A Data Driven Priority Scheduling technique for a Stream Processing Platform

by

Oluwatobi Ajila

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science in
Electrical and Computer Engineering

Ottawa-Carleton Institute or Electrical and Computer Engineering
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada

Abstract

Big data processing has become essential for businesses in recent years as it enables organizations to gather insights from streaming data in near real-time and capitalize on business opportunities. One drawback of stream processing engines is the lack of support for priority scheduling. There are cases where businesses need to ensure that important input data items are processed with low latencies thus avoiding a missed business opportunity. This thesis proposes a technique that enables users to prioritize important input data so that they are processed in time even when the system is under high or bursty input load. Using a prototype this thesis demonstrates the efficacy of the proposed technique. Performance analysis demonstrates that there is a significant latency improvement for high priority data over low priority data especially when there is high system contention.

Acknowledgements

I would like to thank my supervisor, Dr. Shikharesh Majumdar, for his patience and guidance during the process of writing this thesis and also throughout my time in the graduate program. I appreciate the time and effort that he has put into helping me complete my thesis.

I also would like to thank my parents, Samuel and Victoria, for supporting me throughout my academic career. This would not have been possible without their help.

List of Tables

List of Figures

List of Abbreviations

API	Application Program Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDPS	Data Driven Priority Scheduler
DStream	Discretized Stream
EC2	Elastic Compute Cloud
FIFO	First-In, First-Out
GB	Gigabyte
IOT	Internet of Things
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MB	Memory Block
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
TCP	Transmission Control Protocol
XML	Extensible Markup Language

List of Symbols

Ae	Average end-to-end latency for processing a message
Aq	Average job queuing latency for jobs in Spark Streaming job queue
As	Average scanning latency for scanning an RDD
BD	Batch duration
C	Number of logical CPUs
C_a	Coefficient of variation of interarrival time
K	Number of Priority Mappings
L	Length of messages
Lj	Number of jobs in job queue
Lmn	Number of messages in an RDD
Lu	Number of memory blocks in list of unattached memory blocks
msg	Message
PHP	Percentage of high priority messages
PMP	Percentage of medium priority messages
S	Message service time
s	Second
Smb	Size of memory block
SO	Scheduling overhead
Ta	Average latency for processing a job in the Spark Engine
λ	Average message arrival rate

$\Delta 1$	Difference between the average end-to-end latency for low and high priority messages
$\Delta 2$	Difference between the average end-to-end latency for medium and high priority messages
$\Delta 3$	Difference between the average end-to-end latency for low and medium priority messages

Chapter 1: Introduction

In recent years big data has become one of the important factors to consider when designing business applications. Businesses often need to deal with large amounts of data in a timely manner. The characteristics of big data that make its processing challenging are often characterized by 3Vs where the V's are Volume, Velocity and Variety [1]. Volume refers to the amount of data that is processed; for some applications this can range from several Gigabytes to petabytes of data. Velocity is the speed at which data needs to be processed and variety is the diversity of the data. Diversity can result from various data forms such as text and video as well as various data formats such as ASCII, JSON and XML. Real-time stream processing engines such as Apache Spark [2] and Storm [3] are widely used to tackle challenges that concern the velocity and variety characteristics of big data. Some of the major use cases of real-time stream processing engines are sentiment analysis, fraud detection, log monitoring and processing customer behaviour. Platforms like Spark and Storm allow users to write applications that can perform data analytics in near real-time while offering fault tolerance, scalability and guaranteed message processing [4].

1.1 Motivation for the Thesis

Sentiment analysis is the process of gathering and categorizing public opinion in order to determine a feeling or sentiment about a certain issue [5] [6]. It is becoming increasingly common for businesses and government organizations to perform sentiment analysis on message boards or social media to gather actionable intelligence for marketing, security and automation purposes [7]. Some businesses use sentiment analysis to determine what consumers think about a particular product [8]. Sentiment analysis is also used by companies in the financial industry to perform real-time analysis for stock trades. Security agencies use sentiment analysis to analyze public forums for potential illegal or terrorist activity [9]. There may be cases where one may want to assign

priority to certain input data items. For example, suppose newspaper company is analyzing tweets in real-time for a potential news worthy event. A tweet containing the word “fire” would warrant immediate attention. Ideally, one would want to process this tweet immediately to determine if there is an emergency or if the word “fire” is being used in a more benign context. One could also construct a similar example in a security context. Security agencies may be monitoring social media for signs of potential criminal activity. Or, a financial organization may want to take immediate action upon news of a new directive from an elected official. In these cases, it is imperative that the analysis is performed in a timely manner as a delay in analysis could result in a missed opportunity. If incoming data is arriving at a high rate or the input load is very bursty this may delay the processing of high priority data. The ability to assign priorities to data inputs helps to ensure that the most important data is processed first in these situations [10].

Streaming platforms are often the tool of choice for performing data analytics. One of the difficulties with stream processing is responding to changes in input load in an efficient manner. Ideally, one would one would allocate enough resources such that it can handle the input load without causing queuing delays. This requires that the system is able to process the input load as fast as it enters the system. Figure 1.1 demonstrates a batch processing scenario where the Y-axis indicates the amount of time it takes to process a batch of data. The horizontal blue line indicates the time it takes for a new batch of data to be created. A stable system is a system where the current batch will be processed before the new batch is created. This occurs when the black horizontal line is slightly below the blue line.

There are many cases where it is not possible to predict the system load that the system needs to support at any point in time. If the system does not have enough resources to handle the load, queuing delays will occur resulting in large processing delays or a system failure. This is captured

as the red portion of the graph in Figure 1.1. In these cases, one solution is to overprovision the system. Overprovisioning is a technique where the system is allocated enough resources to handle the peak load. This often results in a waste of resources as most of the time only a fraction of the resources will be in use. This is shown as the green portion of the graph in Figure 1.1. Another issue with overprovisioning is it requires that one is able to provision for the worst case scenario, which may not be known.

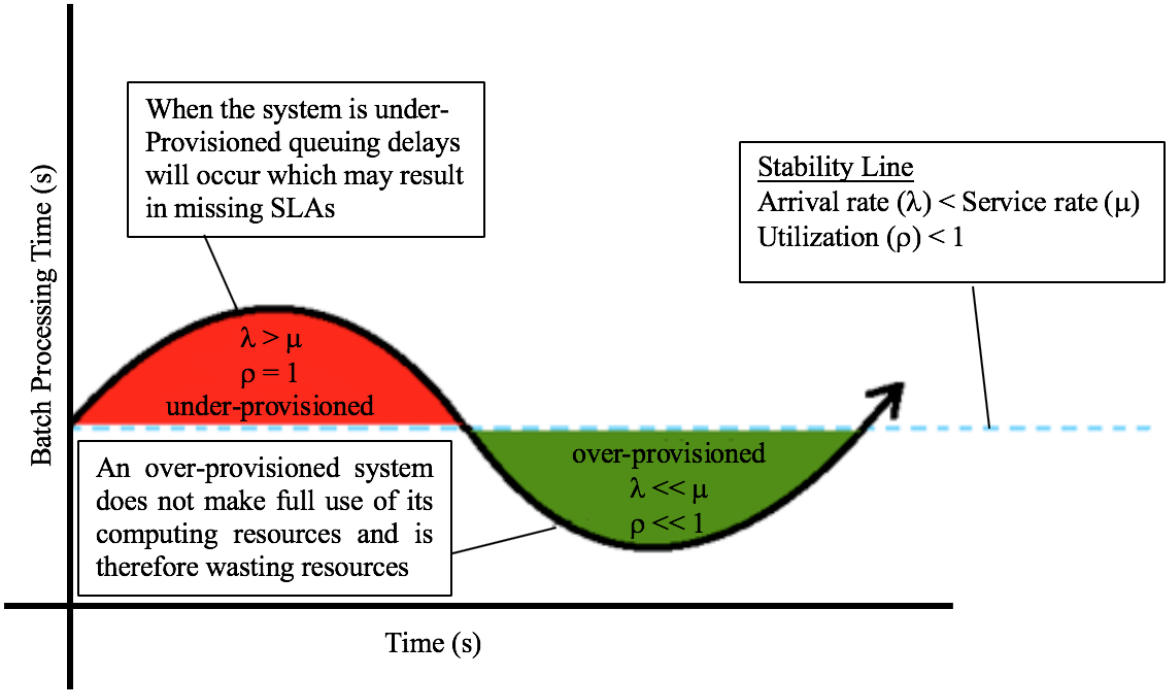

Figure 1.1: Requirements of a Stable System

Another alternative is to dynamically provision resources, where the system adds or removes resources based on the input workload. This is typically done by either vertical or horizontal scaling [11]. Horizontal scaling is a technique where more processing nodes are added to the system. This can be done by adding more machines to a cluster. Vertical scaling is where the processing capabilities of the existing nodes are increased. In a case where one wants to increase the processing resources of a parallel processing application running in a virtualized environment

with vertical scaling, one can increase the number of virtual CPUs that the application has access to. Another example of vertical scaling is increasing the amount of memory available to a memory bound application. The challenges with this approach are that scaling is not instantaneous, it takes time for dynamic provisioning systems to add and remove resources. In cases where the input load is bursty, meaning the interval between arrivals varies dramatically, this may result in processing delays when the load suddenly spikes (see Figure 1.2). The problem is exacerbated if the input load returns to normal just before the system allocates more resources, as the system will be overprovisioned.

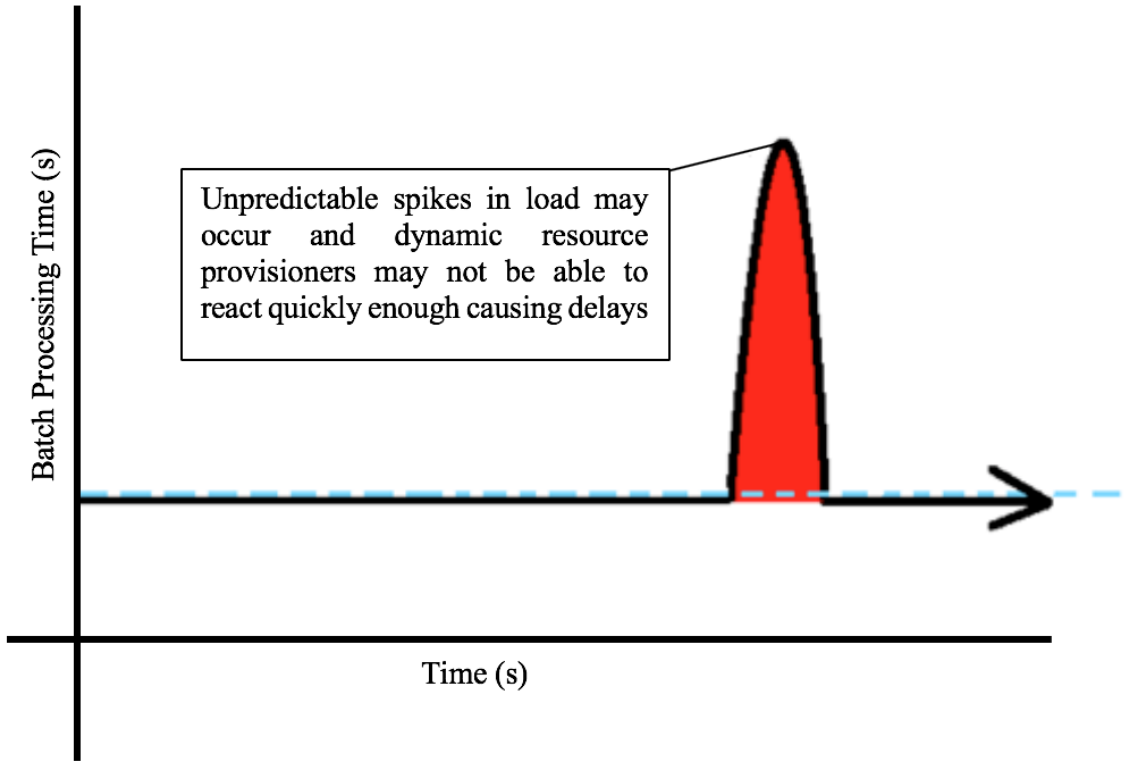

Figure 1.2: Unpredictable Load

1.2 Proposed Solution

This thesis proposes a data driven priority scheduler (DDPS) which helps to provide stability to a data streaming engine under varying loads without requesting more computing resources. This

scheduling technique requires that the user assign priorities to all the different types of input data ahead of time. At runtime the scheduler prioritizes input data with the higher priorities. In cases where the system load increases dramatically, the latency of higher priority jobs remains relatively stable, and the lower priority jobs remain in the queue and as a result incur processing delays. When the input load reduces to a level where all jobs can be processed by the system, both higher and low priority jobs receive a fair share of resources.

This scheduling technique does not require additional resources to function, however, it can be used in conjunction with dynamic resource allocation techniques as well. The data driven priority scheduler can be implemented on any micro-batching streaming engine. Micro-batching engines are streaming engines that divide the stream input data into small batches, after, the data is then processed at a batch granularity. In this thesis, the proposed data driven priority scheduler is implemented on the Spark Streaming framework. A new API is added to the Spark Streaming framework to allow users to assign priorities to input data types as they are writing their application. The Spark Streaming scheduler itself is modified to scan the input data and ensure that high priority data items are given precedence.

1.3 Scope of the Thesis

This thesis focuses on a data driven priority scheduler for handling a single streaming application. This technique requires the user of the given streaming application to specify the priority of the input data types. This lets the scheduler know what input data is most important so that it can prioritize the processing of the high priority data items. Determining the priority mappings automatically from previous history of system operation by using machine learning techniques for example forms an important direction for future research.

The cases identified in the thesis that benefit from this technique are text streaming applications for performing sentiment analysis of twitter feeds, or messages boards for example. In these cases, it is possible to assign priorities to input data that contain certain words, or are sent from a certain sender. The future works section discusses further approaches that can be used to improve the data driven priority scheduler. The DDPS technique does not allow users to update the priorities of input data types at runtime, all priorities must be determined *a priori*. Although this approach is apt for handling situations in which priorities are static, further research is warranted for handling dynamic changes in priority. For example, a priority change may be required for a sentiment analysis application analysing Twitter feeds after the occurrence of an event such as an earthquake or terrorist attack.

The scheduling technique investigated in this thesis focusses on reducing the end-to-end processing latency for high priority messages. Techniques that can handle deadlines associated with messages forms an important direction for future research.

1.4 Contributions of the Thesis

The main contributions of this thesis include:

- A novel data driven priority based scheduling technique for streaming applications that prioritizes high priority data items and can effectively handle bursty loads.
- A proof of concept prototype of the data driven priority scheduler implemented on the Spark Streaming framework.
- Demonstration of the efficacy of the proposed technique using a synthetic workload and a live twitter sentiment analysis scenario.

- Insights into system behavior and performance resulting from a rigorous performance analysis of the prototype that include the impact of various system and workload parameters on performance.

1.5 Thesis Outline

The remainder of this Thesis is organized as follows. Chapter 2 provides background on Apache Spark, Spark Streaming, machine learning and language classification and discusses related work. Chapter 3 introduces DDPS a and the implementation of the DDPS prototype. Chapter 4 presents the performance evaluation of the DDPS prototype and discusses the results. Finally, Chapter 5 concludes the thesis and presents topics for future research directed at enhancing the DDPS technique.

Chapter 2: Background and Related Work

This chapter focuses on the background information for systems and technologies and a literature survey for of the state of the art related to the thesis research. It introduces the Apache Spark and the Spark Streaming systems. It also presents background information on sentiment analysis techniques and machine learning. A literature survey of auto-scaling and scheduling techniques is also included.

2.1 Spark

2.1.1 Apache Spark

Apache Spark is a general-purpose cluster computing platform [2] that is implemented using Scala [12] and runs in a Java Virtual Machine (JVM) environment. Spark provides a functional programming interface that enables users to write algorithms to process data in parallel. Spark is mainly used with big data workloads, much like its predecessor Hadoop. However, Spark differs in that it offers in-memory processing giving it a large performance advantage over Hadoop which writes back to disk after each phase of computation.

Spark represents input data as a Resilient Distributed Dataset (RDD). RDDs are an immutable collections of data elements represented as Scala objects that can be partitioned across a cluster of machines [12]. RDDs are fault tolerant and can be rebuilt if a node storing a partition fails.

Spark enables users to perform parallel operations on RDDs. There are two kinds of operations that can be performed: transformations and actions [13]. Spark defines transformations as operations that produce a new RDD (or multiple RDDs) from an existing one. These transformations include map, which takes a function and applies it to each element of a RDD and generates a new one [14] or join, which combines two RDDs that contain key-value tuples and

produces a new RDD that is a combination of the input RDDs. Spark defines actions as operations that are performed on RDDs which do not generate a new RDD. An example is count, which counts and returns the number of elements in an RDD.

Apache Spark runs with multiple nodes for parallel data processing. The main node is called the driver node, and all others are worker nodes [15]. Figure 2.1 illustrates the relationship between the driver and worker nodes. When running Spark in the cluster mode, each node is a JVM instance which may have multiple threads executing on it. In the local mode all the nodes are in a single JVM instance. The Spark driver program (an instance of SparkContext) is an application that executes on the driver node. An application may have multiple threads. The SparkContext defines a directed acyclic graph (DAG) of RDDs where the operations are edges and the RDDs are nodes [12]. The Spark scheduler splits the graph into stages and attempts to pipeline operations. Tasks are created for each stage, where a task is defined in terms of the data and the computation to be performed on it. The Spark scheduler assigns tasks to executors which run on worker nodes (see Figure 2.1).

2.1.2 Spark Streaming

Spark Streaming is a framework that is built on top of Apache Spark. It offers fault-tolerant near real-time processing of big data. It can process data from input sources such as Kinesis, TCP sockets, Kafka [16] and can output processed data to databases, filesystems and dashboards. Spark streaming allows users to apply transformation on streams. These transformations are similar to the ones that Spark APIs allow on RDDs. An example of these transformations is map which applies a function on every element in a stream.

Spark Streaming internally represents streams as discretized streams (DStreams). A DStream is a continuous stream of RDDs where the elements of each RDD are data items from a

specific time interval. This technique is commonly known as micro batching. The time interval is called the batch interval. The batch interval impacts the latency of processing the data. Figure 2.2 demonstrates an example where an input source sends data items to a Spark Streaming instance. The RDDs contain input data items that were sent within a certain batch interval, t (seen on x-axis of Figure 2.2). The continuous stream of RDDs over time is the DStream.

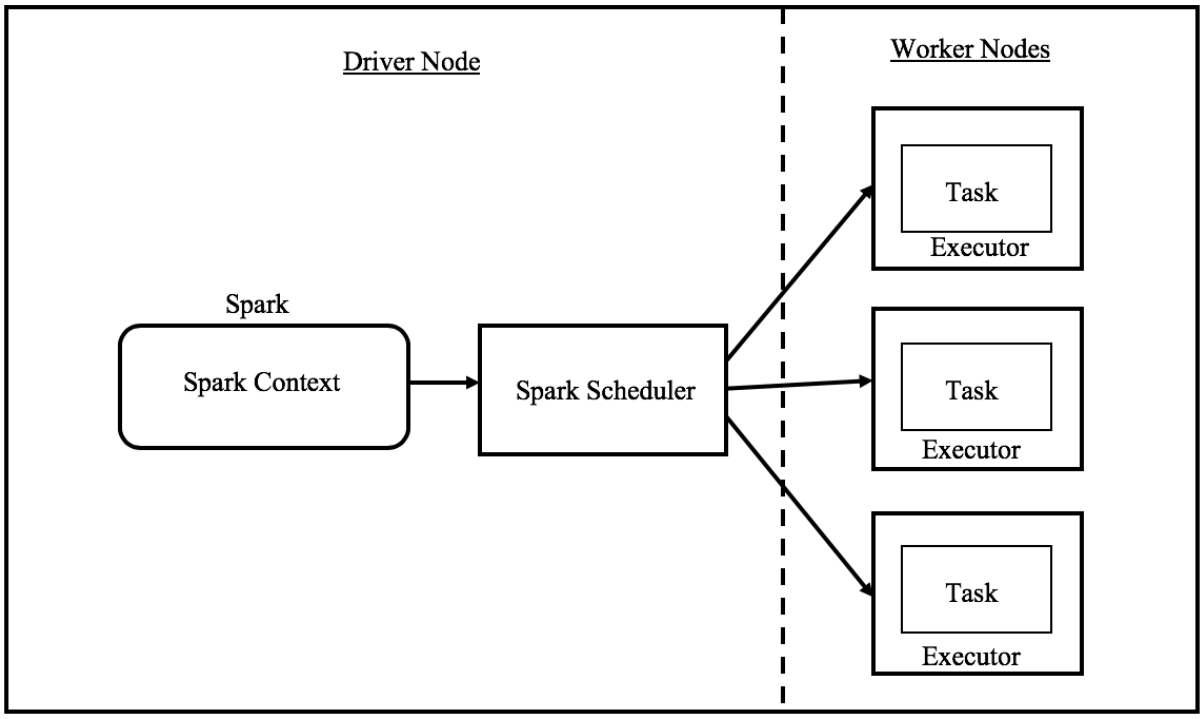

Figure 2.1: Spark system overview

Any transformation performed on the DStream is internally performed on each individual RDD [17]. As a result, DStream transformations are essentially RDD transformations continuously applied over time. The RDDs are eventually processed by the spark computing engine and the result is a series of output batches. A DStream graph is directed acyclic graph where each stream is an edge and transformations are nodes. Figure 2.3 illustrates an example where there are three DStreams. The second and third DStream are produced by the first and second transformation respectively. The output of the preceding DStream is the input of the following DStream in the

next batch interval. For example, the output RDD of DStream1 at time t is the input RDD for DStream2 at time 2t.

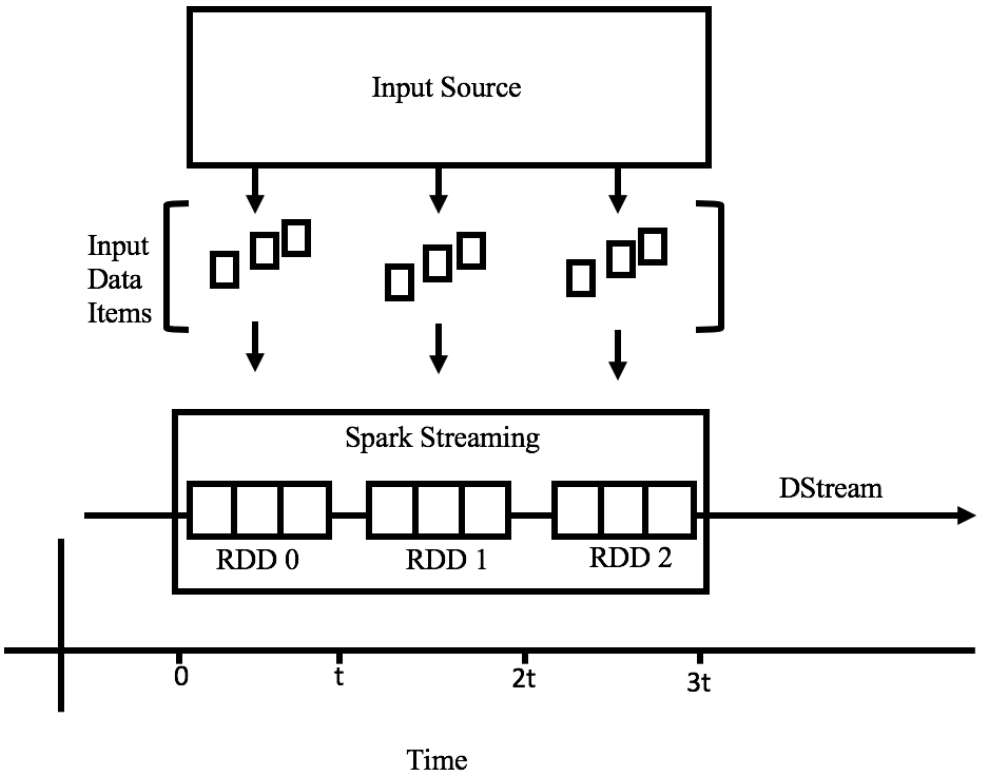

Figure 2.2: DStream

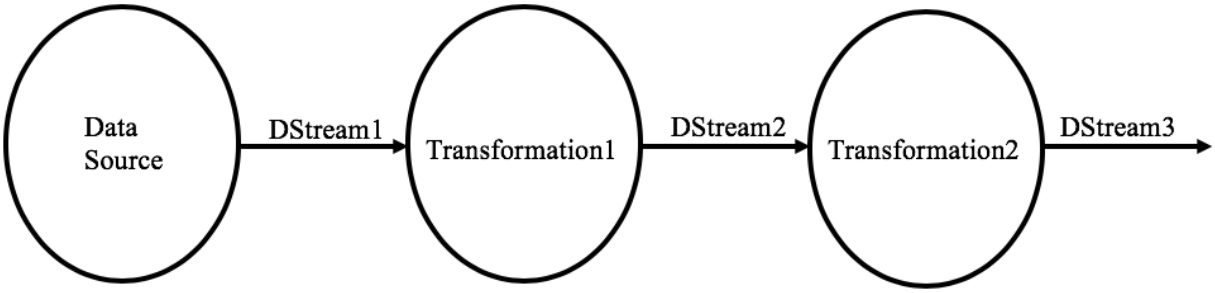

Figure 2.3: Directed Acyclic Graph

The major components that compose Spark Streaming are the receiver, job scheduler, job manager, receiver tracker and the DStream graph. The receiver executes on a worker node and accepts input data items. The receiver tracker keeps track of all the input data items received and

places them into an RDD at the end of each batch interval. For each batch interval the job scheduler creates jobs for each RDD in the DStream graph. A job represents all the input data items (RDD) for a specific time window and the transformations that will be applied to it. After the job is created, it is placed in a job queue. The job queue is a First In First Out (FIFO) queue managed by the job manager. The job manager only removes a single job from the queue at a time and hands it off to the Spark scheduler. The Spark scheduler divides the job into tasks which are then processed on executors. Figure 2.4 shows the high level components of Spark Streaming. Similar to SparkContext for Apache Spark, Spark Streaming has a SparkStreamingContext. This represents an instance of the streaming application. It is the receiver for streaming API calls.

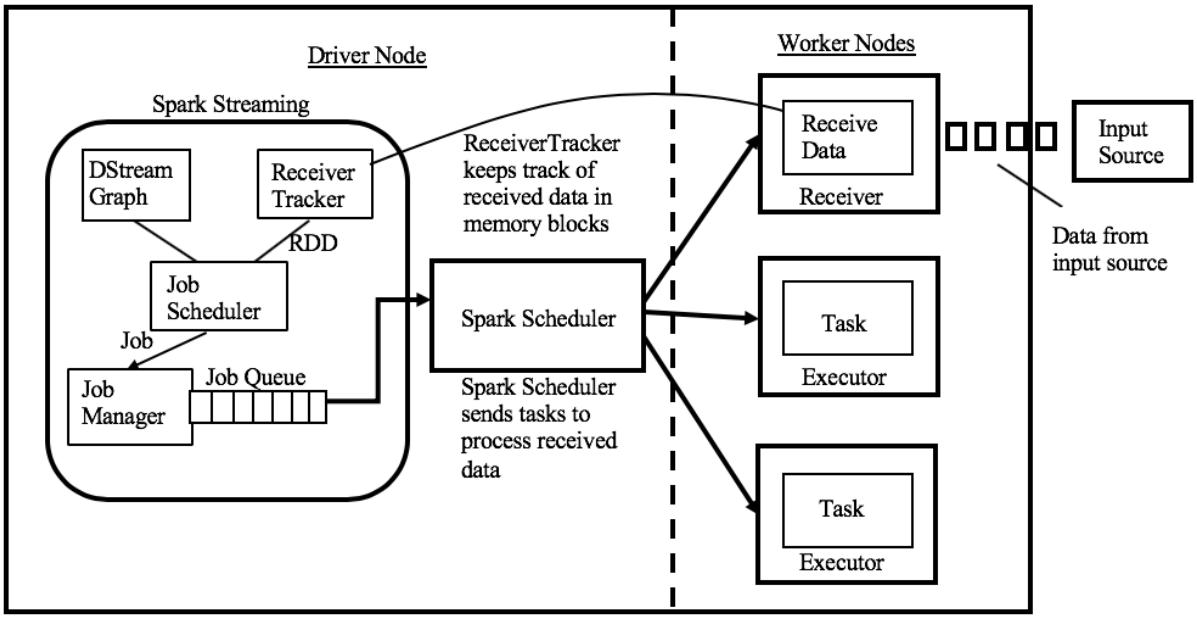

Figure 2.4: Spark Streaming System Overview

2.1.3 Spark Schedulers

Spark Streaming does not support any queueing discipline other than FIFO for processing jobs. However, the underlying Spark framework does support different scheduling modes for scheduling jobs within an application. By default, when an application's threads submit tasks, they

are processed in a first come first served manner. If a thread's tasks are very large, they will use up as many resources as required to process them. This may cause delays for the other application threads that are waiting. If the application does not require all the resources, the next thread in the queue may begin running its tasks. The second mode Spark offers for scheduling within jobs is the fair scheduler. This mode runs jobs in a round-robin fashion so that all the threads in the queue get an equal share of resources regardless of order or size. The fair mode also supports grouping of threads into pools and assigning weights to each pool. Pools make it possible to give a certain thread a higher priority for resources within a system [18].

For scheduling multiple applications (multiple SparkContext instances) on a cluster, Spark's default (standalone) mode provides each application its own JVM and there is no sharing of computing resources between applications. The resources in the cluster are assigned to applications in a FIFO manner. One can also statically partition the clusters resources for each application. Spark also supports third party resource schedulers such as Mesos [19] and YARN [20]. Both of these offer static partitioning. Mesos also supports resources sharing where an application can relinquish its own resource for another application if it doesn't need it. None of the existing schedulers for Spark support dynamic job priority for which the priority of a job can change dynamically based on the data elements it contains. This thesis is directed at addressing this gap in the state of the art by introducing a data driven priority scheduler for Spark-based stream processing systems.

2.2 Computational Intelligence

One of the common uses of streaming technology is in the field of computational intelligence (also known as Artificial Intelligence). This section introduces a subset of computational intelligence that is used to identify the public's emotions or sentiments and act on

it with computational algorithms. This thesis makes use of a sentiment analysis scenario to test the performance of DDPS.

2.2.1 Sentiment Analysis

In recent years sentiment analysis has become an increasingly popular field of study. Its applications range from combating cyber-bullying to election predictions [21]. Sentiment analysis is the process of extracting, identifying and quantifying opinions or emotions of a population [22]. The origin of sentiment analysis dates back to the early 1900's in the form of public opinion analysis. This type of analysis involved creating a series of questions or statements to represent opinions [23]. The questions and statements were then rated by people on a scale. A ranking of opinions was then created using the results from the ratings. More recently in the 1990's computational linguistics was used to conduct subjectivity analysis of fictional narrative texts [21]. This was done by writing algorithms which made use of the syntactic, lexical and morphological elements of the text to determine whether a sentence expresses an opinion, feeling or emotion [24]. In recent times natural language processing techniques, machine learning and lexicon based approaches have been used to extract and analyze sentiment from text [22]. These techniques often involve some sort of sentiment classification and sentiment rating prediction. Sentiment classification involves classifying an opinion or sentiment into distinct categories (e.g. positive, negative). Sentiment rating prediction is the process of determining the intensity of the sentiment or opinion (e.g. very negative, mildly negative) [22].

2.2.2 Machine Learning

Machine learning is a field of study that makes use of computers to simulate the learning capabilities of humans. This is often done to acquire new information, identify existing information or to optimize performance of a system [25]. Machine learning has its roots in computational

learning theory and pattern recognition [26]. In recent years machine learning has become the most commonly used tool for data analytics.

The four main aspects of machine learning are: external environment, learning, repository and execution [25]. The external environment represents the information that is available outside the machine learning system. This could be tweets, traffic data, medical history of patients [26] and so on. Learning is the aspect of processing data from the external environment to obtain knowledge. A repository is a system used to store the knowledge obtained in the learning phase. [25]. Knowledge can be represented in various forms such as semantic networks, production rules and logic statements [25]. The final aspect, execution, is the process of taking the stored knowledge from the repository and using it to complete a task. The results of the execution phase are then fed back into the learning stage.

