology, tuples are not emitted as fast as possible but instead need to be requested by subsequent operators in the topology. When this happens, the value residing in the next slot in the circular buffer is retrieved. This string is subsequently turned into a tuple with relevant values. Part of the tuple is the field `time`, which stores, as an integer, the time in seconds at which the position report was issued. In order to ensure consistency of the tuples in the stream, this value is incremented by 3600, i.e. the number of seconds in an hour, for each complete traversal of the buffer. Thus, finite data for one hour of traffic is transformed into an unbounded stream. A theoretical objection to this approach is integer overflow of the `time` value. This is no cause for concern as the duration of our experiments is just a few minutes, which only results in processing data for a few hours' worth of position reports.[1]

Linear Road Tuple Format

Linear Road generates position reports as comma-separated values. For instance, the very first position report in the input data is the following:

`0,0,109,32,0,0,0,38,201154,-1,-1,-1,-1,-1,-1`.

The first value is the tuple type `Type`. Position reports, the only kind of data we are interested in, are of `Type` value 0. Entries with a different `Type` value are for historical requests, which are related to an aspect of Linear Road that does not concern us. A total of seven of the next eight values are directly relevant for our

[1]Integer overflow is not even a theoretical problem because it could be easily solved by reseting `time` to 0 after each traversal of the buffer. Accident durations are confined within hour intervals, meaning that the start and end time of an accident lie within the interval $[0n, ..., 3600n)$, where $n \in \mathbb{N}^+$, i.e. the set of natural numbers without 0.

implementation. In order, they are, as outlined int the original paper on Linear Road [3]:

`Time`	Simulation time of position report, one of [0..3599].
`VID`	Vehicle identity number $\in [0...max]$, where *max* is the maximum integer value on the system. Linear Road is a 32-bit application, thus $max = 2^{32} - 1$.
`Spd`	Speed of the vehicle. This value is not relevant for us.
`XWay`	Number of the expressway. In our implementation this value is taken into account in order to provide a more general solution. However, it is redundant in practice, as we only process data for `XWay` $= 0$.
`Lane`	Number of the lane of `XWay` $\in [0..4]$.
`Dir`	Direction of the vehicle; 0 for eastbound, 1 for westbound.
`Seg`	Segment of `XWay` the vehicle is on $\in [0..99]$. Each segment is one mile long.
`Pos`	Horizontal position of `VID` $\in [0..527999]$.[2]

Note that `Seg` is redundant as this value could be computed based on the `Pos` value. However a separate `Seg` value makes the implementation more straightforward. Some computations rely on the segment, and others on the position, in which case both values can be directly accessed without the need to convert a `Pos` value into a `Seg` value.

Limiting the Number of Tuples per Second

One of the goals of our experiments is to determine the saturation point at which latency starts to deteriorate. Considering that Apache Storm is pull-based, the topology would request as many tuples as possible, until capacity is reached and even exceeded. In order to enable more precise measurements, we limit the number of tuples emitted from the spout by keeping track of the number of tuples emitted in the current minute. Whenever the predefined limit is reached, the request of a bolt for another tuple is simply ignored.

Tuple format: `RawData` Stream

The stream `RawData` runs between the bolts `source` and `split`. The values `Time`, `VID`, `XWay`, `Lane`, `Dir`, `Seg`, and `Pos` are being extracted from the provided Linear Road data. The output of `source` contains the additional field `EmitTime`, which is the system time of the emission of the current tuple.

[2]Linear Road uses idealized roads, and idealized driving behavior. Thus, the vertical position of vehicles does not change since they all move in a perfectly straight line on perfectly straight roads.

Splitting and Modifying the `RawData` Stream

The bolt `split`, illustrated in figure 5.4, emits two streams.

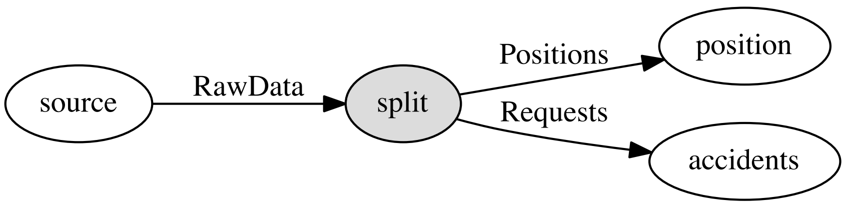

Figure 5.4: Splitting the data stream

The stream `Positions` connects to the bolt `positions`. It retains the following values of tuples coming in via the stream `RawData`: `Time`, `VID`, `XWay`, `Lane`, `Dir`, `Seg`, and `Pos`. The stream `Requests` connects to the bolt `accidents`. Since accidents are recorded by `Seg`, the value `Pos` is dropped. Furthermore, there is no need to submit the value `VID` since accidents are tied to locations. Of course, this entails that a car that is involved in an accident will continuously receive confirmation of the fact that it is involved in an accident. Tuples in this stream contain the values `Delta`, `EmitTime`, `Time`, `XWay`, `Lane`, `Dir`, and `Seg`.
An alternative approach would have been to let `source` emit the two streams `Positions` and `Requests`. However, this would have placed a limitation on the flexibility of our parallelisation strategy. By defining an additional bolt `split`, we retain the flexibility to instantiate different numbers of `source` spouts and `split` bolts. For instance, in the deployed topology, we use one `source` spout but more than one `split` bolt. Had there been no separation between processing input tuples and splitting the stream `RawData` into two separate streams `PositionReports` and `AccidentNotifications`, this approach would not have been possible.

Identifying Stopped Cars

The bolt `positions`, cf. figure 5.5, determines whether there are any vehicles that have stopped. Recall that a vehicle is interpreted as having stopped if it has issued four consecutive position reports from the exact same location, i.e. the same expressway (`XWay`), lane (`Lane`), direction (`Dir`), segment (`Seg`), and position (`Pos`).

Data Structures

The bolt `positions` makes use of hash maps and hash sets. In order to identify stopped cars, a global hash map `allPos` is maintained. This hash map contains a key for each encountered position. The corresponding values for this key are the hash maps `carsAtPos` that record the cars that have been encountered at this position. The keys of the hash map `carsAtPos` are vehicle identifiers (`VID`). The values

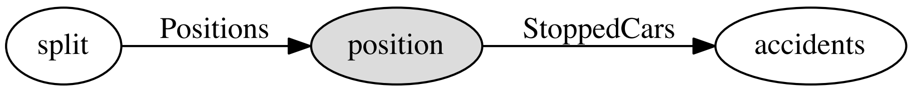

Figure 5.5: Identifying stopped cars in position reports

corresponding to those keys are pairs containing the time of the last position report (`Time`) that was issued by the vehicle, and a counter starting at 1. This counter increases by 1 every time a position report for `VID` is processed that was issued 30 seconds after the last one. This counter is initialized to 1 when encountering a new position report, and when encountering a position report that was issued more than 30 seconds later than the last one from the same location. This ensures that there is a strict upper bound on the amount of memory this hash map consumes, as old entries are constantly pruned and the set of `VID` values is bounded. Using a hash map allows us to determine in constant time whether a vehicle has stopped.

