Thesis

Real Time Stream Processing for Internet of Things and Sensing Environments

Submitted by

Thilina Hewa Raga Munige

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2015

Master's Committee:

Advisor: Shrideep Pallickara

Sangmi Pallickara
Daniel Turk

Abstract

Real Time Stream Processing for Internet of Things and Sensing Environments

Improvements in miniaturization and networking capabilities of sensors have contributed to the proliferation of Internet of Things (IoT) and continuous sensing environments. Data streams generated in such settings must keep pace with generation rates and be processed in real time. Challenges in accomplishing this include: high data arrival rates, buffer overflows, context-switches during processing, and object creation overheads.

We propose a holistic framework that addresses the CPU, memory, network, and kernel issues involved in stream processing. Our prototype, Neptune, builds on the Granules cloud runtime and leverages its support for scheduling packets and communications based on publish/subscribe, peer to peer, and point-to-point. The framework maximizes bandwidth utilization in the presence of small messages via the use of buffering and dynamic compaction of packets based on their entropy. Our use of thread-pools and batched processing reduces context switches and improves effective CPU utilizations. The framework alleviates memory pressure that can lead to swapping, page faults, and thrashing through efficient reuse of objects. To cope with buffer overflows we rely on flow control and throttling the preceding stages of a processing pipeline. Our correctness criteria included deadlock/livelock avoidance, and ordered and exactly-once processing.

Our benchmarks demonstrate the suitability of the Granules/Neptune combination and we contrast our performance with Apache Storm, the dominant stream-processing framework developed by Twitter. At a single node, we are able to achieve a processing rate of $\sim$2 million

stream packets per-second. In a distributed cluster setup, we are able to achieve a processing rate of ∼100 million stream packets per-second with a near-optimal bandwidth utilization.

Acknowledgements

This research is supported by a grant from the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Application of wireless sensor networks and related technologies to efficiently measure and monitor a wide spectrum of environments from weather and ecology to smart households to military battlefields has presented a novel set of challenges to the traditional data processing paradigm. Increasing numbers of ubiquitous sensors, actuators and embedded communication hardware that are seamlessly integrated with the environment surrounding us along with the information systems to store and process voluminous data produced by these monitored environments form a communicating-actuating network, which is called the Internet of Things (IoT) [1].

Middleware for on-demand storage and data analytics is considered a key element in an IoT reference architecture [1]. Stream processing systems [2–9] and batch processing systems [10–12] are two key components in implementing this middleware layer. Being two approaches that are contrasted often with each other, low latency processing or near-realtime processing is one advantage offered by stream processing systems over batch processing systems. This is a key requirement for most of the IoT use cases such as triggering actuators in realtime upon processing of sensor data.

Stream processing systems are developed for real-time processing of voluminous data arriving through continuous unbounded data streams with varying data rates. A stream processing system provides a managed execution runtime for a set of stream processing jobs at a given time. A stream processing job is comprised of multiple stages where a stage encapsulates a domain specific processing logic to process streaming data received over external input data streams or intermediate streams generated within the stream processing job.

Once processing is completed for a stream data object at a particular stage, it is forwarded to the next stage through a data stream which links the two stages. Therefore the managed execution runtime provided by the stream processing system should support ingesting external data streams into the system, managing life cycles of each stream processing job and their constituent stages, scheduling, and orchestrating data flow between the stages of a stream processing job. Further, a stream processing system should accommodate for the inherent heterogeneity of stages within a stream processing system due to differences in their processing and IO requirements. A stream processing systems should be capable of handling streams with high data rates as well as multiple concurrent data streams. The solution is to engineer stream processing systems towards achieving high throughput. It should be emphasized that a high throughput should be achieved while maintaining acceptable levels of end-to-end latency. Further, for certain IoT use cases low latency processing is more critical than high throughput [13]. Hence a stream processing system should be able to provide the required balance between low latency and high throughput for individual stream processing jobs.

1.1. Research Challenges

Achieving high-throughput stream processing in Internet of Things (IoT) and sensing settings involves several challenges that impact the efficiency of network, CPU, and memory utilization.

(1) Small packets: The packet sizes in IoT settings tend to be very small ($\sim$100 bytes). Since these packets are processed in Ethernet-based clusters, given the small payload sizes of stream packets, a significant portion of each Ethernet packet frame (with

an MTU of 1500 bytes) is left unused. This contributes to lower throughputs since a large portion of the available network bandwidth is not utilized.

(2) Context switches: Packets are typically processed in thread pools with each thread processing a packet at time from a shared queue. The processing performed per-packet is not CPU intensive. However, since packets arrive at a high rate, context-switching costs start to dominate the overall processing costs. This is true even when these packets are being processed in thread-pools where the thread-context switching costs are significantly lower than process context switching costs.

(3) Buffer overflows: Processing in IoT settings is performed in stages, some of which may execute on different machines. End-to-end processing in these settings is determined by the slowest stage. When packets arrive at a stage faster than the rate at which that stage can process, queues build up at these stages leading to buffer overflows, and in some cases, subsequent process crashes.

(4) Object creation: Prior to packets being processed, the raw bytes need to be converted from their serialized representations into objects through which data fields (of different types) can be accessed. Object creation costs in these settings can add up, because the objects retain their memory footprint slightly longer than their processing scope; this is applicable regardless of whether the memory reclamation scheme is implicit (as in Java/C#) or explicit (as in C/C++). There are costs associated with the allocation of objects on the heap as well; as the number of objects and the memory utilization increase, the costs increase as well. In extreme cases, as memory utilization increases page faults and thrashing may occur as well.

Challenges are also exacerbated by the interactions between these issues. For example, as object creations increase there is a processing cost involved in identifying the objects

(that have gone out of scope) to garbage collect. Similarly, lack of flow control may trigger unimpeded object creations and the associated memory and processing overheads.

1.2. Research Questions

Efficient real-time processing of data streams generated in IoT and sensing settings must address the challenges described earlier. These challenges necessitate a holistic solution that addresses the CPU, memory, network, and kernel (context switches and page faults) issues involved in stream processing. As part of this study, we will explore the following research questions.

(1) How do we incorporate support for efficiently expressing multi-stage stream processing?
(2) How can we improve bandwidth utilizations in IoT settings?
(3) How can we reduce context-switching overhead during processing? And, how can we minimize the number of context switches even in cases where the processing is backed by thread pools?
(4) How can we avoid buffer overflows when the processing involves multiple stages? Specifically, How can we identify which stages to throttle and how do we throttle the stages effectively? Such flow controls often have ripple effects with multiple, preceding stages being throttled.
(5) How can we minimize object creation in such stream processing settings?
(6) How can we minimize data volumes in such settings?

A key issue that we also consider is that of correctness. Our proposed solution should not result in dropped or corrupted stream packets. Furthermore, packets must be processed in-order and exactly-once. The trade-off space in this setting encompasses the entire processing

stack involving I/O, memory, and CPU utilization. Consider the case where we try to improve bandwidth utilization: compacting messages using efficient serialization techniques or compression introduces additional CPU costs. Furthermore, sometimes compaction may not be possible; for example, depending on the entropy within individual packets compression rates may be quite low.

1.3. Approach Summary

We present Neptune, a system designed for high throughput stream processing use cases, especially IoT applications. The rationale behind many of the design principles in Neptune is to achieve optimal consumption of resources: network IO, CPU and memory. Neptune provides an intuitive stream processing API and a stream processing graph description model that facilitate implementing a stream processing use case as a collated set of modular stages. The framework will initialize individual stages, establishes communication between stages and manages the life cycle of a stream processing job. Besides these primitive constructs, users can augment a stream processing graph with a degree of parallelism for each stage and stream partitioning schemes in order to scale the stream processing job during the runtime to better utilize the available cluster resources and to achieve desired levels of performance.

Neptune's communication module is optimized for high throughput while maintaining the communication latencies at acceptable levels. Neptune buffers stream data objects at the application layer and transfers a batch of buffered messages together over the network instead of sending individual messages one at a time. In addition to improving the bandwidth usage significantly, application level buffering further helps reducing the number traversals of the network stack and queue contention between worker threads and IO reactor threads. Buffer sizes are configurable per stream processing job which enables optimizing each stream

processing job for achieving either high throughput or low latency. Even with a configuration optimized for high throughput, Neptune manages to maintain a sub-second end to end communication latency in the order of tens of milliseconds for most applications. Application level buffering is streamlined with batch processing to reduce the number of context switches for worker threads as well as to improve the use of instruction cache.

Neptune uses a two-tier thread model comprised of two thread pools for worker threads and IO threads. This simplified model reduces the queue contentions caused by inter-thread communication. We rely on an asynchronous IO model based on Java NIO library [14] and Netty IO library [15] to implement a scalable communication module with a minimum resource usage footprint. Neptune has a conservative object creation scheme that reduces the strain on the garbage collector through the reuse of data structures used for serialization/deserialization process. The advantages are reduced instantiation overhead, efficient serialization/deserialization of stream data objects and reduced number of short-lived objects created during runtime.

Neptune supports backpressure, a flow controlling technique, at the framework level in order to cope with discrepancies between processing rates and data arrival rates at certain stages of a stream processing job. Backpressure throttles the upstream stages to avoid queue buildups and possible memory management issues such as excessive garbage collection cycles at latter stages due to low processing rates.

Neptune has a in-built compression module which can be configured to compress portions of a data stream selectively based on their entropy levels. Despite the additional processing overhead incurred, compression can be useful in settings with limited bandwidth and low entropy data streams.

We evaluate the capabilities of Neptune based on the results obtained through a comprehensive set of experiments. This includes experiments that evaluates the validity of individual design principles used in Neptune as well as experiments that evaluates it as a complete solution for stream processing. Neptune is compared with Apache Storm [2], a widely used stream processing system, when necessary. While some of the design principles are used in both systems, Storm does not employ some of the optimizations implemented in Neptune. As evident from the empirical evaluation, Neptune outperforms Storm in all metrics we have considered which in turn validates the suitability Neptune for stream processing in IoT and sensing environments.

1.4. Thesis Contributions

This thesis presents our experiences in building a real-time stream processing framework for IoT and sensing environments. The main contributions of this thesis relate to the exploration of the trade-off space encompassing CPU, memory, network, and kernel issues in such settings. The framework supports buffering of data streams, object reuse, dynamic compaction, flow controls, thread-pool based batched executions and reduced queuing-based contentions. Specifically, the framework:

(1) Supports expressing and orchestrating multistage stream processing.