The common types of machine learning are supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning [27]. Machine learning systems typically use labeled data, unlabeled data or combination of the two as an input to the system [28]. Unlabeled data is any piece of information (natural or artificially created) that is available to the machine learning system. This can be tweets, pictures, seismic data, news articles, etc. Labelled data is an association between unlabeled data and an identifier or classification that describes what the unlabeled data is. Supervised learning is the process of comparing the computed output with the expected output for a given input. The machine learning system records the error between computed and expected output and attempts to minimize this error [27]. Unsupervised learning is the process of learning based on the discovery of input patterns. Reinforcement learning is a type of learning where an incorrect output is penalized and correct output is rewarded. The goal of this approach is to maximize a long term reward. This is different from supervised learning in that the

input and output pairs are never given [27]. Semi-supervised learning is a hybrid of supervised and unsupervised learning where both user input and machine learning discovery are used for predictions [29] [27].

2.3 Automatic Resource Scaling

Automatic resource scaling, also known as auto-scaling, is a technique where a system automatically adjusts the amount of resources an application is using without minimal or no human involvement [30]. As previously stated in Section 1.1, there are two ways for auto-scalers to dynamically allocate and deallocate resources. One is vertical scaling and the other is horizontal scaling. Horizontal scaling involves adjusting the number of nodes that the application is being run on. Increasing the number of nodes is referred to as scaling out, reducing the number of nodes is called scaling in. Vertical scaling typically involves adjusting the CPU or memory capabilities of the existing nodes on which an application is being run on. Increasing the number or computing resources via vertical scaling is called scaling up, reducing the number of resources is called scaling down. Vertical scaling typically involves adjusting the CPU or memory capabilities of the existing nodes that an application is being run on. Typically, this often requires the auto-scaler to reboot the operating system that is hosting the application [30]. For this reason, most cloud providers only offer horizontal auto-scaling services [30].

There are two main techniques for autoscaling, reactive and proactive [30] [31]. Auto-scalers often monitor performance metrics such as CPU utilization, input load, service response time or dropped requests [32]. Reactive auto-scalers will set thresholds for one or more performances metrics to determine when more resources are required or if resources need to be released. The performance metrics are monitored at a specified sampling frequency. Increasing the sampling frequency increases the responsiveness of the auto-scaler, but also increases the overhead

of the auto-scaler [30]. An example of a reactive auto-scaler is presented in [33]. This technique runs an algorithm on every request arrival. The algorithm decides if the system needs to acquire more resources or if the incoming request must be rejected. The goal of the algorithm is to minimize the cost of computing resources while ensuring that the pre-determined grade of service is met. Proactive auto-scalers also monitor performance metrics, however, they use the performance metrics to predict future loads. This lets the auto-scaler pre-emptively allocate or deallocate resources. The analyses required to predict future loads contributes to the overhead of the auto-scaler. An example of this is the proactive automatic resource scheduler presented in [34]. This technique uses machine learning to predict number of resources required to handle the upcoming input load based on previous input load patterns. There are some auto-scalers that combine both proactive and reactive auto-scaling techniques when provisioning computing resources. An example is the work presented in [35] which uses a proactive machine learning approach in conjunction with a reactive scaling algorithm in order scale the machine resources to meet the demands of the input load. It invokes the reactive algorithm after each request arrival and invokes the proactive algorithm after a fixed amount of request arrivals.

2.4 Related Work

The literature suggests that there are two ways to solve the problem of bursty input loads in stream processing environments. These techniques are auto-scaling presented in Section 2.4.1 and scheduling presented in Section 2.4.2.

2.4.1 Literature Survey of Autoscaling Techniques for Stream Processing

Most of the existing studies attempt to mitigate the effects of varying input loads by using auto scaling techniques. Addressing the variance in input loads is important as this could affect the response time of an application. If an application suddenly receives a large burst of input data this

could result in missing SLAs or perhaps running out of memory to store all the received data. An example of an auto-scaler is seen in [36] which implements a dynamic scaling system on the Nephele stream processing engine. In this work, the researchers created a model to estimate the latency of processing data based on the current resource allocations. Using this model they were able to scale in or out the amount of resources required to meet latency constraints. Research presented in [37] also demonstrates an auto scaling technique for real time stream processing engines based on system utilization thresholds and reinforcement learning where past scaling decisions and outcomes are recorded and used to determine future scaling decisions. Similarly, [38] provides an auto scaling mechanism to stream processing engines that makes use of machine learning to classify workloads and predict whether scaling of resources will be required in the near future. The advantage of using the data driven priority scheduler presented in this thesis over auto-scaling techniques is that it functions in situations where it is not possible to acquire more resources.

2.4.2 Literature Survey of Scheduling Techniques for Stream Processing

Another technique to address the effects of bursty input loads is load balancing. Load balancing [39] is the process of taking load from one node that is over-utilised and sharing or moving it entirely to another node which has unused processing capacity [40]. Examples of load balancing techniques are Aurora and Medusa [41]. Both of these run a daemon on each node to monitor its utilization. If the node is operating at maximum capacity it will attempt to either move all computations to other nodes or transfer part of the computations to another node and continue processing. The downside to load balancing is that it requires the system to have extra nodes which are under-utilised or not utilised at all in order to function properly. This thesis proposes a

technique that addresses vary input loads without requiring the system to have under-utilised nodes.

The scheduling techniques discussed earlier rely on having the ability to increase the computational resources of the cluster. This is done either by horizontal scaling where more nodes are added to the cluster or by load balancing where nodes that are not being fully utilized are used to alleviate the pressure on nodes that are over utilized. There may be cases where it is not possible to acquire more nodes or load balance. For example, a company hosting an on premise data center would not be able to easily increase their computing capacity if the input load suddenly increases. Also, if all their nodes are fully utilized, load balancing will not work as there will be no nodes to transfer computations to. Research that attempts to address fluctuating input loads without the use of auto-scaling and load balancing is presented next.

To the best of our knowledge, there are only three other bodies of work that address the effects dynamic of input loads in real-time micro batching engines like Spark Streaming without the use of auto scaling techniques or load balancing. The first is research on adaptive scheduling with Spark Streaming [42]. This research introduced a technique that dynamically modifies the batch duration based on input load. It demonstrates that there is a trade-off between latency and throughput. When running Spark Streaming with a small batch duration the end to end processing latency is smaller, but throughput is reduced. With a large batch duration throughput improves but the latency gets larger. In cases where the input load varies over time, the adaptive stream scheduler attempts to stabilize the system by either increasing the batch duration when the input load increases, or reducing the batch duration when the input load reduces. This technique sacrifices responsiveness at peak loads and minimizes latency when the system is under a reduced load. One advantage the advantage DDPS presented in this thesis research has over adaptive scheduler is that

when the system load increases, only the response time of low priority data items gets worse. However, with the adaptive scheduler, the response time of all data items increases.

The second technique is DSlash proposed by Birke et al. This work [43] highlights the fact that workloads may vary greatly over time and that this may cause issues for time sensitive data analysis. The authors propose a data controller, DSlash, that processes data as fast as it can given an application target latency. It achieves this by delaying data processing and load shedding of data older then a supplied threshold. Load shedding is a technique where input data items are discarded when the system is unable to keep up with the demands [44]. By discarding input data the system is able to recover and improve its responsiveness. The advantage of using the data driven priority scheduler over DSlash is that it doesn't discard input data items to respond to peak loads. Load shedding is a lossy operation which may affect the application's results.

The third approach is the A-Scheduler [45], which is implemented on Spark streaming. It improves performance by dynamically modifying the number of jobs that are processed in parallel. By default, Spark streaming only processes a single job at a time. The A-Scheduler assesses data dependencies between jobs to determine if jobs can be run in parallel. By increasing parallelism, both throughput and latency are improved. The drawback to this approach is that the A-scheduler only improves performance when the system is underutilized. Increasing parallelism requires more resources; if there are no more resources on the system, there will be no improvement. The benefit of using the technique presented in this thesis over the A-scheduler is that it minimizes the latency of high priority data items even when the system is running at maximum capacity.

The main drawback with the three techniques [42] [43] [44] is that they do not allow one to assign priorities to input data items. The ability to assign priority is essential if there is important information that must be analyzed in a timely manner. Load shedding [43] is not an option because

it can cause important input data items to be discarded. Adaptive scheduling techniques [42] can improve the efficiency of the streaming engine which improves throughput but sacrifices responsiveness which can be costly if a high priority data item arrives.

To the best of our knowledge there is only one other body of work that demonstrates priority scheduling in a real-time micro batching engine like Spark Streaming. In this research [46] a priority based resource scheduling technique for real-time distributed streaming processing systems is introduced. The technique enables users to assign priorities to certain nodes in the stream processing graph. Input data is assigned a priority based on the nodes that it is destined to reach. The scheduler re-orders the input data such that the high priority data items are processed first. The difference between the technique proposed in [46] and DDPS presented in this thesis is that DDPS assigns priorities to input data types rather than nodes in the DAG. Thus our proposed technique lets users differentiate between input data items that will reach the same nodes in a DAG by assigning high priority keywords to input data items, whereas, [46] doesn't provide any mechanism to distinguish data items assigned to the same nodes.

There has been similar research done in the context of complex event processing. This work [47] presents a deadline aware scheduling technique where a deadline is assigned to an event stream. The researchers propose a pre-emptive time-sliced scheduling technique that can prioritize event streams based on priority. This work [47] differs from the research described in this thesis in that it [47] assigns a priority to the entire stream whereas this thesis allows users to assign priorities to individual data items within the same stream.

2.5 Summary

This chapter presents the Apache Spark and Spark Streaming systems. It shows that Sparking Streaming is a micro-batching stream processing engine that associates batches with jobs

and executes jobs on the Apache Spark engine in a FIFO manner. This chapter introduces sentiment analysis and machine learning and describes some of its uses in data analytics. The motivations for priority based scheduling is discussed and a survey of priority based scheduling and techniques to address fluctuating input loads in stream processing environments is presented. None of the papers presented have addressed the problem of bursty input loads where input data items are assigned priorities that this thesis focuses on.

Chapter 3: The Data Driven Priority Scheduling Technique

This chapter presents an overview of the Data Driven Priority Scheduling technique. It also introduces a priority mapping strategy and discusses the key algorithms for the DDPS technique with an analysis of the algorithmic complexity. Lastly, it describes the implementation of DDPS on the Spark Streaming platform.

3.1 Overview of the Data Driven Priority Scheduling Technique

In the default Spark Streaming system, the system receives a stream of input data items and places the data items in an RDD. Each RDD is associated with a job and jobs are placed in a job queue. Jobs in the job queue are processed in a First-In-First-Out manner. A FIFO queue is a queue where the item that entered the queue first is removed from the queue first. Figure 3.1 illustrates an example where an input data stream sends input data items to the streaming system. Spark Streaming places the input data item into a RDD. The RDD is associated with a job and the job is placed in the job queue. Note that with a FIFO approach, if there is an important input data item in Job N + 2, the data item cannot be processed until Job N and Job N + 1 have been processed.

The goal of the DDPS technique is to give precedence to higher priority input data items. This is accomplished in two phases. The first is classifying the priority of input data items. This is done by the user before starting the streaming application. In this thesis it is done by the user before starting the streaming application. The second phase is performing priority scheduling. Priority scheduling is done in two steps. The first is step is to scan the input data items to determine the priority of the data items. Once the priority of the data items is known it is used to determine the priority of the job. The second step is to reorganize the job queue such that the highest priority jobs are processed first and lowest priority jobs are processed last. Figure 3.2 shows a scenario where an input data item in Job N + 2 is marked as high priority. With the default Spark Streaming

system the high priority data item would only be processed after the two preceding jobs. But with DDPS, the job containing the high priority input data item is moved to the head of the queue and is processed before the rest of the jobs in the queue.

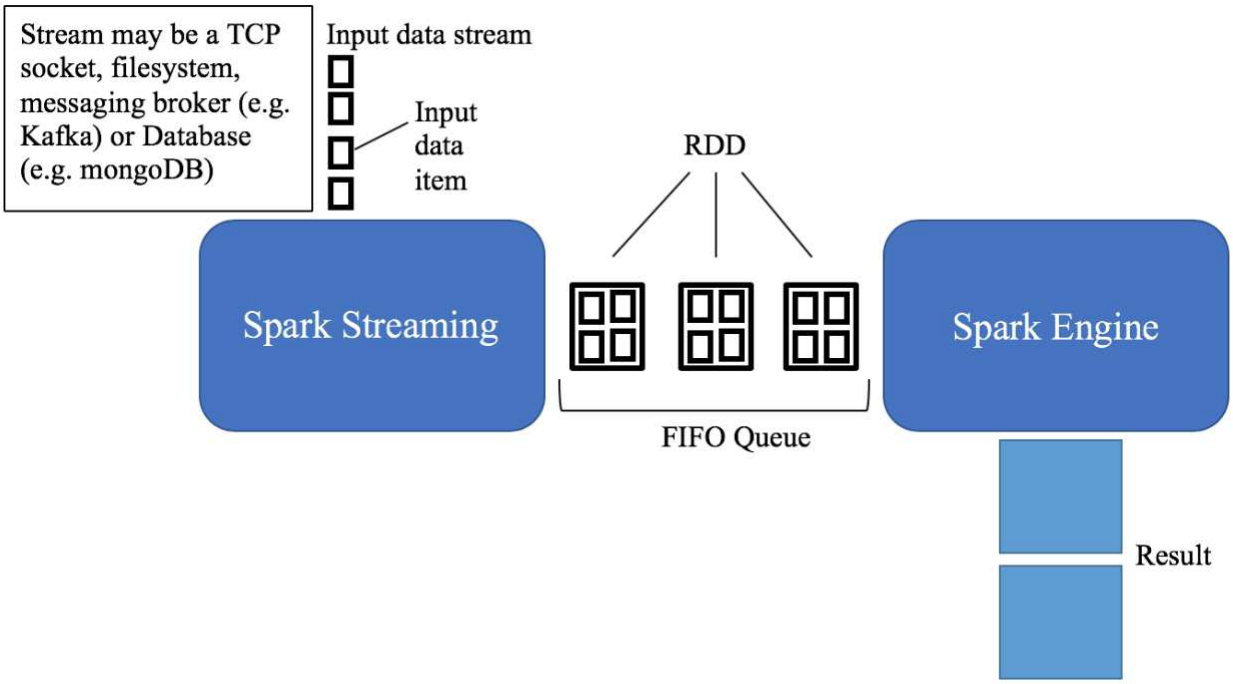

Figure 3.1: FIFO Queue used in Spark Streaming

3.2 Priority Mappings

As input data items enter the streaming system, they are scanned to determine their priority. In order for DDPS to know the priority of an input data item, the user must provide a list of priority mappings before the system is started. A priority mapping is a general technique that is used to assign a priority to an input data item. A priority mapping is a key value pair (i.e. {key, value}) where the key is a feature or characteristic of a data item. For example, if data items were composed of strings, a key could be a word or a phrase. Or, if the input data items were composed of a JSON objects the key could be a specific value for a property in the JSON object. The value in the priority

mapping is an integer that represents the priority. The larger the value of the integer the higher the priority and vice versa. A user may supply multiple priority mappings (i.e. a list of priority mappings {{key1, value1} … {keyN, valueN}}).

Input data types can be composed of ASCII messages, JSON objects, files, tweets, GPS coordinates, database entries, etc. A stream will only be composed of one type of input data item. This means that a stream of JSON objects will only contain JSON objects. There cannot be a mixture of multiple types of input data items in the same stream. For this thesis, the focus is on input data items that are ASCII messages.

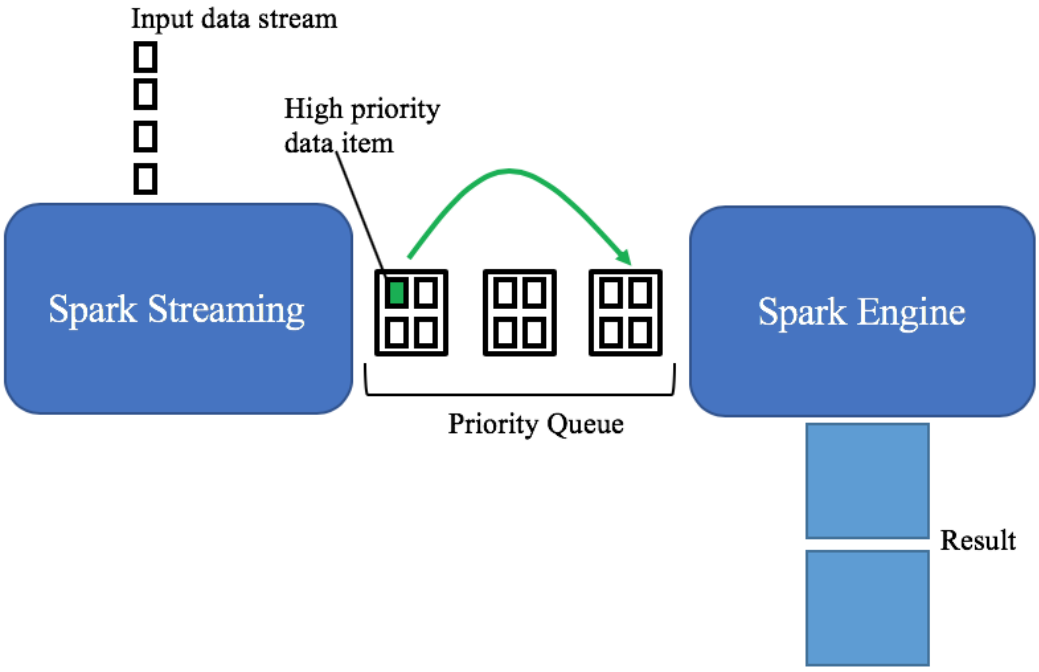

Figure 3.2: DDPS Priority Queue Used in Spark Streaming

In a case where an input data stream contains ASCII sentences, a user could provide the following list of priority mappings for example {{“fire”, 10}, {“explosion”, 100}}. With the supplied list of priority mappings, the words “fire” and “explosion” are the keys and the integers

10 and 100 are the values. Therefore, words “fire” and “explosion” are assigned to a priority of 10 and 100 respectively. This means that the word “explosion” has a higher priority than the word “fire”.

The priority of a job is the priority of all the input data items contained in its RDD summed together. For example, if the system were to contain a RDD with the following input data items:

Input Data Item1 - “I don’t like the weather today.”

Input Data Item2 - “There is a fire downstairs”

Input Data Item3 - “I heard an explosion!”

The priority of the job associated with the RDD would be 10 (Input Data Item2) + 100 (Input Data Item3) = 110. Only the words with mappings contribute to the priority of the Job. This means that words without mappings in effect have a priority of zero. Since Input Data Item 1 does not contain any words in the list of priority mappings it does not contribute to the priority of the job.

In the following example there are two jobs with RDDs that contain the following input data items:

Job 1:

RDD: Input Data Item1 - “Fire, fire, fire, fire, fire, fire, fire, fire, fire, fire, fire!!”

Job 2:

RDD: Input Data Item1 - “I heard an explosion!”

In this case Job 1 has a priority of 110 because the word “fire” appears 11 times in the in its associated RDD. Job 2 has a priority of 100 because the word “explosion appears only once in its RDD. As a result, Job 1 will be treated with higher precedence than Job 2.

There is no restriction on what the value in a key-value pair is allowed to be. Also, multiple keys may be mapped to the same value. This means that the words “loud” and “boisterous” can be assigned the same priority. In addition, there is no limit on the size of the list of priority mappings that can be supplied to the system.

The previous discussion has focussed on textual data streams. The RDD scanning algorithm can detect any keyword specified by the user and has been used effectively in the context of sentiment analysis on live tweets described in Section 4.2. Other examples include scanning data containing text based transaction logs of bank accounts to detect fraudulent activity [48] and live scanning of sales transactions for merchants. Handling a stream characterized by a sequence of structures where each structure corresponds to the current reading of a sensor value in a sensor based system will require changing the data scanning algorithm. The user will need to define specific values for a designated field within the structure that will determine the priority of the given input data item.

3.3 Data Driven Priority Scheduler Algorithm

The priority scheduler proposed in this thesis assigns a priority to input data items, and allows the system to give precedence to higher priority data items over lower priority data items. This is performed in two phases. The first is the classification stage and second is the scheduling stage. A classification of data items by the user is done ahead of time before the streaming application begins, as shown in Figure 3.3. In this stage the user assigns priorities for input data types by creating priority mappings. The priority mappings are then supplied to the streaming engine. The streaming engine will use the priority mappings to determine which input data items it will give precedence to.

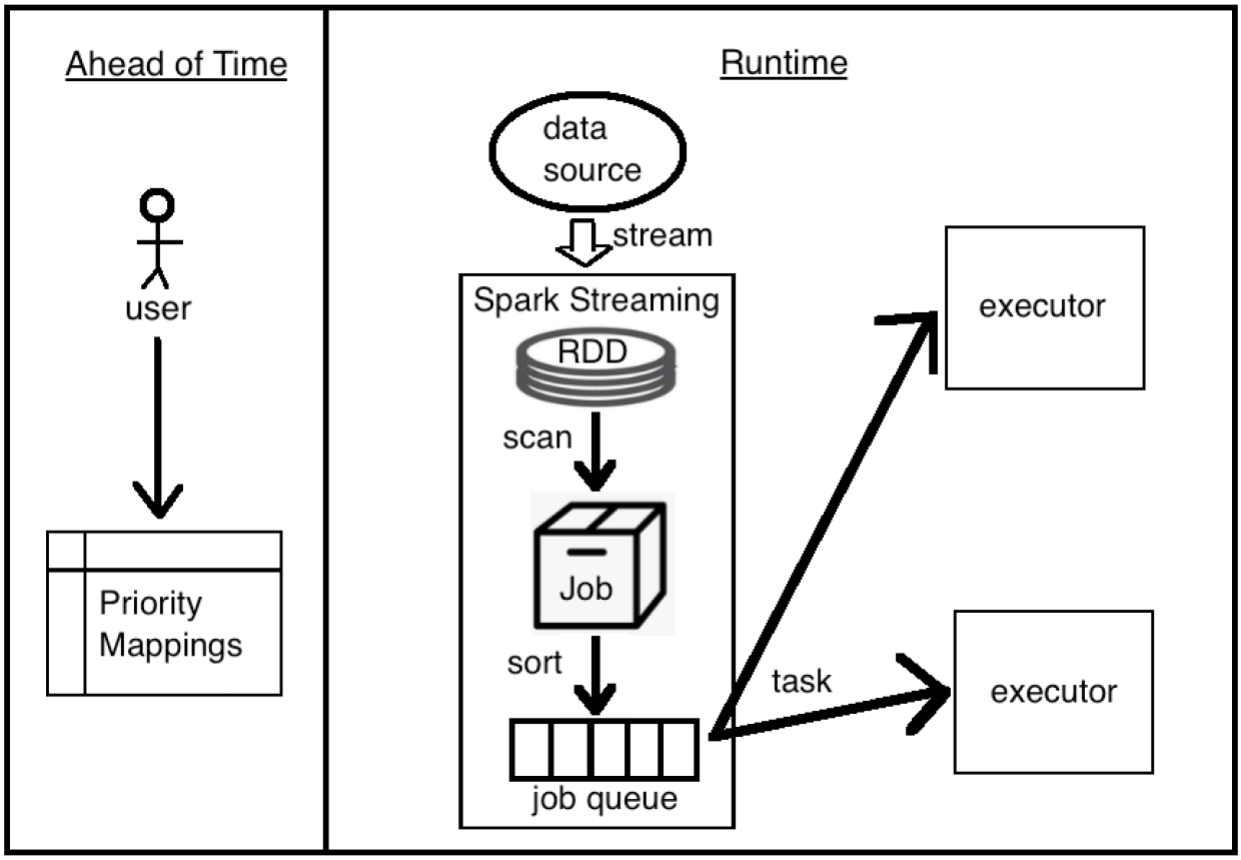

Figure 3.3: Data Driven Priority Scheduler Overview

The scheduler is invoked on every batch interval and schedules the jobs that process the input data items in the current batch. The scheduling stage is performed at runtime as input data items are being placed in the RDD. The data items are scanned for determining their priority and the number of high priority elements in the RDD are added up to determine the priority of the job.

A short description of the scheduling technique is provided next. Algorithm 1 concerns the receiving of messages and their storage into memory blocks (MB). Algorithm 2 assigns the memory blocks to a new RDD. Algorithm 3 is responsible for scanning RDDs, creating Jobs and assigning job priorities. Algorithm 4 is responsible for ordering the jobs in the job queue. Algorithm 5 is responsible for recycling memory blocks that are no longer in use. Algorithm 1, 2, and 5 are required for both DDPS and the default FIFO scheduler. This is because algorithm 1, 2

and 5 are required for any job scheduling policy as they are responsible allocating memory blocks for input data items, creating RDDs and freeing the unused memory blocks. Algorithms 3 and 4 are specific to DDPS, therefore, the only overheads that DDPS introduces are Algorithm 3 and 4. Minor changes are made to Algorithm 5 in order to support DDPS. More details on this are provided in Section 3.4.

Algorithm 1: The input parameter to Algorithm 1 is the unattached blocks list. This is a list of memory blocks that have not yet been assigned to an RDD. First, the receiver reads in input data from a socket connection (line 3). The read is a blocking call and will only complete once there is data to be read. The current memory block is the memory block that the receiver will add the received input data to. If the current memory block is full or has been removed from the list of unattached memory blocks, the receiver will allocate a new memory block (line 5) and add it to the list of unattached memory blocks. The newly allocated memory block becomes the current memory block. Algorithm 1 runs continuously on the receiver for the lifetime of the application. Algorithm 1 is run concurrently with Algorithm 2, 3, 4 and 5.

Algorithm 2: The input parameters to algorithm 2 are the unattached blocks list and the RDD list. The RDD list is a list of all the RDDs that have been created. Algorithm 2 is executed by the receiver tracker. At the beginning of each batch interval the receiver tracker creates a RDD and assigns it an ID that includes the time at which the RDD was created. The RDD is also added to the list of RDDs (line 4). Next, all the memory blocks from the unattached blocks lists are added to the newly created RDD (line 4 – 8). Once this is completed the newly created RDD is returned.

Algorithm 3: The input parameters to algorithm 3 are a new RDD, the Job map and the list of priority mappings. The RDD is the newly created RDD from Algorithm 2. The job map is a key-value map where the key is an ID and the value is a job. The receiver tracker scans the data

items in the RDD to see if they contain any items listed in the priority mappings list (lines 2 – 8). This process calculates the cumulative priority of all the data items in the RDD, and the result is used to determine the priority of the job. Once this is complete, the RDD is assigned to a new job which keeps track of the RDD and all the operations that will be applied to the RDD (line 16). The job will be assigned an ID that is the same as the RDD's ID. This ID represents the age of the job and it is used as the key when entering the job in the job map. Next, the job is handed to the job manager (line 18).

Algorithm 1: Algorithm for receiving data items

```
// Algorithm is run by receiver
// Input:
// UBList list of unattached memory blocks.
1:   Begin
// loop forever
2:   while(TRUE)
// receiving data is a blocking call
3:       data = receiveData()
4:       if (isFull(MBcurrent))
// the new MB will be added to the UBList
5:           MBcurrent = allocateNewMB(UBList)
6:       endif
//add the received data to the memory block
8:       MBcurrent.addData(data)
9:   endwhile
10:  End
```

Algorithm 4: The input parameters to Algorithm 4 are the job queue and a newly created job. Algorithm 4 is executed by the job manager. The job queue is a queue consisting of jobs where the jobs are ordered first by their priority and secondly by their age. If two jobs have the same priority the older job is placed ahead in the queue.

After the receiver tracker executes Algorithm 3, it hands the job to the job manager. The job manager checks and ensures that the queue is sorted in order of job priority: higher the priority of a job, higher is its position in the queue. This is done by comparing the priority of the incoming

job with that of the highest priority job in the priority queue (line 3). If the incoming job has a higher priority it is inserted at the head of the queue (line 4). If it has the same priority as the priority of the job at the head of the queue and is older (created before) it is placed at the head of the queue (line 7-8). Otherwise, the incoming job's priority is compared with that of the next job in the priority job queue and so on.

Algorithm 2: Algorithm for creating RDDs

```
// Algorithm is run by receiver tracker
// Input:
// UBList unattached blocks list.
// RDDList a list of all the created RDDs
// Output:
// a new RDD
1:  Begin
2:      create RDD
3:      add RDD to RDDList
// take all memory blocks in the UBList
// and add them to the RDD
4:      for each MB in UBList
5:          assign MBi to RDD
6:          remove MBi from UBList
7:      endfor
8:      send RDD to algorithm 3
9:  End
```

A Spark Streaming application may have multiple receivers that accept data in parallel. This creates a race condition where two threads are submitting jobs to the job manager at the same time. It is possible that an older job may be submitted later because its thread was de-scheduled by the Operating System. For this reason, the job manager must check the age of the job by looking at the time at which each job was created.

After the currently executing job has completed, the Job manager takes the job at the head of the priority queue and sends it to the Spark engine which divides the job into tasks to be processed on executors. Note that each message is processed by a separate task. The order in which messages are processed on executors is determined by the Spark engine and is not modified by the

DDPS technique proposed in this thesis. Once the job has been processed, it is removed from the Job map.

If no priority mappings are supplied, the DDPS technique will behave as a FIFO scheduling policy for processing all the jobs. This is equivalent to disabling DDPS.

Algorithm 3: Algorithm for creating jobs

```
// Algorithm is run by receiver tracker
// Input:
// RDD a new RDD.
// HPList a list of priority mappings
// JOBMap a key-value map
// Output:
// a new Job
1:  Begin
// scan RDD for priority mappings
2:      for each Keyword in HPList
3:        for each Di in RDD
4:          if Di contains Keyword
// if high priority keyword is found increase the
// priority of the Job based on the priority value
// associated with the keyword
5:              increasePriority (jobPriority)
6:          endif
7:        endfor
8:    endfor
9 :    create Job with jobPriority
10:   assign RDD to Job
// use job ID as the key when adding job to map
11:   add Job to JobMap
12:   send Job to JobManager
13: End
```

Algorithm 5: The input parameters to algorithm 5 are the RDD and the Job map. The Algorithm 5 is run after a job has been processed. The algorithm iterates through all the RDDs in the RDD list and checks if the RDD is older than the remember duration (line 3). The remember duration is the duration for which the streaming engine must keep the input data. This must been done because Spark Streaming offers operations that can operate on input data that was processed by previous jobs. As a result, the remember duration is required for any scheduling policy used by

Spark Streaming. As a default, the remember duration is the batch duration multiplied by 2, however, this duration can be increased if users wish to operate on data from a larger time window. To determine if the RDD is older than the remember duration, the RDDs ID is compared with the current time. If the RDD's age exceeds the remember duration, the Job map is then consulted to determine if the job associated with the RDD has been processed or is still waiting the priority job queue (line 4). The Job map is queried using the RDD's ID, if no entry is found then the RDD is removed from the RDD list and all memory blocks associated with the RDD are freed (line 5). If an entry is found the RDD is not removed and the next RDD in the RDD list is searched.

Algorithm 4: Algorithm for sorting jobs in the job queue

```
// The following algorithm is run each time the
// job manager receives a job
// Input:
// Incoming job Jin
// job queue Q = {J1 … Jn}
1:  Begin
// loop through all the jobs in the priority queue
2:  for each Ji in Q
// check if the incoming job has a higher priority
// than the current job in the queue
3:     if priority(Jin) > priority(Ji)
4:        insert Jin in front  of Ji in Q
5:        break forloop
// handling jobs with equal priorities
// use age of a job to break the tie
6:     else if priority(Jin) == priority(Ji)
7:        if age(Jin) > age(Ji)
8:           insert Jin in front  of Ji in Q
9:           break forloop
10:       endif
11:    endif
// Check the next job in the queue
12: endfor
13: End
```

Algorithm 5: Algorithm for freeing unused memory blocks

```
// The following algorithm is run each time
// a job has completed processing
// Input:
// RDDList a list of all the created RDDs
// JOBMap a key-value map
1:  Begin
// loop through all the RDDs in the RDD list
2:  for each RDDi in RDDList
// check if the RDD is older than
// the remember length
3:     if age(RDDi) > (REMEMBER_DURATION)
// search the Job map to see if the job
// associated with the RDD has been processed
4:        if (JOBMap.get(RDDi.ID) == empty)
// if the job is not in the Job map free
// all memory blocks associated with the RDD
5:              freeAllMemoryBlocks(RDD)
6:        endif
7:     endif
// Check the next RDD in the RDD list
8: endfor
9: End
```

3.4 DDPS Implementation

This section discusses the implementation of the prototype Data Driven Priority Scheduler running on Spark Streaming. It discusses the five algorithms that are used by DDPS and focuses on the implementation details. It also highlights the key changes made to Spark Streaming and some of the APIs that were used in the DDPS implementation. The proof of concept prototype system was built using Spark Streaming Version 2.1.2 running on top of Apache Spark 2.1.2. DDPS was implemented in Scala.