Determining whether a vehicle has stopped is straightforward, as we only need to check whether the associated counter has reached a value of 4 or more. If this is the case, the bolt `positions` emits *four* tuples into the stream `StoppedCars`. Once a vehicle has been identified as having stopped, it is added to the hash set `stoppedCars`, which uses the vehicle identifier (`VID`) as keys, and the time of the last position report, (`Time`), as values. Afterwards, irrespective of whether the current position report led to identifying a vehicle as having stopped, the entire hash set `stoppedCars` is processed in order to determine whether all vehicles that are contained in this hash set are still stopped. This is determined by the difference of the time of the current position report and the recorded time of the last position report of each car in `stoppedCars`. If this difference is greater than some fixed value — we chose 90 seconds as a threshold — the vehicle is marked for later removal. Java, the chosen implementation language, does not allow modifying a set while traversing its elements, which is why we cannot remove those keys immediately.

The chosen threshold value of 90 seconds may warrant an explanation. First, we need to routinely prune the data we collect as we are dealing with streams. A constantly growing data structure would eventually cause us to run out of memory. Second, Linear Road data is internally consistent, meaning that each car emits a position report every 30 seconds. Thus, there may be a delay of up to 29 seconds in Linear Road data between an accident having been resolved, because one of the involved cars has started moving again, and this fact being visible in the data.[3] The chosen value of 90 provides a generous upper bound.

[3]Assume car c is involved in an accident at time $t = 0$; at $t = 1$ it starts moving again. Yet, only at $t = 30$, which is 29 seconds after resolving the accident situation, the data reflects that there is no longer an accident. Note that in Linear Road, an accident always involves exactly two cars, meaning that the accident is resolved once one of the involved cars changes its position on the expressway.

Notifications

The bolts `positions` and `accidents` are connected via the stream `StoppedCars`, which consists of tuples with the values `Time`, `XWay`, `Lane`, `Dir`, `Seg`, `VID`, and `Stopped`. The value `Stopped` is a boolean that indicates whether the vehicle identified by `VID` has stopped. Of course, this is tied to the specified values for location and the current time.

One requirement of Linear Road is to send positive accident notifications for 4 segments, the segment in which the accident occurred, and the three previous segments. There are no notifications after a vehicle has passed an accident site. In order to take this requirement into account, the bolt `positions` sends a total of *four* tuples per stopped car to the bolt `accidents`. Those tuples differ only in their value of the segment `Seg`. One tuple each is sent for the four segments that indicate the interval for accident notifications. Let the value of `Seg` be x. The segments a notification is sent for are thus $x, x-1, x-2$, and $x-3$. Identifying a stopped car sets the value for `Stopped` to `true`.

Marking a `VID` for removal from the set `stoppedCars` triggers sending four tuples to the bolt `accidents`. This time, though, the value for `Stopped` is `false`, which indicates that a car that was previously recorded as having stopped is no longer in that state. Note that only one vehicle is removed per iteration. Considering that this check is performed for every processed position report, this is a feasible approach. After all, the number of stopped cars in the Linear Road data is minuscule compared to the number of position reports per time stamp.

Accident Detection and Notification

The bolt `accidents`, shown in figure 5.6, subscribes to two different streams. The stream `StoppedCars` is sent from the bolt `positions`. It is used for updating information on stopped cars. Furthermore, the stream `Requests` is sent from the bolt `split`.

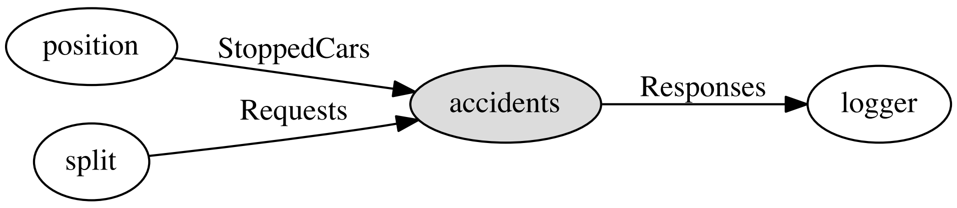

Figure 5.6: Accident notification

Identifying Accidents

In the synthetic traffic data generated by Linear Road, accidents need to be detected based on position reports of vehicles. When a vehicle has remained in the

same *location*, i.e. a particular VID reports the same expressway, lane, direction and position for four consecutive timestamps, it is interpreted as having *stopped*. When two cars have stopped in the same expressway, lane, direction and position, the data is interpreted as an accident. Linear Road does not generate accidents that involve more than two cars. Accidents are identified by the bolt accidents as it processes the stream StoppedCars.
A global hash map named status is used for keeping track of VID values of stopped cars in each location. If a tuple is received from the bolt positions, indicating that a car with a given VID has stopped, then the hash map status is updated by adding that VID as a value to the key specified by the *location*, which is a string that concatenates the values for XWay, Lane, Dir, and Seg. On the other hand, if such a tuple indicates that the car is no longer stopped, the corresponding value is removed from the hash map status. This ensures, due to the fact that there is a finite number of locations, that the hash map status does not grow infinitely large. Tuples indicating that a car is no longer stopped are only ever sent at some indefinite time after a tuple was sent that indicated that a particular car has stopped. It is not possible that the bolt accidents receives a notification that a particular car is no longer stopped without also having received, at some earlier point in time, a notification that this car has stopped.

Accident Notifications

When the bolt accidents receives a tuple from the stream Positions via the operator split, the hash map status is queried, using the location of the car that has issued a position report as a key. If two cars are recorded in the hash map status as having stopped in the provided location, an affirmative accident notification is triggered. Otherwise, the output tuple contains a value indicating that there is no accident in the provided location. An accident is indicated if there is more than one car in status for the given location. Due to the bolt positions sending four tuples whenever a car has stopped, accident notifications can be handled via a simple lookup. Accidents are not permanent occurrences, though. Thus, with each new position report there needs to be a check that the accidents are still ongoing.
After determining whether or not there is an accident site nearby, the bolt accidents emits a tuple, which is sent to the bolt logger. This tuple contains the values EmitTime, Time, Location, and Accident. The first value is used for computing the internal latency of the request, which highlighted further below. The value Location is a concatenation of the input values XWay, Lane, Dir, and Seg. Lastly, Accident is a boolean indicating the result of querying the hash map status. It is true if there is an accident in the current segment, or any of the upcoming three segments, and false otherwise.

Measuring Latency and Throughput

A separate bolt, logger, cf. figure 5.7, keeps track of the total number of tuples received in a predefined interval, which is detailed in Chapter 6. Whenever a tuple is received within that interval, a global counter variable count is incremented by 1. This variable eventually contains the total throughput in the defined interval.

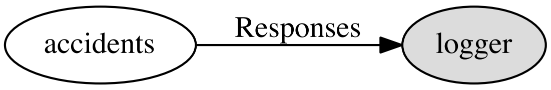

Figure 5.7: Logging

Latency is recorded similarly, using a global counter variable `sum`. Recall that every emitted position report comes with a time stamp, which is retained as the tuple passes through the bolts `split` and `accidents`. Every time a tuple finally reaches the bolt `logger`, the difference between the current time and the time of original emission is calculated.[4] The average latency is computed at the end and is simply the sum of latencies divided by the number of tuples received. The bolt `logger` eventually writes those statistics to a text file. This happens once per test run, and only after relevant statistical data have been taken, in order to avoid distortions that may result from writing to disk.