(2) Provides support for high throughput stream processing while making efficient use of resources.

(3) Supports a mixture of stream processing jobs each with possibly different latency and throughput requirements within the same runtime.

(4) Incorporates an effective backpressure scheme that manages the complexities resulting from mismatches between processing rates and data arrival rates in the stages comprising a processing graph.

(5) Empirical performance evaluation of several aspects of the framework and a comparison with the dominant stream-processing framework, Storm.

1.5. Paper Organization

The remainder of the thesis is organized as follows. In chapter 2, we introduce Granules, the underlying cloud computing runtime of Neptune, followed by various improvements introduced to Granuels while developing Neptune. Chapter 3 introduces Neptune and the key concepts related to its API and stream processing graph description model. The latter half of the chapter 3 discusses in detail how Neptune achieves high performance stream processing. The evaluation criteria, experiments and results are presented in chapter 4. A discussion on related work is provided in chapter 5 and we conclude the thesis in chapter 6 discussing conclusions and future work.

CHAPTER 2

Granules

Neptune is implemented on top of Granules computing framework [16] which is a streaming based computing runtime for cloud. Neptune leverages the general computing abstractions provided by Granules to provide a specialized and intuitive programming model for stream processing. In this chapter, we introduce key concepts of Granules and the improvements that were introduced to Granules as part of implementing Neptune.

A *computational task* is the most fine grained unit of execution in a Granules runtime. It encapsulates a domain specific processing logic to process a fine grained unit of data such as a stream data object, a file or a database record. In practice, a set of computational tasks is cascaded together by defining flow of data between them to form a complex computation graph such as a stream processing job, a map-reduce style computation, a machine learning job, etc.

Granules orchestrates a set of distributed machines to perform a set of concurrent computational tasks. Granules launches one or more *Granules resources* at a single physical machine which act as containers for individual computation tasks. The framework is responsible managing the life cycles of computational tasks in addition to launching and terminating computational tasks running on resources.

A computational task accesses data through a *dataset*. A Dataset encapsulates the access to low level data such as a file system, stream or a database and unifies the access of different types of resources. A computational task may work with a set of datasets which are logically grouped together to form a *dataset collection*. Granules framework manages the initializations and closures of datasets and provides notifications on the availability of data.

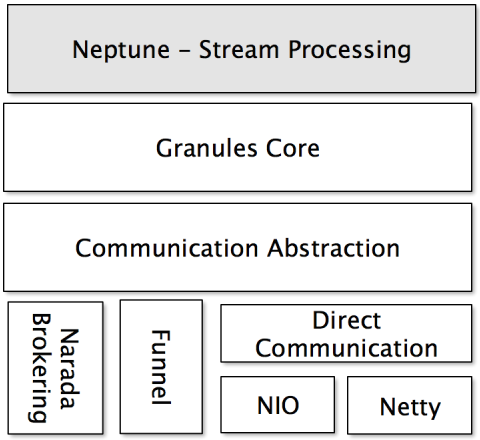


FIGURE 2.1. Abstract design of Granules

Computational tasks are scheduled to run on a shared worker thread pool based on a *scheduling strategy*. A scheduling strategy could be either data driven, periodic, count based or a combination of these three. For instance, a computational task can be scheduled to run every 500 milliseconds or when data is available in a particular dataset. The scheduling strategy associated with a computation task can be changed during the lifetime of a computation.

2.1. IMPROVEMENTS TO GRANULES

While implementing Neptune, we have identified a few areas in Granules that required improvements, especially in its communication substrate. Granules was initially designed to use a publisher/subscriber based model for inter resource communication. If it required for two computational tasks to communicate with each other, a topic has to be created at a central broker to which the sender task should publish. The recipient task should be subscribed to the same topic in order to receive data from the sender task. Granules was using NaradaBrokering [17] as the broker network by default. While this model is useful in certain cases such as implementing fault tolerance with upstream backups [18], it has adverse effects on the performance of the overall system. Sending every message through an intermediate

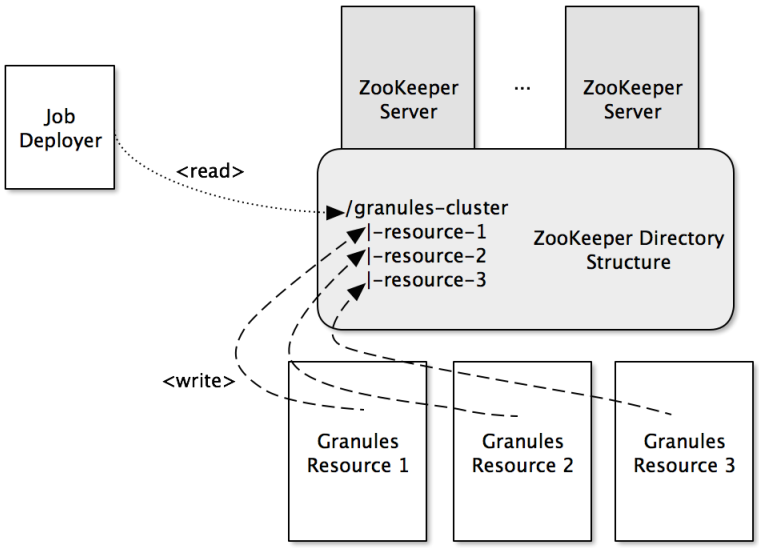


FIGURE 2.2. Resource discovery and meta-data management with Zookeeper

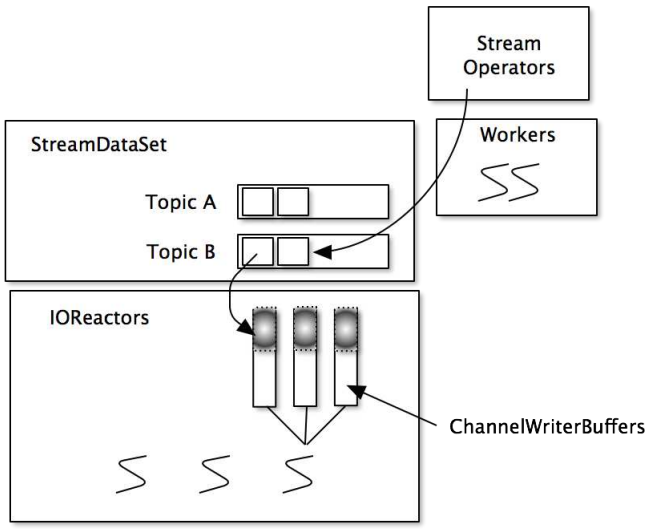


FIGURE 2.3. Direct communication implementation with Java NIO 2

broker can increase the end to end latency, especially in a multi-stage stream processing job. Also the broker may become a bottleneck when achieving high throughput even with a distributed deployment. Finally, given the broker network is a crucial component in the entire deployment, it should be functional all the time along with Granules resources. This incurs

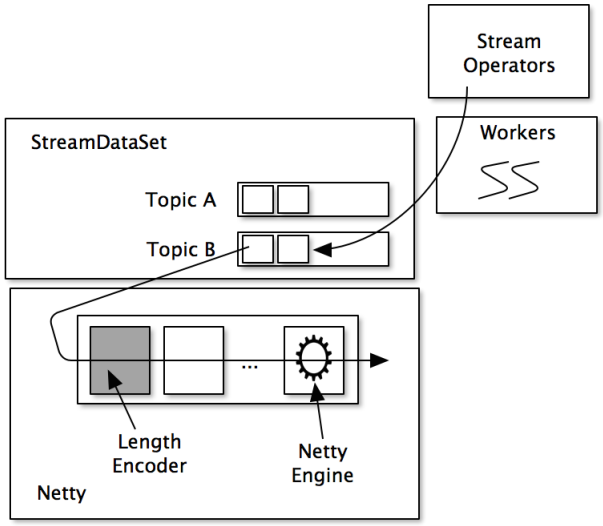


FIGURE 2.4. Direct communication implementation with Netty

additional operational overhead for the users. As an alternative, we have added support for a communication substrate based on direct point to point communication between Granules resources. The evaluation results shows that this model results both in high throughput and low latency.

While adding support for direct communication, the publisher/subscriber based communication abstraction at Granules core was preserved. This is to make sure that Granules is not coupled to any communication model and will remain extensible for future improvements. This will allow users to switch to any communication mode based on their requirements. Figure 2.1 depicts the layered design of Granules. In addition to NaradaBrokering based publisher/subscriber mode and a peer-to-peer communication mode based on a system called Funnel [19], we have added support for two direct communication implementations.

In previous publisher/subscriber based communication model, a special topic created at the broker was used for resource discovery. Since this type of bootstrapping facility is not available with the direct communication based model, we have used Zookeeper [20] for

resource discovery. Every resource, on starting up publishes their meta-data: communication endpoint and resource identifier as a ephemeral zNode at a predefined parent node in Zookeeper. This is depicted in Figure 2.2. When a new stream processing job is deployed, the job deployer polls this specific parent node in Zookeeper to get the current list of available resources. Through the use of ephemeral zNodes and tuning the Zookeeper timeouts appropriately, Granules can detect of failed resources quickly with a high reliability. Apart from the resource discovery, Zookeeper is used to store meta-data related to stream subscriptions as well. Given that Zookeeper cluster is accessed only during the deployment of a stream processing job, the resource requirements for the Zookeeper cluster deployment is much less compared to running a broker network.

Granules supports two inbuilt direct communication implementations: Java New Input/Output module (NIO) [14] based implementation and Netty [15] based implementation. Both implementations use same stream dataset implementation, but the difference is in the way the serialized data gets written into network sockets. Worker threads in Granules which act as threads of execution for computational tasks write data to the appropriate stream dataset. If application level buffering is enabled, the data is written to an internal buffer within the stream dataset. Otherwise data will be directly written to the corresponding IO buffer. In the case of buffering, buffered data is written to the IO buffer only when the cumulative size of buffered messages exceeds a predefined threshold.

For Java NIO based implementation, we have used the reactor model [21] when designing the threading model for IO thread pool. Instead of having a single thread pool to manage the entire set of open connections, in reactor model each thread, known as a IO-Reactor thread, manages a set of connections. In other words, de-multiplexing and dispatching events corresponding to a particular connection is handled by a single thread at all times.