3.4.1 Enhancements to the Spark Streaming API

The Spark Streaming API [49] was extended to support DDPS on Spark Streaming. A new method “addPriorityMappings” was added to the API the Spark Streaming API which enables users to provide priority mappings to DDPS scheduler. The new method takes in a java.util.List of

Tuples and saves it in an internal buffer. Since java.util.List is an interface, users can supply any of the implementing classes such as java.util.ArrayList, java.util.LinkedList or java.util.Vector [50]. The type parameter for the list is a Tuple2. This type is a tuple type that contains two elements. In this case, the elements must be a string and an integer. The string will be the priority keyword, and the integer is the priority value.

3.4.2 Implementing DDPS on Spark Streaming

Implementation of Algorithm 1

Section 3.3 outlines the five key algorithms that comprise DDPS. Algorithm 1 is responsible for receiving incoming data and placing it into memory blocks. When the streaming engine is started a socket connection is established to receive input data items, by launching a receiver thread that receives data items from the socket connection as shown in Algorithm1. The receiver will continue to receive data from the socket connection until the streaming engine is shutdown. Algorithm 1 is required for any scheduler that runs on Spark Streaming.

Implementation of Algorithm 2

Algorithm 2 is responsible for taking the data in memory blocks and placing it into RDDs. This process is started when main thread launches a job scheduler instance. The job scheduler is responsible triggering the creation of jobs. It accomplishes this by starting a timer that is set to generate a job creation event on every batch interval. The job creation event is asynchronous, so after the job scheduler sends the job creation event, it does not wait for a response but sleeps for the duration specified by the batch duration and repeats this process. The job creation event is sent to the ReceiverTracker which then creates the RDD. This is done by first collecting all the memory blocks in the unattached memory blocks list and assigning it to the RDD as shown in lines 4 – 7 of Algorithm 2. The RDD is also given an ID that contains a time stamp which is identical to the

time at which the job scheduler sent the job creation event. Algorithm 2 is required for any scheduler that runs on Spark Streaming.

Implementation of Algorithm 3

Algorithm 3 is responsible for scanning RDDs to determine if they contain any priority keywords. The scanning stage occurs immediately after the RRD is created. The following describes the implementation: First, an accumulator used to track the priority of the RDD is set to zero. The next step is to iterate over all the mappings in the internally stored list of priority mappings. In each iteration a "foreach" [51] operation is done on the RDD to search each element in the RDD for the priority keyword, this is shown in line 4 of Algorithm 3. A foreach operation applies a function to all the elements in a collection [51]. The elements of the RDD will be instances of "java.lang.String". The search is done by looking for the first occurrence of the priority keyword in the RDD element. To do this, the "java.lang.String.indexOf(String str, int fromIndex)" [52] utility function is used. This "indexOf" function takes in a string and an index as the parameters. It searches for the first occurrence of the string starting at the given index. If the string is found it returns a positive value indicating the index at which the string was found. If the string is not found it returns "-1". When scanning the element, the "indexOf" function is given the priority keyword as a parameter and "0" as the offset. If "-1" is returned this means the element does not contain a priority keyword and the next element in the RDD is searched. If a non-negative number is returned this means the priority keyword was found. The priority value associated with the priority keyword is then added to an accumulator for the RDD (line 5 of Algorithm 3). The "indexOf" function is then called again on the same RDD element except this time the index of the previous call to the function is used as a parameter. This is done in case the same priority keyword shows up multiple times in the same RDD element.

In Spark Streaming jobs are represented as a Job class. The Job class contains a callsite field which corresponds to the operations that will be applied to the RDD associated with the job. The Job class also contains a start and end time field which are used to record when the SparkEngine began and finished processing the job. The association between the Job and the RDD is done through the Job "id" field. The "id" for the job and RDD will correspond to each other (line 10 of Algorithm 3). The job also has a time field which is a time that corresponds to the ID. The time field can be used to perform comparisons with other time instances. The receiver tracker then sends the job to the Job manager. The job manager adds the job to a job map where the key is the time attribute of the job and the value is the job. Algorithm 3 is implemented specifically to support DDPS. Other schedulers on Spark Streaming do not require this.

Implementation of Algorithm 4

Algorithm 4 is responsible for ordering jobs within the job queue. In order to accomplish this, the Job class was modified to implement the "java.lang.Comaparable" [53] interface as shown in Figure 3.4. The Comparable interface declares one abstract function "int compareTo(Object o)". This instance function compares the input parameter "o" with the receiver of the method call. If they are the same the method returns zero. If the parameter is greater the result is negative, if the parameter is smaller the result is positive.

A new method "compareTo" is added to the Job class to implement the Comparable interface. This method compares two Job instances and returns the result. Consider for example Job A that represents the receiver of the call and Job B that represents the parameter for the "compareTo" method call (ie. A.compareTo(B)). If the priority of Job A and Job B are the same, then the result of subtracting the time field of Job B from the time field of Job A is returned. Recall that the time field of a job indicates the time at which the job was created. If Job B is older a

negative number is returned and if Job A is older a positive number is returned. This technique works because the time field is the Unix Epoch time [54]. This is the time in milliseconds from the first of January, 1970. This means that the larger the value of the time, the more recent it is. The smaller the value of the time, the older it is.

If the priorities of the Jobs are different, the result of subtracting the priority of Job A from the priority of Job B is returned. If Job A has a higher priority the result is positive, if Job B has a lower priority the result is negative. The method "compareTo" is shown in lines 3 – 10 of Algorithm 4.

The Job manager was modified to replace the existing job queue with a "java.util.concurrent.PriorityBlockingQueue" [55]. This queue only accepts elements that implement "Comparable". The queue orders the elements according to the ordering determine by the "compareTo" method. The two key functions are "put" and "take" which place and remove an element from the queue.

Once the receiver tracker creates a job it sends it to the Job manager. After, the job manager places the job into the queue. The PriorityBlockingQueue ensures that all the elements are ordered with the highest priority jobs first and the lowest priority jobs last as shown in Algorithm 4. With the implementation of "compareTo", if no priority mappings are supplied to the streaming engine, the queue behaves as a FIFO queue because if all the priorities are the same the older jobs are given precedence. The job manager removes the job at the head of the queue and hands it to the Spark Engine to be processed. After the job has been processed, the Job manager sends an event to the receiver tracker to free unused memory blocks then it hands the next job in the queue to the Spark Engine. Algorithm 4 is required to support DDPS on Spark Streaming. Other schedulers on Spark Streaming do not require this.

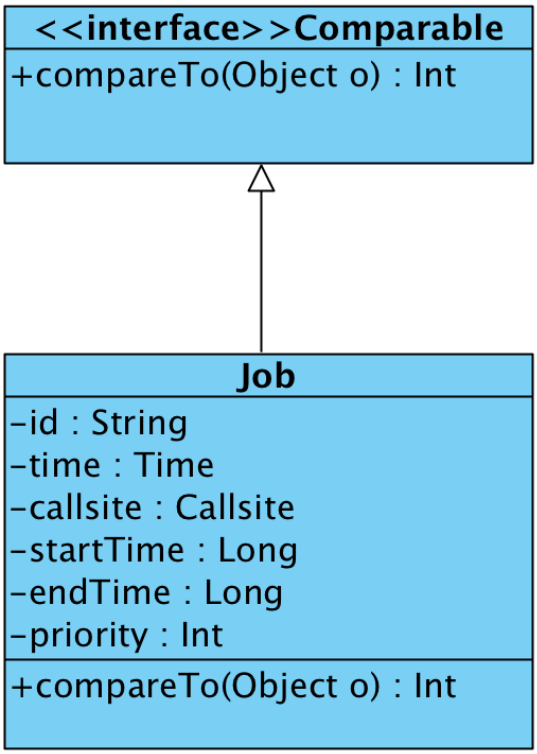

Figure 3.4: Job Class Diagram

Implementation of Algorithm 5

Algorithm 5 is responsible for recycling memory blocks after the job has been processed. This process begins when the receiver tracker receives the event to free memory blocks, it searches through the list of RDDs and filters out the RDDs that are younger than the "rememberDuration" by doing a "filter(test : ()=> Boolan) " operation on the RDD list (line 3 of Algorithm 5). The filter operation takes a lambda function as a parameter. The lambda function is the test that will be applied to each element in the list. In this case the test will simply check to see if the age of the RDD is higher than the current time minus the "rememberDuration". If that is the case, True is returned, False is returned otherwise. The "filter" operation iterates through all the elements and tests each one to see whether the element passes the condition specified by the filter operation, if

it does the element is added to a list. Once all the elements have been tested the list is returned. The receiver tracker then checks to see if the any of the RDDs in the list is associated to any jobs that are in the job map (line 4 of Algorithm 5). This is done by taking the ID of the RDD and converting it to a time, and the using the time as a key to query the job map. If querying the job map doesn't provide any results, it means that all the memory blocks that belong to the RDD can be freed. This process continues for the rest of the RDDs. Querying the job map (line 4 of Algorithm 5) is the additional step required to support DDPS. This step is necessary because DDPS reorders the jobs in the job queue which means that a job waiting to be processed can be older than the "rememberDuration". The remainder of Algorithm 5 is required for any scheduler that runs on Spark Streaming

A sequence diagram of the DDPS implementation is shown in Figure 3.5. On the left is the Receiver which is run in a continuous loop to populate the unattached memory blocks. To the right of the receiver is the Job scheduler which has a timer to trigger job events. The receiver tracker creates RDDs and a Job which is then passed and to the Job manager. The Job manager orders all the jobs based on the priority in a job queue. The job at the head of the queue is sent to the Spark Engine which processes the job. When the job has been processed the unused memory blocks are freed by the Receiver tracker.

3.5 Summary

This chapter describes the Data Driven Priority Scheduler and how it differs from the default FIFO scheduler on Spark Streaming. It also introduces a priority mapping technique which is used by DDPS to determine the priority of input data items. This chapter presents the 5 key algorithms required by DDPS and provides an analysis of the runtime performance of these algorithms in terms of the workload and system parameters. The next chapter will focus the

performance assessment of DDPS and compare its performance to that of the default FIFO scheduler used by the Spark Streaming framework.

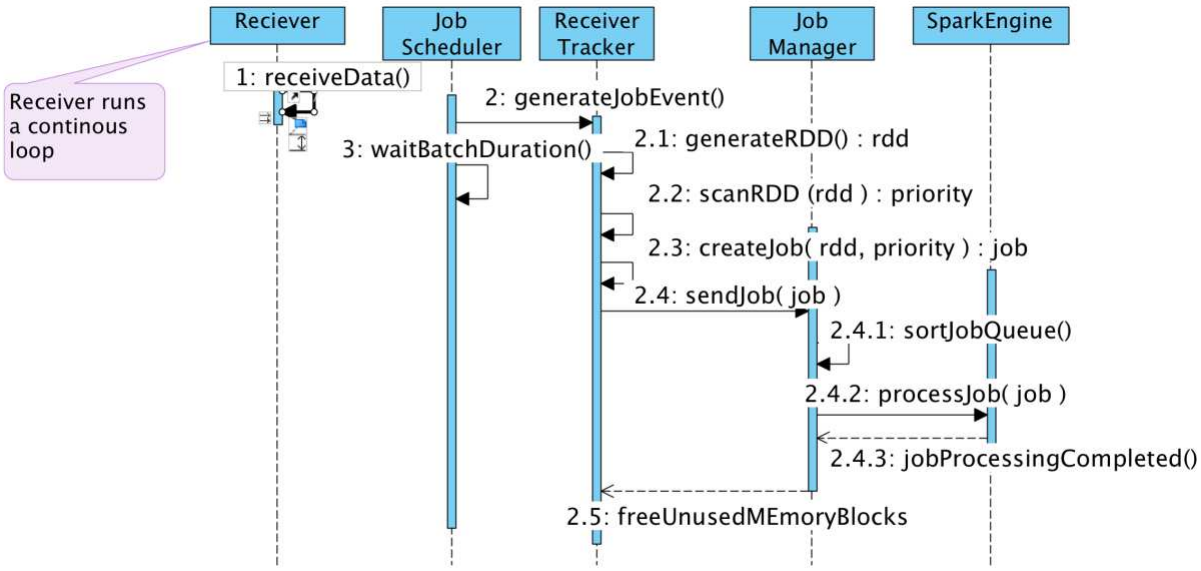

Figure 3.5: Sequence Diagram for DDPS

Chapter 4: Performance Evaluation of DDPS

In this chapter the performance of DDPS is evaluated using a prototype and measurement approach under two scenarios. The first scenario constructs a synthetic workload and measures the performance of DDPS when workload and system parameters are varied. The second scenario performs real-time sentiment analysis on live tweets and records the performance of DDPS. In both scenarios the performance of DDPS is analyzed revealing key insights on the behaviour of DDPS. Section 4.1 describes the synthetic workload scenario. Section 4.2 describes the sentiment analysis scenario.

4.1 Synthetic Work Load Scenario

In this scenario a series of experiments are conducted to evaluate the performance of DDPS with a synthetic workload. The experiments consist of launching a message sending application and a Spark Streaming application, no other applications are run. The message sending application connects to the Spark Streaming application and sends messages to the streaming application. Upon receiving the messages the streaming application performs some computations to model the consumption of CPU cycles by a program. In these experiments the streaming engine is run with both the default the FIFO scheduler and DDPS. In each experiment only one workload or system parameter (Section 4.1.1and Section 4.1.2) is varied at a time. Key performance metrics (in Section 4.1.3) are measured and the results (Section 4.1.5) are presented. Section 4.1.4 describes the implementation of the message sending application and the Spark Streaming application in more detail.

4.1.1 Workload Parameters

The parameters of the synthetic workload used in the evaluation of the proposed scheduling technique are described next.

Mean Arrival rate (λ) is the rate at which the streaming application receives messages measured in messages per second.

Coefficient of variation of interval time (C_a) captures the variability in message inter-arrival times. The greater C_a is the greater the variation in inter-arrival times.

Batch duration (BD) is the duration of the time interval in which the streaming engine collects messages for a given batch measured in seconds.

Number of priority mappings (K): is the number of priority mappings registered with the streaming engine. For the experiments there are three priority levels: high, medium and low. For N priority levels only N-1 priority mappings need to be supplied because any message that does not contain a priority keyword is treated as a low priority message. Each time the streaming engine scans messages it searches all of the priority mappings specified by the user.

Message Length (L) (characters) is length of the message sent by the message sending process measured in characters. All messages sent by the message sending application are the same length.

*Message Service T*ime (S) is the amount of time in seconds a CPU takes to process a message. Message service times are fixed and do not vary from message to message.

Percentage of high priority messages (PHP) is the proportion of messages (measured as a %) that contain at least one high priority keyword. In the synthetic workload scenario a high priority message will only contain a single high priority keyword and no medium priority keywords.

Percentage of medium priority messages (PMP) is the proportion of messages (measured as a %) that contain at least one medium priority keyword and no high priority keywords. In the synthetic workload scenario a medium priority message will only contain a single medium priority keyword and no high priority keywords.

4.1.2 System Parameters

Number of Cores (C) is the number of cores (logical processing units) available to the Spark Streaming application.

4.1.3 Performance Metrics

The primary performance metrics used to evaluate system performance are discussed in this section.

Computation of latency: Processing a message contains the following sequential steps.

- The message is received by the receiver and placed in a memory block. At the end of the batch interval the message is placed in a RDD. The batch delay is the time from when the message is received by the receiver to the time at which the current batch interval ends. On average this will be half of the batch duration. A further discussion on this is presented in Section 4.1.5.2. Let the batch delay for the ith message be given by t^i_{Bdelay}
- The messages in the RDD are scanned by the receiver tracker to detect priority keywords. Let the time required to scan the ith message be given t^i_{scan}.
- The RDD is assigned to a job that is placed in the priority job queue. The job remains in the queue until it is dispatched by the job manager. Let the time the job for the ith message waits in the job queue be given by t^i_{queue}.
- Once the job is dispatched it is processed by the Spark engine on executors. Let the time taken to process the ith message in a job be given by $t^i_{process}$.

Since all the previous steps are performed in sequence the end-to-end processing latency for the ith message is given by:

$$t^i_{latency} = t^i_{Bdelay} + t^i_{scan} + t^i_{queue} + t^i_{process} \quad (4.1)$$

Average End-to-End Latency (Ae) (ms): The end-to-end latency for a message is the difference between the time at which the message processing is completed and the time at which the message is sent by the message sending application. The average end to end latency is the average of the end-to-end latencies of all the messages processed during an experiment.

The average end-to-end latency in a given experiment is computed as:

$$\mathrm{Ae} = \sum_{i=0}^{N} \frac{t_{latency}^{i}}{N} \tag{4.2}$$

Where N is the total number of messages that arrive during the experiment.

Latency is recorded and grouped by message priority level: (ie. high priority (HP), medium priority (MP), low priority (LP), and LP, MP and HP combined).

Average job queuing latency (Aq) (ms): The duration the i_{th} job waits in the job queue is given by $t_{jobqueue}^{i}$. The average job queuing latency in a given experiment is computed as:

$$\mathrm{Aq} = \sum_{i=0}^{N} \frac{t_{jobqueue}^{i}}{J} \tag{4.3}$$

Where J is the total number of jobs that are created during the experiment. Each message has a priority level associated with it. For these experiments there are only three priority levels for messages (high, medium and low). The priority of a job is calculated by summing the priority of all the messages that it contains. Therefore, there may be a very large number of different priority levels for jobs. For this reason, the average job queuing latency is not grouped in terms of priority levels, instead, the average job queuing latency for all the jobs is presented.

Average scanning latency (As) (ms): The scanning latency for all the messages in an RDD is given by:

$$t_{rddscan} = \sum_{i=0}^{M} \frac{t_{scan}^{i}}{M} \tag{4.4}$$

Where M is the number of messages in the RDD. The average scanning latency is given by:

$$\mathrm{As} = \sum_{i=0}^{R} \frac{t_{rddscan}^{i}}{R} \tag{4.5}$$

Where R is the total number of RDDs created during the experiment and $t^{i}_{rddscan}$ is the time taken to scan the ith RDD.

Computation of overhead: Processing a message contains the following scheduling overheads:

- The message is received by the receiver and placed in a memory block. The memory block must be allocated before messages are stored. When all the messages contained in a memory block are processed the memory block is freed. Let the time spent allocating and freeing all memory blocks for the duration of the experiment be given by t_{mb}^{total}.
- On every batch interval, memory blocks are associated with an RDD. Let the time spent associating memory blocks with RDDs for the duration of the experiment be given by $t_{mbToRdd}^{total}$.
- On every batch interval, the messages in the RDD are scanned by the receiver tracker to detect high priority keywords. Let the time taken to scan all the messages in the RDDs for the duration of the experiment be given by $t^{total}_{rddscan}$, where $t^{total}_{rddscan}$is computed as:

$$t^{total}_{rddscan} = \sum_{i=0}^{R} t^{i}_{rddscan} \quad (4.6)$$

- On every batch interval, a newly created RDD is assigned to a job that is placed in the priority job queue. Let the time taken to assign all RDDs to jobs for the duration of the experiment be given by $t_{rddToJob}^{total}$.
- On every batch interval, the job manager places a new job in the queue. The job manager then orders the job queue based on job priority. Let the time the job manager spends ordering the job queue for the duration of the experiment be given by $t^{total}_{orderqueue}$.

Let t_{so}^{total} denote the total scheduling overhead that Spark Streaming incurs. t_{so}^{total} is computed as:

$$t_{so}^{total} = t_{mb}^{total} + t_{mbToRdd}^{total} + \mathrm{t}_{\mathrm{rddscan}}^{total} + t_{rddToJob}^{total} + \mathrm{t}_{\mathrm{orderqueue}}^{total} \quad (4.7)$$

Scheduling Overhead (SO) (%): The total number of cores used by Spark Streaming during the experiments in denoted by C (as stated in 4.1.1). Let the processing time for ith core be given by $\mathrm{t}_{\mathrm{core}}^{\mathrm{i}}$. Let $\mathrm{t}_{\mathrm{core}}^{\mathrm{total}}$ denote the total CPU time consumed for the duration of the experiment. $\mathrm{t}_{\mathrm{core}}^{\mathrm{total}}$ is computed as:

$$\mathrm{t}_{\mathrm{core}}^{\mathrm{total}} = \sum_{\mathrm{i}=0}^{\mathrm{C}} \mathrm{t}_{\mathrm{core}}^{\mathrm{i}} \quad (4.8)$$

The scheduling overhead percentage is the proportion of time spent on scheduling overheads for the duration of the experiment. The scheduling overhead percentage is computed as:

$$\mathrm{SO} = \frac{t_{so}^{total}}{\mathrm{t}_{\mathrm{core}}^{\mathrm{total}}} * 100 \quad (4.9)$$

4.1.4 Implementation of the Message Sending Application and the Synthetic Spark Streaming Application

Each experiment begins by first launching the synthetic message sending application written in Java that starts a server socket and blocks until it receives data. When a connection is established the application sends ASCII messages via the socket connection. The messages can be appended with a special keyword to indicate the priority of the message. For these experiments there are three special keywords. One for low, medium and high priority messages. The message sending application can be modified to vary the percentage of low, medium and high priority messages. This is done by setting a range for each priority level where the range represents the proportion of messages that will contain that priority. For example, if the goal is to ensure that 99% of the messages are low priority, 0.9% of the messages are medium priority and 0.1% are high priority, the ranges would be: less than 1, for high priority, between 1 and 9 for medium priority, and between 10 to 1000 for low priority. The message sending application uses the Math.random() [56] function to generate a random number. This number is multiplied by a 1000

and used to determine which priority is selected for the message. The proportion of low, medium and high priority messages vary depending on the experiment. The message proportions used can be found in Table 4.1.

After each message is sent, the message ending application sleeps for a certain amount of time before sending the next message. The time that the application sleeps for is determined by a by a random number generator. The random number generator generates values uses an exponential distribution. The generator takes two parameters, a mean arrival rate and a coefficient of variation for the mean inter-arrival time. The generator returns an inter-arrival time based on the input parameters. Using the generator one can specify a message arrival rate. For example, if the goal is to send an average of 18 messages per second with second and if the Poisson arrival process is used, one would pass in 18 as the mean arrival rate and 1 as the coefficient of variation. The generator would generate inter-arrival times which would be used as the duration that the message sending application would sleep before sending the next message. After a message is sent, the message sending application generates a new number and repeats the process.

Next the Spark Streaming application is launched. The streaming application defines a set of operations that can be characterized as a DAG shown below in Figure 4.1. The first node of the DAG is the SocketTextStream connection to the message sending application. All the received messages are passed to a MapToPair transformation through an InputDStream of strings. The ‘MapToPair’ transformation first performs a series of mathematical operations. This is done to simulate a service time for processing messages. The computations are performed in a loop and the loop iterations can be increased or decreased to increase or decrease the service time. Other researchers [57] [58] have constructed artificial workloads in a similar manner to assess the performance of a streaming system. Before the ‘MapToPair’ transformation completes it creates a

key-value tuple. This is represented as '<Priority, Int>' in Figure 4.1, where Priority is the priority of the message (high, medium or low) and Int is an integer value. The tuples generated from the 'MapToPair' transformation will contain the priority of the message and an integer value of '1'. An example of this is shown in Figure 4.1 between the 'MapToPair' and 'ReduceByKey' vertices. The result of the 'MapToPair' transformation is a PairDStream (stream of tuples) which is passed to the 'ReduceByKey' transformation. The 'ReduceByKey' transformation receives the tuples and adds together all the values that correspond to the same priority. For example, as sequence of the following tuples: {Low, 1}, {Low, 1}, {High, 1}, {Low, 1} would result in {Low, 3}, {High, 1}. The 'ReduceByKey' transformation produces PairDStream containing the results.

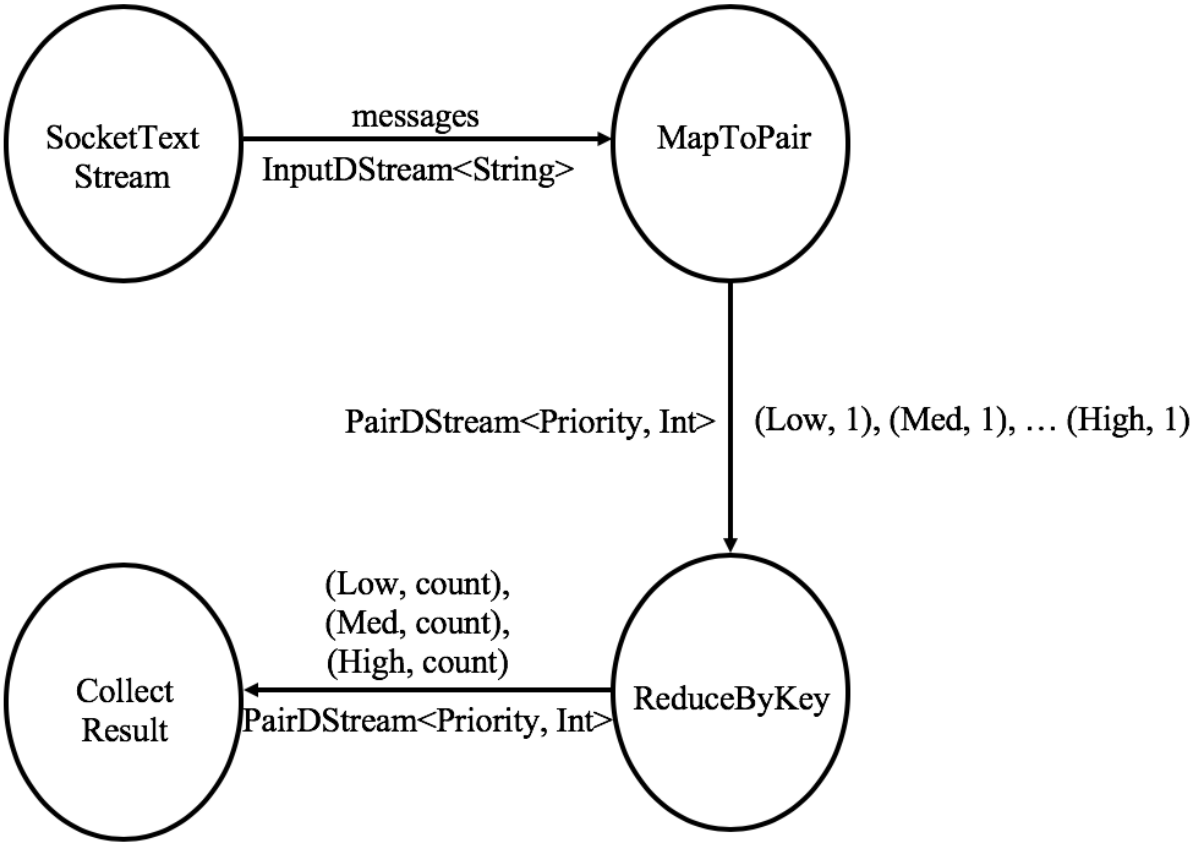

Figure 4.1: DAG for Spark Streaming Application

4.1.5 Experimental Results

The experiments were conducted on a 4core (8 logical CPUs) Intel i7-3720QM (3.6GHz) machine with 16GB of RAM and on an Amazon cluster consisting of four m4.2xlarge [59] nodes where each node is rated as having 8 vCPUs (2.4GHz) and 32GB of RAM. Running the experiments on a single machine and on the AWS cloud yielded similar trends in the performance metrics of interest. For this reason the experimental results on a single machine is presented in the main text and results for running on the cloud are presented in Appendix A: The only exception is the experiment which varies the number of cores. This experiment is only run on the cloud and is presented in the main text. Table 4.1 and Table 4.2 shows the workload and system parameters used in the experiments presented in the main text. The experiments follow a factor at a time method meaning that only a single workload or system parameter is varied during an experiment while the other parameters are held at their default values. The range of the workload and system parameters is shown in the 'Possible Values' column. This is the range that is used when varying workload and system parameters for the experiments. The range is chosen such that the results will demonstrate some meaningful change in the performance metrics. For example, the range of the mean arrival rate is chosen such that the knee in the average end-to-end latency curve is shown. The default values are the values of the workload parameters used when that parameter is not being varied. The default values for batch duration and message service time fall within the range of default values used by other researchers [60] [58]. These values tend to lie somewhere near the middle of the possible values range. The experiments are run for 2 hours and each run of the experiment is repeated such that a confidence interval less than or equal to $\pm$ 3% is achieved at a confidence level of 95%

Table 4.1: Workload Parameters

Workload Parameters	Possible values	Default value
Mean Arrival Rate (**λ**)	18.0 … 20.0	19.0
Coefficient of Variation of Inter-arrival Time (C_a)	1 … 4	1
Percentage of High Priority Messages (PHP)	0.001 …10	0.1
Percentage of Medium Priority Messages (PMP)	0.1 … 5	0.5 when running with 3 priority levels, 0 otherwise
Batch duration (BD)	1.2 … 3.5	2
Number of Priority Mappings (K)	1 … 12	1 when running with two priority levels. 2 when running with 3 priority levels
Message Length (L)	2288 … 3028	2860
Message Service Time (S) (ms)	128 … 145	139

Table 4.2: System Parameters

System Parameters	Possible values	Default value
Number of Cores (**C**)	14 … 26 on cloud, 4 on single machine	22 on cloud, 4 on single machine

4.1.5.1 Effect of Mean Arrival Rate

In these experiments messages are sent using a random number generator to generate the inter-arrival times. In all experiments except the one that investigates the impact of C_a on performance, arrival of messages is modeled by a Poisson process (C_a = 1). These experiments varies mean arrival rate of messages and measures the performance impact on various performance metrics. Increasing the λ increases the number of messages that the streaming system receives per unit time. To the best of our knowledge there is no other priority based data driven scheduling algorithm for Spark Streaming. We have thus compared the effect of the mean arrival rate on average end-to-end latency of DDPS with that of the well-known First In First Out (FIFO) scheduler. In Spark

Streaming when the priority scheduler is turned off, the system uses the default scheduler which is a FIFO scheduler in which the jobs are executed based on their arrival time: a job with a lower arrival time is executed prior to the jobs that arrive later. Two experiments are run. The first is run with two priority levels: high and low. The second experiment is run with three priority levels: high, medium and low.

Figure 4.2 shows four lines: HP and LP correspond to the system running the data driven priority scheduler proposed in this thesis while HP-FIFO and LP-FIFO correspond to a system where the priority scheduler is turned off. Figure 4.2 shows that as λ increases the average end-to-end latency for HP and LP messages with both the DDPS and FIFO scheduler (LP, HP, LP-FIFO and HP-FIFO) increase. Figure 4.2 shows that for any given λ, LP messages are processed with a higher average end-to-end latency than the HP messages when running with DDPS. This demonstrates the utility of the data driven priority scheduler. The performance difference is observed to increase with an increase in λ. When running the FIFO scheduler, the system performance achieved with low (LP-FIFO) and high (HP-FIFO) priority messages are comparable to one another. Figure 4.2 shows the benefit of using DDPS; the average end-to-end latency of HP messages with the priority scheduler is significantly lower than that achieved with the HP messages when the FIFO scheduler is used. However, this comes at a cost to LP messages. Figure 4.2 shows that the average end-to-end latency of LP messages when running with DDPS is higher than both LP-FIFO and HP-FIFO. This occurs as a result of DDPS favouring HP messages at the expense of LP messages. When jobs containing HP messages are prioritized by DDPS, jobs without HP messages spend more time in the job queue and as a result the average end-to-end latency of LP jobs is higher in comparison to LP jobs with the FIFO scheduler.

A similar relationship between average end-to-end latency and λ was captured on a cloud system and the results are presented in Figure A.1 of Appendix A.2.