5.5 The Deployed Topology

The deployed topology is depicted by figure 5.8 below. In order to better exploit multiple CPU cores, it makes selective use of parallelism. Concretely, the bolts `split` and `positions` have multiple instances. The source and sink nodes, `source` and `logger`, are labelled as s and t, respectively. The other bolts are indicated by Greek letters: `split` by σ, `position` by π, and `accidents` by α. Parallelizing the bolt `positions` does not pose a problem. If there are multiple instances of this bolt, then position reports for a given location are always sent to the same bolt, which ensures that stopped cars can be reliably detected.
The deployed topology is the same for the central sever and the edge device scenario. A single server has to process all position reports. On the other hand, using multiple edge devices may require a different approach. Of course, there is the corner case that only one edge device is present, which would need to process all position reports as well. On the other hand, as the number of edge devices grows, they are assigned different data. This approach is realistic, considering that RSUs only have a limited reach. Also, accident notifications are only of rather confined local interest as they are restricted to a total of four one-mile segments: the segment in which an accident has occurred, and the three preceding segments.
In order to adequately divide the given input data, which covers a 100-mile highway, over n edge devices, it is not fully sufficient to designate n non-overlapping intervals of length $100/n$ and assign one to each device. Figure 5.9 below illustrates why this is potentially problematic. The segments are called a, b, c, d, e, and f. The first *interval* ends with segment c, while the second interval starts with segment d.

[4]Both time stamps for this calculation are taken on the same machine. This entails that the implementation is not affected by the clock synchronization problem.

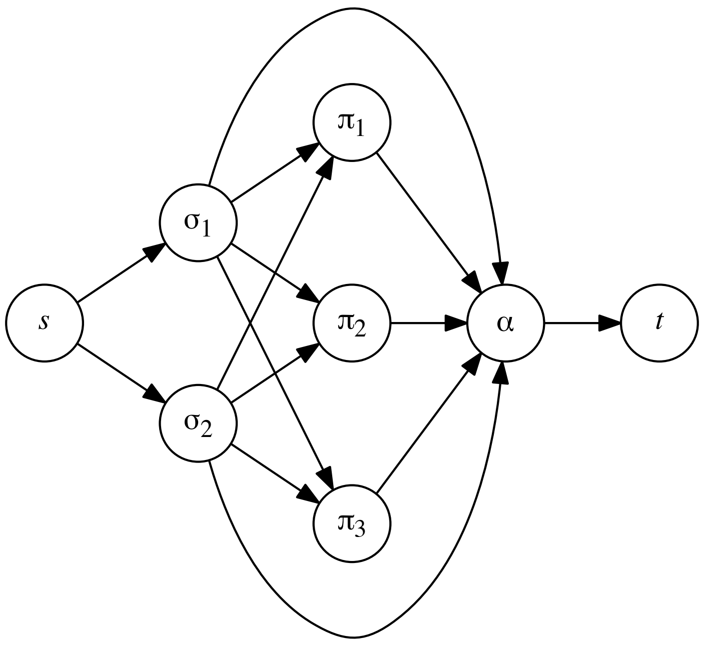

Figure 5.8: Deployed topology

Assume an accident occurs in segment f. As a consequence, vehicles in segments f, e, d, and c need to be notified. However, the edge device processing the first interval would not be aware of the accident and therefore could not notify any cars in segment c.

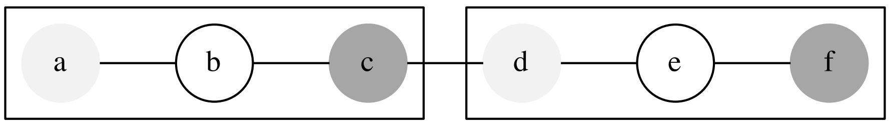

Figure 5.9: Accident notifications across segment intervals

Even worse would be the case of an accident in segment d. The device processing the second interval would issue correct accident warnings to cars entering that segment. However, there would be no advance warnings for vehicles in segments a, b, and c, thus leaving drivers in those segments in the dark until they enter segment d, in which the accident occurred.
In order to avoid such issues, the first $n - 1$ edge devices would have to process position reports of vehicles in the first three segments of the next interval. Of course, this does not apply to edge device n since the last segment in the interval it processes is the last segment overall, and roads are assumed to be non-circular. Another limitation is that an edge device would need to be able to process at least four segments in real time. The last three segments of any interval i covered by the

first $n-1$ devices would have to processed by the device covering interval $i+1$ as well. When processing less than four intervals, an edge device cannot notify all affected cars if there is an accident in the rightmost segment. If it does not also process the three first segments that are covered by the next interval, the problem illustrated by figure 5.9 will occur whenever there is an accident in those three segments.

6 Evaluation

The previous chapter concluded with a discussion of the topology that was used for solving the accident notification problem. In this chapter, we present the results of executing that topology on a server as well as an edge device. We will show that under certain circumstances, edge devices can replace a server, while maintaining near real-time characteristics of accident notifications. A particular focus is a proposal for comparing the total latency of the server and edge device scenario, which addresses the issue that the measured latency values in both experiments are not directly comparable. The first section describes the experimental setup, which details the used hardware, software, and data. The second section presents the results of our measurements, i.e. latency and throughput. Lastly, the third section discusses latency and throughput in the center and edge steam processing scenarios.

6.1 Evaluation Setup

The topology described in the last chapter was deployed on a server as well as a computationally-bound edge device.

Server

The server uses a 2009 quad-core AMD Opteron 2374 HE CPU with a variable clock speed of 800 MHz to 2200 MHz and 16 GB RAM. It runs Red Hat Enterprise Linux 6 (Santiago), compiled for the *amd64* instruction set. The Linux kernel version was 2.6.32. The installed software versions were Apache Storm 0.9.2, Apache Zookeeper 3.4.8, Java 1.6, and Maven 3.0.4.

Edge Device

As an edge device, the Odroid-XU4 by Hardkernel co., Ltd. was used. This device is equipped with a Samsung Exynos5422, which holds a quad-core Cortex-A15 CPU with 2000 MHz as well as a Cortex-A7 octa-core CPUs with a maximum frequency of 1400 MHz. This setup follows the ARM big.LITTLE architecture, where a relatively powerful CPU (Cortex-A15) is paired with a much weaker one (Cortex-A7). The Odroid-XU4 has 2 GB RAM. It ran Ubuntu Linux 15.04 for the *arm* instruction set, using kernel version 3.10.82. The installed software versions were Apache Storm 0.9.2, Apache Zookeeper 3.4.8, Java 1.8, and Maven 3.0.5.