An abstract design on this implementation is depicted in Figure 2.3. This results in reducing the context switching between threads as well as eliminates the necessity for concurrency control to guard shared data structures.

Netty is a dominant asynchronous IO library for Java with a lot of built-in optimizations. In Netty, a message is injected to an ordered sequence of interceptors, called channel handlers, each implementing a certain post-processing operation before sending the message over the network socket. Similarly, at the receiving end a message needs to be processed by a sequence of corresponding set of input handlers before reaching the message processing layer. In the Netty based direct communication implementation of Granules, serialized messages are directly injected into the out-going channel handler chain. This is illustrated in Figure 4.7. We have implemented a channel handler, that will prepend the length of the serialized message to the message being transmitted, which enables the recipient to partition the incoming stream at correct boundaries to retrieve the individual messages. We have not discussed the implementation at the receiver's end, since it is quite intuitive once the details of the message sending are discussed.

CHAPTER 3

Neptune

Stream processing tasks in Neptune are extensions of Granules computational tasks, but present a specialized application programming interface (API) for stream processing. The

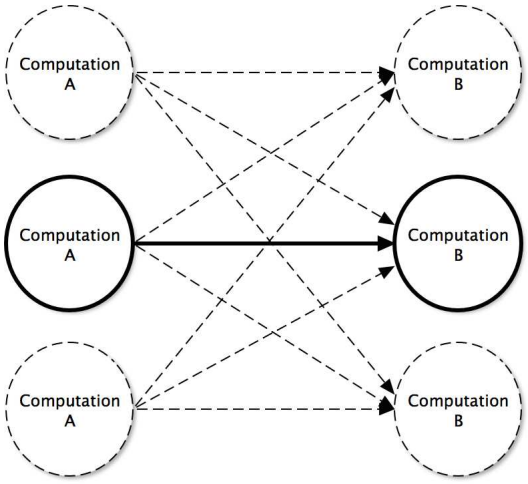


Figure 3.1. Fanning out of a stream processing graph during runtime

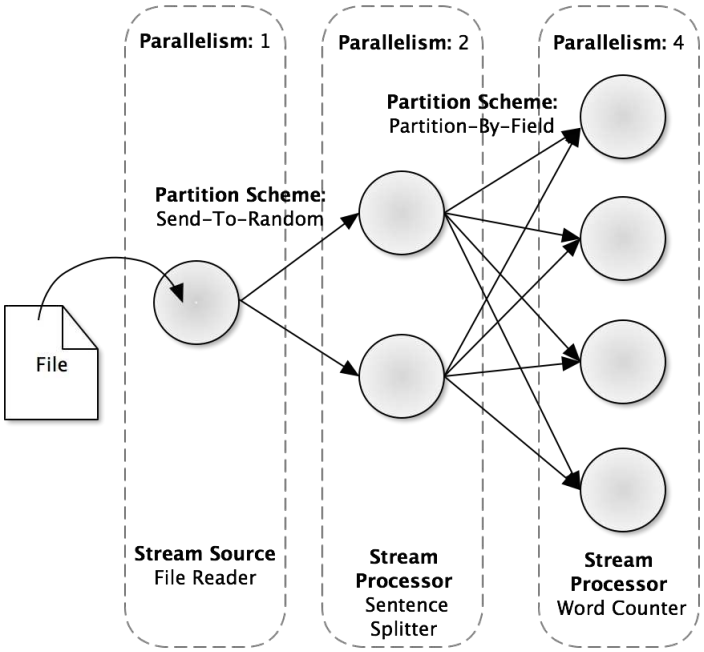


Figure 3.2. Implementing word count in Neptune

communication between stream processing tasks is implemented using datasets in Granules in a way that is oblivious to the end user. In this section, we will introduce the key concepts of Neptune stream processing and how they extend Granules primitives.

3.1. Key Concepts

3.1.1. Stream Data Objects. A stream data object (SDO), also called a stream data packet, is the most fine grained element of data inside a Neptune runtime. An ordered unbounded set of stream data objects forms a stream. Users can define stream data objects by combining one or more data fields as required. Neptune supports a set of primitive data types and data structures out of the box to aid defining data fields within a stream data object.

3.1.2. Stream Sources. Stream sources are used to ingest external data streams into a stream processing graph. During an invocation, it employs a pull based approach to fetch data in and emit stream data objects into one or more internal streams connected to stream processors that are part of the same stream processing graph. Typical implementations of stream sources may read data from message brokers and message queues. This design makes Neptune an ideal candidate for the stream processing engine which retrieves and processes data from an IoT gateway as outlined in IoT reference architectures [13]. Usually a stream source is placed at the root of a stream processing graph. Stream sources are scheduled using a new scheduling scheme introduced to Granules called *continuous execution*. This scheduling scheme ensures that the underlying computational task created for a stream source will be scheduled for execution immediately at the end of an execution.

3.1.3. Stream Processors. Domain specific processing logic to process a stream data object is encapsulated within a stream processor. A stream processor receives stream data

objects from one or more streams, process them and eventually emit stream data objects to one or more streams. Stream processors are scheduled only if data is available in any of the input streams using the data driven scheduling scheme available in Granules. We will use the term *stream operators* to represent both stream sources and stream processors in remainder of the thesis.

3.1.4. Links. A link is used to connect an instance of a stream source or a stream processor with an instance of another stream processor. Links model the streams of stream data objects flowing between stream operators within a stream processing graph. Users define the links between stream operators in the stream graph description in addition to the definitions of stream operators. Within a stream operator, users can decide which link to use when emitting a stream data object. Neptune will mediate the stream data objects between stream operators according the definitions of links and the link to which the stream data object is emitted. Links are internally implemented using a publisher-subscriber model where the source of the link publishes to a topic to which the destination operator is subscribed.

3.1.5. Parallelism. Even though a stream processing graph may contain one instance of each stream operator, at run time the graph may need to fan out by instantiating multiple instances of stream operators as depicted in Figure 3.1. This is a useful construct in implementing solutions to domain specific problems because it provides the ability to partition a stream based on a particular scheme and let one instance of a stream processor to deal with a particular partition of the stream. Further, it is useful when horizontally scaling a given operator in order to load balance the network IO and computational load across multiple nodes and to optimize the cluster utilization.

3.1.6. Stream Partitioning Schemes. Stream partitioning schemes are mandatory to make parallelism work. Partitioning schemes define how a stream should be partitioned when it is routed to different instances of the same stream processor. In other words, given a stream data object emitted by a particular instance of a stream operator, a partitioning scheme decides to which instance of the destination stream processor it should be routed to. Neptune supports four partitioning schemes by default while allowing users to define their own partitioning schemes. *Partition by field* allows destination to be defined based on the value of a particular field of the stream data object. This is based on to widely used `hash(key) mod (no. of destination instances)` partitioning scheme. Other default partitioning schemes supported are *send to one*, *send to all* and *send to random*.

3.1.7. Stream Processing Graphs. A stream processing graph or a stream processing job models a domain specific use case. A stream processing graph is comprised of multiple logical phases called stages where each stage focuses on implementing a portion of the use case. Each stage is implemented using one or more stream operators. The data flow between stages are modeled using links. The definition of a stream processing graph in Neptune is comprised of stream sources and stream processors for different stages, parallelism levels for stream operators, links connecting stream operators and stream partitioning schemes for each link. A stream processing graph can be created by directly invoking the Neptune stream processing graph API or through a JSON descriptor file.

Figure 3.2 depicts a word count stream processing graph implemented using Neptune to illustrate the concepts described above. Stream source reads lines from a file and send to the stream processor which splits each line into a set of words. Finally, each word is sent to Word Counter stream processor for counting. The partitioning scheme for the links connecting the Stream Source with Sentence Splitters is *send-to-random* which will load balance the stream

between the two instances of the Sentence Splitter. Each word is sent to appropriate counter instance using the *partition-by-key* partitioning scheme which will make sure that the same word will be sent to the same Word Counter instance.

3.2. Optimizing for High Throughput and Scalable Stream Processing

Neptune's design is optimized primarily for achieving high throughput and scalability while maintaining latency at acceptable levels. Further it even allows a stream processing job to be optimized for low latency instead of high throughput depending on the requirement. In this section, we introduce the main optimizations that are built into Neptune in order to realize the objectives outlined above.

3.2.1. Application Level Buffering. Stream processing often involves dealing with streams containing small stream data objects. This is applicable to majority of the stream processing jobs used in IoT applications. These streams could be either input streams originated in external sources as well as intermediate streams originated within the stream processing jobs. Sending these small stream data objects individually can cause poor utilization of resources. Systems like Neptune are run on clusters of commodity machines which are connected through an ethernet. Given that the significant mismatch in sizes between the MTU of ethernet frames and serialized versions of these small stream data objects will cause the available network bandwidth to be under utilized. Also sending individual small sized stream data objects can introduce a lot of computational overhead as well. This is mostly due to increased number of network stack traversals [22]. In the absence of low level buffering, this could also lead to a large number of system calls at the network IO layer of the operating system. Due to these reasons, if the size of the stream data object is smaller, the ratio between the overhead of a data transfer and the effective work carried out during the

transfer tend to increase. The poor utilization of systems resources will degrade the system throughput.

Instead of sending individual stream data objects, Neptune implements application level buffering at the stream dataset layer to optimize the throughput. The size of these buffers are defined in terms of their capacity as opposed the number of messages being buffered. The rationale behind this design decision is to flush the buffer as soon as the required level of threshold is reached irrespective of the number of the messages in the buffer and their sizes. We have found this to be quite useful in practice when a stream operator is producing messages with different sizes. This size is configurable and can be set separately for each stream processing graph. Additionally, buffering has helped us to reduce the amount of thread contention when the worker threads are writing the buffered messages to the data buffers of IOReactor threads in the NIO based direct communication implementation.

One of the challenges in buffering is to handle data streams with low data rates. This could be due low data rates in the input streams, due to an under performing stream operator or even due to the nature of the processing logic. For instance, if a stream operator calculates a descriptive statistic over a sliding window of incoming stream data objects and emits a new stream data object only if it detects a significant change in the value that is of interest, the outgoing stream will have a low and a variable data rate. This will increase the time it takes to trigger a buffer flush resulting an increased queuing delay consequently increasing the end to end latency. The end result can be as drastic as failing to satisfy strict real-time processing constraints and latency related quality-of-service (QoS) requirements [23]. To counter this problem, each buffer in Neptune is equipped with a timer which guarantees flushing of the buffer after a certain time period of time since arrival of the first message in the current unflushed buffer. A buffer is flushed either if it is full or if the timer is expired.

