NEPTUNE: Real Time Stream Processing for Internet of Things and Sensing Environments

Thilina Buddhika and Shrideep Pallickara
Department of Computer Science
Colorado State University, USA
Email: {thilinab, shrideep}@cs.colostate.edu

***Abstract*—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, 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 our Granules cloud runtime. The framework maximizes bandwidth utilization in the presence of small messages via the use of buffering and dynamic compactions of packets based on payload entropy. Our use of thread-pools and batched processing reduces context switches and improves effective CPU utilizations. NEPTUNE 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 benchmarks demonstrate the suitability of the Neptune 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 setup, we achieved a rate of $\sim$100 million packets per-second.

I. INTRODUCTION

Deployments of networked sensors and related technologies to efficiently monitor a wide spectrum of environments in domains from weather and ecology to smart households and military battlefields have presented a novel set of challenges to the traditional data processing paradigm. The term IoT encompasses settings where large numbers of ubiquitous sensors, actuators and embedded communication hardware are seamlessly integrated in the environment alongside information systems to store and process voluminous data produced by these monitored environments to form a communicating-actuating network [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 approaches to realizing this middleware layer. Low latency processing or near-real time processing is one advantage offered by stream processing systems over batch processing systems. This is a key requirement in IoT use-cases such as triggering actuators in real time after processing sensor data.

Stream processing systems must support processing continuous, unbounded data streams with varying data rates. These systems provide a managed execution runtime for a set of stream processing jobs. Stream processing proceeds in stages, with each stage encapsulating domain-specific logic. During orchestrations, the system must accommodate the inherent heterogeneity of stages due to differences in their processing and IO requirements. The crux of this paper is to achieve real time high throughput stream processing.

A. RESEARCH CHALLENGES

Achieving high throughput stream processing in IoT and sensing settings involves challenges that impact the efficiency of network, CPU, and memory utilization.

- **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, the small payload sizes results in a significant portion of each Ethernet packet frame (with an MTU of 1500 bytes) being unused. This contributes to lower throughputs due to network bandwidth underutilization.
- **Context switches:** Packets are typically processed in thread pools with each thread processing a packet at a 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 packets are being processed in thread pools where the context switching costs are significantly lower than those for processes.
- **Buffer overflows:** Stream processing 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.
- **Object creation:** Prior to packets being processed, the raw bytes need to be transformed 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 of the rates at which packets arrive; this is applicable regardless of whether the memory reclamation scheme is implicit (as in Java/C#) or explicit (as in C/C++). 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.

B. Research Questions

We posit that inefficiencies in stream processing spanning memory, scheduling, networking, and kernel issues preclude high throughput stream processing. Each of these aforementioned issues and challenges introduces delays that adversely impact the rate at which stream processing is performed. These challenges necessitate a holistic solution that addresses the CPU, memory, network, and kernel issues (context switches and page faults) involved in stream processing. Specific research questions we explore include:

- How do we incorporate support for efficiently expressing multi-stage stream processing?
- How can we improve bandwidth utilization?
- How can we minimize the number of context switches even in cases where processing is backed by thread pools?
- How can we avoid buffer overflows when the processing involves multiple stages? Specifically, how can we identify stages to throttle and do this effectively?
- How can we minimize object creation in such stream processing settings?
- 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.

C. Approach Summary

Here, we describe our system NEPTUNE that is designed for real time, high throughput stream processing. The rationale behind many of the design principles in NEPTUNE is to achieve efficient consumption of resources: network IO, CPU and memory. NEPTUNE provides an intuitive stream processing API alongside a processing graph description model that describes processing as a collated set of modular stages. The framework initializes individual stages, establishes communication between stages and manages the lifecycle 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 at runtime to better utilize the available cluster resources and to achieve desired levels of performance.

NEPTUNE makes effective use of the network bandwidth to achieve high throughput while maintaining the communication latencies at acceptable levels. To accomplish this, it buffers stream packets at the application layer and transfers a batch of buffered messages over the network rather than send individual messages one at a time. In addition to improving the bandwidth usage significantly, application level buffering reduces the number of traversals of the networking stack and alleviates queue contention between worker threads and IO threads when accessing shared data buffers. Even with a configuration optimized for high throughput, NEPTUNE achieves a sub-second end-to-end communication latency in the order of tens of milliseconds for most applications. NEPTUNE's buffering schemes are streamlined with batch processing to reduce the number of context switches among worker threads and to improve use of the instruction cache.

To reduce queue contention caused by inter-thread communication, NEPTUNE uses a two-tier thread model comprised of two thread pools for worker threads and IO threads. We rely on an asynchronous IO model based on the Java NIO library and Netty [13] to implement a scalable communication module with reduced resource usage footprint. NEPTUNE relieves memory pressure through a frugal object creation scheme that reduces strain on the garbage collector via reuse of objects and data structures. The advantages of this scheme include reduced instantiation overhead, efficient serialization/deserialization of stream packets, and reduced number of short-lived objects created during runtime.