Experimental Setup

We were using synthetic traffic data, which was generated by Linear Road. It was provided as a single text file with a size of about 90 megabyte, containing a total of 1838000 entries, of which 1819503 were position reports. The rest were requests for historical data, which are only relevant for Linear Road in general, but not for our experimen.
Measurements were taken after a warmup period of one minute for the server and three minutes for the edge device. The assumption was that the performance would be consistent after the warmup period. Since the edge device is considerably slower, we chose a longer warmup period in order to avoid distortions in the measurements. We measured latency and throughput over a five-minute period. The reported results are the averages of four executions each for different maximum rates of tuples per seconds. Relevant for our research is the saturation point at which the system reaches its maximum throughput. Apache Storm was used with *exactly-once* semantics, which implies a pull-based system, i.e. the spout only emits a tuple when requested. As a consequence, latency was constant in all our measurements.[1]

6.2 Measurements

All measurements are replicated in appendix C. By taking the average of four executions at the saturation point, the results are as follows. The server is able to process 245039 position reports in five minutes, which amounts to a throughput of 816.8 position reports per second. The latency was 2.8 milliseconds on average. With *one* edge device, we measured a throughput of 7992 position reports in five minutes, which amounts to a throughput of 26.6 position reports per second. The measured internal latency amounted to 74.8 milliseconds.
The input data contains one hour of position reports, amounting to 1819503 position reports in total, which leads to an average of 505 position reports per second. Thus, the server is able to process 161.6% of the number of position reports that are needed for real-time processing. On the other hand, a single edge device processes 5.3% of the position reports.

6.3 Comparing the Server with Edge Devices

Latency

The internal latency measures between the deployment on a server and on edge devices are not directly comparable as they model different real world scenarios. For an illustration, refer to the server scenario in 6.1, which illustrates that position reports are transmitted from OBUs via RSUs to a central server, and back. The deployed topology, though, relates only to the server. Consequently, the measured

[1]An alternative approach would have been to steadily emit tuples from the spout. The problem with this approach was that it lead to uneven performance. Either approach would arguably have been viable, as long as the execution was similar on both kinds of hardware.

latency refers to the *internal* latency on the server. In addition, there are latencies Δ_O and Δ_R, of which the former refers to the latency between OBUs and RSUs, and the latter to the latency between RSUs and the central server.

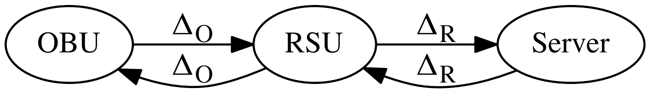

Figure 6.1: Total latency in center stream processing

On the other hand, the edge scenario models computations on RSUs. As figure 6.2 illustrates, there is only the additional latency Δ_O from OBUs to RSUs to consider. Since Δ_O is unknown in both scenarios, it can be omitted for the purpose of comparison, under the plausible assumption that it would be identical in both cases. Thus, the resulting comparison is between the measured latency in the edge scenario (L_E) and the measured latency in the center scenario (L_C) in addition to Δ_R. We make the assumption that each latency Δ_i is identical, irrespective of their direction.

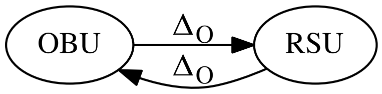

Figure 6.2: Total latency in edge stream processing

Thus, if one wanted to decide between center and edge stream processing, while merely considering latency, edge stream processing would be preferable only if $L_E < (L_C + 2\Delta_R)$. Using our measurements, and assuming that enough edge devices are deployed in order to make real-time processing feasible, the total latency of the edge scenario amounts to $2\Delta_O + 74.8ms$, while the total latency of the server scenario is $2\Delta_O + 2\Delta_R + 2.8ms$. Thus, the edge scenario is preferable if $2\Delta_R + 2.8ms > 74.8ms$.

Throughput

Comparing throughput is more straightforward. Considering that one edge device is able to process 5% of the position reports in real-time, the computational power of 20 edge devices would be needed to process all of them. Of course, this assumes that the same amount of position reports occurs for either each assigned interval, or for each segment. Thus, one edge device processes position reports for five segments, which amounts to five miles of the expressway. Note that, as was discussed in section 3.2, this would be unfeasible with current hardware, as RSUs do not even remotely have such a reach. This is partly an issue of the chosen data, though. After all, the expressway Linear Road simulates is not particularly busy. For simplicity, let us assume that traffic is evenly distributed among all 100 one-mile segments of an expressway. There are around 500 position reports per second, and vehicles emit a position report every 30 seconds. Thus, there are, on average, $500/100 * 30 = 150$ vehicles per mile on the expressway, which amounts to one car roughly every 11.7 yards or 10.7 meters. Most drivers, though, are familiar with roads with a much higher density. Thus, there are many real-world situations where the same amount of traffic that is modeled in Linear Road is reached on roads that are significantly shorter than 100 miles.

Note that our approach deliberately does not address the issue of overlapping intervals, which was discussed in section 5.5. The discussion in this chapter shows how using multiple edge devices scales, as they all process an interval of segments in isolation. However, as one edge device is only able to process five segments, some vehicles may receive accident notifications a few segments too late. Using our simplified model for scaling, the Linear Road requirement of issuing accident notifications for four segments thus cannot be fully met in all cases. This could be remedied by deploying a greater number of edge devices. With one overlapping segment, $100/(5-1) = 25$ edge devices would be needed, which all process four segments as well as the first segment of the next interval of the expressway. Thus, in the worst case, an accident notification may not be received in only the first two of the four affected segments. This happens if there is an accident in the second segment of the next interval. With two overlapping segments, $\lceil 100/(5-2) \rceil = 34$ devices would be needed, which implies that an accident in the third segment of the next interval would not be correctly reported in the current interval. Finally, with three overlapping segments per interval, $100/(5-3) = 50$ devices would be needed, which would lead to accident notifications in all segments.

7
Conclusion and Outlook

In this thesis report we have presented a comparison of the performance of edge devices and servers for solving the accident notification problem. Our formalization presents a solution to the general case. In addition, we used concrete measurements that show when a server may be replaceable by edge devices.
Our results show that a modest number of edge devices could replace a central server. While this is only theoretically true for the roads Linear Road models, considering the currently limited reach of RSUs, the potential for regular streets is much higher. In fact, given the rather low traffic density of Linear Road, it is arguably easy to find roads with a length that only amounts to a fraction of a Linear Road expressway, but which carry just as much traffic, if not more. In that case, the limited reach of RSUs would be much less of an issue. They would adequately cover much shorter roads, while their computational power would suffice to process position reports in real-time. Considering that RSUs need to be deployed anyway in a vehicular network, it seems that a good argument could be made that they should be used for performing computations instead of merely passing data along.
It would be interesting to work with real traffic data in future work, so that a realistic accident notification system could be modeled. Note that there is an important implication: given the still rapid advance of embedded systems, the next generations of edge devices can be expected to be significantly faster. Yet, roads have physical constraints, meaning that there is a limit with regards to how much traffic they can carry. Thus, while there is not yet an end in sight with regards to the increase of computational power of CPUs in embedded devices, there is a relatively fixed upper bound with regards to the number of vehicles that can travel on any given road. This implies that traffic data analysis on RSUs will only get easier, and may one day be the norm.

Bibliography

[1] Abadi, Daniel J., Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Aurora: a new model and architecture for data stream management." The VLDB Journal — The International Journal on Very Large Data Bases 12, no. 2 (2003): 120-139.

[2] Aniello, Leonardo, Roberto Baldoni, and Leonardo Querzoni. "Adaptive online scheduling in storm." In Proceedings of the 7th ACM international conference on Distributed event-based systems, pp. 207-218. ACM, 2013.

[3] Arasu, Arvind, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. "Linear road: a stream data management benchmark." In Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, pp. 480-491. VLDB Endowment, 2004.

[4] Button, Kenneth, and Erik Verhoef. Road pricing, traffic congestion and the environment. Edward Elgar, 1998.