This design has helped Neptune to set a soft upper bound on the expected level of end to end latency even in the presence of buffering.

3.2.2. Batch Processing. Processing multiple stream data objects during a single scheduled execution at a stream processor can improve the throughput of the system. This is mostly through amortizing warm-up costs in instruction cache and reduced context switches between worker threads [23]. In Neptune, batch processing is tightly integrated with application level buffering. Stream processors process a set of messages buffered together as a batch in a single scheduled execution. Users need to write the processing logic for a single stream data object while the batched execution is managed by Neptune in a way that is oblivious to the user. Neptune stream sources are also invoked multiple times once they are scheduled by a worker instead of invoking only once. This invocation count is configurable and should be chosen carefully to avoid starvation for other stream operators.

3.2.3. Ability to Optimize for High Throughput or Low Latency. A stream processing job in Neptune can be optimized for high throughput, low latency or a combination of the two metrics at acceptable levels. This allows a mix of stream processing jobs optimized either for high throughput or low latency executing simultaneously in a single Neptune cluster. Application level buffering and batch processing improves throughput while increasing the latency. Optimizing a stream processing job for low latency can be done through controlling the size of the application level buffers of a particular stream processing job. Given that application level buffering is tightly integrated with batch processing, configuring the buffer size implicitly controls the batch size.

3.2.4. Backpressure Management. A stream processing job often involves multiple stages which are heterogeneous with respect to the stream processing rate they can support.

This could be mainly due to the nature of their processing logic as well as due to other external factors such as running on a over provisioned Granules resource. This could lead to situations where the processing rate is lower than the data arrival rate causing the queues to build up. Also if the queues are unbounded, it may often cause long and inefficient garbage collection cycles and eventual out of memory errors at the stream processor. Some frameworks may employ a fail-fast technique where the senders drop messages in the presence of such conditions which causes loss of messages as well as wasted computation cycles if the dropped messages are already processed at previous stages [3].

Neptune uses a backpressure model that leverages the TCP flow control to control the flow of data from upstream stream operators. Each inbound buffer of an stream processor has two threshold values which are called high and low watermarks. Once the buffer is filled up to the high watermark, the IO worker threads are not allowed to write to the buffer unless the buffer contents are consumed by the worker threads until the buffer usage reaches the low watermark level. Consequently, receive buffers associated with the corresponding TCP connections reach their maximum capacity narrowing down the TCP sliding window. This causes sending buffers at the senders to remain filled. Since Neptune uses shared bounded buffers at IO reactors that are handling out-bound traffic, this prevents worker threads from writing to these shared buffers. The stream processors are not scheduled again until these write operations are successful. The high and low watermarks of the inbound buffers are set apart from each other to avoid system oscillating between the two states rapidly.

The disadvantage of this approach is its coarse grained nature. Usually the communication between two resources takes place through a single connection which results in multiple streams of data flowing through a single connection. If the consumer of a single stream slows down, this model of backpressure will effectively control the flow of data for all streams that

share the same connection. Currently we are working on a more fine grained model that will control the flow of data at individual stream level via control messages that flow upstream. These control messages makes modifications to the scheduling policy of the schedulers at the Granules resources where corresponding upstream stream operator is running. This scheme gradually propagates the backpressure to the higher stages of the stream processing graph without affecting the other stream processing jobs.

3.2.5. Compression. We have included support for compression as an experimental feature in Neptune to evaluate its impact on the overall throughput of the system. Compression can compact the amount of data that needs to be transmitted especially when dealing with data streams with low entropy. This could increase the effective amount of data (when uncompressed) transmitted within a given time. Despite probable improvements in efficiency during data transmission, it introduces extra processing overhead at both sending and receiving end. So it would be interesting to explore if this extra computational overhead can outweigh the gains due to compacted data size. Neptune supports two modes of compression: total compression and selective compression. In total compression mode, every batch of stream data objects is compressed whereas in the selective compression mode, compression is performed based on the compressibility of data. In selective compression, if the ratio of the size of compressed data to the size of actual data is greater than a configurable threshold called *selectivity coefficient*, data will not be compressed. Compression and decompression are implemented at the immediate layer to the communication layer and need to be enabled explicitly. To reduce the extra processing latency that can be introduced due to compression, we have used LZ4 [24] compression algorithm which provides faster compression and decompression rates at a reasonable compression ratio.

CHAPTER 4

Evaluation

In this section, we present a series of evaluations carried out on Neptune to demonstrate its performance, scalability and other features. Three metrics were primarily used for evaluation: throughput, latency and bandwidth consumption. We have compared Neptune with another leading open source stream processing system system: Apache Storm [2] when necessary. The hardware and software experimental setup used for evaluation is explained prior to the discussion on experiments.

4.1. Experimental Setup

An in-house cluster comprised of 80 physical machines connected through a local area network with 1 Gbps bandwidth was used for experiments. Each node in the cluster is an Intel Xeon 2.4 GHz 4 core duo machine with 16 GB of memory.

We have used version 0.9.5 of Storm with reliable message processing feature disabled. This is to ensure that the throughput of Storm is not adversely affected by the additional overhead added due to acknowledgements. Also Neptune does not support reliable message processing at the moment; hence it was a measure taken to maintain the fairness in the evaluation. In our experiments, we have optimized Neptune and Storm for high throughput. We configured Storm for high throughput based on the following settings as recommended by its developers and research literature [25, 26].

```
topology.transfer.buffer.size - 32
topology.receiever.buffer.size - 8
topology.executor.send.buffer.size - 16384
topology.executor.receive.buffer.size - 16384
```

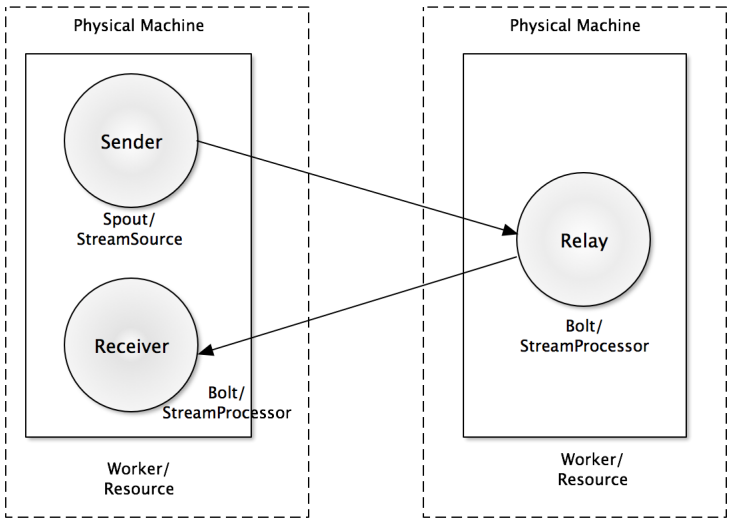


FIGURE 4.1. Three-stage stream processing job acting as a message relay

For Neptune, we have used the default configurations where the buffer size is set to 1 MB. Thread pool sizes are determined automatically depending on the number of cores in the machine it is running on. Heap sizes of both Storm workers and Granules resources were set to 1 GB.

We have used a few different stream processing jobs for the evaluation depending on the objective of the experiment. If the experiment is focused on the underlying communication framework, we have used stream processing jobs with less CPU intensive processing to ensure a minimum interference on the communication layer from the stream processing logic. We have used complex multi-stage stream processing jobs and the associated datasets otherwise.

4.2. Effect of Internal Buffer Size on Performance

Neptune's internal buffer size is the primary mechanism to optimize a stream processing job either for high throughput or low latency. We observed how throughput, latency and bandwidth usage varied with the buffer sizes for different message sizes. Buffer size was varied

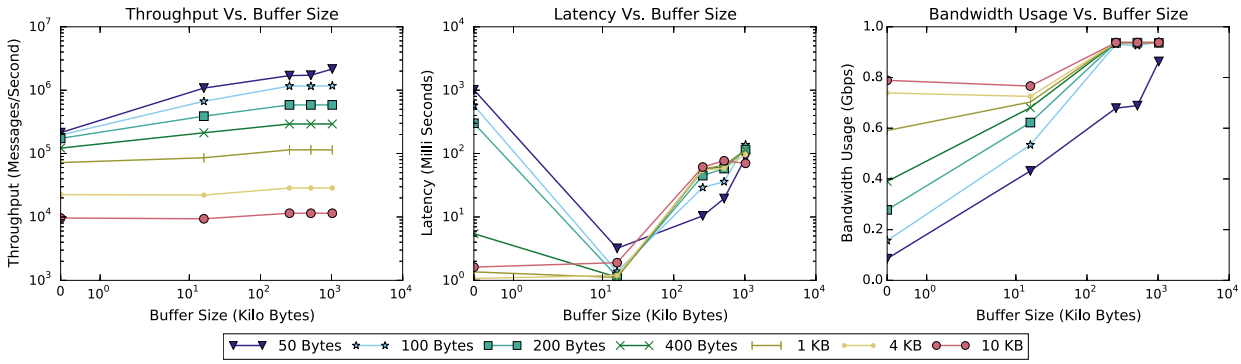


FIGURE 4.2. Throughput, end-to-end latency and bandwidth usage Vs. application level buffer size for different message sizes

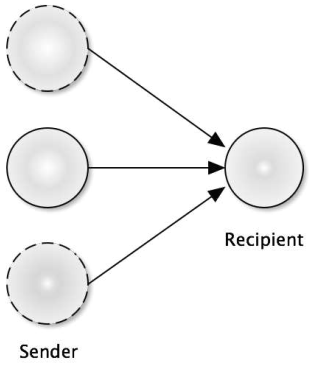


FIGURE 4.3. Two-stage stream processing graph with multiple senders and a single recipient