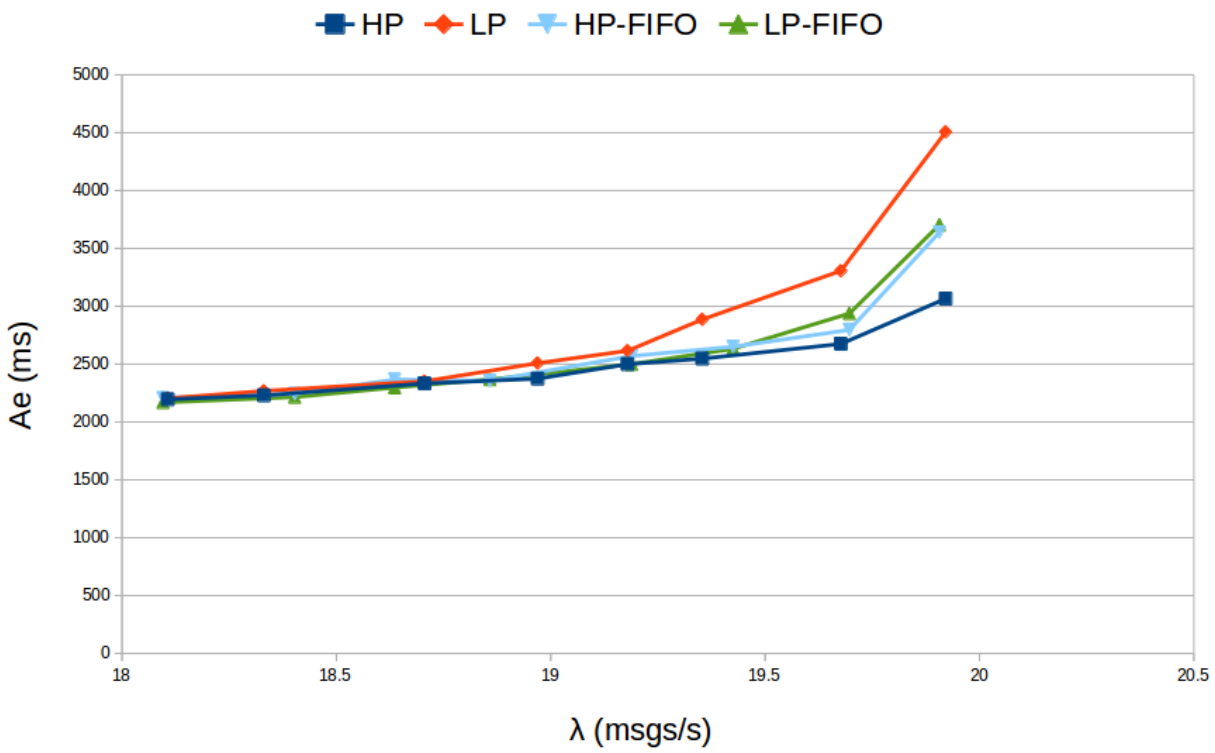

Figure 4.2: Effect of Mean Arrival Rate on Average End-to-end Latency

Figure 4.3 shows the effect of λ on the average job queuing latency when running with the DDPS and FIFO schedulers. Figure 4.3 shows that the effect of λ on average job queuing latency is very similar for both the DDPS and FIFO schedulers. As λ increases so will the average number of messages in each RDD. This means that it will take longer for jobs to be processed resulting in increased job queuing latencies. DDPS prioritizes jobs with HP messages, so these jobs will spend less time in the job queue. Conversely, jobs without HP messages will spend more time in the job queue. The results show that when the average of all jobs (jobs with and without HP messages) is considered, the measured average job queuing latency with DDPS is the same as the FIFO scheduler. This implies that on average, the reduction of the average queuing time of jobs

containing HP messages is accompanied by a corresponding in average queuing time of jobs that do not contain HP messages.

The results in Figure 4.3 also show that as λ increases the average job queuing latency also increases. The rate of change remains fairly constant until about the point where λ reaches 19.5 at which the rate of change increases dramatically. This also appears to be the same point at which the average latencies of HP and LP messages begin to differ in Figure 4.2. DDPS prioritizes HP messages by reordering jobs in the job queue based on priority. This means that the more jobs there are in the job queue the greater the effect of DDPS. The less jobs are in the queue, the smaller the effect of DDPS. As the average job queuing latency increases, so will the average number of jobs in the job queue. This explains why the difference between the average end-to-end latency of HP and LP jobs increases when the average job queueing time increases.

A similar relationship between average job queuing latency and λ is captured on a cloud system and the results are presented in Figure A.2 of Appendix A.2.

Figure 4.4 shows the effect of λ on the average job scanning time when running with DDPS. The results show that as λ increases the average scanning latency increases also. The increase in average scanning latency appears to be linear with respect to λ. The value of λ determines the rate at which messages arrive. As the mean arrival rate of messages increases the average number of messages in a RDD will increase also. The scanning time is proportional to the number of messages in a RDD since the each message in the RDD must be scanned. This explains why increasing λ increases the average scanning latency.

A similar relationship between average job scanning latency and λ is captured on a cloud system and the results are presented in Figure A.3 of Appendix A.2.

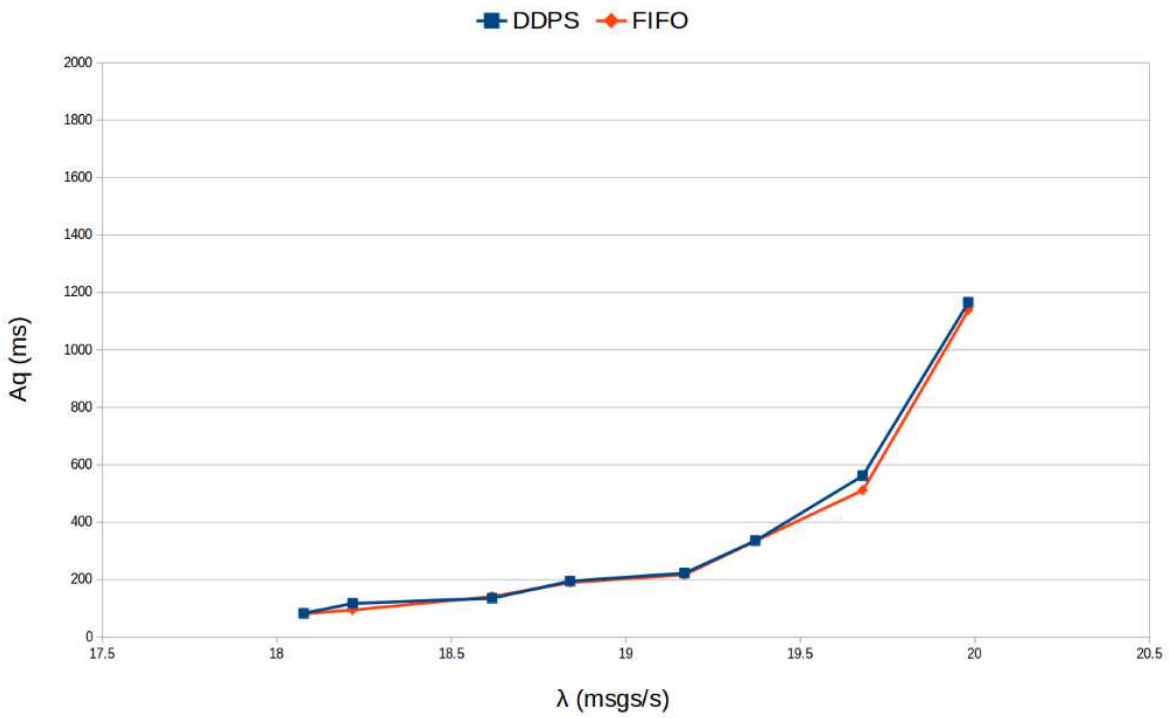

Figure 4.3: Effect of Mean Arrival Rate on Average Job Queuing Latency

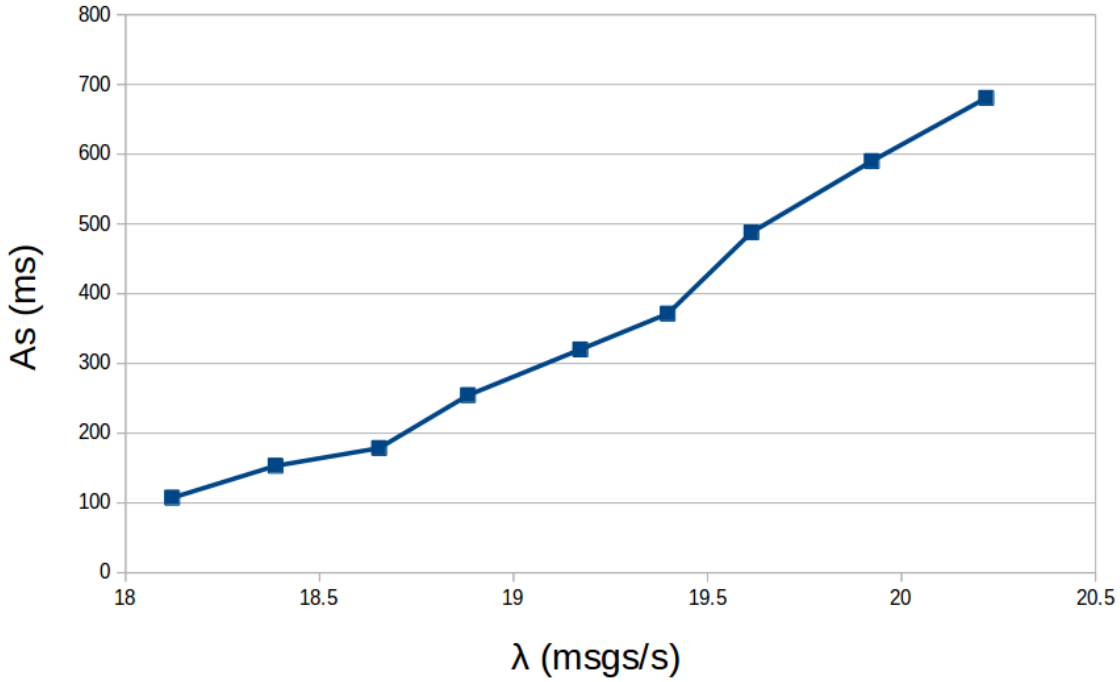

Figure 4.4: Effect of Mean Arrival Rate on Average Scanning Latency

Figure 4.5 shows the effect of λ on the scheduling overhead when running with DDPS and the FIFO scheduler. The results show that as λ increases the scheduling overhead also increases for both the DDPS and FIFO schedulers. As λ increases the number of messages that enter the system per unit time also increases. This means more memory blocks need to be allocated (and eventually deallocated) to store the messages. The allocation and deallocation of memory blocks contributes to the scheduling overhead.

A similar relationship between average scheduling overhead and λ is captured on a cloud system and the results are presented in Figure A.4 of Appendix A.2.

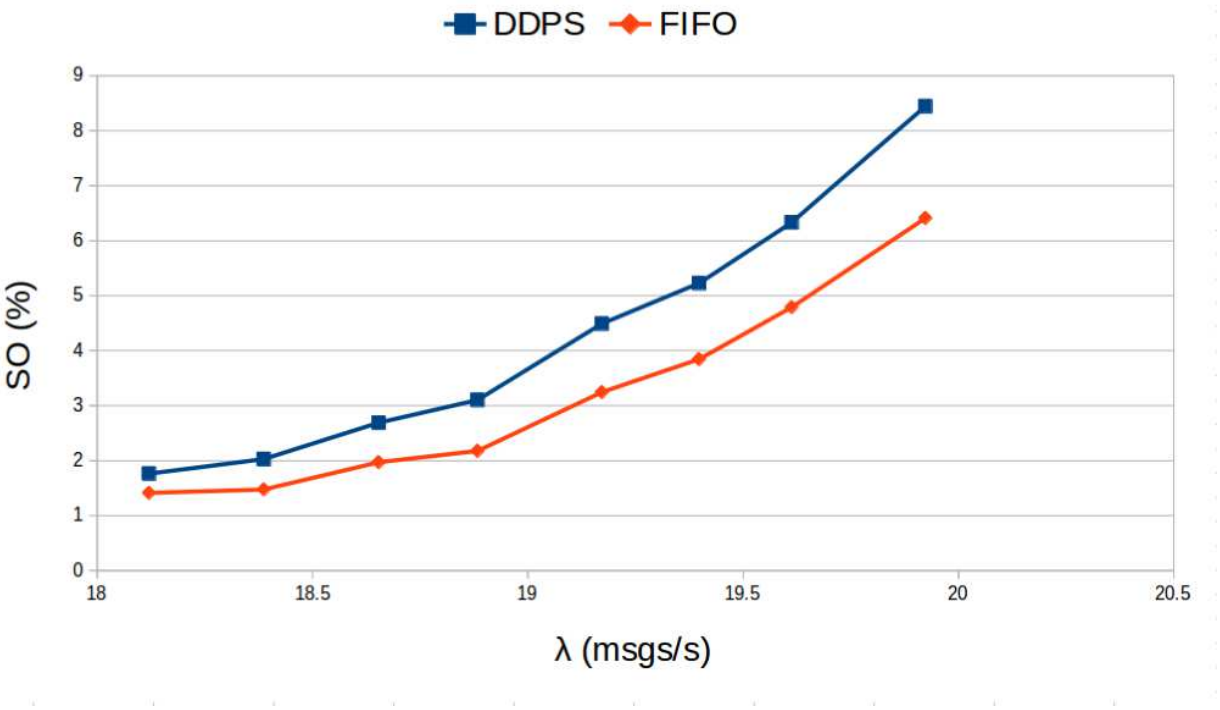

Figure 4.5: Effect of Mean Arrival Rate on Scheduling Overhead

The results in Figure 4.5 also show that for any given λ the scheduling overhead of DDPS is larger than the scheduling overhead of the FIFO scheduler. DDPS incurs additional overheads when it scans the RDDs for priority mappings, whereas, the FIFO scheduler does not incur this

overhead. As a result, DDPS will have a larger scheduling overhead in comparison to the FIFO scheduler. This analysis is confirmed in Figure 4.6. Figure 4.6 shows the difference between the scheduling overheads of the DDPS and FIFO scheduler. It is the result of subtracting DDPS line with the FIFO line in Figure 4.5. Figure 4.6 shows that the difference between the scheduling overhead of DDPS scheduler and the FIFO scheduler seems to grow monotonically as λ increases. The scanning time of the RDD also grows linearly as λ increases. Since DDPS has to scan RDDs and the FIFO scheduler does not, this would explain the difference between the scheduling overheads of the DDPS and FIFO scheduler.

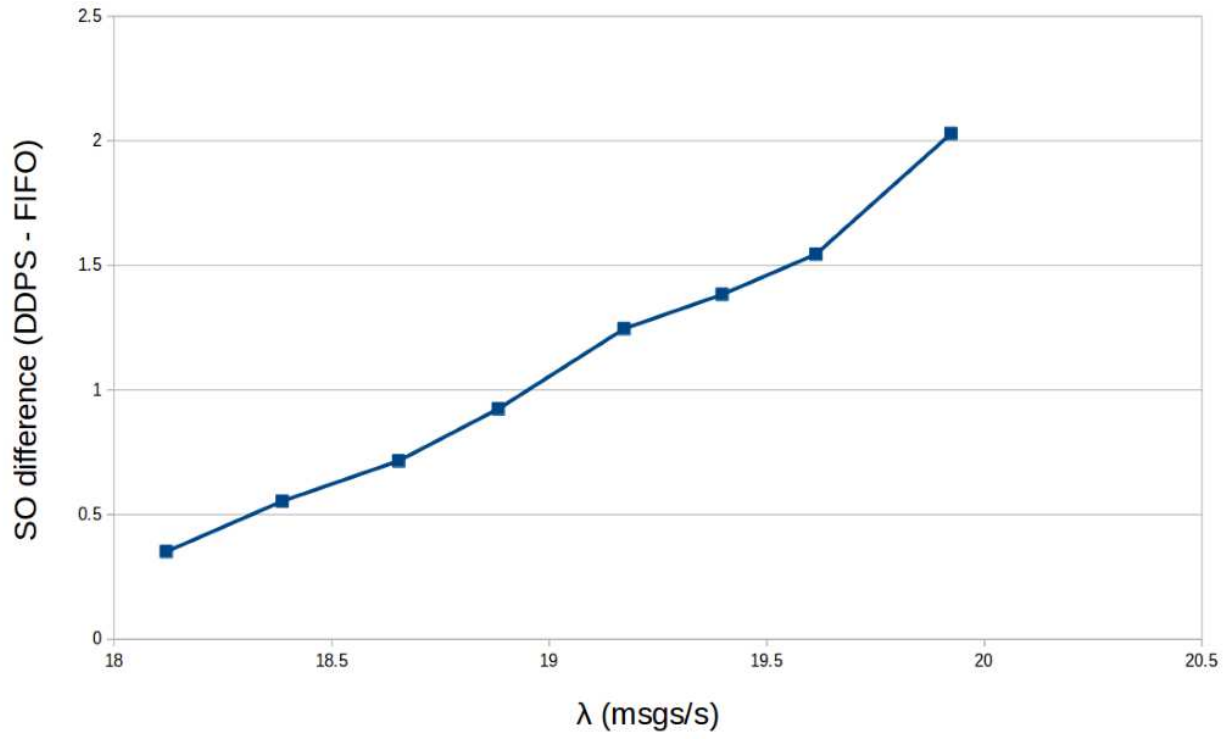

Figure 4.6: Difference in Scheduling Overhead between DDPS and FIFO

The previous discussion has concerned a system with two priority levels, the performance of a system having three priority levels (HP, MP and LP) is discussed next. Figure 4.7 shows three lines: HP, MP and LP. These lines correspond to the average end-to-end latency of high, medium

and low priority messages respectively. Figure 4.7 shows that for any given λ, LP always has the largest latency followed by MP and HP messages always have the lowest latencies. This highlights the impact of DDPS when multiple priority mappings are supplied as DDPS is able to prioritize the messages according to their assigned priorities.

A similar relationship between average end-to-end latency and λ is captured on a cloud system and the results are presented in Figure A.5 of Appendix A.2.

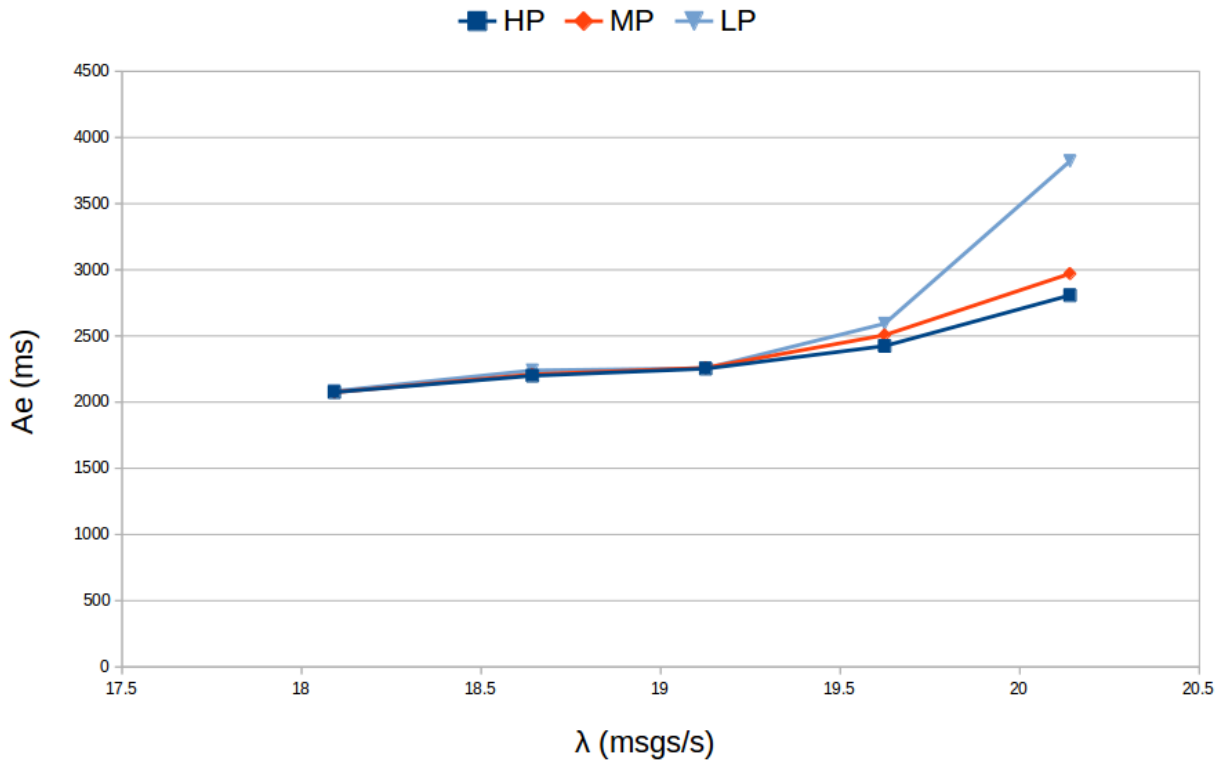

Figure 4.7: Effect of Mean Arrival Rate on Average End-to-end Latency with 3 Priority Levels

4.1.5.2 Effect of Batch Duration

This experiment focuses on the impact of varying the batch duration on various performance metrics. The batch duration determines the amount of time in which the streaming

system collects data for a batch (RDD). Increasing the batch duration will result in more messages being placed in a single RDD, and reducing the batch duration reduces the number messages being placed in the RDD. Since each experiment is run for a fixed duration of time, increasing the batch duration reduces the number of RDDs that are created during the experiment, and reducing the batch duration increases the number of RDDs that are created.

Figure 4.8 shows the results of varying the batch duration with the DDPS and FIFO scheduler. The two lines HP and LP show the average end-to-end latency for high and low priority messages achieved with DDPS. HP-FIFO and LP-FIFO show the average end-to-end latency for high and low priority messages when running with the FIFO scheduler. The results show that with the FIFO scheduler there is minimal difference between low and high priority messages as expected. Figure 4.8 shows that as batch duration increases the average end-to-end latency of LP-FIFO and HP-FIFO increases. Figure 4.8 shows that when the batch duration increases the average end-to-end latency of LP and HP also increases. However, when running with DDPS at small batch durations the results show a large difference between high and low priority messages which does not appear with the FIFO scheduler. Figure 4.8 shows that when the batch duration is between 1.2 and 1.4 seconds there is a large difference between the average end-to-end latency of HP and that of LP. When the batch duration is greater than two seconds the difference between the average end-to-end latency of HP and LP begins to decrease.

A similar relationship between average end-to-end latency and batch duration is captured on a cloud system and the results are presented in Figure A.6 of Appendix A.3.

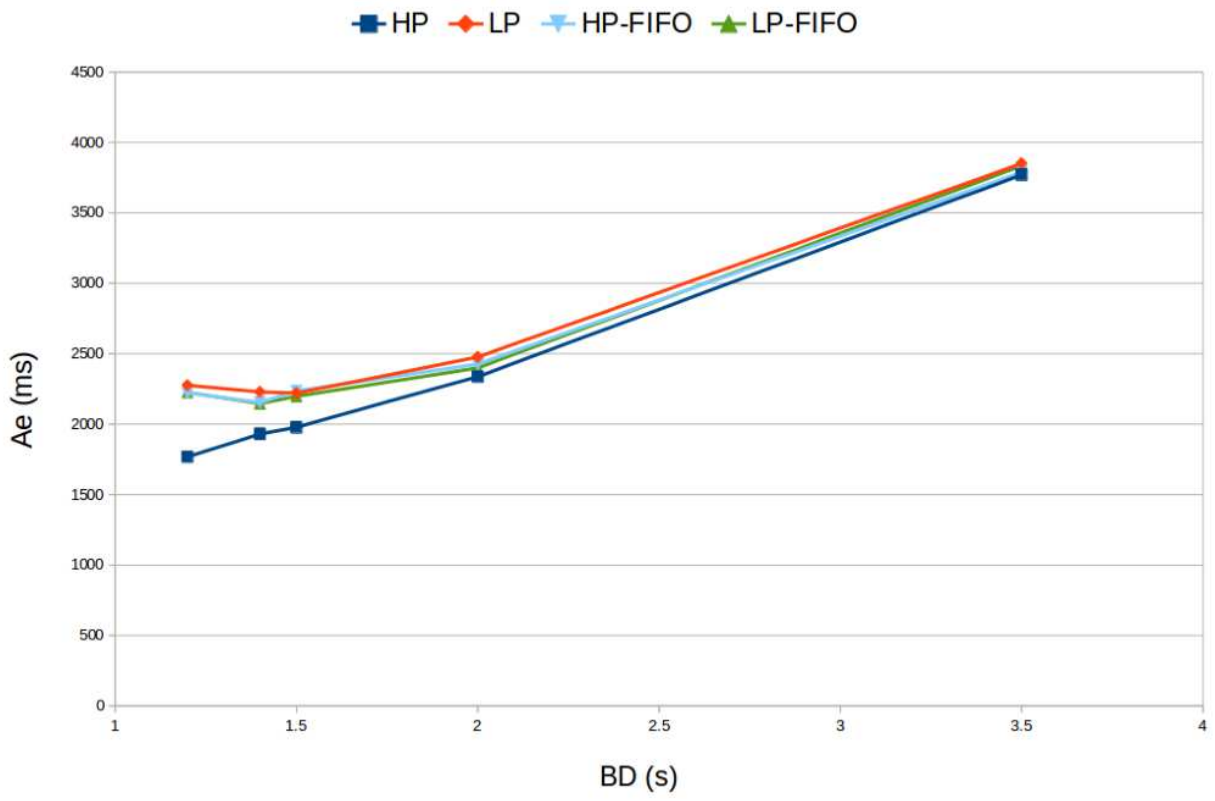

Figure 4.8: Effect of BD on Average End-to-end Latency

Figure 4.9 shows the effect of batch duration on the scheduling overhead with the DDPS and FIFO scheduler. The results show that with a small batch duration the scheduling overhead with both the DDPS and FIFO scheduler is large and as the batch duration the scheduling overhead reduces. When the batch duration is small more batches are created during the course of the experiment. As a result, all the events that occur on every batch interval such as, assigning memory blocks to RDDs, scanning RDDs, creating Jobs, recycling memory blocks, etc. occur more frequently. The effect is that the scheduling overheads consume a larger proportion of computing resources in comparison to an experiment where fewer batches are created. This means that there will be fewer computing resources to process jobs, which will result in increased job queuing times. This is why Figure 4.9 shows that as the batch duration increases the scheduling overhead reduces. The scheduling overhead is higher when running with DDPS because of the additional

overheads of scanning RDDs. The effect of batch duration on scheduling overheads is seen in Figure 4.8. Between 1.2 and 1.4 seconds the average end-to-end latency of LP, LP-FIFO and HP-FIFO decrease. This occurs because the scheduling overheads are decreasing.

A similar relationship between scheduling overhead and batch duration is captured on a cloud system and the results are presented in Figure A.7 of Appendix A.3.

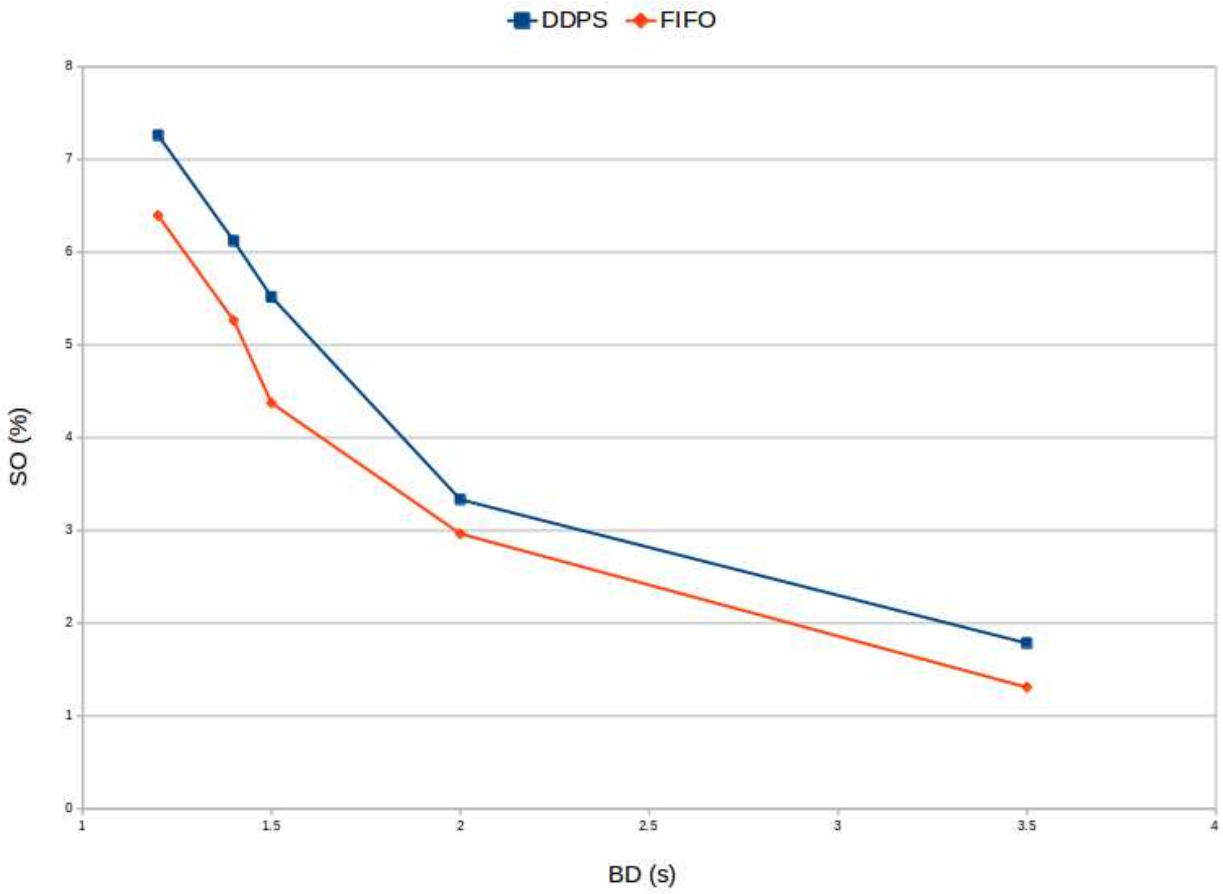

Figure 4.9: Effect of BD on Scheduling Overhead

In Section 4.1.3 batch delay (t^{i}_{Bdelay}) is introduced. Batch delay is the time difference between when the receiver receives a message to the end of the batch duration as shown in Figure 4.10. In Figure 4.10 the message arrives at time t. Time t will occur somewhere within a batch interval. The start and end of the batch interval is indicated as BStart and BEnd respectively.

Therefore time t is greater than BStart and smaller than BEnd. The batch delay is the difference between BEnd and time t. The batch delay is the duration the message waits in the receiver queue. As shown in equation 4.1, the batch delay contributes to the overall latency of processing a message. This means that any workload parameter that has an impact on the batch delay will also impact the average processing latency.

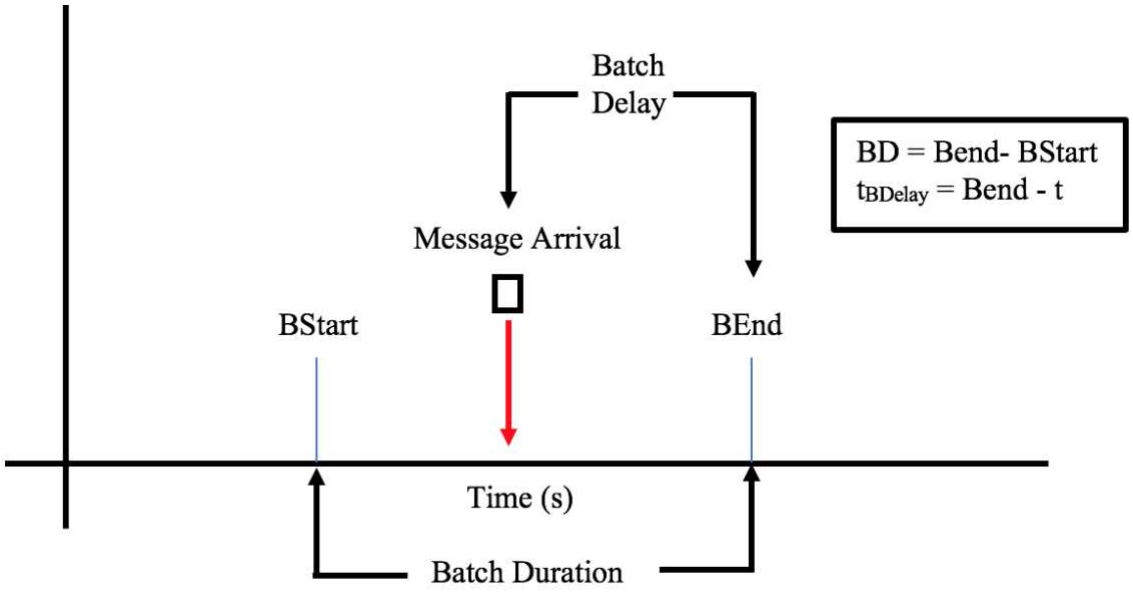

Figure 4.10: Batch Delay

All the experiments use a random number generator with an exponential distribution to determine inter-arrival times. This means that a message arrival can occur at any time within a batch duration. On average message arrivals will occur at the midpoint between BStart and BEnd as shown in Figure 4.10. A message that arrives at the midpoint between BStart and BEnd will have a batch delay that is equal to half of the batch duration. Thus, the batch delay is linearly proportional to the batch duration. This means that increasing the batch duration will increase the batch delay.