[5] Chen, Hsinchun, Roger HL Chiang, and Veda C. Storey. "Business Intelligence and Analytics: From Big Data to Big Impact." MIS quarterly 36, no. 4 (2012): 1165-1188.

[6] Clark, Don. "Intel Rechisels the Tablet on Moore's Law". WSJ.com. http://blogs.wsj.com/digits/2015/07/16/intel-rechisels-the-tablet-on-moores-law/ (accessed February 15, 2016).

[7] Costache, Stefania, Vincenzo Gulisano, and Marina Papatriantafilou. "Understanding the data-processing challenges in Intelligent Vehicular Systems." In 2016 IEEE Intelligent Vehicles Symposium (IV), pp. 611-618. IEEE, 2016.

[8] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51, no. 1 (2008): 107-113.

[9] Dittmar, Hank, Karen Frick, and David Tannehill. Institutional and political challenges in implementing congestion pricing: case study of the San Francisco Bay Area. No. 242. 1994.

[10] Dwork, Cynthia, Frank McSherry, Kobbi Nissim, and Adam Smith. "Calibrating noise to sensitivity in private data analysis." In Theory of Cryptography Conference, pp. 265-284. Springer Berlin Heidelberg, 2006.

[11] Eliasson, Jonas, and Lars-Göran Mattsson. "Equity effects of congestion pricing: quantitative methodology and a case study for Stockholm." Transportation Research Part A: Policy and Practice 40, no. 7 (2006): 602-620.

[12] Evans-Pughe, Chris. "The connected car." IEE Review 51, no. 1 (2005): 42-46.

[13] Gedik, Bugra. "Discriminative Fine-Grained Mixing for Adaptive Compression of Data Streams." Computers, IEEE Transactions on 63, no. 9 (2014): 2228-2244.

[14] Greenhalgh, Peter. "Big. little processing with arm cortex-a15 & cortex-a7." ARM White paper (2011): 1-8.

[15] Gulisano, Vincenzo, Magnus Almgren, and Marina Papatriantafilou. "Online and scalable data validation in advanced metering infrastructures." In IEEE PES Innovative Smart Grid Technologies, Europe, pp. 1-6. IEEE, 2014.

[16] Gulisano, Vincenzo, Valentin Tudor, Magnus Almgren, and Marina Papatriantafilou. "BES: Differentially Private and Distributed Event Aggregation in Advanced Metering Infrastructures." In Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security, pp. 59-69. ACM, 2016.

[17] Jiang, Daniel, and Luca Delgrossi. "IEEE 802.11 p: Towards an international standard for wireless access in vehicular environments." In Vehicular Technology Conference, 2008. VTC Spring 2008. IEEE, pp. 2036-2040. IEEE, 2008.

[18] Kuroda, Tadahiro. "CMOS design challenges to power wall." In Microprocesses and Nanotechnology Conference, 2001 International, pp. 6-7. IEEE, 2001.

[19] Lee, Junghoon, and Cheol Min Kim. "A roadside unit placement scheme for vehicular telematics networks." In Advances in computer science and information technology, pp. 196-202. Springer Berlin Heidelberg, 2010.

[20] Manyika, James, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H. Byers. "Big data: The next frontier for innovation, competition, and productivity." (2011).

[21] Ottenwälder, Beate, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. "MigCEP: operator migration for mobility driven distributed complex event processing." In Proceedings of the 7th ACM international conference on Distributed event-based systems, pp. 183-194. ACM, 2013.

[22] Peña-López, Ismael. "OECD Digital Economy Outlook 2015." (2015).

[23] Rothensee, Matthias. "User acceptance of the intelligent fridge: empirical results from a simulation." In The Internet of Things, pp. 123-139. Springer Berlin Heidelberg, 2008.

[24] Rouhani, Omid M., and Debbie Niemeier. "Flat versus spatially variable tolling: A case study in Fresno, California." Journal of Transport Geography 37 (2014): 10-18.

[25] Schaller, Robert R. "Moore's law: past, present and future." Spectrum, IEEE 34, no. 6 (1997): 52-59.

[26] Sheykh Esmaili, Kyumars, Tahmineh Sanamrad, Peter M. Fischer, and Nesime Tatbul. "Changing flights in mid-air: a model for safely modifying continuous queries." In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pp. 613-624. ACM, 2011.

[27] Silver, Nate. The signal and the noise: Why so many predictions fail-but some don't. Penguin, 2012.

[28] Srivastava, Mani B., and Miodrag Potkonjak. "Transforming linear systems for joint latency and throughput optimization." In European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation.

ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings., pp. 267-271. IEEE, 1994.

[29] Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing." ACM SIGMOD Record 34, no. 4 (2005): 42-47.

[30] Surdu, Sabina, and Vasile-Marian Scuturici. "Addressing resource usage in stream processing systems: sizing window effect." In Proceedings of the 15th Symposium on International Database Engineering & Applications, pp. 247-248. ACM, 2011.

[31] Tonguz, Ozan, and Wantanee Viriyasitavat. "Cars as roadside units: a self-organizing network solution." Communications Magazine, IEEE 51, no. 12 (2013): 112-120.

[32] Toshniwal, Ankit, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson et al. "Storm@ twitter." In Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 147-156. ACM, 2014.

[33] United States Congress, "Energy Independence and Security Act of 2007." Public law 110, no. 140 (2007): 19.

[34] Verhoef, Erik T., Peter Nijkamp, and Piet Rietveld. "The social feasibility of road pricing: a case study for the Randstad area." Journal of Transport Economics and Policy (1997): 255-276.

[35] Viel, Emeric, and Hiroshi Ueda. "Data stream partitioning re-optimization based on runtime dependency mining." In Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on, pp. 199-206. IEEE, 2014.

[36] Vydyanathan, Nagavijayalakshmi, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. "Toward optimizing latency under throughput constraints for application workflows on clusters." Euro-Par 2007 Parallel Processing (2007): 173-183.

[37] Vydyanathan, Nagavijayalakshmi, Umit Catalyurek, Tahsin Kurc, Ponnuswamy Sadayappan, and Joel Saltz. "Optimizing latency and throughput of application workflows on clusters." Parallel Computing 37, no. 10 (2011): 694-712.

[38] Wu, Sai, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. "Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?." In Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 278-289. ACM, 2012.

[39] Yang, Qi, and Haris N. Koutsopoulos. "A microscopic traffic simulator for evaluation of dynamic traffic management systems." Transportation Research Part C: Emerging Technologies 4, no. 3 (1996): 113-129.

[40] Yu, Xiaohui, Ken Q. Pu, and Nick Koudas. "Monitoring k-nearest neighbor queries over moving objects." In Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on, pp. 631-642. IEEE, 2005.

[41] El Zarki, Magda, Sharad Mehrotra, Gene Tsudik, and Nalini Venkatasubramanian. "Security issues in a future vehicular network." In European Wireless, vol. 2. 2002.

[42] Zeitler, Erik, and Tore Risch. "Massive scale-out of expensive continuous queries." VLDB Endowment 4, no. 11 (2011).