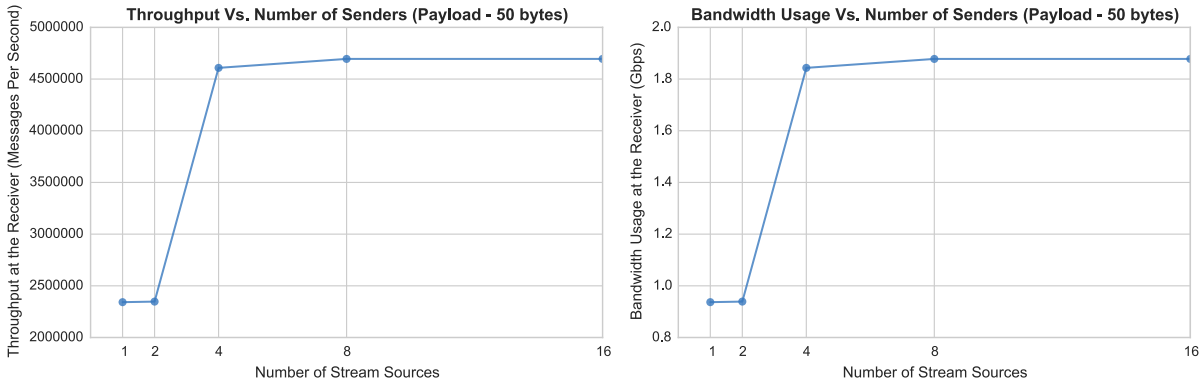


FIGURE 4.4. Throughput, and bandwidth usage Vs. number of senders in a two-stage stream processing graph

from 1 KB to 1 MB at different step sizes. Message sizes were chosen to cover a wide spectrum from 50 Bytes to 10 KB. We have focused more on relatively small sized messages, which are

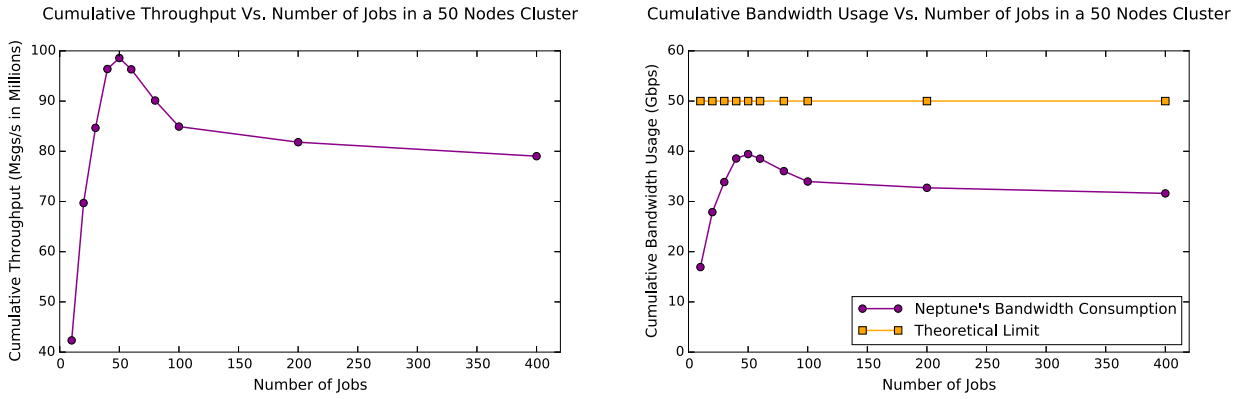


FIGURE 4.5. Cumulative throughput and cumulative bandwidth usage with the number of concurrent jobs

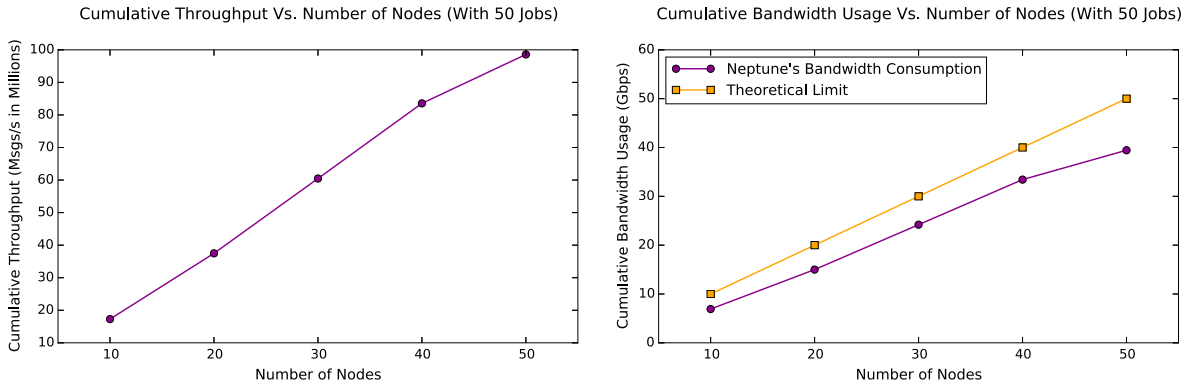


FIGURE 4.6. Cumulative throughput and cumulative bandwidth usage with the number of nodes in the cluster

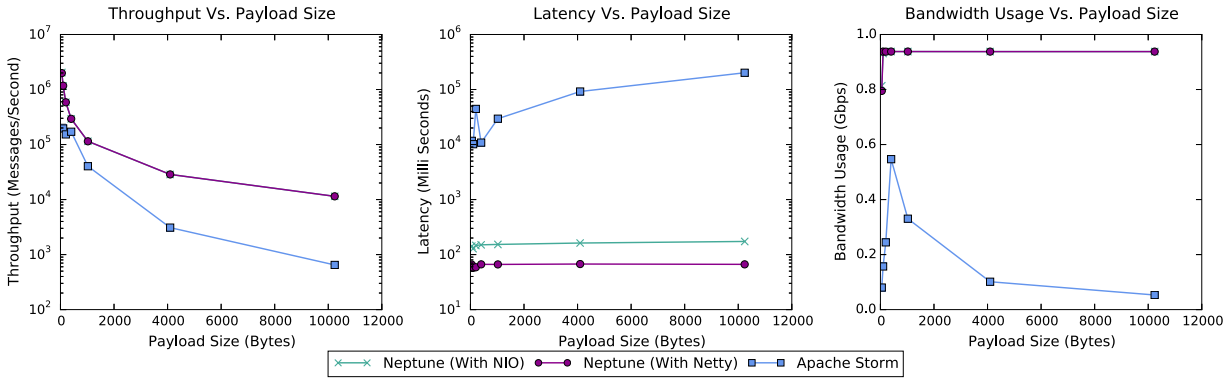


FIGURE 4.7. Throughput, end-to-end latency and bandwidth usage Vs. message size in Neptune and Storm

in the range of 50 to 400 bytes, since majority of the message sizes found in Internet of Things (IoT) datasets are within that range. A three-stage stream processing job, as depicted in

Figure 4.1, was used for this experiment. This simulates a message relay where a stream processor in the second stage relays messages it receives from the stream source at stage 1 to a stream processor at stage 3 which is the final destination. Sender and final receiver are deployed in the same Granules resource whereas the message relay was deployed in a different resource running on a separate physical machine. This deployment plan helped us to measure the end-to-end message latency with a higher accuracy without being affected by clock synchronization issues. We used the Netty based direct communication implementation in Granules for this experiment.

Figure 4.2 shows the results of this experiment. As expected, the system throughput increases until it reaches a steady state with the buffer size. The bandwidth usage reaches 0.937 Gbps (out of 1 Gbps) for message sizes greater than 200 KB when the buffer size increases. Stabilization at the bandwidth consumption causes the throughput to reach and stay at a steady state for larger message sizes. The latency on the other hand slightly increases with the buffer size due to increased queuing delay at the application layer. For smaller message sizes, we are seeing a very high latency when the buffer size is set to 1 KB due to the overhead of meta data. When buffering is enabled, Neptune attaches meta-data only once for each batch. Hence the ratio of meta-data to the actual data is much higher with small buffer sizes. With a lower middle range buffer sizes like 16 KB, the observed latency is less than 10 ms for all message sizes.

4.3. Scalability of Neptune

A stream processing system should be scalable with respect to the number of concurrent stream processing jobs as well as the complexity of stream processing jobs. Complexity of a stream processing job can have many faces: number of processing stages, complexity of

the processing logic embedded in stream processors, level of parallelism and complexity of partitioning schemes.

As the first experiment to test the scalability of Neptune, we tested how its communication layer performs in the presence of multiple traffic sources. A simple two staged stream processing job was used for this experiment by varying the parallelism of the sender as shown in Figure 4.3 with the message size set to 50 bytes. Each of the stream sources and stream processors were deployed on separate physical machines. The results of the experiment are shown in Figure 4.4. The throughput and bandwidth usage increase and stabilize with the number of senders. Neptune could process around 4.7 million messages per second with 4 cpu cores. Please note that our cluster nodes can receive data from two network interface cards at a give time which will allow us to achieve an effective read bandwidth of 2 Gbps.

Figure 4.5 depicts the cumulative throughput and cumulative bandwidth usage of a Neptune cluster with 50 nodes when the number of concurrent jobs is increased. Both cumulative metrics increase until the number of jobs is equal to 50. This phase of the plot corresponds to an sufficient provisioning of resources. Beyond this point, when the number of jobs increased further, the cluster reaches an over provisioned stage and there is a drop in both cumulative throughput and cumulative bandwidth usage. We carried out another experiment by fixing the number of jobs to 50, which corresponds to the maximum throughput Neptune was able achieve in the previous experiment, and varied the number of machines in the cluster. Figure 4.6 shows the cumulative throughput and cumulative bandwidth usage with the cluster size. Both these metrics linearly scale with the cluster size and it is expected to reach a maximum and stabilize once the cluster grows beyond a certain number of machines.