Since batch delay also contributes to the overall latency of processing messages, this would imply that increasing the batch duration would also increase the average end-to-end latency. Figure

4.8 shows that increasing the batch duration increases the average end-to-end latency of HP, LP, HP-FIFO and LP-FIFO. This behaviour occurs because the batch delay is a major contributing factor to the average end-to-end latency.

The effect of batch duration on scheduling overhead, batch delay and are similar with both the DDPS and the FIFO scheduler. However, the effect of batch duration on average end-to-end latency differs between the DDPS and the FIFO scheduler when it comes to high priority messages but not low priority messages. DDPS always gives precedence to high priority messages, however if there is no contention the difference in the average end-to-end latency between higher priority messages and lower priority messages will be small. When the batch duration is small, the resource contention is at its highest due to the high scheduler overheads (as shown in Figure 4.9). This means that when the batch duration is small the difference in average end-to-end latency between HP and LP messages will be at its highest. This explains why the HP curve is lower than the LP curve in Figure 4.8 when the batch duration is less than 1.5 seconds. As the batch duration increases, contention for resources decreases and average end-to-end latency for both LP and HP exhibit comparable values. Since the FIFO scheduler does not prioritize high priority messages, this behaviour is not observed in the LP-FIFO and HP-FIFO curves. The reason why the HP curve does not continue to remain smaller than the LP curve after batch duration exceeds 1.5 seconds is because at that point the impact of the batch delay is greater. The result is that the DDPS and the FIFO scheduler will behave in the same manner after 1.5 seconds.

Figure 4.11 shows the effect of the batch duration on the average job queuing latency. Two lines are shown: DDPS and FIFO which correspond to the streaming engine being run with the DDPS and FIFO schedulers respectively. Figure 4.11 shows that the average job queuing latencies of DDPS and FIFO are comparable for a given value of BD. The figure shows that when the batch

duration is small the average job queuing latency is high. But as the batch duration increases, the average job queuing latency also decreases. A similar behaviour was seen in Figure 4.9 where the scheduling overhead reduced as the batch duration increased. The reason why the average job queuing latency is high when BD is small is because the high scheduling overhead (shown in Figure 4.9) causes resource contention. This will increase the time it takes to process jobs and therefore increase job queuing latency. As the scheduling overhead reduces the so will the resource contention and as a result jobs will be processed faster thus reducing the average job queueing latency.

A similar relationship between average job queuing latency and batch duration is captured on a cloud system and the results are presented in Figure A.8 of Appendix A.3.

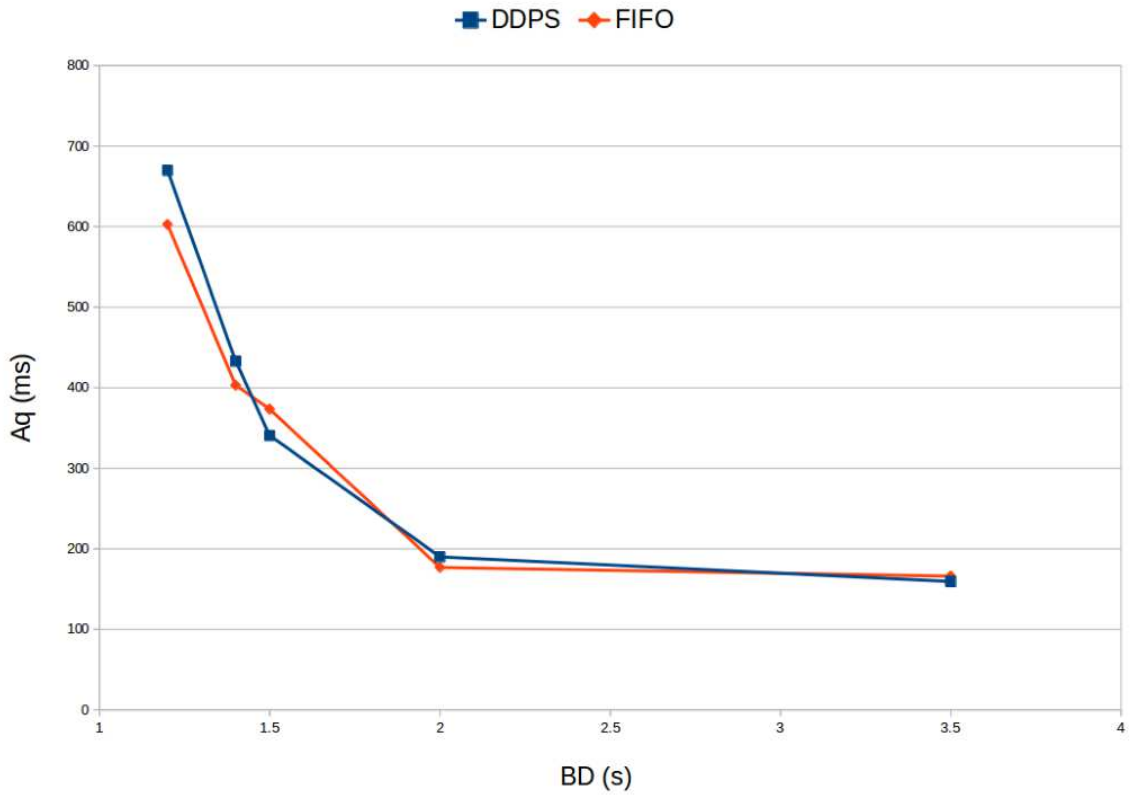

Figure 4.11: Effect of Batch Duration on Average Job Queuing Latency

4.1.5.3 Effect of Coefficient of Variation for Interval Time

This experiment measures the effect of varying the C_a. The C_a determines the variability of inter-arrival times, so increasing C_a increases the burstiness of message arrivals. Figure 4.12 shows four lines: HP, LP, HP-FIFO and LP-FIFO. HP and LP are the average end-to-end latencies of the high and low priority messages when running with DDPS. HP-FIFO and LP-FIFO are the average latencies of low and high priority messages when running with the FIFO scheduler. The results in Figure 4.12 show that as C_a increases the average latencies HP, LP, HP-FIFO and LP-FIFO all increase. This means that increasing the variability of inter-arrival times increases the average end-to-end latency when using DDPS and the FIFO scheduler. The results also show that for any given value of C_a HP messages have a lower latency than LP messages. It also shows that as message arrivals become more bursty (as C_a increases) the performance difference between the average end-to-end latency of LP and HP messages increases. This indicates that DDPS processes HP messages earlier than LP messages when the system has a bursty input load. However, with the FIFO scheduler this behaviour is not seen. For any value of C_a the average end-to-end latency values for HP-FIFO and LP-FIFO are comparable to one another as expected. This is because the FIFO scheduler makes no distinction between jobs that contain high priority messages and jobs that do not contain high priority messages.

A similar relationship between average end-to-end latency and C_a is captured on a cloud system and the results are presented in Figure A.9 of Appendix A.4.

Figure 4.13 shows the effect of C_a on the average job queueing time. The results show that as C_a increases so does the average job queuing latency for both the DDPS and the FIFO scheduler. In fact the increase in average end-to-end latency seen in Figure 4.12 is caused in largely by the increases in average job queuing latency. The reason why increasing C_a increases queuing delays

is because a high C_a increases the variability of message inter-arrival times which makes the message arrivals more bursty. As a result the number of messages owned by a Job will become more varied as C_a increases. The time it takes to process a job depends on the number of messages that it contains. Therefore, as C_a increases the variability of job processing times will also increase. Increasing the variability of processing times in queues is known to be one of the factors that increases waiting time [61]. The increase in average job queuing for DDPS in Figure 4.13 also seems to correspond with an increase in the difference between HP and LP in Figure 4.12. This means the effect of C_a on average job queuing latency is the same when running with the DDPS and FIFO scheduler.

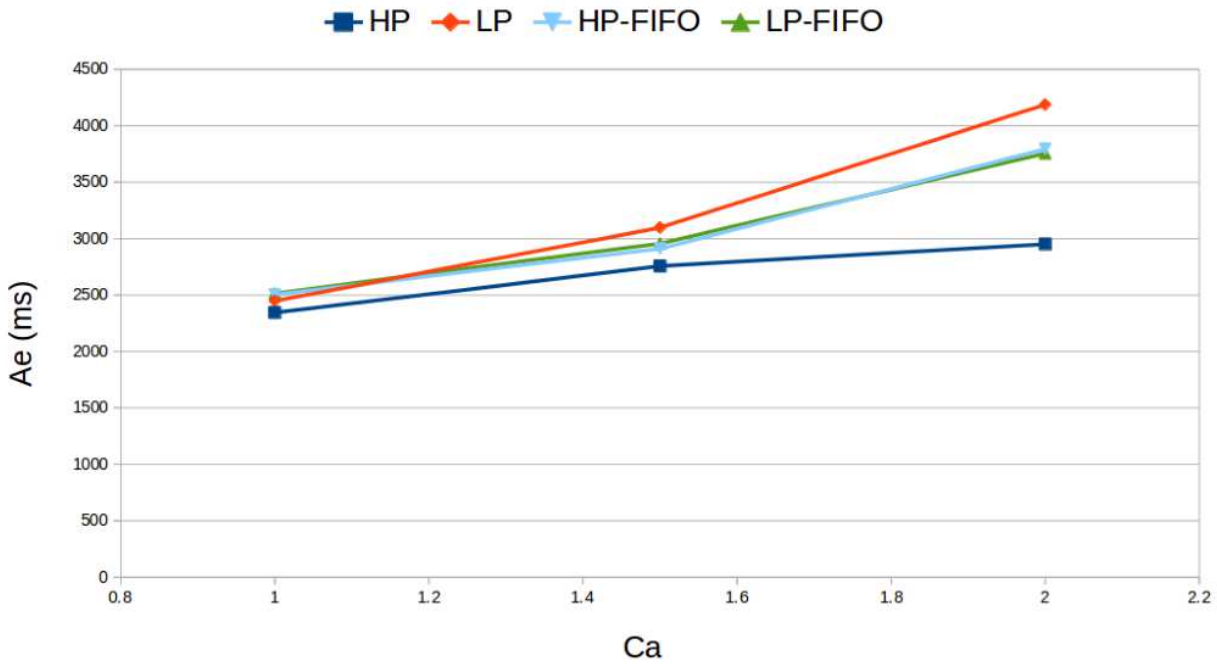

Figure 4.12: Effect of C_a on Average End-to-end Latency

A similar relationship between average job queuing latency and C_a is captured on a cloud system and the results are presented in Figure A.10 of Appendix A.4.

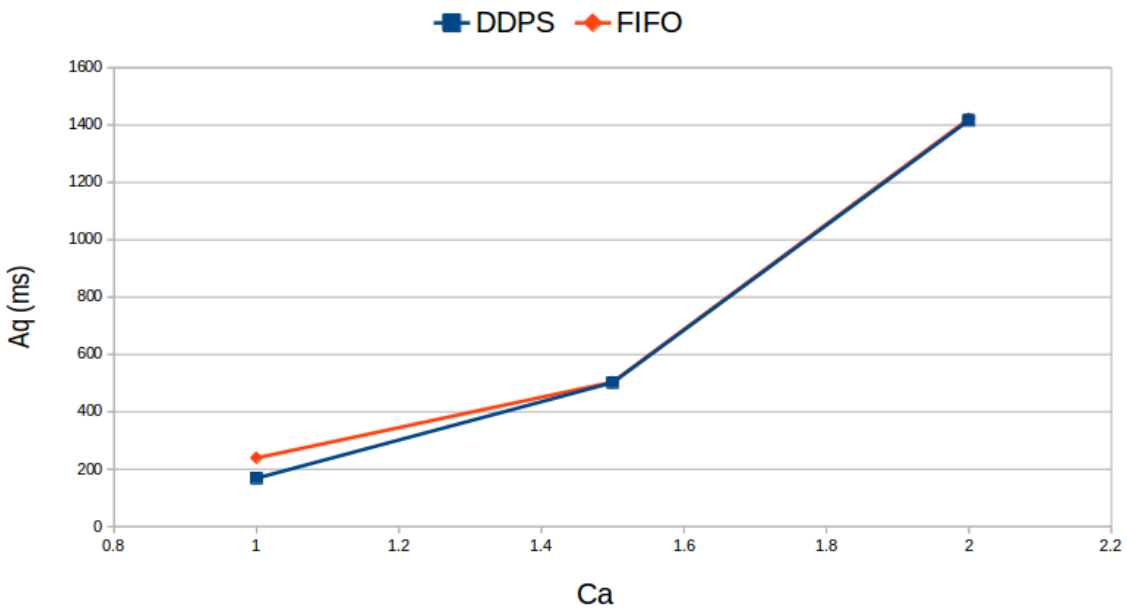

Figure 4.13: Effect of C_a on Average Job Queuing Latency

The previous discussion has concerned a system with two priority levels, the performance of a system having three priority levels (HP, MP and LP) is discussed next. The effect of C_a on average end-to-end latency with three priority levels when running with DDPS is shown below in Figure 4.14. There are three lines: HP, MP and LP. These correspond to the average end-to-end latencies of high, medium and low priority messages respectively. The results show that as C_a increases the average latencies for high, medium and low priority messages also increase. For every value of C_a HP has the lowest average end-to-end latency, followed by MP and LP. This demonstrates the ability of DDPS is able to give precedence to high priority messages over medium priority messages, and medium priority messages over low priority messages and shows the effectiveness of the priority scheduler with multiple priority levels.

A similar relationship between average end-to-end latency and C_a is captured on a cloud system and the results are presented in Figure A.11of Appendix A.4.

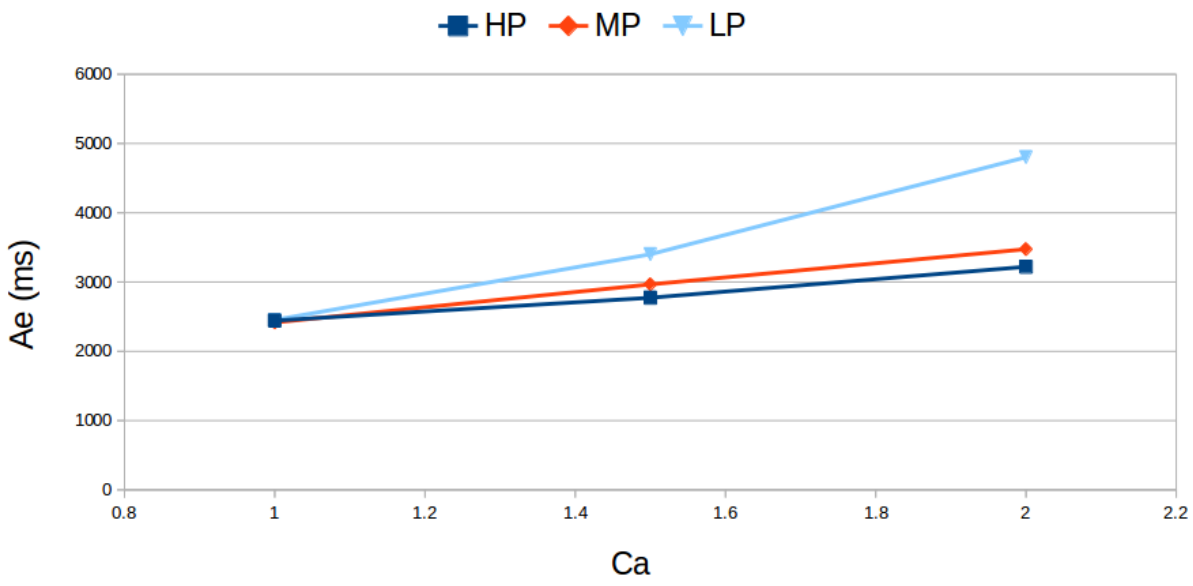

Figure 4.14: Effect of C_a on Average End-to-end Latency with 3 Priority Levels

4.1.5.4 Effect of Number of Priority Mappings

In this experiment the number of priority mappings, K, are varied and the impact on various performance metrics is measured. The number of priority mappings determines the keywords that DDPS will search for during the scanning phase. If no priority mappings are supplied the scanning phase is skipped and all jobs will be processed in a FIFO manner.

Figure 4.15 shows the effect of K on average end-to-end latency. The results show that when K is low the average end-to-end latency is also low. As K increases so does the average end-to-end latency. The increase in average end-to-end latency with respect to K appears to be linear. The reason why this occurs is that as the number of priority mappings increases, the number of key words that need to be searched for when scanning RDDs also increases. The rationale for this behaviour is discussed in the following paragraph

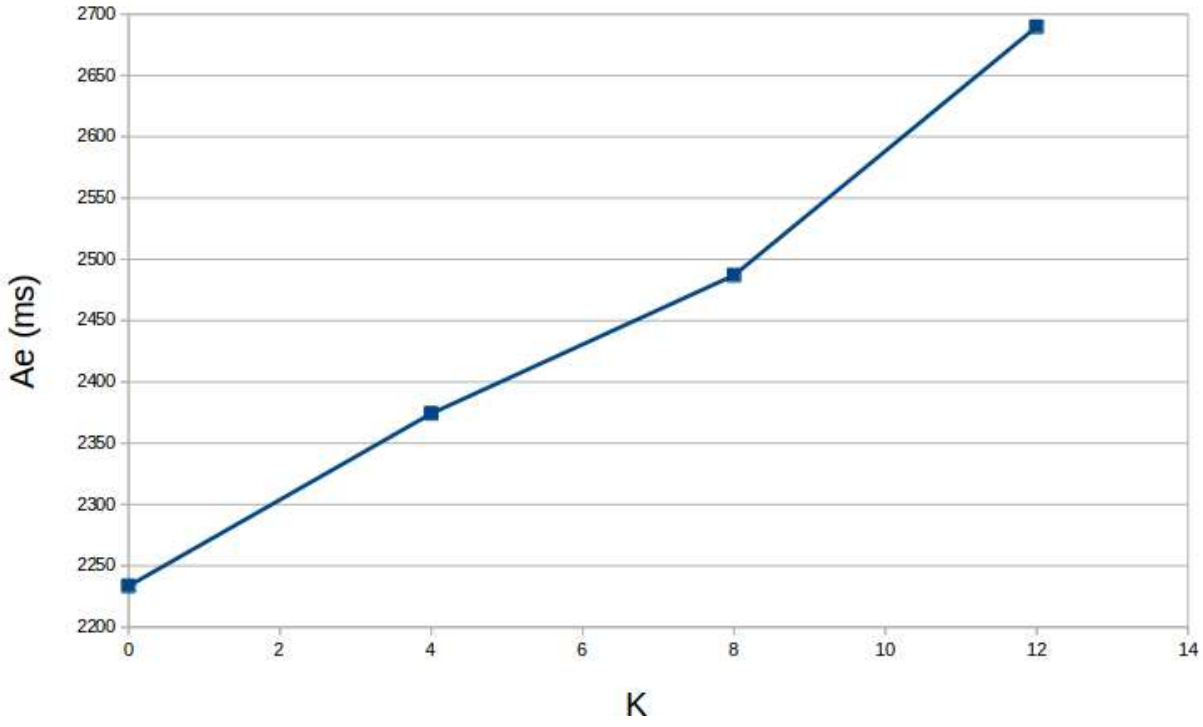

Figure 4.15: Effect of K on Average End-to-end Latency

The effect of K on the average scanning latency is shown below in Figure 4.16. The results show that when K is zero the average scanning is also zero. As K increases the average scanning latency increases in a linear fashion. The scanning time increases linearly because all messages need to be scanned for each priority mapping. This increase in scanning time contributes to the increase in average end-to-end latency captured in Figure 4.15.

Figure 4.17 shows the effect of K on scheduling overhead. The results as K increases the scheduling overhead increases. This is similar to the impact of K on average scanning latency that was captured in Figure 4.16. The average scanning latency contributes to the overall scheduling overhead. Therefore, when the average scanning latency increases the scheduling overhead also increases.

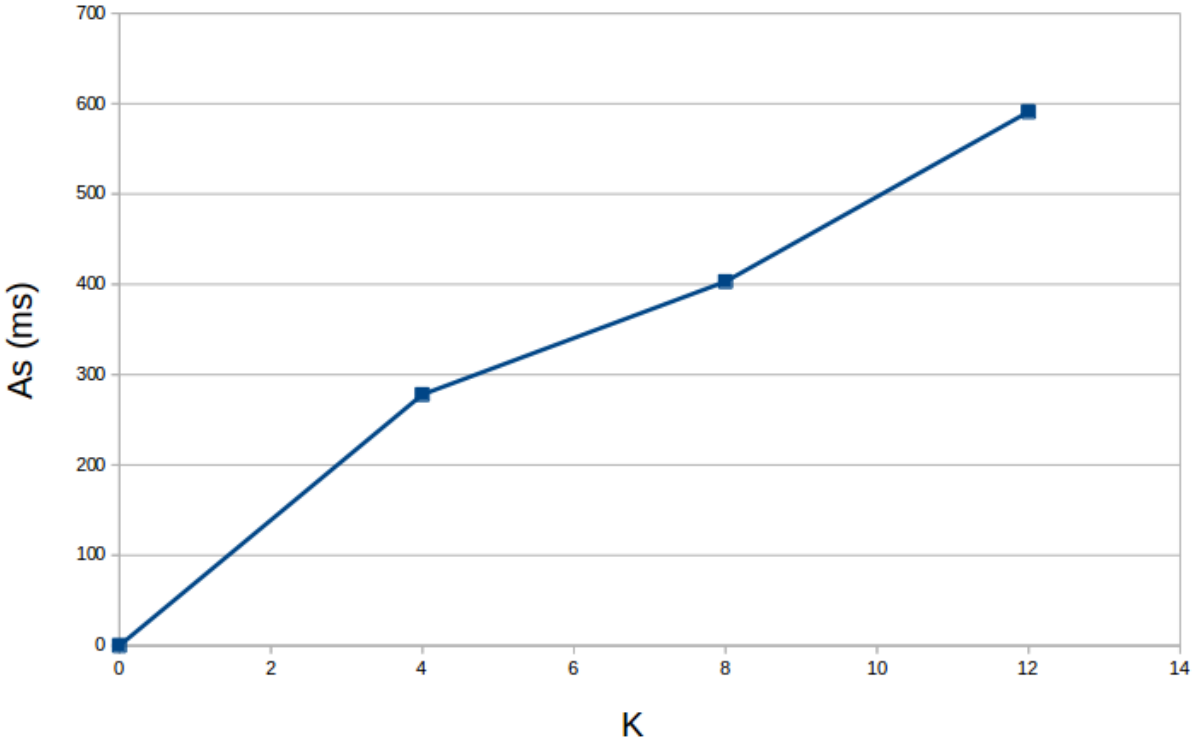

Figure 4.16: Effect of K on Average Scanning Latency

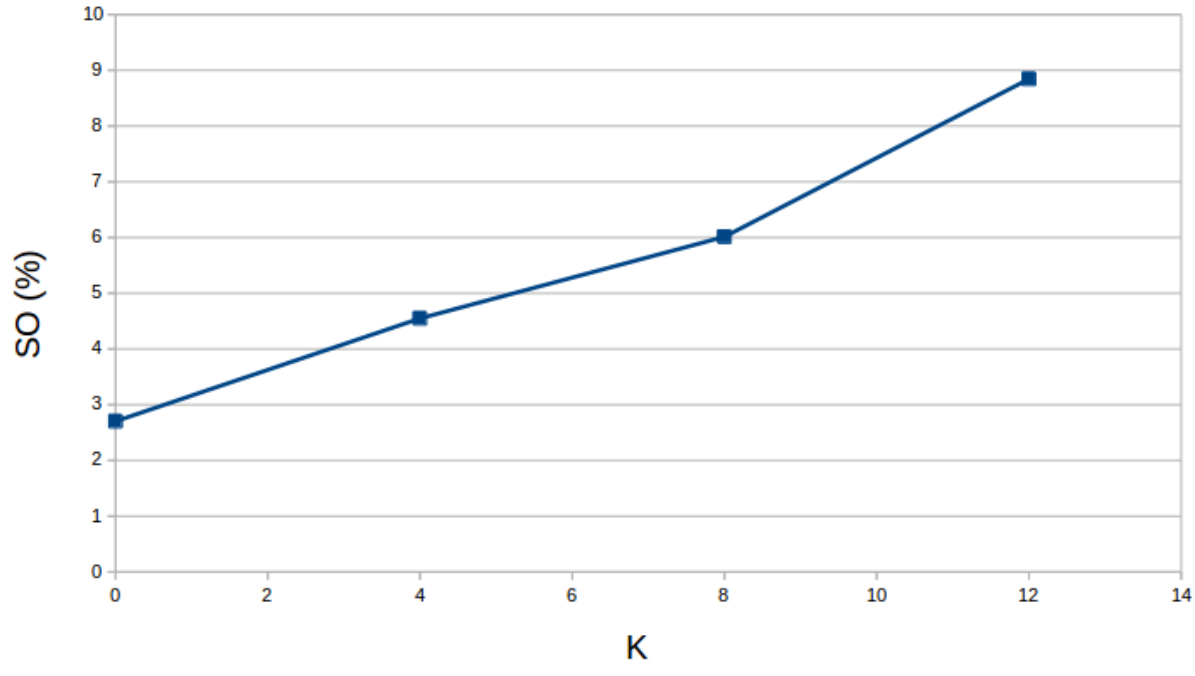

Figure 4.17: Effect of K on Scheduling Overhead

The effect of K on the average job queuing latency is shown below in Figure 4.18. The results show that for any value of K the average job queuing latency remains fairly constant. This means that K does not have any effect on the average job queuing latency. Increasing K increases the average scanning latency which affects the overall latency. However, messages are scanned before the job is placed in the job queue therefore do not prolong the duration the job spends in the job queue. As a result, any delay that causes an increase in the average scanning latency does affect the average job queuing latency.

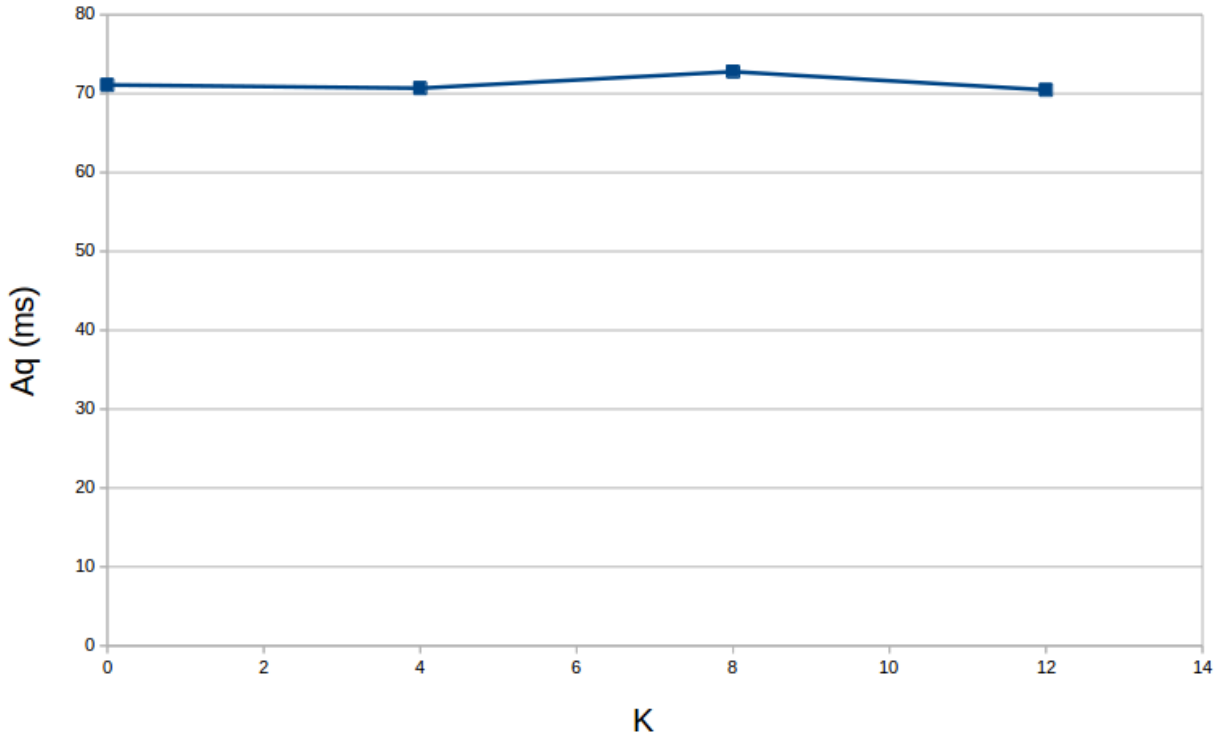

Figure 4.18: Effect of K on Average Job Queuing Latency

4.1.5.5 Effect of Message Length

In this experiment the length of messages sent by the message sending process is varied. The effect of message length on average end-to-end latency when running with DDPS is shown in Figure 4.19. The figure shows the average end-to-end latency on HP and LP messages. Both HP and LP appear to increase as the message length increases. As expected, for every value of L the

average latency of LP messages is higher than that of HP messages as expected. The difference between the average latencies of HP and LP also appear to increase as message length increases. The effect of message length on average job queuing latency is shown in Figure 4.20. The results show that the average job queuing latency remains fairly constant with respect to message length. This means the message length does not have an impact on job queuing latency. The reason for this is presented. The time it takes to process a job depends on the number of messages contained in a job and the amount of time it takes to process each message. The number of messages contained in job is determined by λ which is a constant in this experiment. The time it takes to process each message is determined by the message service time which is a workload parameter (as described in Section 4.1.1). In this experiment neither λ or the message service time are affected by increasing the message length. As a result the average queuing time remains fairly constant.

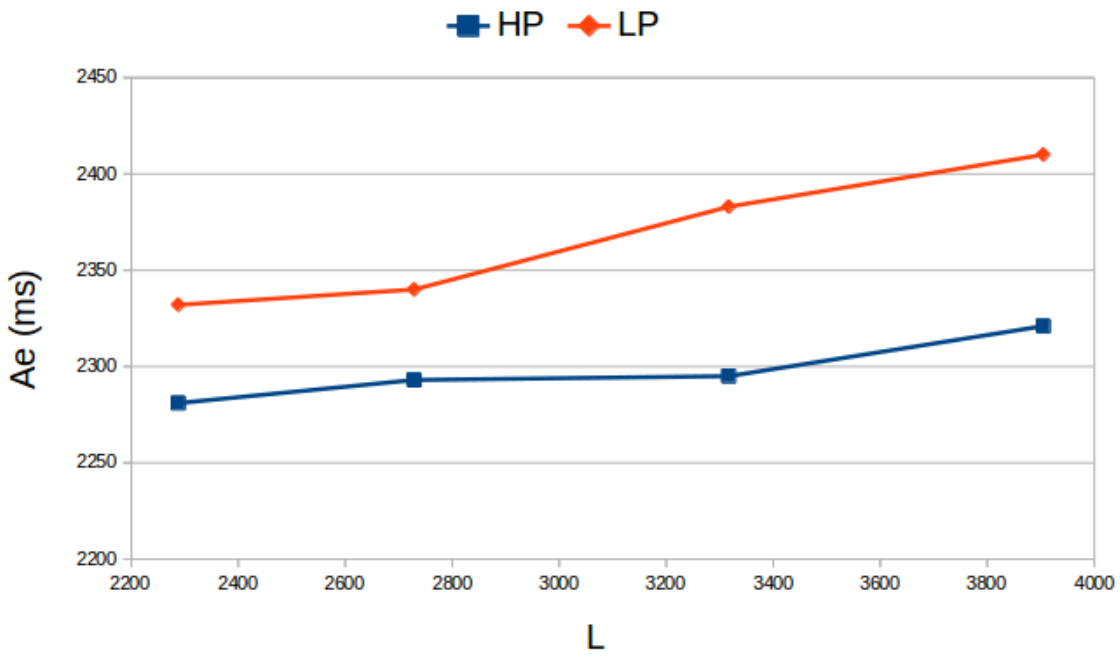

Figure 4.19: Effect of Message Length on Average End-to-end Latency

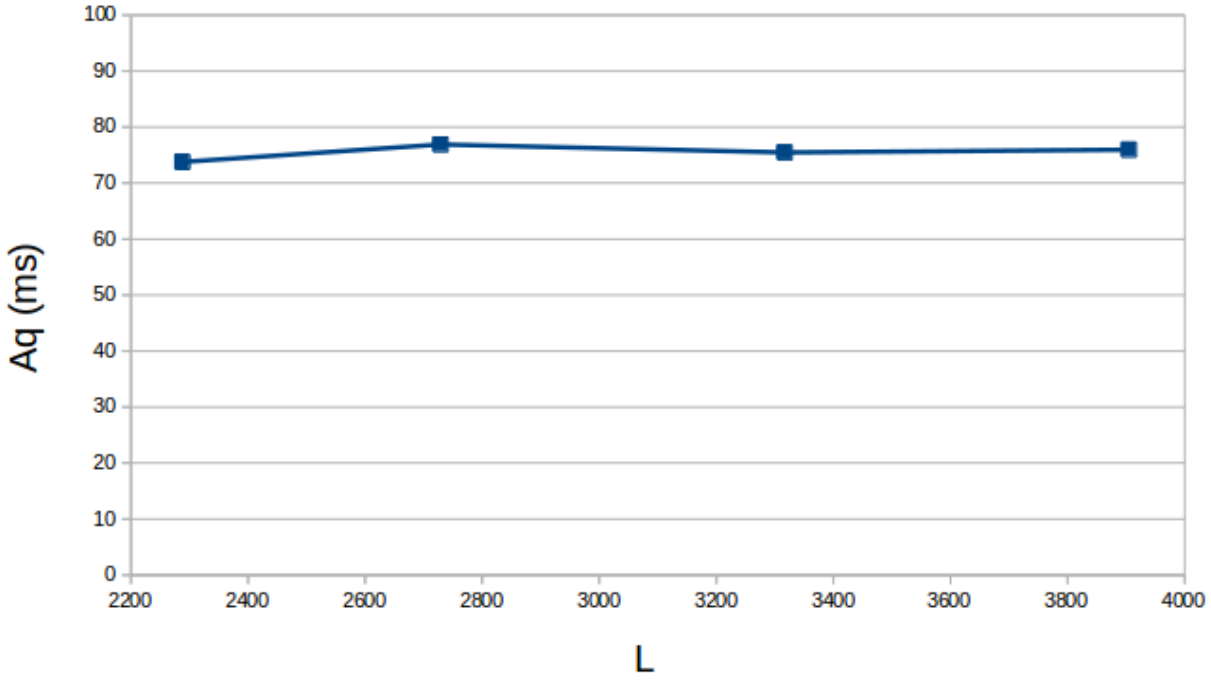

Figure 4.20: Effect of Message Length on Average Job Queuing Latency

The effect of message length on average scanning latency is shown in Figure 4.21. The results show that as message length increases there is also an increase in average scanning latency. This increase appears to be linear with respect to message length. The reason for this result is that DDPS scans all messages to search for priority keywords. The longer the message is, the longer it takes to scan the message. As a result, increasing the message length will increase the average scanning latency. The average scanning latency contributes to the total latency for processing a message (equation 4.1). Therefore increasing the average scanning latency will also increase the average end-to-end latency. This explains why in Figure 4.19 the average end-to-end latency increases for both HP and LP when the message length increases.