NEPTUNE supports backpressure, a flow control mechanism, 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 subsequent downstream stages.

NEPTUNE also incorporates a dynamic compression scheme that selectively compresses portions of a data stream 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 profile NEPTUNE using a comprehensive set of experiments that include evaluating the validity of individual design principles used in NEPTUNE, and as a complete solution for stream processing. NEPTUNE is contrasted 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. In our empirical evaluations, NEPTUNE outperforms Storm in metrics relating to throughput, bandwidth utilization and latency.

D. Paper Contributions

This study presents our experiences in building a real time stream processing framework. The primary contributions relate to the exploration of the trade-off space encompassing CPU, memory, network, and kernel issues in such settings. NEPTUNE supports buffering of data streams, object reuse, dynamic compaction, flow control, thread pool based batched execution and reduced queue contentions. Specifically, the framework described here:

- Supports expressing and orchestrating multistage stream processing.
- Provides support for high throughput stream processing while making efficient use of resources.
- 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.
- Includes comprehensive empirical evaluations of several aspects of the framework and a comparison with the dominant stream-processing framework, Storm.

E. Paper Organization

The remainder of the paper is organized as follows. In section 2, we introduce Granules. Section 3 describes NEPTUNE and the key concepts related to its API and programming model. The latter half of the section 3 discusses in detail how NEPTUNE achieves high performance stream processing. Experiments contrasting NEPTUNE with Storm and their results are presented in section 4. Related work is reviewed in section 5. Section 6 outlines conclusions and future work.

II. Granules

NEPTUNE is implemented on top of our Granules computing framework [14]. In the Granules runtime, a *computational task* is the most fine grained unit of execution in the Granules runtime. Tasks encapsulate a domain specific processing logic to process a fine grained unit of data such as a file, a packet, or a database record.

Granules orchestrates a set of distributed machines to perform a set of concurrent computational tasks. Granules launches one or more *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 these resources.

A computational task accesses data through a *dataset*. The dataset unifies the access of different types of resources and encapsulates the access to low level data such as files, streams or databases. Granules framework manages the initializations and closures of datasets and provides notifications on the availability of data.

Computational tasks are scheduled to run based on a *scheduling strategy* that can be changed during execution. The scheduling strategy could be data driven, periodic, count based or a combination of these. For instance, a computational task can be scheduled to run every 500 milliseconds or when data is available in a particular dataset.

III. Neptune

NEPTUNE leverages the general computing abstractions provided by Granules to provide a specialized and intuitive programming model for stream processing.

A. Key Concepts

1) Stream Packets: A stream packet is the most fine grained element of data in NEPTUNE. An ordered, unbounded set of stream packets forms a stream. Users can define stream packets by combining one or more data fields as required. NEPTUNE natively supports a set of primitive data types and data structures to aid in defining data fields within a stream packet.

2) Stream Sources: Stream sources are used to ingest external data streams into a stream processing graph and emit stream packets to the next stage of stream processing graph over one or more internal streams. Typical implementations of stream sources may read data from message brokers and message queues. A NEPTUNE stream source can ingest streams using a pull-based approach from an IoT gateway as outlined in IoT reference architectures [15].

3) Stream Processors: Domain specific processing logic to process a stream packet is encapsulated within a stream processor. A stream processor receives stream packets from one or more incoming streams, processes them and eventually emits stream packets over one or more outgoing streams. Stream processors are scheduled only if data is available in any of the input streams using the data driven scheduling scheme provided by Granules. We will use the term *stream operators* to represent both stream sources and stream processors in remainder of the paper.

4) Links: Links model the flow of data between stream operators within a stream processing graph by connecting an instance of a stream source or a stream processor with another stream processor instance. Within a stream operator, users can configure the link to use when emitting packets.

5) Parallelism: Though a stream processing graph is initialized with one instance of each stream operator, at run time the graph may need to *fan out* by instantiating multiple instances of stream operators. This is a useful construct because it allows partitioning streams and lets stream processors deal with a particular portion of a stream. Further, this allows horizontal scaling a given operator in order to load balance the network IO and computational load across multiple nodes and to optimize the cluster utilization.

6) Stream Partitioning Schemes: Stream partitioning schemes are necessary 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 packet emitted by a particular instance of a stream operator, a partitioning scheme decides the instance of the destination stream processor that it should be routed to. NEPTUNE supports a set partitioning schemes natively and also allows users to design custom partitioning schemes.

7) Stream Processing Graphs: A stream processing graph/job models a domain specific use case. A stream processing graph is composed of multiple logical phases called stages where each stage focuses on implementing a

portion of the domain-specific logic. A stream processing graph in NEPTUNE comprises: (1) stream sources and stream processors for different stages, (2) parallelism levels for stream operators, (3) links connecting stream operators, and (4) stream partitioning schemes for each link. A stream processing graph can be created by directly invoking the NEPTUNE API or through a JSON descriptor file.

B. OPTIMIZING FOR HIGH THROUGHPUT

NEPTUNE's design is optimized primarily for achieving high throughput and scalability while maintaining latency at acceptable levels. Here, we describe optimizations built into NEPTUNE to realize above objectives.