A
Preliminaries

A.1 Running the Linear Road Data Generator on Linux

Considering that Linear Road is, at the time of writing, about 12 years old, it is of little surprise that there are several pitfalls when trying to install this benchmark on a modern Linux installation. As this was quite an ordeal, this appendix records how to execute the Linear Road data generator on Linux. The bulk of our work was carried out on an Apple MacBook Pro running on OS X. For the Linear Road data generator, we set up a virtual machine with Xubuntu 15.10 on VirtualBox. This is a long-term release, which will be supported until October 2018.
This section describes how to set up VirtualBox, Perl, and Postgres. Afterwards, we describe how to set up the Linear Road data generator.

VirtualBox

The first pitfall is that Linear Road comes with precompiled 32-bit binaries, which cannot be executed on 64-bit Linux versions without jumping through additional hoops. Thus, we used the 32-bit version of Xubuntu 15.10, and installed it on VirtualBox. An 8 GB virtual hard drive was sufficiently large as generated data can be written to a shared directory of the host operating system. In order to enable automatic resizing of the desktop and activate shared folders between the virtual machine and the host machine, the VirtualBox guest additions are required. These can be installed via the command line:

```
sudo apt-get update
sudo apt-get install virtualbox-guest-dkms
```

After setting up a shared folder, a restart is required. After rebooting, it is necessary to add permissions for accessing the shared folder. For instance, in order to enable user `foo` to access the shared folder, type `sudo adduser foo vboxsf`.

Perl

Perl is preinstalled in Xubuntu 15.10. However, some modules that are required for Linear Road are missing. In order to proceed, type `cpan` at the command line, and select the option `sudo`. This leads to an automatic setup. Afterwards, the following commands need to be executed in `cpan`:

```
install DBI
install DBD::PgPP
install Math::Random
exit
```

Postgres

Postgres is not preinstalled on Xubuntu 15.10. In the following, we are assuming that a user `foo` exists on the system. In a terminal window, type:

```
sudo apt-get install postgresql
sudo -i -u postgres
psql
```

In the `psql` console, type:

```
CREATE USER foo;
CREATE DATABASE linear;
ALTER USER "foo" WITH PASSWORD 'foo';
ALTER USER "foo" WITH SUPERUSER;
\q
```

Ubuntu and most if not all its derivative distributions switched to the init system `systemd` in version 15.10. Thus, the following commands are required to start a Postgres server:

```
su - foo
sudo systemctl start postgresql
sudo systemctl status postgresql
```

The last command is used for checking that the Postgres server is active.

Linear Road Data Generator

Download and unzip the data generator from the Linear Road home page. In the terminal, navigate to the directory the file was unpacked into. Type `chmod -R 777 .`, including the period, in order to set the correct permissions. Open `mitsim.config`, and enter the required data for path, database name (`linear`), user name (`foo`) and password (`foo`). Afterwards, a symbolic link between the location of the Postgres database and the location Linear Road expects needs to be created. In the terminal, type the following to find out the path of the Postgres database:

```
sudo -u postgresql psql -c "SHOW unix_socket_directories;"
sudo -u postgresql psql -c "SHOW port;"
```

On our machine, the required symlink was created by:

```
sudo ln -s /var/run/postgresql/.s.PGSQL.5432 /tmp/.s.PGSQL.5432
```

After navigating to the binary file `mitsim` in the terminal and typeing `ldd mitsim`, the reader may notice that a required object file is missing. To solve this issue, the file `libstdc++2.10-glibc2.2_2.95.4-27_i386.deb` needs to be retrieved online. It is part of a rather old and still available Debian distribution. The two object files in the folder `/lib` need to be extracted, and moved to `/usr/lib/gcc-compat`. Afterwards, the library path needs to be exported. Type the following in the terminal: `export LD_LIBRARY_PATH=/usr/local/lib/gcc-compat`. At long last, it is now possible to execute the Linear Road data generator by typing `./run mitsim.config`.

A.2 Python Script for Identifying Accidents

In order to verify that accidents were correctly identified, a Python script was used for computing location, duration, and involved vehicles of each accident that is recorded in the provided traffic data. The solution below is not in full generality, as it takes peculiarities of the data into account. For instance, there is not a single case in which a car stops for two or more minutes without being involved in an accident.
The script works as follows: First, all position reports are extracted from the input file. Recall that position reports are of `Type` $= 0$. All valid position reports are added to a hash map, where the key is a tuple of identifiers for expressway, lane, direction and position. Those four identifiers uniquely identify each possible location that is described by the traffic data. The value corresponding to the key just mentioned consists of `VID` and `Time`.
For convenience, we afterwards filter entries and exclude all keys from consideration where the corresponding value has less than 8 elements. This is due to the fact that the lowest possible number of reports that constitute an accident is given by four consecutive position reports by two different cars for the same location. Lastly, the remaining entries are narrowed down and reduced to a list of tuples that give a `VID` and the time interval, with start and end points, during which it was stopped.
Before providing the code, we give an example of slightly reformatted output of this script, which illustrates how accidents are identified per location. The example below was extracted from traffic data for one hour. The number of traffic accidents is small enough to verify them manually. The key specifies the accident location, while the value identifies cars by `VID` and the interval during which they were stopped. An accident occurs when two cars are stopped in the same location. For instance, cars with `VID` 71 and 88 are in an accident in the interval $[333, 930]$, according to the data.

```
{(0, 1, 0, 485759): [(71, (333, 933)), (88, (330, 930))],
 (0, 1, 1, 438240): [(5844, (1521, 2151)), (11181, (1500, 2130))],
 (0, 3, 0, 249159): [(19461, (2725, 3565)), (30297, (2700, 3570))]}
```

This is the Python 2.7 script:

```
posReports  = dict()
candidates  = dict()
stoppedCars = dict()


def readFile():
    with open("cardatapoints_1.out0") as f:
        for line in f:
            tmp = line.strip().split(",")

            reportType = int(tmp[0])

            if not reportType == 0:
                continue

            else:
                time      = int(tmp[1])
                vid       = int(tmp[2])
                xway      = int(tmp[4])
                lane      = int(tmp[5])
                direction = int(tmp[6])
                pos       = int(tmp[8])
                key = (xway, lane, direction, pos)
                if key not in posReports:
                    posReports[key] = [(vid, time)]
                else:
                    val = posReports[key]
                    val.append((vid, time))
                    posReports[key] = val


def getCandidates():
    # need to sort by vid!
    for (key, val) in posReports.iteritems():
        if len(val) >= 8:
            val.sort()
            candidates[key] = val
            print key, val


def reduceCandidates():
    for (key, val) in candidates.iteritems():
        val            = getTimespan(val)
        candidates[key] = val


def getStoppedCars():
    for (key, val) in candidates.iteritems():
        stopped = []
        for (vid, (from_time, to_time)) in val:
            if to_time - from_time >= 120:
                stopped.append((vid, (from_time, to_time)))
        if not stopped == []:
            stoppedCars[key] = stopped


def getTimespan(lst):
    assert not len(lst) == 0

    (vid, time) = lst[0]
    tmp         = (vid, (time, time))
    return getTimespanAux(lst[1:], tmp, [])


def getTimespanAux(lst, tmp, acc):
    if lst == []:
        acc.append(tmp)
        return acc
    else:
        (vid, time)                     = lst[0]
        (tmp_vid, (from_time, to_time)) = tmp
        if not vid == tmp_vid:
            acc.append(tmp)
            return getTimespanAux(
                lst[1:], (vid, (time, time)), acc)
        else:
            assert time - to_time == 30
            return getTimespanAux(
                lst[1:], (vid, (from_time, time)), acc)


readFile()
getCandidates()
reduceCandidates()
getStoppedCars()

print stoppedCars
```