Figure 4.22 presents the effect of message length on scheduling overhead. The results show that as the message length increases the scheduling overhead also increases. This is because the

time taken to scan messages contributes to the scheduling overhead. Increasing the message length increases the average scanning latency that in turn increases the scheduling overhead.

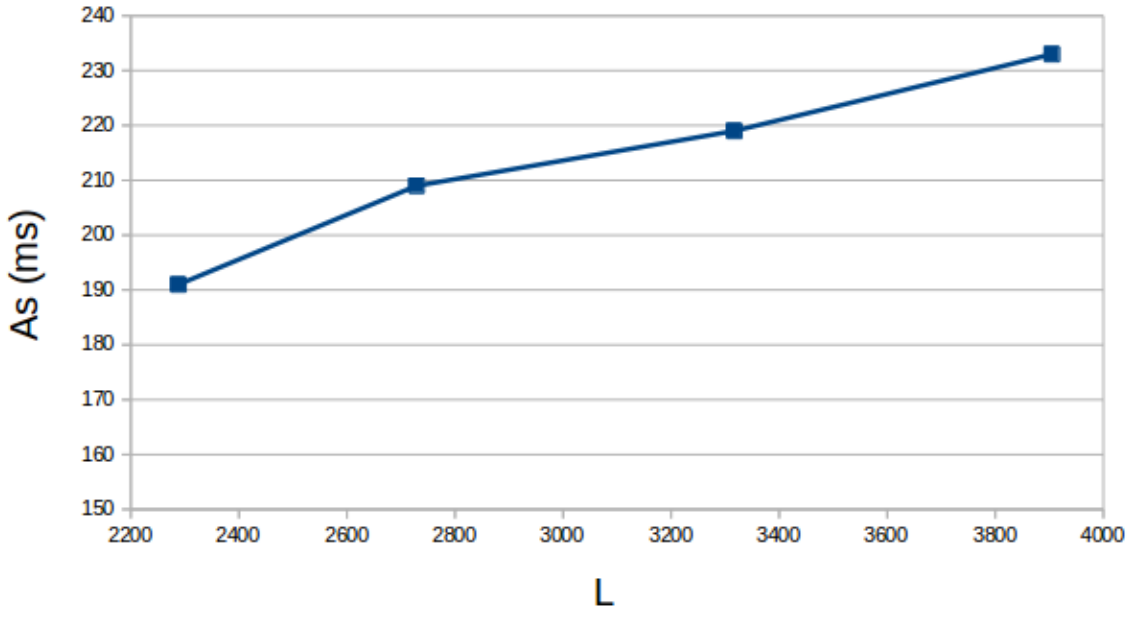

Figure 4.21: Effect of Message Length on Average Scanning Latency

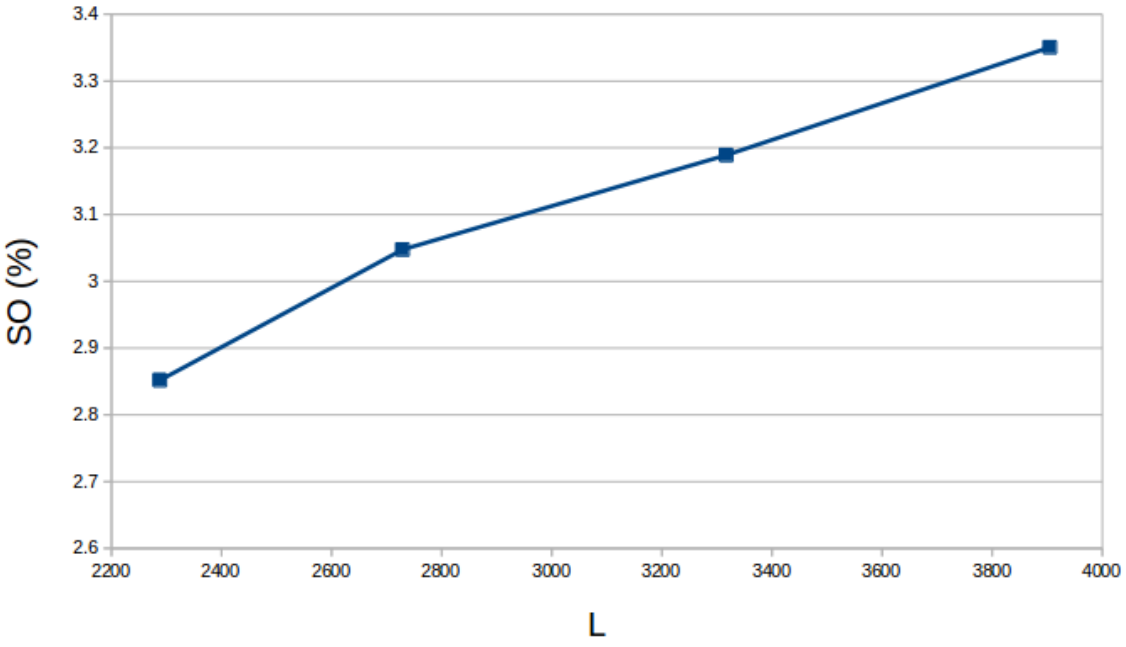

Figure 4.22: Effect of Message Length on Scheduling Overhead

4.1.5.6 Effect of Message Service Time

In this experiment the message service time is varied and the average end-to-end latency and average job queuing latency is measured when running with DDPS. The message service time determines the amount of time taken to process a message. The effect of message service time on average end-to-end latency is shown below in Figure 4.23. The results show that when message service time increases the average end-to-end latency of both HP and LP messages also increase. It also shows that that for any message service time HP messages have a lower average end-to-end latency than LP messages. As the message service time increases, the difference between HP and LP also increases. This is similar to the results seen when varying λ (Figure 4.2) and C_a (Figure 4.12) with respect to average end-to-end latency.

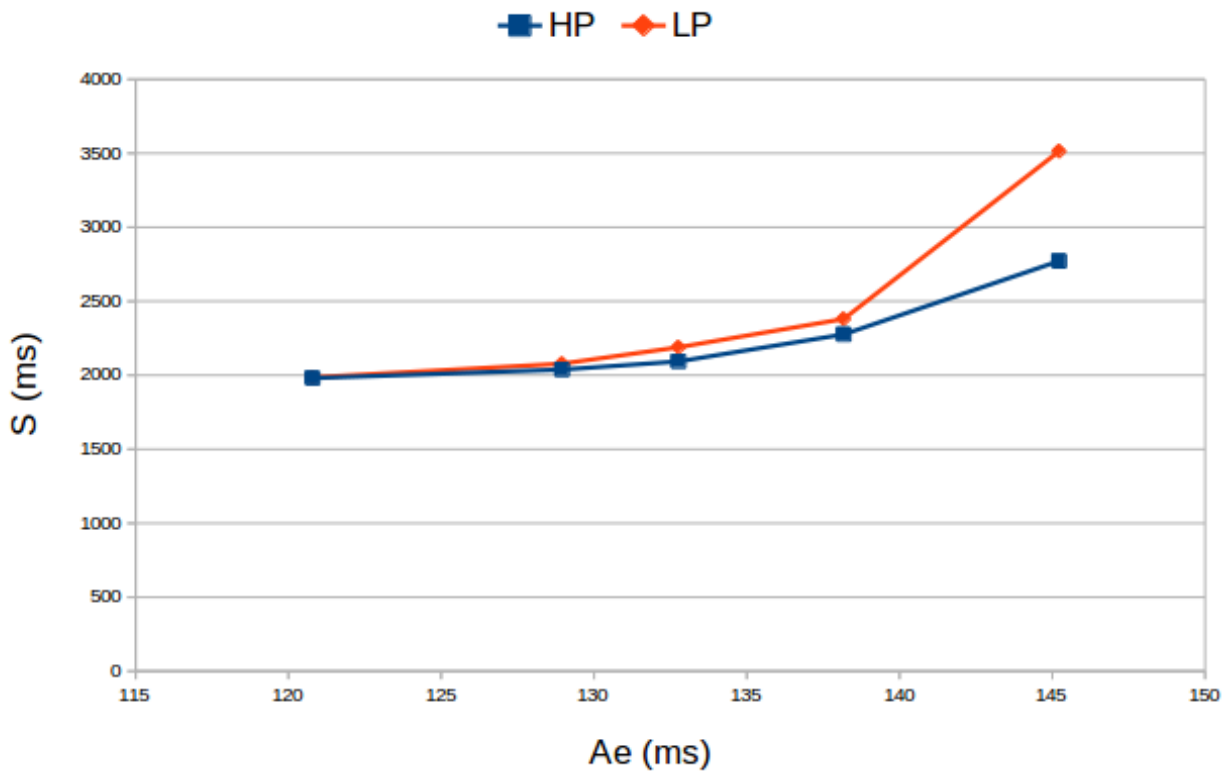

Figure 4.23: Effect of Message Service Time on Average End-to-end Latency

Figure 4.24 shows the effect of message service time on the average job queuing latency. The results show that as message service time increases the average job queuing also increases. When the message service time is around 137ms, the increase in average job queuing latency is more dramatic. This is also the same point where the difference between the average end-to-end latency of HP and LP message begins to increase in Figure 4.23. Increasing the mean service time increases the amount of time each message takes to be processed. As a result, the amount of time it takes to process a job will increase also. An increase in job processing time will increase the job queuing delays.

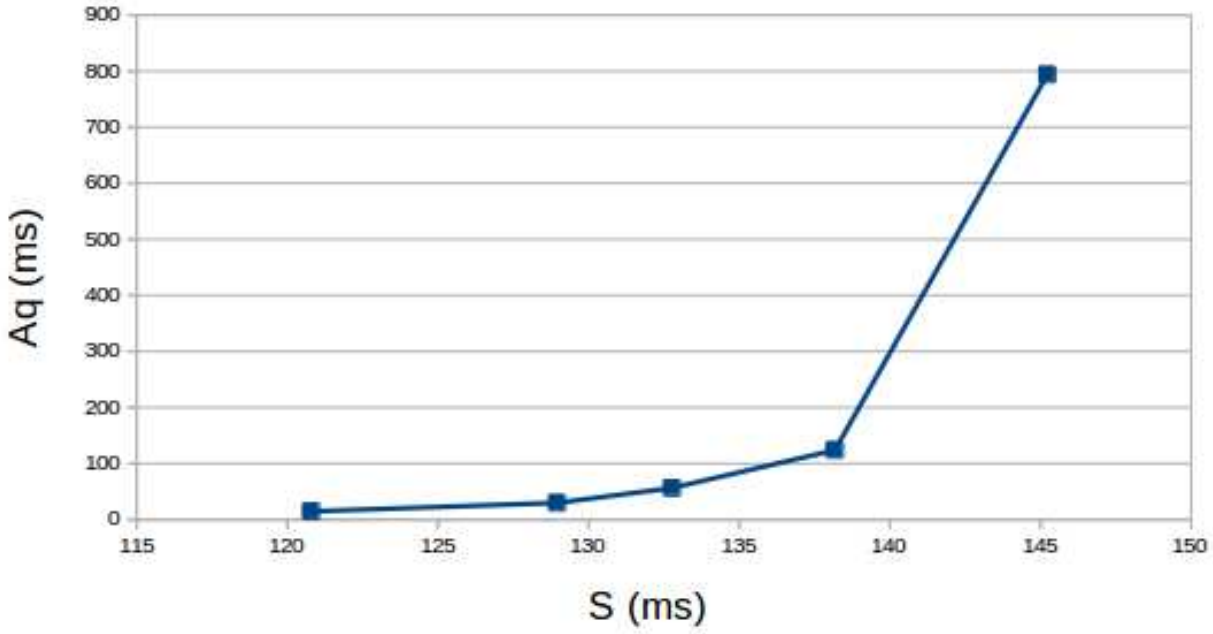

Figure 4.24: Effect of Message Service Time on Average Job Queuing Latency

4.1.5.7 Effect of the Number of Logical CPUs

In this experiment the number CPUs assigned to the Spark Streaming engine are varied. This experiment is run on the Amazon EC2 cloud. Figure 4.25 shows the effect C on average end-to-end latency. The figure has four lines, HP, LP, HP-FIFO and LP-FIFO. These correspond to high and low priority messages with the DDPS and FIFO scheduler respectively. The results show

that when C is low, the average end-to-end latency of all four lines is at its highest. As C increases the average end-to-end latency reduces. The results also show that for any value of C HP has a lower priority than LP. When C is low the difference between HP and LP is high. As C increases the difference becomes smaller. Figure 4.25 shows that the average end-to-end latency of HP-FIFO and LP-FIFO is similar for all values of C.

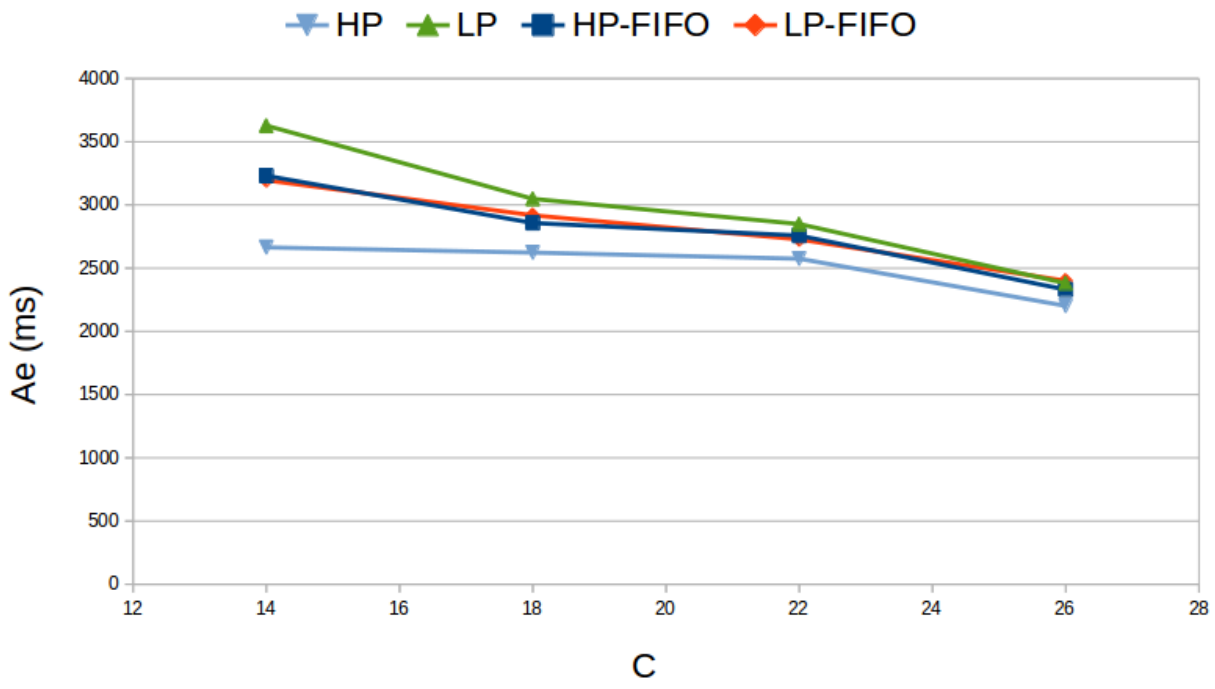

Figure 4.25: Cloud: Effect of Number of Cores on Average End-to-end Latency

Figure 4.26 shows the effect of C on the average job queuing latency. The results show that the average job queuing latency is similar when running with the DDPS and the FIFO scheduler. The results also show that when there are fewer CPUs, there average job queuing latency is high. But as the number of CPUs increases the average job queuing latency reduces. This occurs because increasing the number of CPUs increases the parallelism when processing a job. This means the amount of time it takes to process a job is reduced. This explains why in in Figure 4.25 the average latencies are high when C is low.

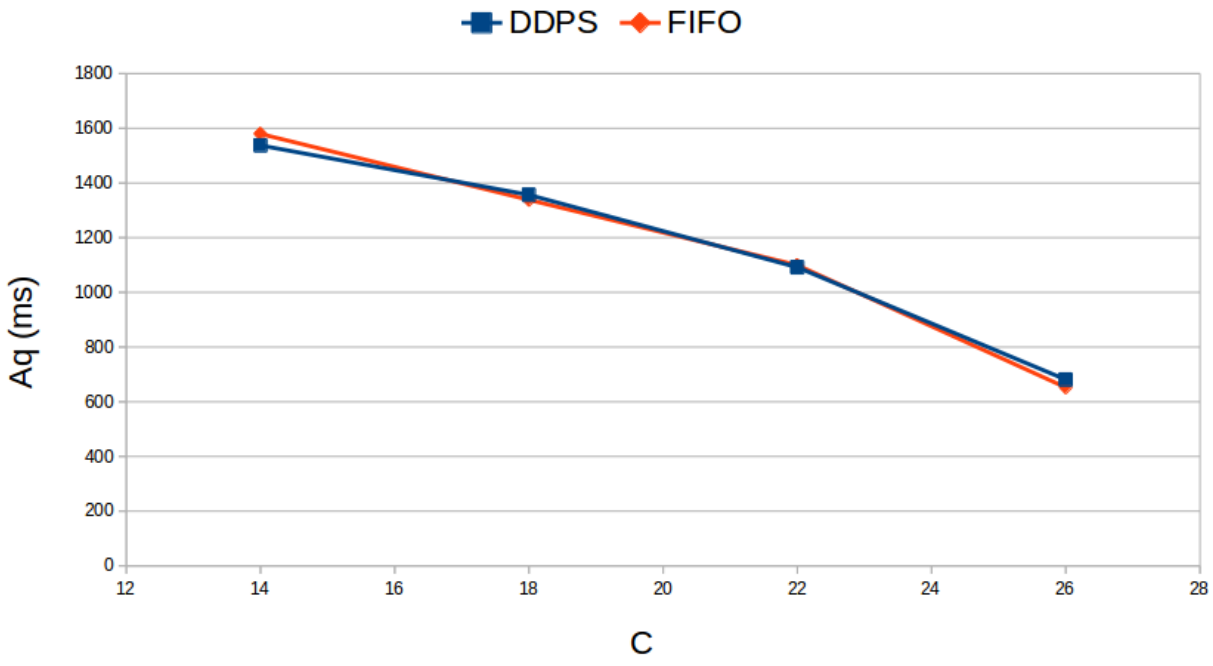

Figure 4.26: Cloud: Effect of Number of Cores on Average Job Queuing Latency

4.1.5.8 Effect of the Percentage of High Priority Messages

In this experiment the percentage of high priority messages is varied and the results are shown in Figure 4.27. The y-axis on Figure 4.27 shows the result of subtracting the average end-to-end latency of high priority messages from the average end-to-end latency of low priority messages (Δ1). On the x-axis is the percentage of high priority messages presented in a logarithmic scale. The results show that when the percentage of HP messages is low the latency difference is high. But as the percentage of HP messages increase the difference reduces. This occurs because when there are fewer HP messages, the job that contains the HP message takes precedence over other jobs in the priority queue, and as a result the average latency of processing the messages in that job are lower than jobs without HP messages. When the number of HP messages increases there are more jobs containing HP messages, and these jobs will all have a higher priority than jobs without HP messages. When multiple jobs have the same priority, they are processed in a

FIFO order with respect to each other. Therefore, as the number of high priority jobs increases the effect of priority scheduling diminishes.

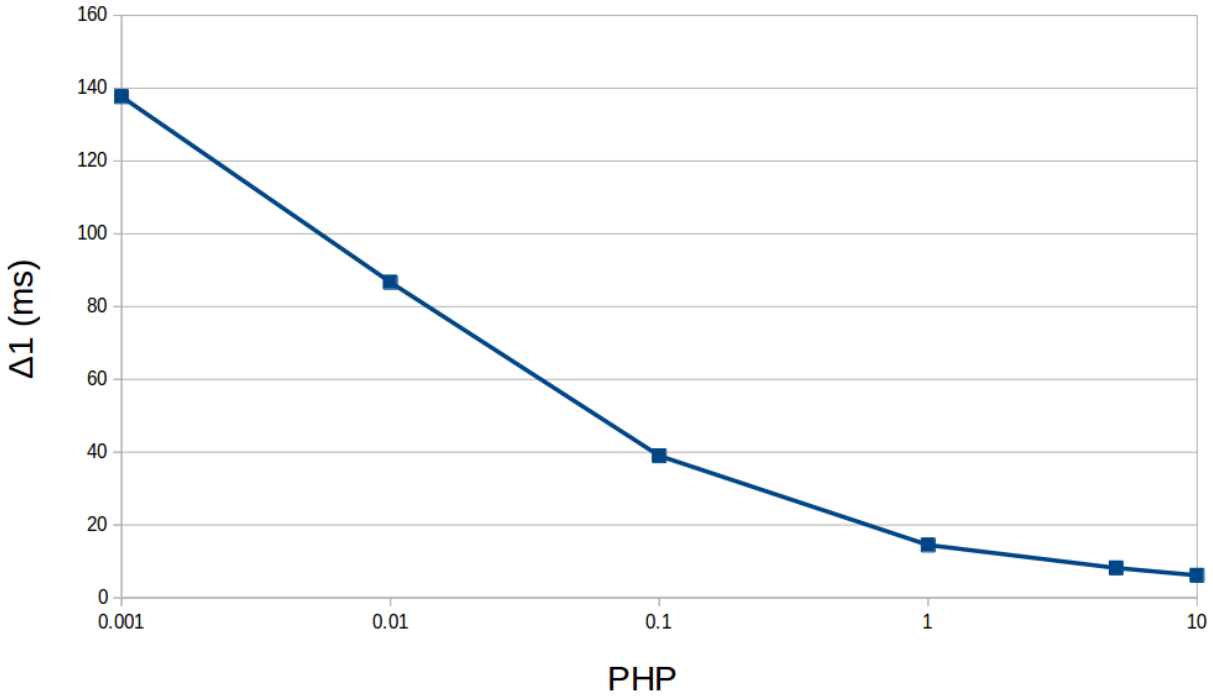

Figure 4.27: Effect of Percentage of High Priority Messages on Δ1

4.1.5.9 Effect of the Percentage of Medium Priority Messages

In this experiment three priority levels for messages are used, high, medium and low. The percentage of medium priority messages is varied and the average end-to-end latency for low, medium and high priority messages is recorded while running with DDPS. Only the difference between two priority levels is presented at a time for clarity. The difference between the average end-to-end latency of medium and high priority messages is shown in Figure 4.28. The difference between the average end-to-end latency of low and medium priority messages is shown in Figure 4.29. Figure 4.28 shows the results of varying the percentage of medium priority messages. The y-axis shows the result of subtracting the average end-to-end latency of high priority messages from that of medium priority messages (Δ2). The results show that Δ2 is always greater than zero.

This means that the average end-to-end latency of medium priority messages is always larger than that of high priority messages. The results also show that when there is a small percentage of medium priority messages, Δ2 is small. As the percentage of medium priority messages increases, Δ2 begins to increase. Figure 4.29 shows the results of varying the percentage of medium priority messages and recording the difference in average end-to-end latencies between low priority messages and medium priority messages (Δ3). The results show that Δ3 is always greater than zero. This means that the average end-to-end latency of low priority messages is always larger than that of medium priority messages. The results also show that when there is a small percentage of medium priority messages Δ3 is large, but as the percentage of medium priority messages increases Δ3 reduces.

As the percentage of medium priority messages increases the proportion of high priority messages remains the same, the proportion of medium priority messages increases and the proportion of low priority messages reduces. Therefore, as the percentage of medium priority messages increases, a given medium priority message on average will see a higher number of medium priority messages ahead of it in the queue. As a result the average end-to-end latency of medium priority messages will increase. Since Δ2 is the difference between the average end-to-end latency of high and medium priority messages, when the percentage of medium priority messages increases, Δ2 will increase also. Similarly, Δ3 is the difference between the average end-to-end latency of medium and low priority messages, therefore, as the average end-to-end latency of medium priority messages increases, Δ3 decreases.

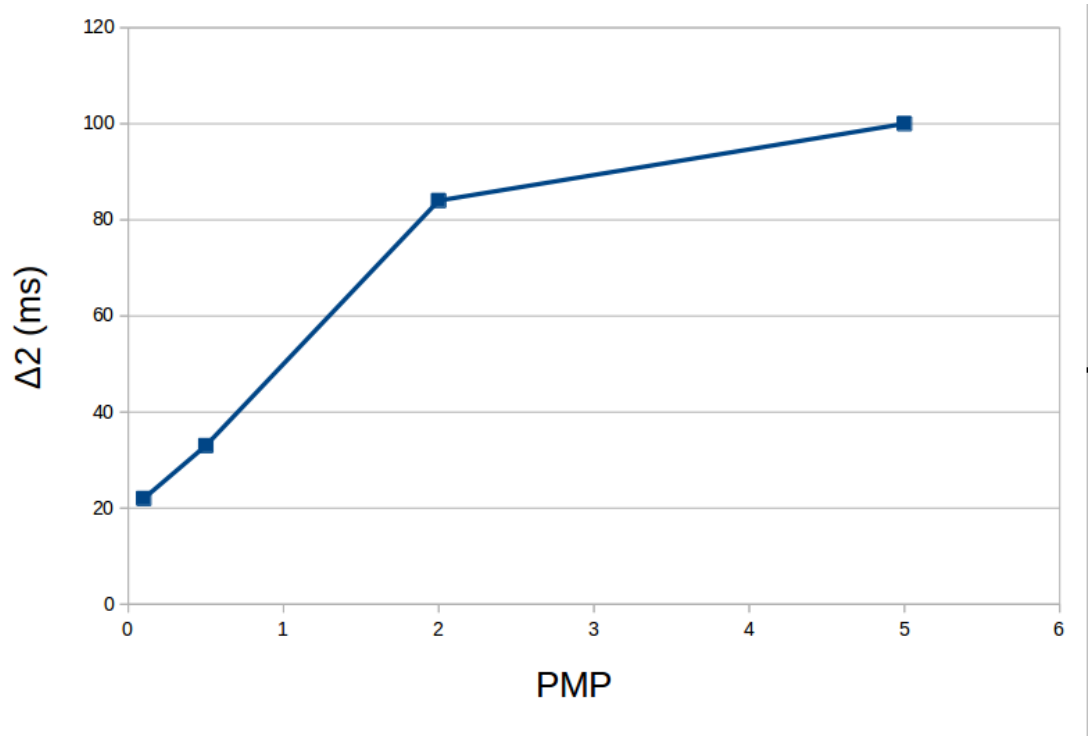

Figure 4.28: Effect of Percentage of Medium Priority Messages on Δ2

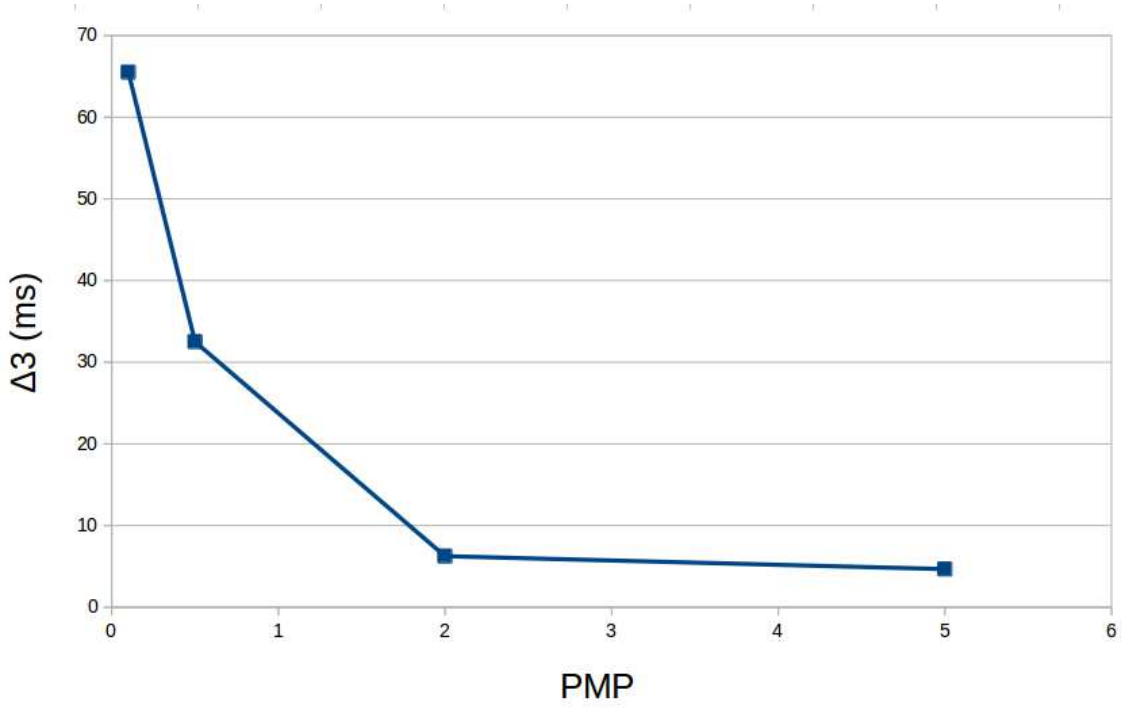

Figure 4.29: Effect of Percentage of Medium Priority Messages on Δ3

4.2 Live Twitter Sentiment Analysis Scenario

This experiment aims to test the performance of DDPS in a real-world sentiment analysis scenario. The experiment begins by launching a Spark Streaming Application. The streaming application trains a language model classifier [62] by feeding it messages and as well as data to describe whether the message has a 'positive', 'neutral' or 'negative' sentiment. This data comes from a training set [63] of over one million entries containing English tweets. Each tweet is marked as having a negative, neutral or positive sentiment. A sentiment is a feeling or attitude about a specific topic. This training set has been used by other researchers [64] [65] [66] to conduct research on sentiment analysis techniques. The experiment uses a supervised machine learning technique [67] where a dataset (in this case a series of tweets) is passed to the classifier along with the category (positive, neutral or negative sentiment) that it is associated with. Once this process is complete, the classifier can be queried with a message and it will produce an estimate describing whether the message is associated with a positive, neutral or negative sentiment.