Message Size	Neptune		Storm		Two Sample T-Test		
	Mean	Std. Dev.	Mean	Std. Dev.	t-value	df	P-value
50 B	1987600.95	267582.82	200523.3	51209.70	65.84	102.68	< 0.0001
100 B	1172239.33	51410.19	195943.15	62955.95	154.11	206.19	< 0.0001
200 B	586026.05	7527.83	152759.19	63654.30	125.81	392.67	< 0.0001
400 B	293131.43	2541.31	170864.27	44226.59	52.64	375.83	< 0.0001
1 KB	114496.22	530.59	40328.75	34353.70	31.05	207.21	< 0.0001
4 KB	28620.50	50.36	3090.77	4438.46	64.56	126.04	< 0.0001
10 KB	11447.95	8.62	648.05	1403.28	46.81	37.00	< 0.0001

TABLE 4.1. Results of the two sample t-test for comparing the throughput of Neptune and Storm

Message Size	Neptune			Storm			Two Sample T-Test		
	Mean	Std. Dev.	99^{th} Perc.	Mean	Std. Dev.	99^{th} Perc.	t-value	df	P-value
50 B	65.58	20.21	109.02	11667.11	5718.05	22509.36	-28.40	196.01	< 0.0001
100 B	57.46	6.13	68.0	10140.63	9743.56	48508.8	-17.77	295	< 0.0001
200 B	58.82	5.76	70.0	44398.21	17184.58	71949.01	-48.89	359	< 0.0001
400 B	66.45	4.78	77.02	10833.43	6108.73	22437.16	-33.77	367	< 0.0001
1 KB	66.25	4.08	74.02	29484.76	17307.34	56146.61	-24.46	207	< 0.0001
4 KB	67.31	4.87	76.0	91973.57	20939.42	104507.46	-49.27	126	< 0.0001
10 KB	66.44	7.47	87.82	201598.55	53237.60	225254.14	-23.03	37	< 0.0001

TABLE 4.2. Results of the two sample t-test for comparing the latency of Neptune and Storm

Message Size	Neptune		Storm		Two Sample T-Test		
	Mean	Std. Dev.	Mean	Std. Dev.	t-value	df	P-value
50 B	0.80	0.11	0.08	0.03	65.84	102.68	< 0.0001
100 B	0.94	0.04	0.16	0.05	154.11	206.19	< 0.0001
200 B	0.94	0.01	0.24	0.10	125.81	392.67	< 0.0001
400 B	0.94	0.01	0.55	0.14	52.64	375.83	< 0.0001
1KB	0.94	0.00	0.33	0.28	31.05	207.21	< 0.0001
4 KB	0.94	0.00	0.10	0.15	64.56	126.04	< 0.0001
10 KB	0.94	0.00	0.05	0.12	46.81	37.00	< 0.0001

TABLE 4.3. Results of the two sample t-test for comparing the bandwidth consumption of Neptune and Storm

4.4. Comparison of Neptune and Apache Storm

We compared the performance of communication layers of Neptune with Apache Storm using the message relay setup depicted in Figure 4.1. The evaluation metrics: throughput, latency and bandwidth usage were recorded by varying the message size from 50 bytes to 10 KB.

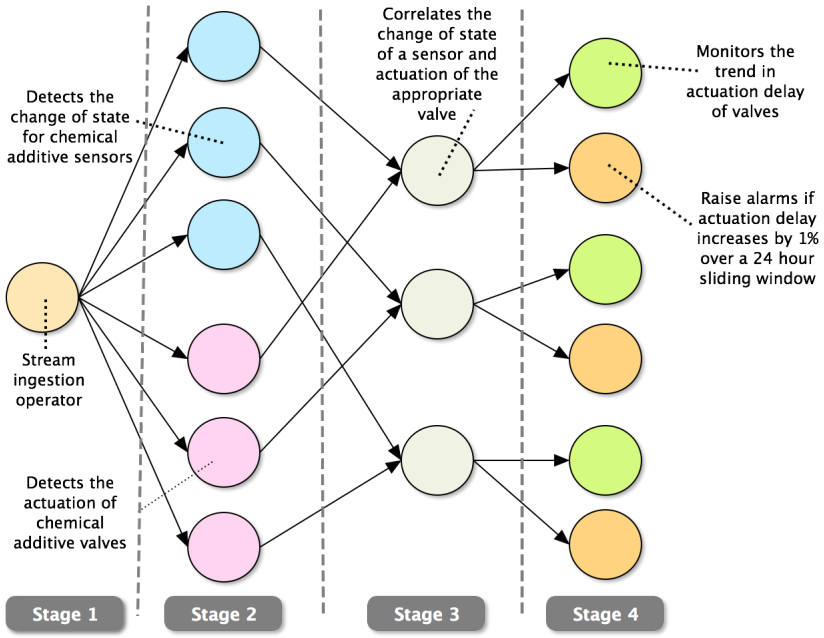


FIGURE 4.8. Multi-stage stream processing graph for monitoring a manufacturing equipment

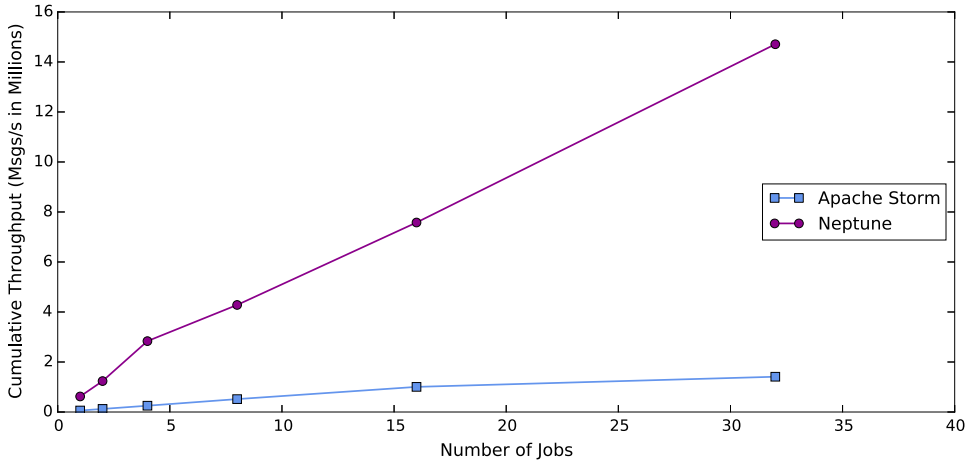


FIGURE 4.9. Cumulative throughput Vs. number of concurrent jobs for the manufacturing equipment monitoring use case

As illustrated in Figure 4.7, the results of this experiment show that Neptune outperforms Storm with respect to all three metrics. One-tailed t-tests were performed for each metric to compare the performance of Neptune (with Netty) and Storm. The results of the t-tests are reported in Table 4.1, Table 4.2 and Table 4.3 with the t-statistic value, degrees of freedom(df) and the P-value. As it can be observed from the results, the P-value is

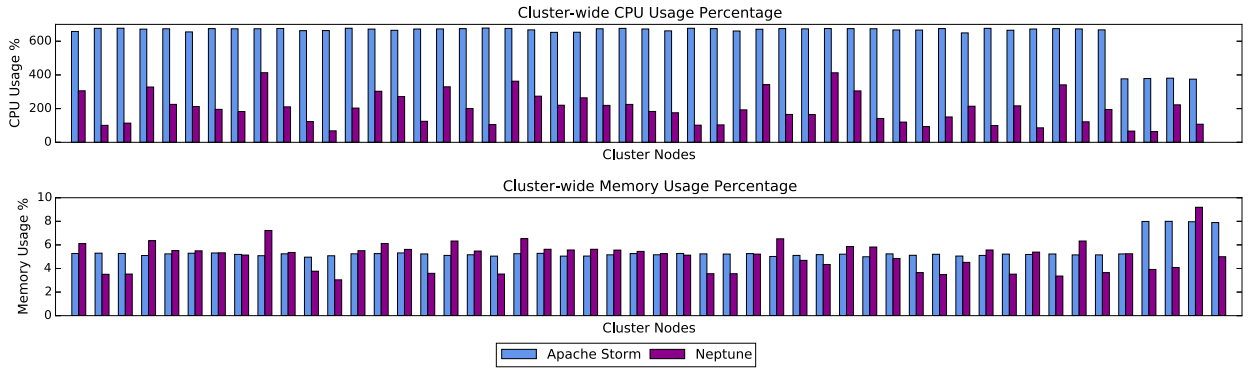


FIGURE 4.10. Cluster-wide resource consumption by Storm and Neptune

insignificant compared to any practical test significance level, α. This provides sufficient evidence to reject our null hypothesis: Storm's performance is equal or better than that of Neptune. Hence it is a strong statistical evidence to support our claim: Neptune performs better than Storm with respect to throughput, latency and bandwidth consumption. The latency of Storm was drastically increasing with the message size. This was mainly due to the absence of backpressure in Storm. The storm spout was emitting a single tuple in every invocation. The relay processor (which is a Storm bolt) is relatively slower than the sender (which is a Storm spout) which creates a bottleneck in the entire Storm topology. If we add a small wait after emitting a tuple (through a sleep to the thread of execution of the Spout), then Storm was able to perform well with a very low latency. But this was not a viable option given its adverse effect on throughput and bandwidth usage. Also we could not use the `topology.max.spout.pending` parameter to throttle the spout because it relies on acknowledgments provided through reliable message processing. The same issue is discussed in other performance benchmarks that used Storm [3, 26].

We implemented a more complex multi-stage stream processing graph to model a real world stream processing use case using both Neptune and Storm. The use case is to monitor a manufacturing equipment using real-time processing data captured by sensors attached to it.

This use case is presented as part of the Grand Challenge competition [27] held in parallel to 6th ACM International conference on Distributed Event-Based Systems. The system ingests a continuous stream of readings captured by sensors that operate at a rate of 100Hz. A single stream data object contains 66 different data fields including the timestamp during which the data is captured. For this particular use case, we will be using 6 sensor readings along with the timestamp. Three of these sensor readings correspond to states of three chemical additive sensors whereas the remaining three readings capture the states of corresponding valves. When the state of a sensor changes, consequently the valve should actuate resulting in a change of its state. The objective of this stream processing job is to monitor the delay between the change of state in a sensor and the actuation of the corresponding valve over a 24 hour time window. Figure 4.8 depicts components of the four-stage stream processing graph implemented for this use case.