B
Accident Notification Source Code

B.1 Topology

LinearRoadMain.java

```
package storm.starter;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;

import storm.starter.lr_spout.Source;
import storm.starter.lr_bolts.Split;
import storm.starter.lr_bolts.Position;
import storm.starter.lr_bolts.Accidents;
import storm.starter.lr_bolts.Logger;


public class LinearRoadMain {

  public static void main(String[] args) throws Exception {

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout(
      "spout",
      new Source(), 1)
        .setMaxSpoutPending(4);
        // avoids congestion and also ensures that there are
        // enough tuples in flight

    builder.setBolt(
      "split",
      new Split(), 2)
        .shuffleGrouping(
          "spout",
          "RawData");

    builder.setBolt(
      "position",
      new Position(), 3)
        .fieldsGrouping(
          "split",
          "Positions",
          new Fields("XWay", "Lane"));

    builder.setBolt(
      "accidents",
      new Accidents(), 1)
        .shuffleGrouping(
          "position",
          "StoppedCars")
        .shuffleGrouping(
          "split",
          "Requests");

    builder.setBolt(
      "logger",
      new Logger(), 1)
        .shuffleGrouping(
          "accidents",
          "Responses");

    Config conf = new Config();
    conf.setDebug(true);

    if (args != null && args.length > 0) {
      conf.setNumWorkers(2);
      StormSubmitter.submitTopologyWithProgressBar(
        args[0], conf, builder.createTopology());

    } else { // run locally for testing
      conf.setMaxTaskParallelism(1);

      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("linear-road", conf, builder.createTopology());

      // run topology for 7 minutes
      Thread.sleep(420000);

      cluster.shutdown();
    }

    System.out.println("Done.\n");
  }

}
```

B.2 Spout

Source.java

```
package storm.starter.lr_spout;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.File;
import java.io.IOException;
import java.io.FileNotFoundException;

import java.util.ArrayList;
import java.util.Map;


public class Source extends BaseRichSpout {

  SpoutOutputCollector collector;

  final int LIMIT  = 500;

  String lr_file   = "cardatapoints_1.out0";
  int    tupleID   = 0;
  int    bufferPos = 0;
  int    iteration = 0;

  String[] buffer;

  long thisSec;
  long nextSec;
  int emittedInCurrentSec;

  @Override
  public void open(Map conf,
                   TopologyContext context,
                   SpoutOutputCollector collector) {

    thisSec             = System.currentTimeMillis();
    nextSec             = thisSec + 1000;
    emittedInCurrentSec = 0;
    this.collector      = collector;

    try {

      BufferedWriter out = new BufferedWriter
      (new FileWriter("start.txt", true));
      out.write(System.currentTimeMillis() + "\n");
      out.write("OK.\n");
      out.close();

      FileReader        fileReader = new FileReader(lr_file);
      BufferedReader    reader     = new BufferedReader(fileReader);
      int               count      = 0;
      String            entry      = null;
      ArrayList<String> tmp        = new ArrayList<String>();

      while ((entry = reader.readLine()) != null){
        tmp.add(entry);
        count++;
      }

      buffer = new String[count];

      for (int i = 0; i < tmp.size(); i++) {
        buffer[i] = tmp.get(i);
      }
    }

    catch (FileNotFoundException e) {
      System.out.println("File␣not␣found.");
      String cwd = System.getProperty("user.dir");
      System.out.println("cwd:␣" + cwd);

    } catch(Exception e){
      throw new RuntimeException("Error␣reading␣tuple.", e);

    } finally {}

  }


  @Override
  public void nextTuple() {

      long val = System.currentTimeMillis();

      if (val >= nextSec) {
        emittedInCurrentSec = 0;
        long tmp = nextSec;
        thisSec  = tmp;
        nextSec  = thisSec + 1000;
      }

      if (val < nextSec  && emittedInCurrentSec > LIMIT) {
          return;
      }


      if (bufferPos >= buffer.length) {
        // process buffer anew
        iteration++;
        bufferPos = 0;
      }

      String   entry     = buffer[bufferPos];
      String[] allValues = entry.split(",");
      int[]    vals      = new int[allValues.length];

      for(int i = 0; i < vals.length; i++) {
        vals[i] = Integer.parseInt(allValues[i]);
      }

      int type = vals[0];
      int time = vals[1];
      int vid  = vals[2];
      //int spd  = vals[3];
      int xway = vals[4];
      int lane = vals[5];
      int dir  = vals[6];
      int seg  = vals[7];
      int pos  = vals[8];

      // add one hour (3600 sec) for each iteration
      time += iteration * 3600;

      long emitTime = System.nanoTime();

      // only consider position reports
      if (type == 0) {
        collector.emit(
          "RawData",
          new Values(
            emitTime,
            time, vid, xway, lane, dir, seg, pos), tupleID++);
      }

      emittedInCurrentSec++;
      bufferPos++;

}


  @Override
  public void ack(Object id) {
  }


  @Override
  public void fail(Object id) {
  }


  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {

    declarer.declareStream(
      "RawData",
        new Fields(
          "EmitTime", "Time",
          "VID", "XWay", "Lane", "Dir", "Seg", "Pos"));
  }

}
```

B.3 Bolts

Split.java

```java
package storm.starter.lr_bolts;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;

import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;


public class Split extends BaseBasicBolt {

  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {

    int  time     = input.getIntegerByField("Time");
    int  vid      = input.getIntegerByField("VID");
    int  xway     = input.getIntegerByField("XWay");
    int  lane     = input.getIntegerByField("Lane");
    int  dir      = input.getIntegerByField("Dir");
    int  seg      = input.getIntegerByField("Seg");
    int  pos      = input.getIntegerByField("Pos");
    long emitTime = input.getLongByField("EmitTime");


    collector.emit(
      "Positions",
      new Values(time, vid, xway, lane, dir, seg, pos));

    collector.emit(
      "Requests",
      new Values(emitTime,
        time, xway, lane, dir, seg));

  }


  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {

    declarer.declareStream(
      "Positions",
      new Fields("Time", "VID", "XWay", "Lane", "Dir", "Seg", "Pos"));

    declarer.declareStream(
      "Requests",
      new Fields("EmitTime",
        "Time", "XWay", "Lane", "Dir", "Seg"));

  }

}
```