The streaming application defines a set of operations that can be characterized by a DAG shown in Figure 4.30 that displays the different stages of processing after it receives the input data from Twitter Server. The application establishes a connection to Twitter Server using the Twitter API for Spark Streaming [68]. Once the connection is made the twitter stream will produce tweets. Each tweet is represented as a 'twitter4j.status' object. Note that "twitter4j" is a shorter version of "Twitter for java" that is the API used for connecting to Twitter Server. This object contains the contents of the tweet as well as the name of the sender, the location that the tweet was sent from and the time at which the tweet was sent. These tweets are from all over the world and can be written in any language. The tweets are passed to a Filter transformation which filters out the English tweets. The English tweets are then passed to a MapToPair transformation through an

InputDStream. The MapToPair transformation evaluates the message using the classifier and determines its category. The transformation returns a key-value tuple where the key is the category of the message and the value is an integer of 1 via a PairDStream. This stream is passed to a ReduceByKey transformation which takes the key-value tuples and groups them based on their key, then adds up the values. The final result are two tuples containing the number of messages belonging to positive and negative sentiment categories (i.e. {positive, 1}, {neutral, 1}, {positive, 1}, {positive, 1}, {negative, 1} becomes {positive, 3},{neutral, 1}),{negative, 1}). After, the results are collected.

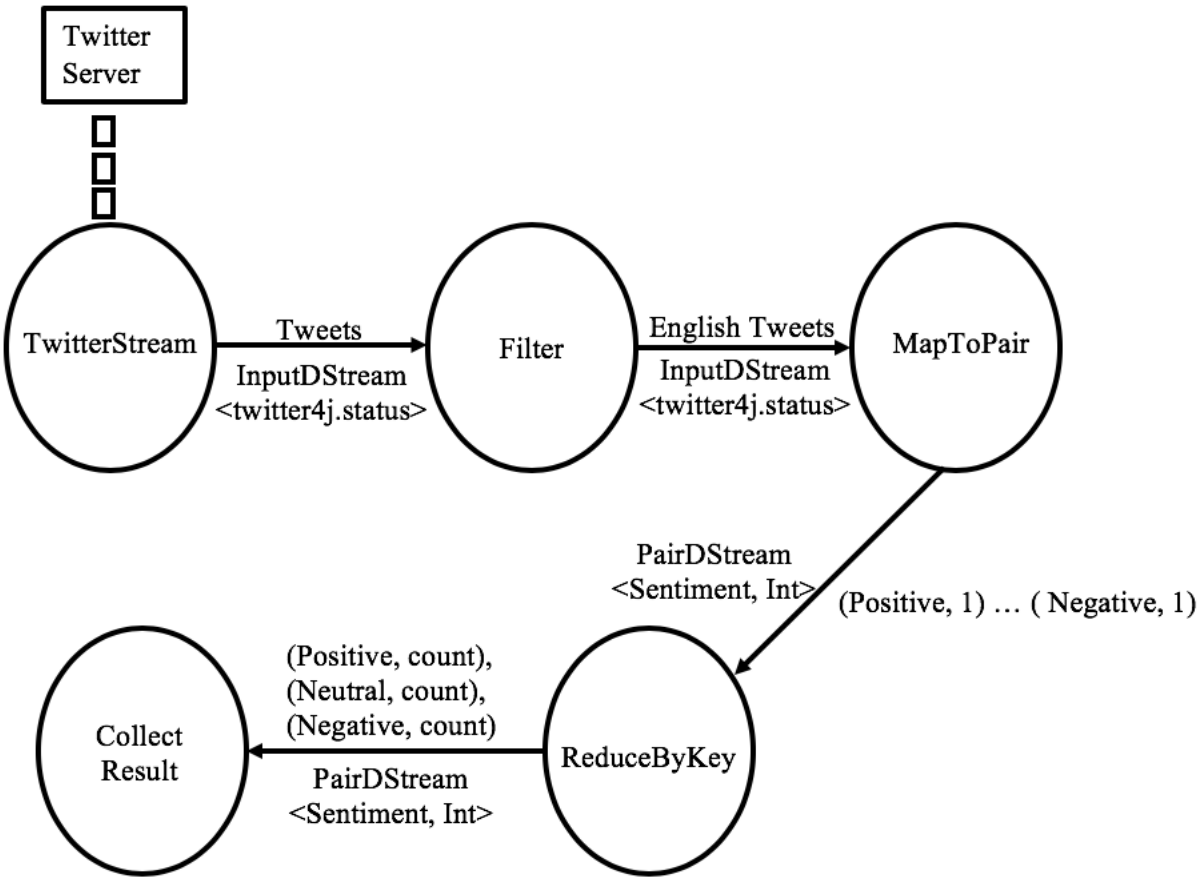

Figure 4.30: DAG for Twitter Sentiment Analysis Application

The experiment was conducted on a 4core (8 logical CPUs) Intel i7-3720QM machine. The spark streaming application was lauched with access to 3 logical CPUs. One for the receiver and

two for the executors. The experiment was run on May 12, 2019 between 3pm and 3am EST. Any tweet contain the name Trump was treated as a high priority tweet. The Results were recorded and divided into 15 minute sections. The average of the 15minute sections is displayed in the graphs below. The performance metrics used to measure the performance of the system are average end-to-end latency, average job queuing latency and average scanning latency. These are defined in Section 4.1.3

Figure 4.31 shows the average end-to-end latency of processing tweets. The resutls show two lines HP and LP, where HP coressponds to high priority messages and LP coressponds to low priority messages. The results in Figure 4.31 show that between 3pm and 7pm the average end-to-end latency of LP tweets is significantly higher than HP tweets. Between 7 and 9pm the average end-to-end latency of HP tweets reduces and becomes comparable to that of LP tweets. Between 9 and 11pm the average end-to-end latency of LP tweets increases again, but the average end-to-end latency of LP tweets remains relatively stable. After 11pm the average end-to-end latency of LP reduces and becomes similar to the average end-to-end latency of HP tweets. This behavour continues until 3am. One possible explanation for the spikes in LP latency between 3pm to 7pm, and 9pm to 11pm is that 3 to 7pm is around the time most people in North America finish school and work. So there would be an increase in twitter activity. Between 7 and 9pm most, people in North America would be eating dinner, so twitter activity would reduce. Twitter activity would increase after dinner and reduce after 11pm when most people would have gone to bed. The results in Figure 4.32 are in line with these observations. The figure plots the number of tweets received for each 15minute window for the duration of the experiment. The results show high twitter activity between 3 to 7pm and 9 to 11pm. An increase in tweets would increase resource contention and as a result job queuing latencies will increase. The impact of DDPS on performance is more

prominent when resource contention is higher. When there is high frequency of tweets there is a high resource contention and the average end-to-end latency for HP tweets is significantly lower than that of the LP tweets. When there is a low frequency of tweets the resource contention is low and the average end-to-end lantecy of HP tweets is comparable to that of LP tweets.

Figure 4.33 shows the average job queuing latencies for the experiement. The average job queuing latencies increases between 3pm and 7pm and between 9pm and 11pm. This is inline with Figure 4.31 in which the difference between the avaerage end-to-end latency achieved with HP and LP tweets is highest between 3 and 7pm and between 9 and 11pm.

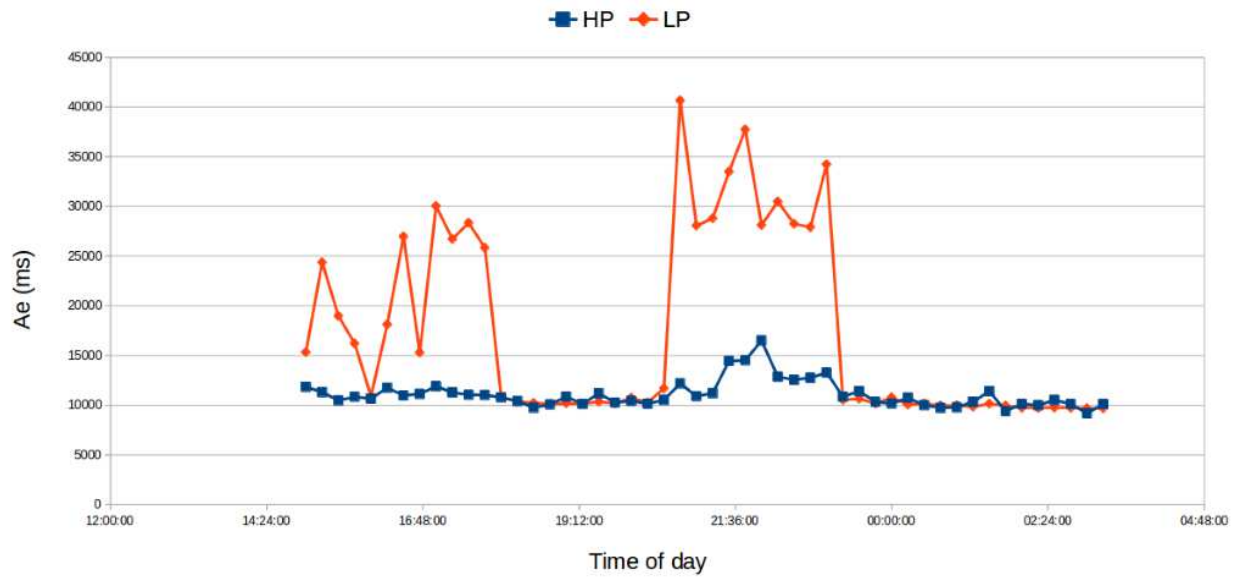

Figure 4.31: Sentiment Analysis Scenario: Average End-to-end Latency

Figure 4.34 shows the average scanning latency during the course of the experiment. The figure shows a correlation between the number of messages received during a 15minute window and the average scanning latency within that window. The reason for this is that the time it takes to scan the tweets is proportional to the number of tweets that are in the RDD. As the number of tweets received increases, the average scanning latency increases.

The results in Figure 4.31 show the performance of DDPS in a real world sentiment analysis scenario. It shows that DDPS is able to effectively prioritize HP tweets when the input load increases.

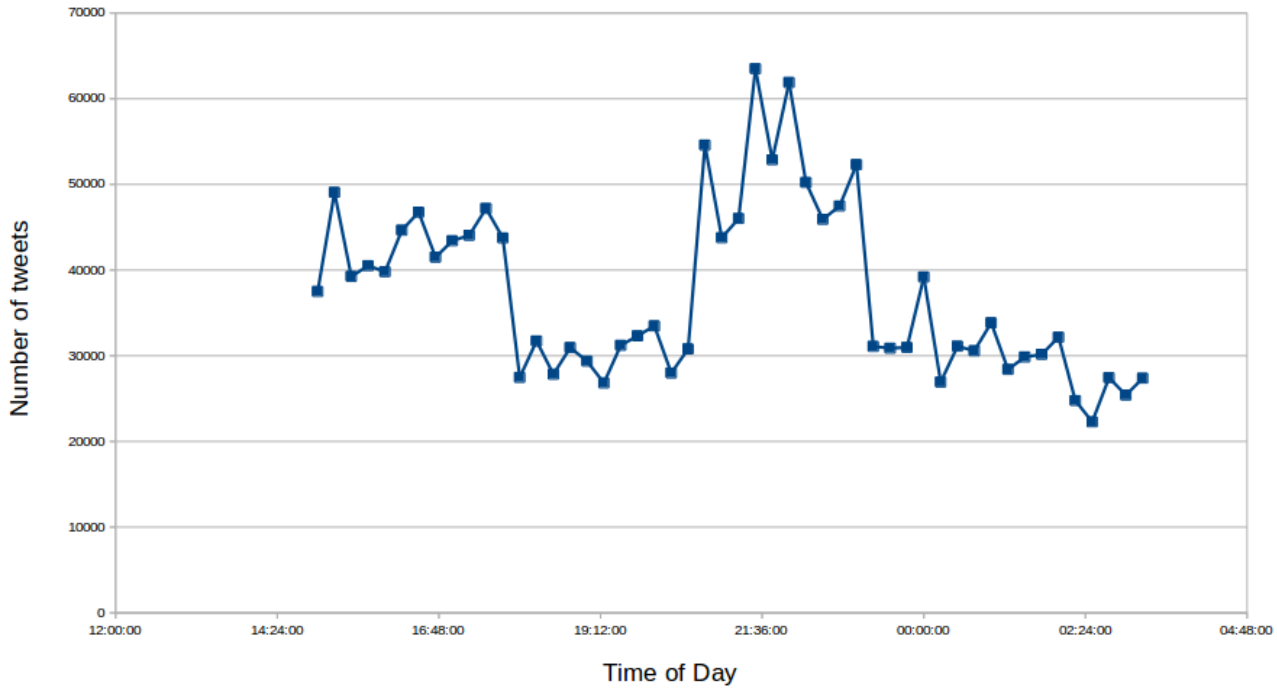

Figure 4.32: Sentiment Analysis Scenario: Number of Tweets

4.3 Summary

This chapter discusses the evaluation of DDPS using the prototype implementation on Spark Streaming. Two scenarios were constructed: the first exercised DDPS with a synthetic workload and the second concerned a real-time sentiment analysis on tweets. The results of the evaluation showed that DDPS is able to give preference to high priority data items. It also showed that the higher the resource contention the more DDPS favoured high priority data items.

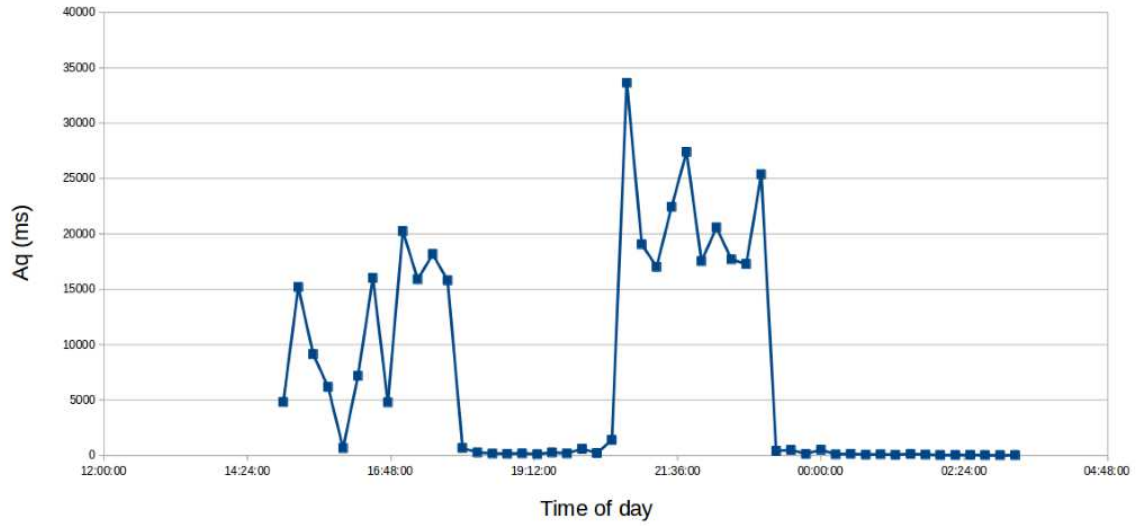

Figure 4.33: Sentiment Analysis Scenario: Average Job Queuing Latency

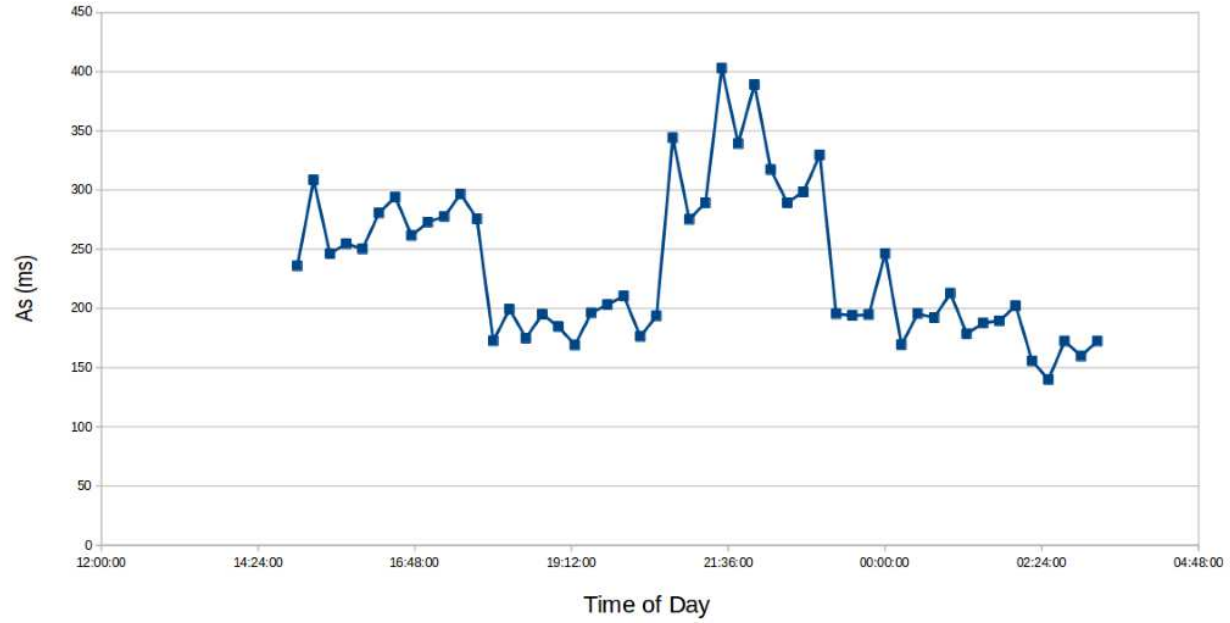

Figure 4.34: Sentiment Analysis Scenario: Average Scanning Latency

Chapter 5: Summary and Conclusions

This chapter presents a summary of the thesis and provides concluding remarks. A summary of the thesis research is presented in Section 5.1. Section 5.2 presents the key conclusions while direction for future research are discussed in Section 5.3.

5.1 Summary

One of the main problems with the state of the art is that it does not provide mechanisms for prioritizing certain input data items over others. The ability to prioritize input data items is essential in cases where one needs to ensure that important data is always processed with low latency. Ensuring this becomes more challenging under bursty loads as the streaming system may be temporarily overwhelmed resulting in large queuing delays. This thesis addresses this challenge by proposing a scheduling technique called the Data Driven Priority Scheduler that can be used on stream processing engines where certain input data items can be given higher priority than others. This thesis implements the DDPS on the Spark Streaming platform. The default scheduler that comes with Spark Streaming is a FIFO scheduler. In cases where the input load suddenly increases, the latency for processing input data items will increase due to resource contention. If there are input data items that are more important than others, these data items will also suffer from increased processing latencies because the FIFO scheduler does not prioritize the processing of important data items over less important data items. To solve this problem, this thesis introduces a priority-based scheduling technique in which the priority of an input data item is based on its content. The technique assigns priorities to input data items such that higher priority data items are processed before lower priority data items. A user can define the priority of input data types by creating priority mappings. A priority mapping is a key value pair where the key is feature or characteristic (e.g. words, phrases, locations, etc.) of the input data item, and the value is an integer

representing the priority of the input data item. Once the user has constructed the priority mappings, it is then supplied to the streaming engine. At runtime the streaming engine groups input data into jobs. The priority of each job is calculated based on the collective priority of all the data items that it contains. The job manager ensures that jobs with highest priority are processed first.

To evaluate the performance of DDPS a prototype of DDPS is constructed using the Spark Streaming platform. Performance analysis is performed both on a single machine and on a cloud comprising multiple nodes. The conclusions derived from the results of the experiments performed in the prototype are presented in the following section.

5.2 Conclusions

The performance analysis of the prototype demonstrates the efficacy of the DDPS scheduler. The experiments performed on the single machine and on the cloud lead to a similar set of conclusions regarding the effect of various system and workload parameters on performance and are described in the following subsections each of which focuses on experiments corresponding to a different scenario. The first scenario exercises DDPS under a synthetic workload and compares DDPS with the FIFO scheduler. The second scenario exercises DDPS with live sentiment analysis of Twitter tweets and analyzes the effectiveness of DDPS on this real-world scenario. Key performance metrics are such as average end-to-end latency, average job queueing latency and average scanning latency are measured for each of these scenarios.

5.2.1 The Scenario Based on a Synthetic Workload

In this scenario a message sending application is run in parallel with the Spark Streaming application. The message sending application sends messages to the Streaming engine. The workload parameters are the mean arrival rate of messages, coefficient of variation of inter-arrival times, message length, message service time, batch duration, and the number of cores. The

experiments vary one workload parameter at a time and measure the following performance metrics: average latency, average job queuing time, scheduling overhead and average scanning time. The key insights into system behaviour and performance resulting from this performance analysis are presented:

- High priority messages are processed with a lower average latency than low priority messages when running with DDPS. As the average arrival rate increases, the difference between the average latency of LP and HP messages gets larger. With the default FIFO scheduler high and low priority messages are processed with the same average latency. When running DDPS with multiple priority levels, the highest priority messages have the lowest average latency, and the lowest priority messages have the highest processing latency.
- The average job queuing time increases as the arrival rate increases. This behaviour occurs with both the DDPS and the FIFO scheduler. As the arrival rate increases contention for resources increases leading to an increase in in the average job queuing time. When the average job queuing time is high, the difference between the average processing latency of high and low priority messages is also high.
- The average scanning latency for messages in an RDD increases as the arrival rate increases. The scheduling overhead (which comprises of scanning messages as well as allocating and freeing memory blocks, associating RDDs with jobs, re-ordering the job queue, etc.) also increases also as the arrival rate increases.
- The experiments conducted show that the smaller the batch duration the larger is the difference between that average end-to-end latency for HP and LP messages when running

with DDPS. It also shows that the when the batch duration increases the average end-to-end-latency for both high and low priority messages also increases.

- Increasing C_a increases the average latency of processing both HP and LP messages. However, for any given value of C_a the average latency of processing LP messages is always larger than that of the HP messages. As C_a increases, so does the difference between the average latency of LP and HP messages. Increasing C_a also increases the average job queuing latency.
- Increasing the number of cores reduces the average processing of both high and low priority messages. As the number of cores increases, the difference between the average latencies of low and high priority messages reduces. Increasing the number of cores reduces the average end-to-end latencies of all messages.

5.2.2 The Scenario Based on Live Sentiment Analysis

In this scenario a streaming application was constructed to perform sentiment analysis of live tweets. The application first constructed a classifier based on a large dataset of historical tweets. The classifier was used at runtime to evaluate the sentiment of tweets. The word "Trump" was used to signify a high priority tweet. Key insights into system behavior and performance are summarized:

- The rate at which tweets arrived varied over time. When there was a sudden increase in tweets DDPS was able to prioritize tweets contained the high priority keyword. This resulted in a high priority tweets being processed with a lower average latency than low priority tweets. When the tweet arrival rate of tweets reduced the difference between the average latency of processing high and low priority tweets also reduced.

- When the tweet arrival rate was high the measured average job queuing latency was also high.
- As the tweet arrival rate increased, the average scanning time also increased.

5.3 Future Research

This section suggests topics for further investigation.

- Investigate the impact of additional workload and system parameters on DDPS system performance.
- A variable batch duration is a batch duration that can be dynamically increased or decreased while the streaming application is running. Variable batch durations may be able to improve DDPS ability to process high priority data items with low latency. In order to give immediate attention to high priority data items one can collapse the batch duration as soon as a high priority data item arrives. By reducing the batch duration this will reduce the batch delay for the items in the batch. Also, reducing the batch duration will reduce the number of data items contained in a job. A high priority job with a small number of data items will be processed faster. Investigation of such a variable batch duration for DDPS warrants further investigation.
- DDPS prioritizes high priority data items at the expense of low priority data items. This can lead to starvation of low priority data items. Investigation of aging techniques for low priority data items (similar to OS task schedulers) such that the starvation of low priority data times can be bounded based on user requirements forms an important direction for future research.
- DDPS requires users to input priority mappings in order to determine which data items should be given precedence. Investigation into machine learning techniques that enable

DDPS to learn how to distinguish high priority data items from low priority data items automatically would be ease the burden placed on users.

- This thesis demonstrated the efficacy of DDPS with a live twitter sentiment analysis scenario. A similar experiment with Internet of Things (IOT) technology, stock trading applications or geo physics log monitoring applications would strengthen the case for DDPS and may uncover other areas of improvement for this scheduling technique.
- Investigate dynamic priority mapping techniques where priorities for keywords can be added or modified in real-time.

References

[1] D. Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety," [Online]. Available: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. [Accessed August 2019].

[2] Apache Spark, "Spark Overview," [Online]. Available: https://spark.apache.org/docs/latest/. [Accessed February 2018].

[3] Apache Storm, "Apache Storm," [Online]. Available: http://storm.apache.org/. [Accessed February 2019].

[4] E. Thompson, "Use cases for Real Time Stream Processing Systems," [Online]. Available: https://www.linkedin.com/pulse/use-cases-real-time-stream-processing-systems-emma-thompson. [Accessed January 2018].

[5] R. D. Desai, "Sentiment Analysis of Twitter Data," in *Proceedings of the 2nd International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, 2018, pp. 114-117.

[6] V. A and S. S.S, "Sentiment Analysis of Twitter Data: A Survey of Techniques," *International Journal of Computer Applications,* vol. 139, no. 11, pp. 5-15, 2016.

[7] D. A. Bermingham, "Sentiment Analysis Use Cases," [Online]. Available: http://www.isin.ie/assets/38/903B8F83-111B-4DCA-A84606B4F6E557B6_document/0915_Sentiment_Analysis_Dr._Adam_Bermingham.pdf. [Accessed March 2018].

[8] F. Neri, C. Aliprandi, F. Capeci, M. Cuadros and T. By, "Sentiment Analysis on Social Media," in *Proceedings of the Advances in Social Network Analysis and Mining (ASONAM)*, Istanbul, Turkey, 2012, pp. 919-926.

[9] A. Bermingham, M. Conway, L. McInerney, N. O'Hare1 and A. F. Smeaton, "Combining Social Network Analysis and Sentiment Analysis to Explore the Potential for Online Radicalisation," in *Proceedings of the International Conference on Advances in Social Network Analysis and Mining*, Athens, Greece, 2009, pp. 231 - 236.

[10] Y. Zhao and H. Zeng, "The Concept of Unschedulability Core for Optimizing Real-Time Systems with Fixed-Priority Scheduling," *IEEE Transactions on Computers,* vol. 68, no. 8, pp. 926-938, 2019.

[11] B. Lohrmann, P. Janacik and K. Odej, "Elastic Stream Processing with Latency Guarantees," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, Columbus, USA, 2015, pp. 399-410.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10)*, Boston, USA, 2010, pp. 10-17.

[13] Apache Spark, "Spark RDD Operations-Transformation & Action with Example," [Online]. Available: https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/. [Accessed February 2018].

[14] W. Huang, L. Meng, D. Zhang and W. Zhang, "In-Memory Parallel Processing of Massive Remotely Sensed Data Using an Apache Spark on Hadoop YARN Model," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing,* vol. 10, no. 1, pp. 3-19, 2016.

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Communications of the ACM,* vol. 59, no. 11, pp. 56-65, 2016.

[16] Apache Spark, "Spark Streaming Programming Guide," [Online]. Available: http://spark.apache.org/docs/latest/streaming-programming-guide.html. [Accessed February 2018].

[17] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, USA, 2013, pp. 423-438.

[18] Apache Spark, "Scheduling Within an Application," [Online]. Available: https://spark.apache.org/docs/2.0.0/job-scheduling.html#scheduling-within-an-application. [Accessed March 2018].

[19] The Apache Software Foundation, "Apache Mesos," [Online]. Available: http://mesos.apache.org/. [Accessed July 2018].

[20] Apache Hadoop, "Apache Hadoop YARN," [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. [Accessed July 2018].

[21] M. V. Mäntylä, D. Graziotin2 and M. Kuutila, "The Evolution of Sentiment Analysis - A Review of Research Topics, Venues, and Top Cited Papers," *Computer Science Review,* vol. 27, no. 1, pp. 16-32, February 2018.

[22] J. Serrano-Guerrero, J. A. Olivas, F. P. Romero and E. Herrera-Viedma, "Sentiment Analysis: A Review and Comparative Analysis of Web Services," *Information Sciences,* vol. 311, no. 1, pp. 18-38, 2015.

[23] D. D. Droba, "Methods Used for Measuring Public Opinion," *American Journal of Sociology,* vol. 37, no. 3, pp. 410-423, 1931.

[24] J. M. Wiebe, "Recognizing Subjective Sentences: A Computational Investigation of Narrative Text," PhD Thesis, Department of Computer Science, State University of New York at Buffalo, 1990.

[25] H. Wang, C. Ma and L. Zhou, "A Brief Review of Machine Learning and Its Application," in *Proceedings of the International Conference on Information Engineering and Computer Science*, Wuhan, China, 2009, pp. 1-4.

[26] S. Angra and S. Ahuja, "Machine Learning and its Applications: A Review," in *Proceedings of the International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, Chirala, India, 2017, pp. 57-60.

[27] S. Das, A. Dey, A. Pal and N. Roy, "Applications of Artificial Intelligence in Machine Learning: Review and Prospect," *International Journal of Computer Applications,* vol. 115, no. 9, pp. 31-41, 2015.

[28] S. Wu and P. A. Flach, "Feature Selection with Labelled and Unlabelled Data," in *Proceedings of ECML Workshop on Integration and Collaboration Aspects of Data Mining, Decision Support and Meta-Learning (PKDD'02)*, Helsinki, Finland, 2002, pp. 156-167.

[29] H. Wang, Q. Zhang, J. Wu, S. Pan and Y. Chen, "Time Series Feature Learning with Labeled and Unlabeled Data," *Pattern Recognition,* vol. 89, no. 1, pp. 55-56, 2019.

[30] T. Lorido-Botran, J. Miguel-Alonso and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing,* vol. 12, no. 4, pp. 559-592, 2014.

[31] A. Ali-Eldin, M. Kihl, J. Tordsson and E. Elmroth, "Efficient Provisioning of Bursty Scientific Workloads on the Cloud using Adaptive Elasticity Control," in *Proceedings of the 3rd workshop on Scientific Cloud Computing (ScienceCloud '12)*, Delft, Netherlands, 2012, pp. 31 - 40.

[32] H. S. B. L. M. I. G. Ghanbari, "Exploring Alternative Approaches to Implement an Elasticity Policy," in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, Washington DC, USA, 2011, pp. 716-723.

[33] A. Biswas, S. Majumdar, B. Nandy and A. El-Haraki, "An Auto-Scaling Framework for Controlling Enterprise Resources on Clouds," in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, China, 2015, pp. 971-980.

[34] A. Biswas, S. Majumdar, B. Nandy and A. El-Haraki, "Automatic Resource Provisioning: A Machine Learning Based Proactive Approach," in *Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science*, Singapore, Singapore, 2014, pp 168-173.

[35] A. Biswas and S. M. N. El-Haraki, "A Hybrid Auto-scaling Technique for Clouds Processing Applications with Service Level Agreements," *Journal of Cloud Computing,* vol. 6, no. 29, pp. 1-14, 2017.

[36] B. Lohrmann, P. Janacik and O. Kao, "Elastic Stream Processing with Latency Guarantees," in *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, Columbus, USA, 2015, pp. 399-410.

[37] T. Heinze, V. Pappalardo, Z. Jerzak and C. Fetzer, "Auto-scaling Techniques for Elastic Data Stream Processing," in *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops*, Chicago, USA, 2014, pp. 318-321.

[38] T. M. Ahmed, F. H. Zulkernine and J. R. Cordy, "Proactive Auto-Scaling of Resources for Stream Processing Engines in the Cloud," in *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16)*, Toronto, Canada, 2016, pp. 226-231.

[39] X. Liao, Y. Huang, L. Zheng and H. Jin, "Efficient Time-Evolving Stream Processing at Scale," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, no. 1, pp. 1-14, 2019.

[40] L. J. d. l. C. Llopis, A. V. Rodas, E. S. Gargallo and M. A. Igartua, "Load Splitting in Clusters of Video Servers," *Computer Communications,* vol. 35, no. 8, pp. 993-1003, 2012.

[41] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing and S. Zdonik, "Scalable Distributed Stream Processing," in *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, US, 2003, pp. 500-511.

[42] T. Das, Y. Zhong, I. Stoica and S. Shenker, "Adaptive Stream Processing using Dynamic Batch Sizing," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*, Seatle, USA, 2014, pp. 1-13.

[43] R. Birke, M. Bjorkvist, E. Kalyvianaki and L. y. Chen, "Meeting Latency Target in Transient Burst: a Case on Spark Streaming," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, Vancouver, Canada, 2017, pp. 149-158.