We have used a fixed size cluster of 50 nodes for this experiment. Multiple instances of the stream processing job were deployed at the same time and the cumulative throughput at the stream source was measured across the cluster. We have deployed a single Neptune resource per each physical machine and each of them were running one or more stream processors corresponding to one or more stream processing jobs. Storm dedicates a worker process to run a part of a single topology. Since we were measuring the throughput by doubling the number of jobs, with a 50 worker setup the cluster was not fully utilized with 32 concurrent jobs in this setup. Due to this reason, we experimented with a 100 worker setup in addition to the 50 worker setup to ensure that all 50 cluster nodes are utilized. We did not observe any significant difference in the cumulative throughput, hence we used a 50 worker setup for our experiment. The results of this experiment are plot in Figure 4.9. Both systems scale with the number of concurrent jobs. But the degree of increase in throughput is higher in

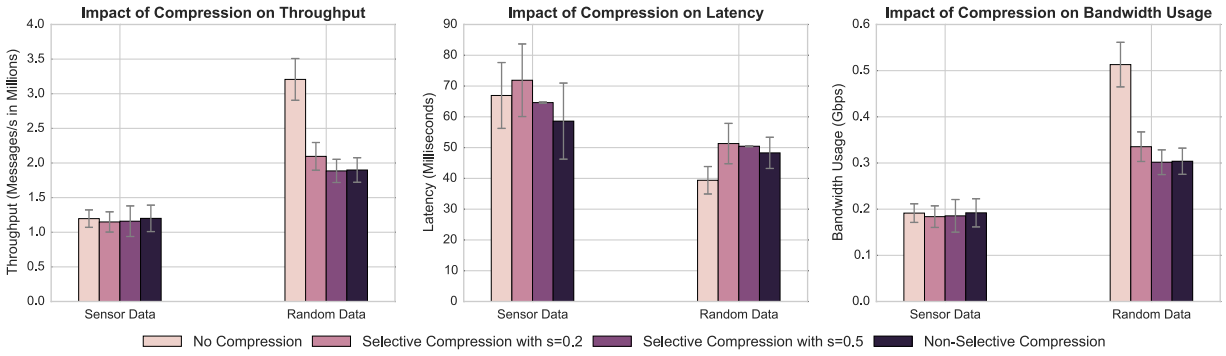
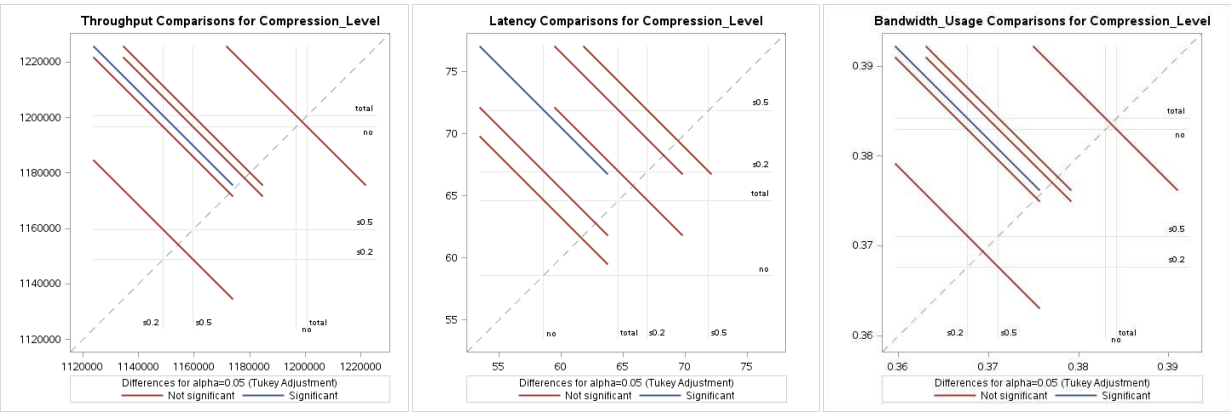


FIGURE 4.11. Impact of compression on stream processing performance

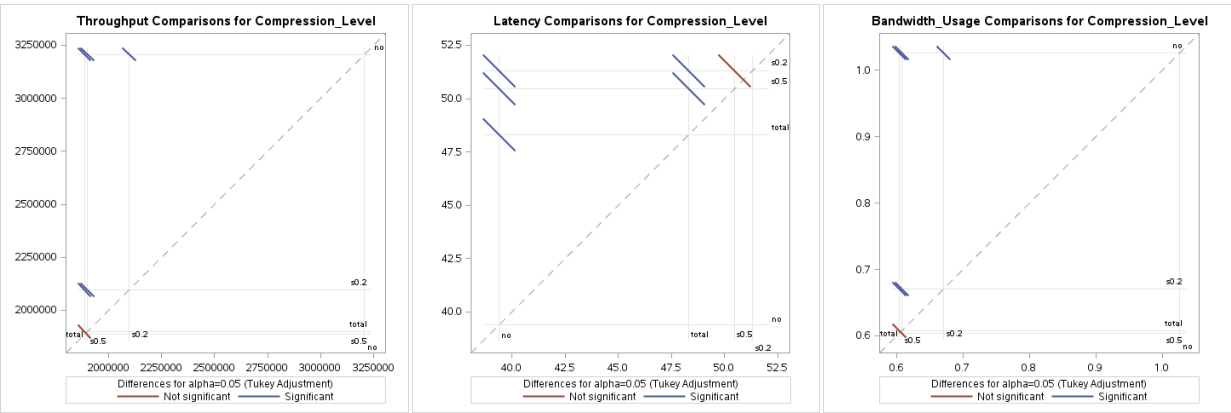
Neptune. With 32 jobs, Neptune can process approximately 8 times more messages than Apache Storm.

We used the same stream processing job to measure the cluster-wide resource usage by both frameworks. A 50 node cluster was used similar to the previous experiment and 50 concurrent stream processing jobs were deployed. CPU usage and memory usage for each system were measured at each individual cluster node in the cluster. Data collected during this monitoring process is plotted in Figure 4.10. Please note that the CPU usage shows the cumulated value of 8 virtual cores (with hyper threading). Memory usage is the amount of memory consumed by the system as a percentage of total available memory. Neptune's CPU consumption is consistently lower compared to the CPU consumption of Storm across all 50 nodes. The high CPU consumption in Apache Storm is due to the overhead introduced by its threading model which requires every message to go through four different threads from the point of entry to exit from a stream processor [3]. On the other hand, Neptune uses a simple two tier thread model, which results in less overhead and reduced queue contention. With respect to memory consumption, we do not see any noticeable difference in both systems.



(A) Throughput (B) Latency (C) Bandwidth Usage

FIGURE 4.12. Results of Tukey's HSD procedure to evaluate the significance of compression for sensor data



(A) Throughput (B) Latency (C) Bandwidth Usage

FIGURE 4.13. Results of Tukey's HSD procedure to evaluate the significance of compression for random data

4.5. EFFECTIVENESS OF COMPRESSION

The impact of compression on the performance of a stream processing job was evaluated using two data sets. One dataset was from the manufacturing equipment monitoring use case [27] that was discussed earlier. The sensor readings do not change frequently over time which results in a low entropy when consecutive stream data objects are batched together. To simulate a data stream with higher entropy, we created a synthetic data stream with random binary data with stream data objects of the same size as the first dataset. We have

used the three-stage stream processing graph as depicted in Figure 4.1 with four different settings: no compression, selective compression with a selectivity coefficient of 0.2 and 0.5 and total compression. The results of this experiment are shown in Figure 4.11. There is a clear improvement in performance when the compression is completely disabled for random data. This is because the effective level of compaction is very low for random data and often compression resulted in an expansion of the data due to their high entropy. Therefore there is no significant reduction in transmission overhead to outweigh the additional overhead incurred due to compression and decompression operations. For the manufacturing equipment monitoring dataset, there is no strong evidence to support any negative or positive impact of the compression. The positive impact caused by the compaction of data is at a comparable level of the negative impact incurred due to additional processing overhead for this dataset. We used a multiple comparison procedure called Tukey-Kramer HSD (Honestly Significant Difference) procedure to compare the data recorded for each compression level for each performance metric for both datasets. The results of the Tukey's HSD procedure for sensor data is appearing in Figure 4.12. The difference between compression levels are insignificant except for a few cases, which do not show a significant difference. Results of the same test for random dataset is appearing in Figure 4.13. In contrast to sensor data, there is a significant statistical difference in data when compression is disabled. To this end, effectiveness of compression depends on the nature of the stream data, hence should be enabled and configured for each stream individually even within the same stream processing job.

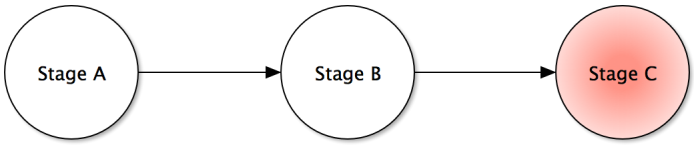


FIGURE 4.14. Three-stage stream processing graph used to trigger backpressure

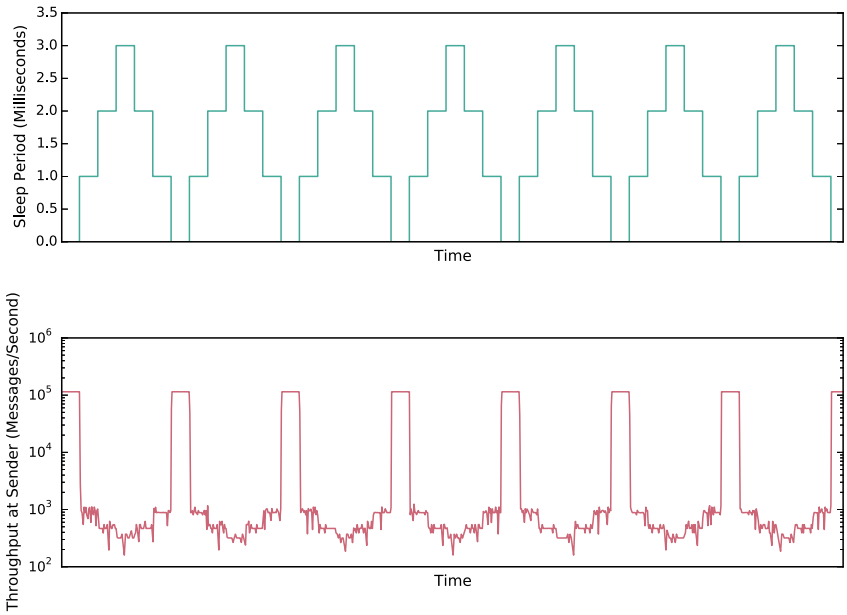


FIGURE 4.15. Demonstrating the backpressure in Neptune

4.6. DEMONSTRATING BACKPRESSURE MANAGEMENT

We used the setup shown in Figure 4.14 to simulate a stream processing job with a stream processor with varying performance. The thread of execution for stream processor at stage C sleeps for some time after processing a stream data object. The sleep interval varies between 0 ms and 3 ms in a cycle in steps of 1 ms as shown in first sub-figure of Figure 4.15. The backpressure should be propagated to stream source at stage A through the stream processor at stage B. In the second sub-figure, we plot the observed throughput at stream source at stage A. The throughput at the stream source is inversely proportional

to the sleep interval at stage C. As expected, the stream source controls the rate of emission of new stream data objects according to the processing rate of the stage C processor.