Position.java

```java
package storm.starter.lr_bolts;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.util.HashMap;
import java.util.Map;
import java.util.HashSet;


public class Position extends BaseBasicBolt {

  // all stopped cars
  // key: VID, value: time of last position report
  Map<Integer, Integer> stoppedCars = new HashMap<Integer, Integer>();

  // key: location, val: car status
  Map<String, Map<Integer, Pair<Integer, Integer>>> allPos =
    new HashMap<String, Map<Integer, Pair<Integer, Integer>>>();


  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {

    if (input.getSourceStreamId().equals("Positions")) {

      int    time     = input.getIntegerByField("Time");
      int    vid      = input.getIntegerByField("VID");
      int    xway     = input.getIntegerByField("XWay");
      int    lane     = input.getIntegerByField("Lane");
      int    dir      = input.getIntegerByField("Dir");
      int    seg      = input.getIntegerByField("Seg");
      int    pos      = input.getIntegerByField("Pos");
      String location = xway + "." + lane + "." + dir + "." + seg + "." + pos;

      Map<Integer, Pair<Integer, Integer>> carsAtPos = allPos.get(location);

      // is there an entry for the current position?
      if (allPos.get(location) == null) {
        carsAtPos = new HashMap<Integer, Pair<Integer, Integer>>();
        allPos.put(location, carsAtPos);
      }

      // has the current vid been encountered at the current location?
      if (allPos.get(location).get(vid) == null) {
        Pair<Integer, Integer> initStats = new Pair<Integer, Integer>(time, 1);
        carsAtPos = allPos.get(location);
        carsAtPos.put(vid, initStats);
        allPos.put(location, carsAtPos);
      }

      // car has been seen (take into account "send at least once semantics")
      Pair<Integer, Integer> carStatus = allPos.get(location).get(vid);
      int vidTime  = carStatus.getKey();
      int vidCount = carStatus.getValue();

      if (vidTime != time) {  // values are identical if tuple was resent
        carsAtPos = allPos.get(location);
        Pair<Integer, Integer> stats;

        if ((time - vidTime) <= 30) { // reset
          vidCount += 1;
          stats = new Pair<Integer, Integer>(time, vidCount);
        } else {
          stats = new Pair<Integer, Integer>(time, 1);
        }
        carsAtPos.put(vid, stats);
        allPos.put(location, carsAtPos);
      }

      // now we can use the count of the car status to determine whether
      // a car has been stopped, i.e. count >= 4
      carStatus = allPos.get(location).get(vid);

      if (carStatus.getValue() >= 4) {
        // if vid already in stoppedCars, then only update value:
        // otherwise, insert, and emit tuples
        if (stoppedCars.get(vid) == null) {
          boolean stopped = true;
          // emit values for current seg and last three segments;
          // car stopped
          for (int i = 0; i < 4; i++) {
            collector.emit(
              "StoppedCars",
              new Values(time, xway, lane, dir, seg - i, vid, stopped));
          }
        }
        stoppedCars.put(vid, time);
      }

      // determine whether all cars in stoppedCars are still stopped
      // if not, send new tuple to next bolt
      int remove = -1;

      for (int key : stoppedCars.keySet()) {  // key is vid
        int last_time = stoppedCars.get(key);
        if (time - last_time > 90) {  //
        boolean stopped = false;
        remove = key;

        // emit values for current seg and last three segments;
        // car no longer stopped
        for (int i = 0; i < 4; i++) {
          collector.emit(
            "StoppedCars",
            new Values(time, xway, lane, dir, seg - i, key, stopped));
        }

        break;
        }
      }
      /*
        we break out of the for loop to avoid concurrent modification errors;
        this is not problematic due to the very small number of stopped cars,
        the fact that cars stop and restart at different times, and the large
        number of position reports (this check is performed for every position
        report!)
      */
      if (remove != -1) {
        stoppedCars.remove(remove);
      }

    }
  }


  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {

    declarer.declareStream(
      "StoppedCars",
      new Fields("Time", "XWay", "Lane", "Dir", "Seg", "VID", "Stopped"));

  }
```

```
}
```

Accidents.java

```
package storm.starter.lr_bolts;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.util.HashMap;
import java.util.Map;
import java.util.HashSet;


public class Accidents extends BaseBasicBolt {

  // key: unique identifier for location, val: car status
  Map<String, HashSet<Integer>> status =
    new HashMap<String, HashSet<Integer>>();

  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {

    int     xway     = input.getIntegerByField("XWay");
    int     lane     = input.getIntegerByField("Lane");
    int     dir      = input.getIntegerByField("Dir");
    int     seg      = input.getIntegerByField("Seg");
    String  location = xway + "." + lane + "." + dir + "." + seg;


    if (input.getSourceStreamId().equals("StoppedCars")) {

      int     time     = input.getIntegerByField("Time");
      int     vid      = input.getIntegerByField("VID");
      boolean stopped  = input.getBooleanByField("Stopped");


      // update car status
      HashSet<Integer> cars = status.get(location);

      if (stopped) {
        if (cars == null) {
          cars = new HashSet<Integer>();
        }
        cars.add(vid);
      }

      // remove car if it was counted as stopped
      if (!stopped) {
        if (cars != null) {
          cars.remove(vid);
        }
      }

      status.put(location, cars);

    }


    if (input.getSourceStreamId().equals("Requests")) {

      HashSet<Integer> cars = status.get(location);

      long    emitTime = input.getLongByField("EmitTime");
      int     time     = input.getIntegerByField("Time");
      boolean accident = true;

      if (cars == null || cars.size() < 2) {
         accident = false;
      }

      collector.emit(
         "Responses",
         new Values(emitTime, time, location, accident));
    }
  }


  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {

    declarer.declareStream(
      "Responses",
      new Fields("EmitTime", "Time", "Location", "Accident"));
  }

}
```

Logger.java

```java
package storm.starter.lr_bolts;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.File;
import java.io.IOException;


public class Logger extends BaseBasicBolt {

  final long nanoSecMin = 60000000000L;

  double  sum     = 0;
  long    count   = 0;
  boolean written = false;
  boolean flag    = false;

  long    startTime;
  long    intervalFrom;
  long    intervalTo;

  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {

    // set boundaries of measurement interval
    if (!flag) {
      startTime    = System.nanoTime();
      intervalFrom = startTime + nanoSecMin;
      intervalTo   = startTime + nanoSecMin * 6;
      flag         = true;
    }

    long    emitTime = input.getLongByField("EmitTime");
    int     time     = input.getIntegerByField("Time");
    String  location = input.getStringByField("Location");
    boolean accident = input.getBooleanByField("Accident");

    long    timeNow  = System.nanoTime();
    long    latency  = timeNow - emitTime;

    boolean inInterval =
      timeNow >= intervalFrom && timeNow < intervalTo;

    if (inInterval) {
      count++;
      sum += (double) latency / 1000000;  // latency in ms
    }


    if (timeNow >= intervalTo && !written) {

      try {
        BufferedWriter out =
          new BufferedWriter(new FileWriter("results.txt", true));
        double latency_ms = (double) (sum / count);
        out.write("500," + count + "," + latency_ms + "\n");
        out.close();
      }
      catch (IOException e) {}
      finally { written = true;}
    }

  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

}
```

C Listing of Measurements

The columns in the two tables below indicate the maximum number of issued position reports per second, the total number of position reports that were processed in five minutes, and the average internal latency of the executed topology in milliseconds.

C.1 Server

500	148773	2.73
500	148743	2.76
500	148805	2.75
500	148886	2.77
500	148677	2.80
1000	245280	2.78
1000	245251	2.77
1000	242611	2.82
1000	244170	2.78
1500	244680	2.79
1500	244568	2.78
1500	245167	2.77
1500	245740	2.76

Table C.1: Measurements on an AMD Opteron 2374 HE

C.2 Edge Devices

500	7960	75.23
500	8021	74.52
500	8008	74.70
500	7991	74.91
1000	8029	74.53
1000	8040	74.66
1000	7941	75.06
1000	7958	75.16

Table C.2: Measurements on a single Odroid-XU4