[44] A. L. S. i. N. Monitoring, "Pere Barlet-Ros; Gianluca Iannaccone; Josep Sanjuas-Cuxart; Diego Amores-Lopez; Josep Sol´e-Pareta," in *Proceedings of the USENIX Annual Technical Conference (ATC'07)*, Santa Clara, USA, 2007, pp. 47 - 50.

[45] D. Cheng, Y. Chen, X. Zhou, D. Gmach and D. Milojicic, "Adaptive Scheduling of Parallel Jobs in Spark Streaming," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, Atlanta, USA, 2017, pp. 1 - 9.

[46] P. Bellavista, A. Corradi, A. Reale and N. Ticca, "Priority-Based Resource Scheduling in Distributed Stream Processing Systems for Big Data Applications," in *Proceedings of the 7th IEEE International Conference on Utility and Cloud Computing*, London, U.K, 2014, pp. 363-370.

[47] N. Li and Q. Guan, "Deadline-aware Complex Event Processing Models over Distributed Monitoring Streams," *Mathematical and Computer Modelling,* vol. 55, no. 1, pp. 901-917, 2012.

[48] R. Moffatt, "ATM Fraud Detection with Apache Kafka and KSQL," [Online]. Available: https://www.confluent.io/blog/atm-fraud-detection-apache-kafka-ksql. [Accessed August 2019].

[49] Apache Spark, "Spark Streaming API," [Online]. Available: https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.streaming.package. [Accessed July 2018].

[50] Oracle, "Javadocs java.util.Collection," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html. [Accessed July 2019].

[51] Apache Spark, "Apache Spark API," [Online]. Available: https://spark.apache.org/docs/2.0.1/api/java/org/apache/spark/rdd/RDD.html. [Accessed July 2019].

[52] Oracle, "Javadocs: java.lang.String," [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/String.html. [Accessed April 2019].

[53] Oracle, "Javadocs: java.lang.Comparable," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html. [Accessed April 2019].

[54] T. O. G. B. Specifications, "The Open Group Base Specifications," [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16. [Accessed July 2019].

[55] Oracle, "Javadocs: java.util.concurrent.PriorityBlockingQueue," [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html. [Accessed April 2019].

[56] Oracle, "Javadocs: java.lang.Math," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html. [Accessed May 2019].

[57] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi and R. J. Meijer, "Dynamically Scaling Apache Storm for the Analysis of Streaming Data," in *Proceedings of the IEEE First International Conference on Big Data Computing Service and Applications (BigDataService)*, Redwood City, USA, 2015, pp. 154-161.

[58] Y. Zhou, B. C. Ooi, K.-L. Tan and J. Wu, "Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System," *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE,* vol. 42, no. 75, pp. 54-71, 2006.

[59] Amazon AWS, "Amazon EC2 Instance Types," [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed May 2019].

[60] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker and I. Stoica, "Discretized streams: Fault-tolerant Streaming Computation at Scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farminton, Pennsylvania, 2013, pp. 423-438.

[61] J. Medhi, Stochastic Models in Queueing Theory, San Diego, USA: Elsevier Science & Technology, 2002.

[62] T. Yan and G.-L. Gao, "Research on the Methods of Chinese Text Classification using Bayes and Language Model," in *Proceedings of the Chinese Conference on Pattern Recognition*, Beijing, China, 2008, pp. 1-6.

[63] Sentiment140, "Sentiment140," [Online]. Available: http://help.sentiment140.com/for-students. [Accessed May 2019].

[64] K. Korovkinas, P. Danėnas and G. Garšva, "SVM and Na¨ýve Bayes Classification Ensemble Method for Sentiment Analysis," *Baltic Journal of Modern Computing,* vol. 5, no. 4, pp. 398-409, 2017.

[65] W. Park, Y. You and K. Lee, "Detecting Potential Insider Threat: Analyzing Insiders' Sentiment Exposed in Social Media," *Security and Communication Networks,* vol. 1, no. 1, pp. 1 - 8, 2018.

[66] S. U. S. K. Young Gyo Jung College of Software, K. T. Kim, B. Lee and H. Y. Youn, "Enhanced Naive Bayes Classifier for Real-time Sentiment Analysis with SparkR," in *Proceedings of the International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, South Korea, 2016, pp. 141-146.

[67] S. B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques," in *Proceedings of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, Amsterdam, Netherlands, 2007, pp. 3-24.

[68] H. MEDHAT, "Apache Spark Streaming Tutorial: Identifying Trending Twitter Hashtags," [Online]. Available: https://www.toptal.com/apache/apache-spark-streaming-twitter. [Accessed May 2019].

[69] S.-H. Kim and W. Whitt, "Statistical Analysis with Little's Law," *Operations Research Journal,* vol. 61, no. 4, pp. 1030-1045, 2013.

[70] M. C. T. D. A. D. J. M. M. M. M. F. S. S. a. I. S. M. Zaharia, "Resilient distributed datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, USA, 2012, pp. 15-29.

[71] A. Muhammad, N. Wiratunga and R. Lothian, "Contextual Sentiment Analysis for Social Media Genres," *Knowledge-Based Systems,* vol. 108, no. Complete, pp. 92-101, 2016.

Appendix A: Performance Measurements on the Cloud

A.1 Workload Parameters

Table A.1: Workload Parameters on Amazon EC2 Cloud

Input Parameters	Possible values	Default value
Mean Arrival Rate (λ) (msg/s)	22 … 34	28
Coefficient of Variation of Inter-arrival Time (C_a)	1 … 7	1
Batch Duration (BD) (s)	1.2 … 3.5	2
Service Time (S) (ms)	242	242
Percentage of High Priority Messages (PHP)	0.1	0.1
Percentage of Medium Priority messages (PMP)	0.5	0.5 with three priority levels, 0 otherwise
Message Length (L)	2860	2860
Number of Priority Mappings (K)	1, 2	1

A.2 Effect of Mean Arrival Rate

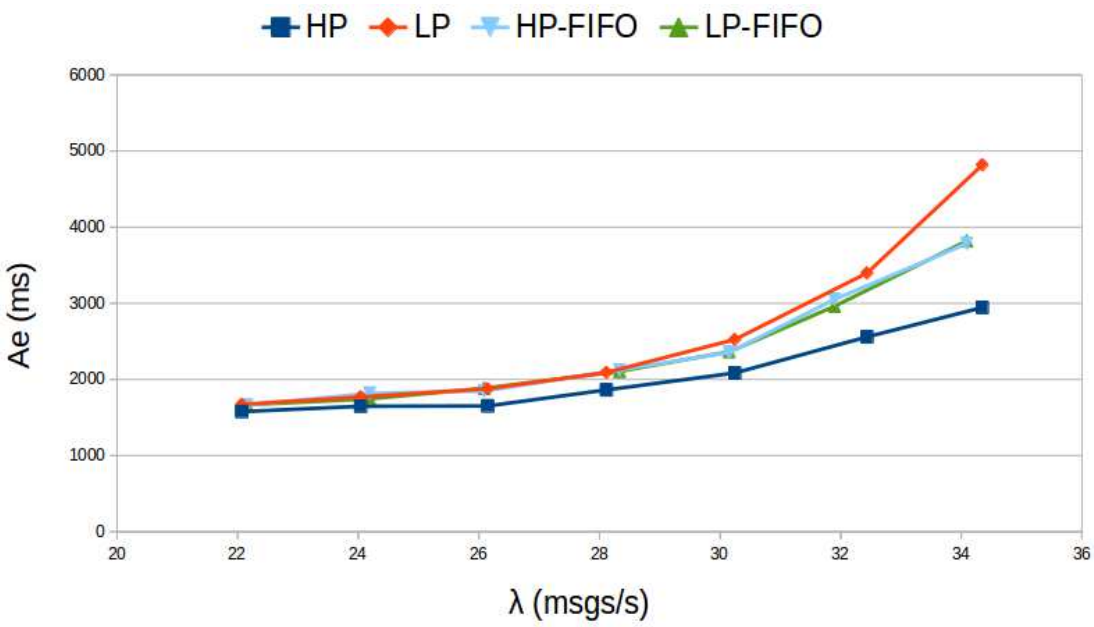

Figure A.1: Cloud: Effect of λ on average End-to-end latency

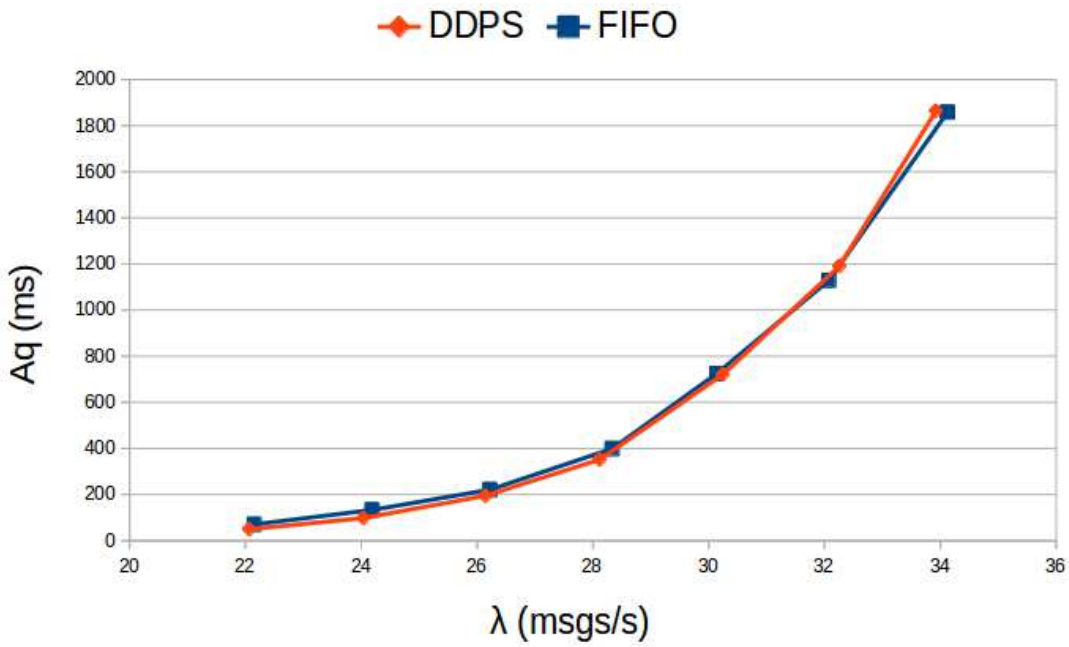

Figure A.2: Cloud: Effect of λ on Average Job Queuing Latency

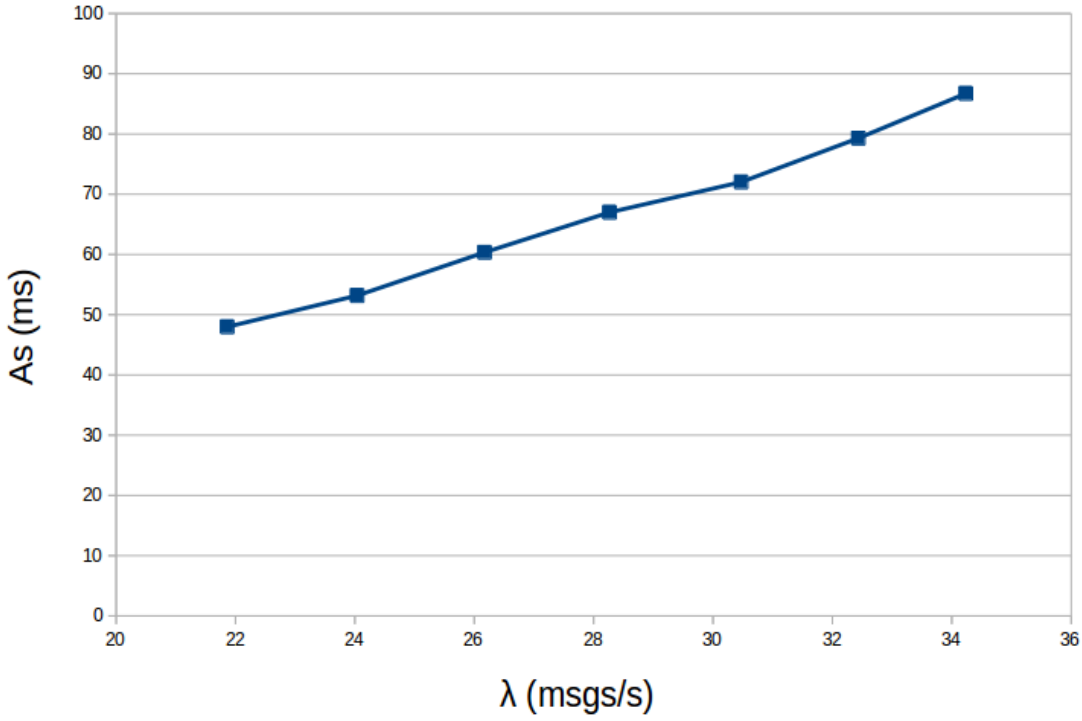

Figure A.3: Cloud: Effect of λ on Average Scanning Latency

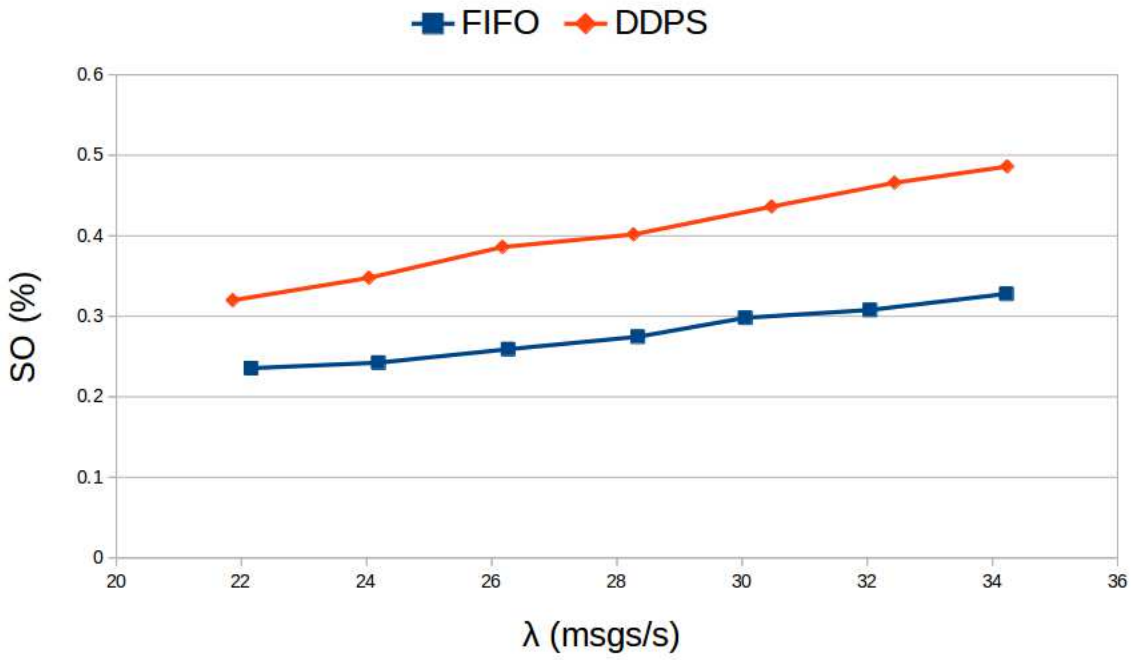

Figure A.4: Cloud: Effect of λ on Average Scheduling Overhead

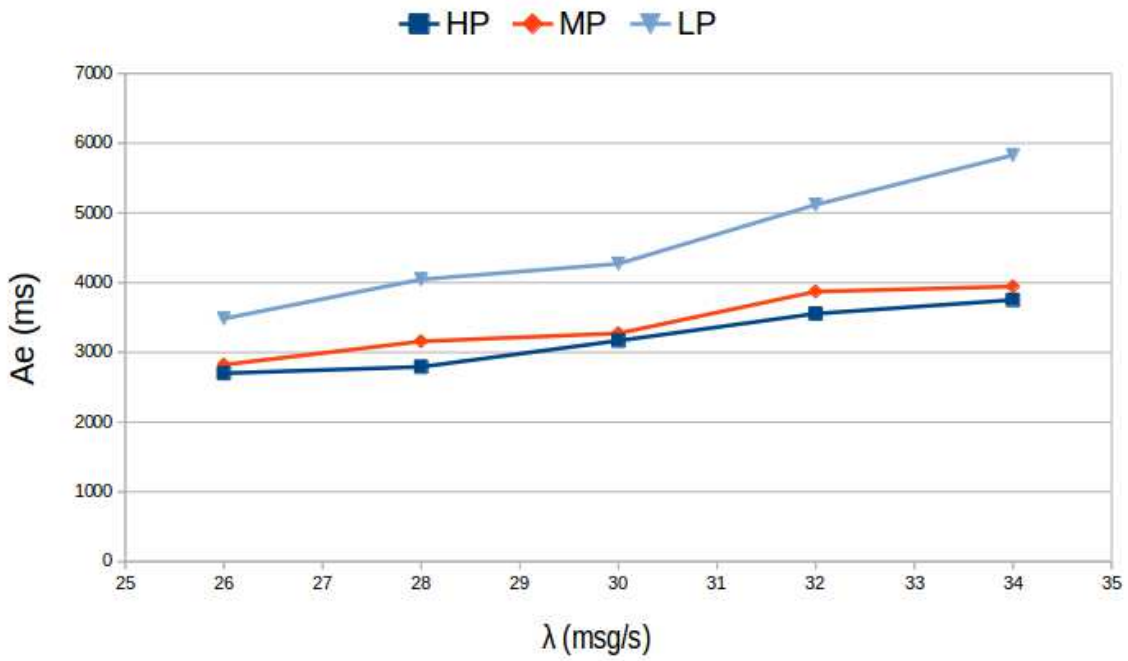

Figure A.5: Cloud: Effect of λ with 3 Priority Levels

A.3 Effect of Batch Duration

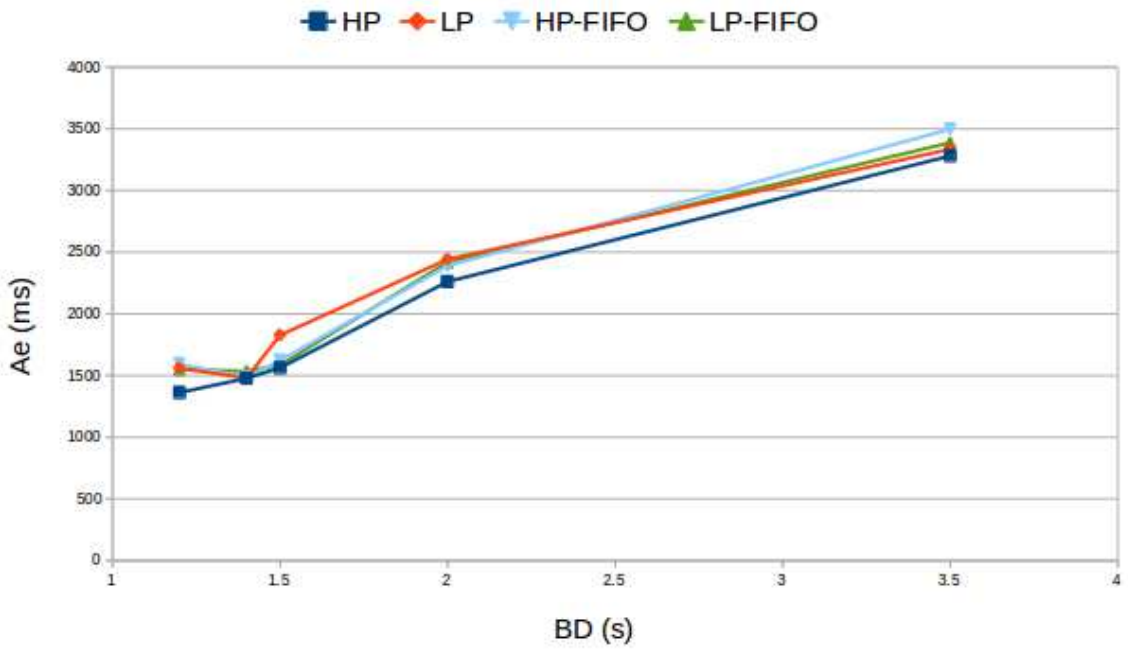

Figure A.6: Cloud: Effect of Batch Duration on Average End-to-end Latency

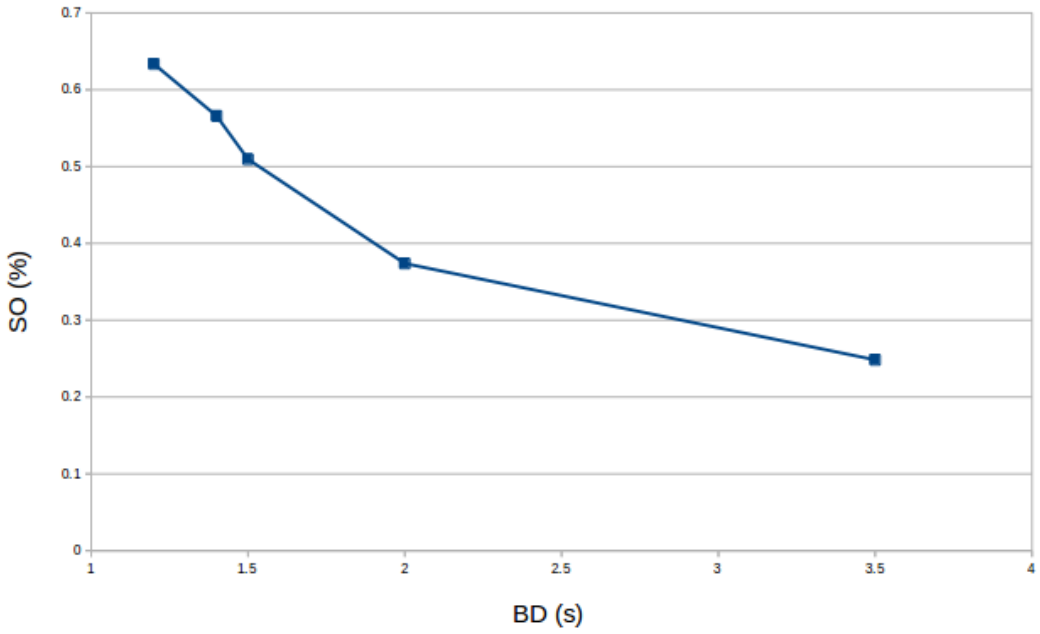

Figure A.7: Cloud: Effect of Batch Duration on Scheduling Overhead

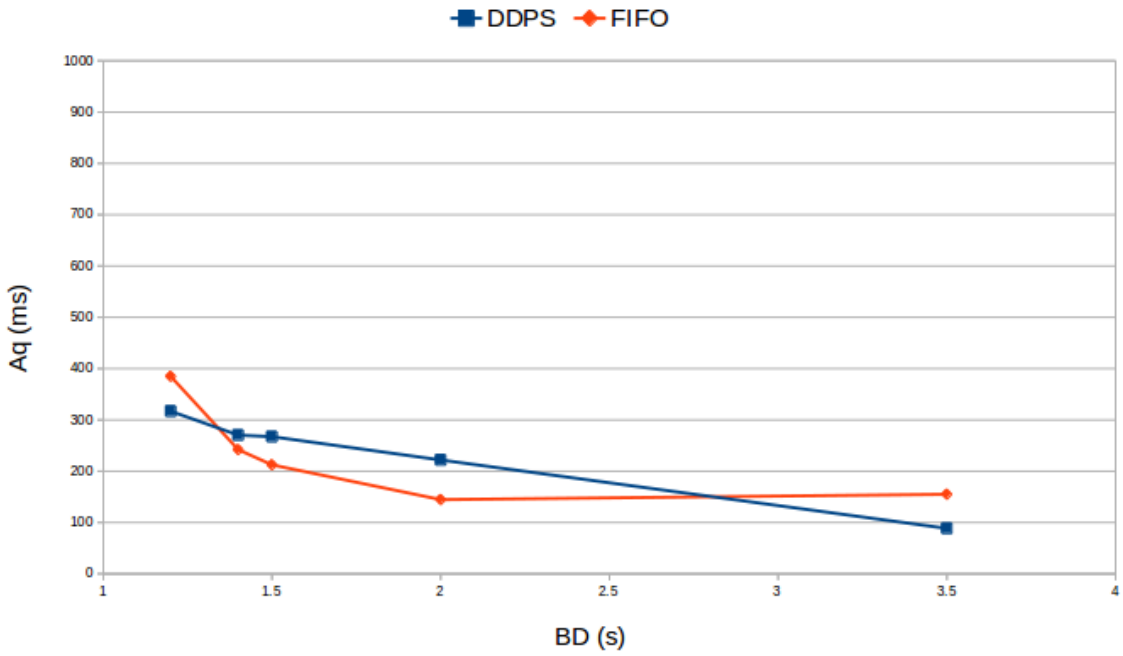

Figure A.8: Cloud: Effect of Batch Duration on Average Job Queuing Latency

A.4 Effect of Coefficient of Variation of Interval Time

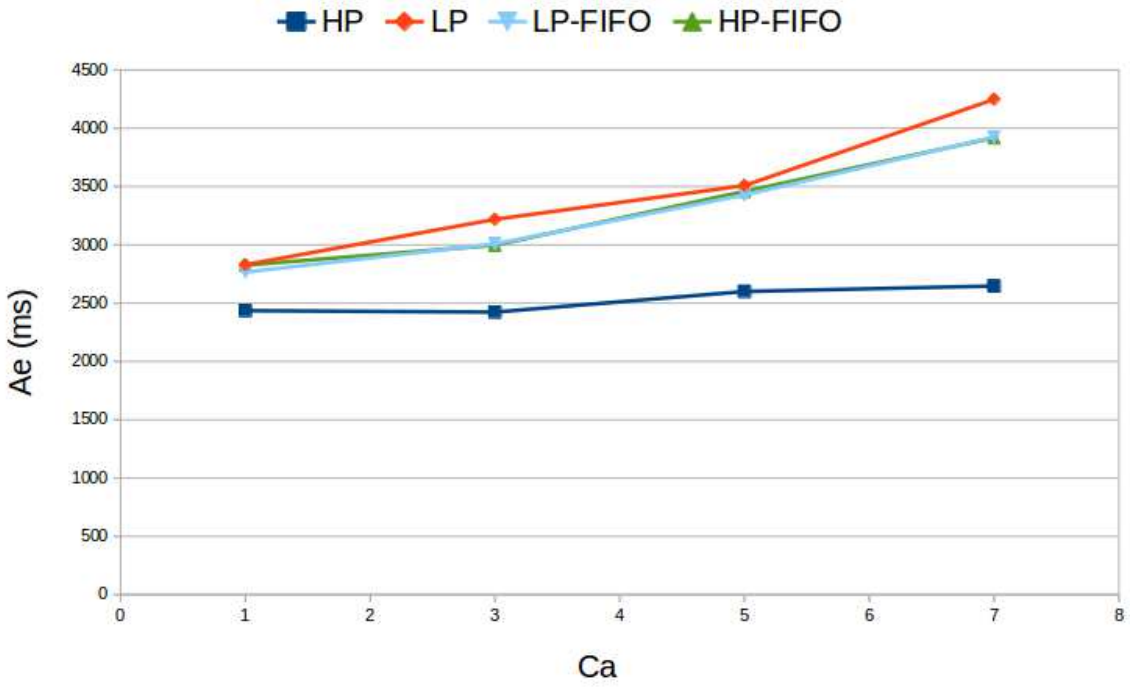

Figure A.9: Cloud: Effect of C_a on Average End-to-end Latency

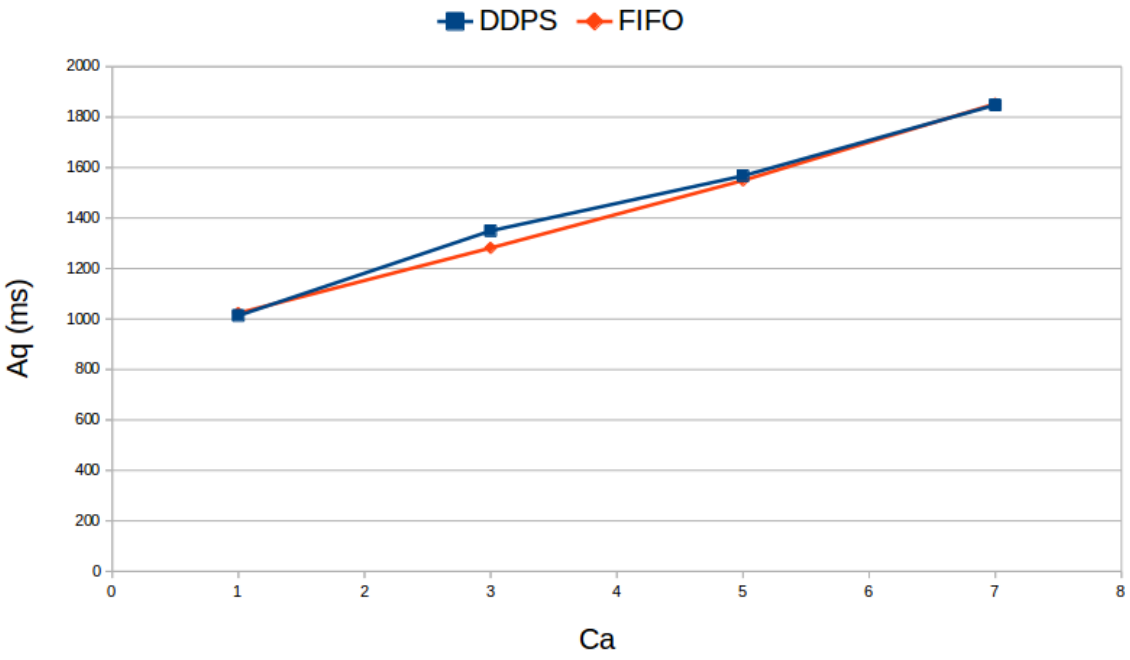

Figure A.10: Cloud: Effect of C_a on Average Job Queuing Latency

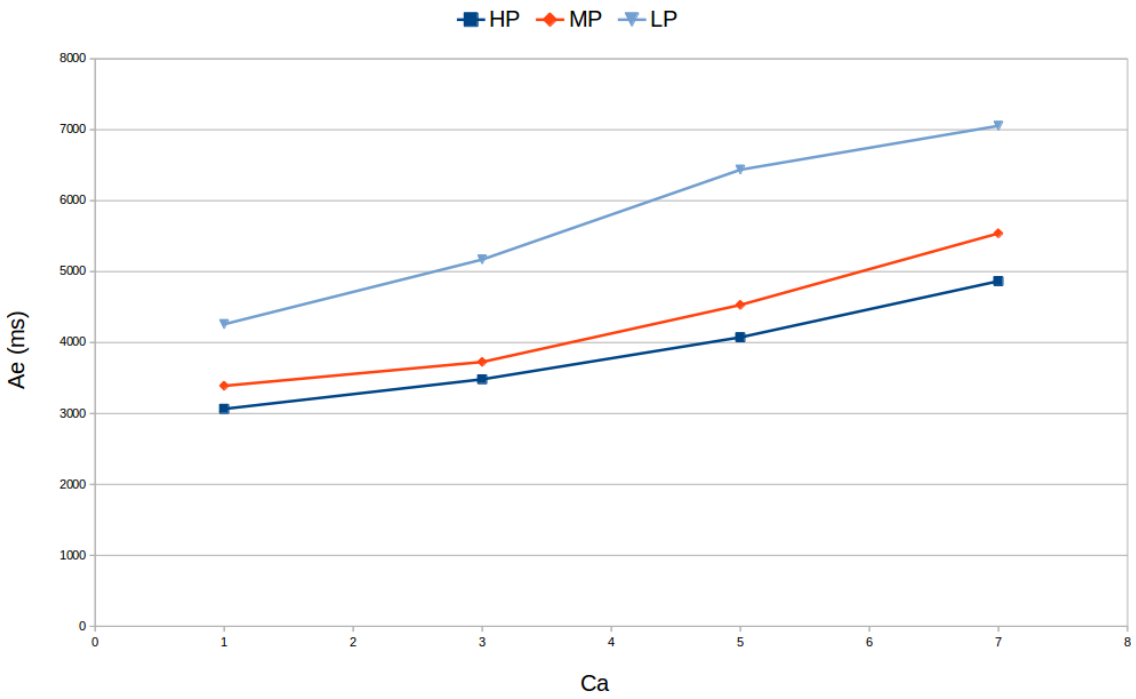

Figure A.11: Cloud: Effect of C_a with 3 Priority Levels