CHAPTER 5

Related Work

Streaming Databases or Data Stream Management Systems [4, 5, 28] are considered as one of the initial efforts in data stream processing. These systems attempt to extend the concepts of relational database systems to perform data stream processing. Streaming databases materializes an unbounded set of stream data into a continuous series of bounded datasets through *data windows*. Similar to relational databases, streaming databases often provide a query language such as Continuous Query Language (CQL) [29] through which users can express the processing logic. Even though query languages constrain the capabilities of the system due to the limited expressiveness offered by the language itself compared to a general purpose programming language, they simplify the generation and optimization of query plans from a user expressed query.

Most of the recent attempts [6, 7, 30, 8, 9, 2, 31, 32, 3] to stream processing let users to express the processing logic using imperative languages as opposed to system specific query languages. S4 [6] models a stream processing job as a set of *processing elements*. Each processing element is responsible for processing events with a specific key. *Processing Nodes* act as containers for processing elements and the framework routes events between processing elements based on keys of events. Stream Processing Core (SPC) [30] also uses processing elements as the basic unit of computation. The processing elements define their input and output streams through a publisher/subscriber model similar to Granules' internal messaging layer. Communication in SPC is abstracted to a layer called *data fabric* which transparently uses one of its communication modes: pointer transport, shared memory or network transport based on the location of the destination processing element.

Apache Storm [2] uses two types of stream processing elements, namely, Spouts and Bolts. Spouts are used to ingest streams into the system whereas Bolts are used to process event streams and to generate intermediate streams if necessary. Spouts and Bolts can be wired through different streams to form a topology, which will be deployed in a cluster of Storm workers. Storm provides different levels of reliable processing guarantees: *at most once* and *at least once* through an acknowledgement scheme. Through the use of Trident abstraction layer [33], Storm can be extended to provide *exactly once* processing guarantee if necessary in the expense of topology throughput. Heron [3] is a successor to Storm which mainly improves upon Storm's execution model and resource utilization.

Apache Samza [8] relies on a message broker system called Kafka [34] for ingesting streams into the system and for inter-task communication. A stream is modeled as a topic in Kafka which is partitioned and distributed across the broker network. The partition for an event published to a given topic is decided based on the key associated with the event. A topic partition corresponds to a stream partition and each of these stream partitions is processed by a separate Samza task.

Apache Flink [9] provides support for both stream processing and batch processing in a single framework based on the same data-flow oriented framework core. Flink supports *exactly-once* semantics, backpressure and state check-pointing for stream operators. Similar to Flink, Spark Streaming [31] also uses the same programming model used in batch processing for stream processing. Spark Streaming breaks up incoming streams into a series of small batches known as *discretized streams* based on time intervals before processing. Spark runtime treats these discretized streams as *resilient distributed datasets (RDDs)* [35] and process them as regular RDDs created during batch processing. These RDDs are kept in-memory, but replicated to multiple nodes in the cluster to provide fault tolerance. In the

case of a node failure, the lost state can be recalculated by applying the series of the RDD transformations starting from one of the raw data replicas. Both Flink and Spark Streaming support data windows and a set of operators such as joins, union, map, flatmap, etc. as first class members in their programming model. Google's Millwheel [32] provides fault-tolerant stream processing framework through exactly-once delivery, efficient check-pointing and idempotency built into the framework. Further it provides in-order deliveries of streaming data based on a notion of logical time implemented using a watermarking approach on event timestamps.

At a high level, complex event processing (CEP) systems [36–38] may appear to provide functionality that overlaps with the functionality offered by stream processing systems to a certain extent. CEP systems are designed to ingest events from multiple sources simultaneously in order to detect relationships and patterns between these events that are of interest. These are known as event clouds [39] where events are partially ordered based on the time and causality. On the other hand, event streams are initiated at a single source and are totally ordered based on time which makes them a special case of even clouds. Stream processing systems are designed to process event streams whereas CEP systems are designed to process event clouds [40].

CHAPTER 6

Conclusions and Future Work

Achieving real-time stream processing in IoT and sensing environments requires a holistic framework that accounts for the CPU, memory, network, and kernel issues that arise in such settings.

Efficient scheduling of workloads through the use of thread pools and minimizing context-switches by processing streams in batches reduces the number of context switches that need to be performed during stream processing. This, in turn, effectively utilizes the CPU. As our results demonstrate, this allows Neptune to do more with less; we are able to achieve higher throughput than Storm while maintaining a lower average CPU utilization across the entire cluster.

Reusing objects reduces memory utilization; this, in turn, forestalls kernel issues stemming from swapping, page faults, and thrashing. Reusing objects also eliminates issues of memory space reclamation including reduction of CPU overheads relating to garbage collection that involves tracking the scope of individual objects.

Buffering streams utilizes the bandwidth far more effectively. This is especially important when dealing with small packets that may leave Ethernet packet frames underutilized. In fact, as our results demonstrate, our bandwidth utilization is near optimal.

Buffer overflows, where packets arrive at a rate faster than the rate at which they can be processed, can be avoided via effective throttling of streams. Such flow control simplifies memory management and obviates the need to resort to sampling when processing stream packets.

Cumulatively, our methodology results in a novel framework that allows us to demonstrably perform stream processing at scale in real time and high throughput. Our empirical evaluations are performed in large-scale settings and contrast performance with Apache Storm.

In a three-stage message relay benchmark, Neptune was able to achieve a throughput of 2 million messages per second with a 93.2% bandwidth consumption. The same experiment in a 50 node cluster setup recorded a cumulative throughput closer to 100 million messages per second with a near optimal bandwidth consumption. The processing latencies (for 10KB packets) for the 99% of the packets was less than 87.8ms even with a configuration optimized for high throughput. For a four-stage stream processing application that modeled real-time monitoring of manufacturing equipment, Neptune was able to achieve a cumulative throughput of 15 million messages per second. During the course of evaluation, Neptune outperformed Storm in all three metrics that are critical in stream processing: throughput, latency and bandwidth consumption. Neptune's cluster-wide CPU consumption is significantly less than Storm as demonstrated when the cluster is loaded up with multiple instances of the four-stage stream processing job.

Neptune has provided us a solid basis to introduce a series of new features in future. Future work will target development of algorithms for fault tolerant processing while reducing overheads that often accompany such schemes. We also plan to add support for a dynamic deployment model for Neptune that leverages the available capabilities of cluster nodes, properties of the stream processing graph and the data arrival patterns of data streams.

Bibliography

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] Apache Software Foundation, "Apache Storm." `https://storm.apache.org`, 2014. Online; accessed 17 July 2015.

[3] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, ACM, 2015.

[4] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," *Book chapter*, 2004.

[5] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[6] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177, IEEE, 2010.

[7] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: Mapreduce-style processing of fast data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, 2012.

[8] Apache Software Foundation, "Samza." http://samza.apache.org, 2014. Online; accessed 7 August 2015.

[9] Apache Software Foundation, "Flink." https://flink.apache.org/index.html, 2015. Online; accessed 7 August 2015.

[10] 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*, vol. 10, p. 10, 2010.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 59–72, ACM, 2007.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] S. Kamburugamuve, L. Christiansen, and G. Fox, "A framework for real time processing of sensor data in the cloud," *Journal of Sensors*, vol. 2015, 2015.

[14] Oracle Inc., "Package java.nio." http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html, 2014. Online; accessed 29 June 2015.

[15] The Netty project, "Netty Project." http://netty.io, 2015. Online; accessed 3 July 2015.

[16] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–10, IEEE, 2009.

[17] G. Fox and S. Pallickara, "The narada event brokering system: Overview and extensions," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications-Volume 1*, pp. 353–359, CSREA Press, 2002.

[18] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 779–790, IEEE, 2005.

[19] Department of Computer Science, Colorado State University, "Funnel: Autonomous High-throughput Dissemination of Data Streams." `http://funnel.cs.colostate.edu`, 2015. Online; accessed 4 June 2015.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX Annual Technical Conference*, vol. 8, p. 9, 2010.

[21] D. C. Schmidt, "Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching," 1995.

[22] Martin Sústrik, "ZeroMQ." `http://aosabook.org/en/zeromq.html`. Online; accessed 4 August 2015.

[23] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[24] Y. Collet, "Lz4: Extremely fast compression algorithm," *code. google. com*, 2013.

[25] Michael G. Noll, "Understanding the Internal Message Buffers of Storm." `http://www.michael-noll.com/blog/2013/06/21/understanding-storm-internal-message-buffers/`, 2013. Online; accessed 17 July 2015.

[26] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas, "Of streams and storms," *IBM White Paper*, 2014.

[27] "6^{th} acm international conference on distributed event-based systems - grand challenge." `http://www.csw.inf.fu-berlin.de/debs2012/grandchallenge.html`.

[28] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, *et al.*, "Nile: A query processing engine for data streams," in *Data Engineering, 2004. Proceedings. 20th International Conference on*, p. 851, IEEE, 2004.

[29] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006.

[30] L. Amini, H. Andrade, and R. Bhagwan, "SPC: A distributed, scalable platform for data mining," *... on Data mining ...*, pp. 27–37, 2006.

[31] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pp. 10–10, USENIX Association, 2012.

[32] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[33] Apache Software Foundation, "Trident API Overview." `https://storm.apache.org/documentation/Trident-API-Overview.html`, 2014. Online; accessed 7 August 2015.

[34] The Apache Software Foundation, "Apache Kafka: A high throughput distributed messaging system." `http://kafka.apache.org`. Online; accessed 7 August 2015.

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "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*, pp. 2–2, USENIX Association, 2012.

[36] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: A second look at complex event processing architectures," in *Proceedings of the 2011 ACM workshop on Gateway computing environments*, pp. 43–50, ACM, 2011.

[37] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: a high-performance event processing engine," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 1100–1102, ACM, 2007.

[38] EsperTech Inc., "Esper: Event Processing for Java." `http://www.espertech.com/products/esper.php`, 2015. Online; accessed 7 August 2015.

[39] D. Luckham, "A Brief Overview of the Concepts of CEP1." `http://complexevents.com/wp-content/uploads/2008/07/overview-of-concepts-of-cep.pdf/`, 2007. Online; accessed 7 August 2015.

[40] David Luckham, "What's the Difference Between ESP and CEP?." `http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/`, 2006. Online; accessed 7 August 2015.