1) Application Level Buffering: Stream processing jobs, especially in IoT settings, require processing streams comprising stream packets with small payload sizes. These streams could be either input streams that originate at external sources as well as intermediate streams that originate within stream processing jobs. NEPTUNE is designed for deployment in commodity clusters inter-connected using Ethernet links. Because of the significant mismatch in sizes of the MTU of Ethernet frames and serialized versions of these small stream packets, the available network bandwidth is underutilized. Also sending individual small stream packets introduces computational overheads due to the increased number of network stack traversals [16]. In the absence of low level buffering, this could also lead to a large number of system calls at the kernel network IO layer.

Instead of sending individual stream packets, NEPTUNE implements application level buffering at the stream dataset layer to increase 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 threshold is reached irrespective of the number of the messages in the buffer and their sizes. We have found this to be quite useful when a stream operator is producing stream packets of different sizes. The buffer size is configurable for each stream processing graph. Additionally, buffering has helped us reduce the amount of queue contention between worker threads and IO threads.

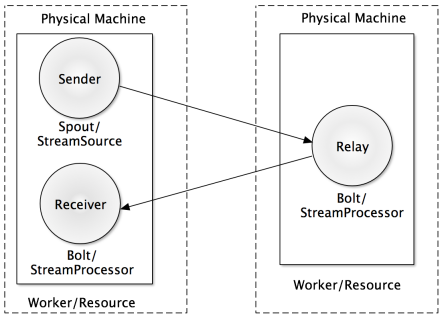

Figure 1. Three-stage stream processing job acting as a message relay

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, an underperforming stream operator, or the nature of the processing logic. For instance, if a stream operator calculates a descriptive statistic for a sliding window over incoming stream packets and emits a new stream packet 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 causing an increased queuing delay consequently increasing the end-to-end latency. This can result in failing to satisfy strict real time processing constraints and violations of latency related quality-of-service (QoS) requirements [17]. To circumvent this problem, each buffer in NEPTUNE is equipped with a timer that guarantees flushing of the buffer after a certain time period since arrival of the first message. This allows NEPTUNE to set a soft upper bound on expected end-to-end latency even in the presence of buffering.

We observed how throughput, latency and bandwidth usage varied with the buffer size for different message sizes; this is depicted in Figure 2. Buffer size was varied 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 in the range of 50 to 400 bytes, since majority of the message sizes found in IoT and sensing environment datasets are within that range [18]. A three-stage stream processing job, as depicted in Figure 1, was used for this experiment. This simulates a message relay where a stream processor in the second stage relays messages that it receives from the stream source at stage 1 to a stream processor at stage 3. The sender and 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 helps us measure the end-to-end message latency with high accuracy without having to account for clock synchronization issues such as skews and drifts.

Figure 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 of the bandwidth consumption causes the throughput to reach and stay at a steady state for larger message sizes. The latency, on the other hand, increases slightly with the buffer size due to increased queuing delay at the application layer. For smaller message sizes, we see a very high latency when buffering is disabled due to high number of context switches as explained in section III-B2. With a lower, middle-range buffer sizes like 16 KB, the observed latency is less than 10 ms for all message sizes.

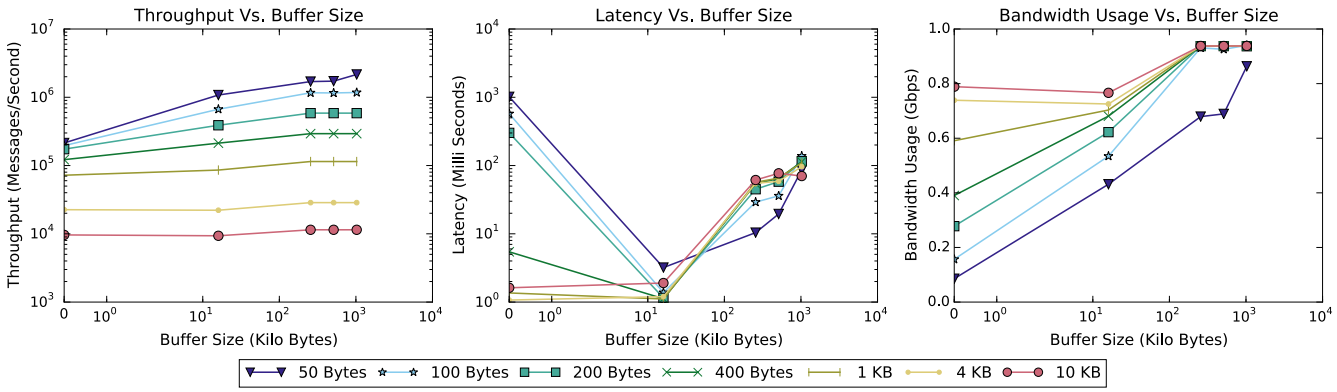

Figure 2. Throughput, end-to-end latency and bandwidth usage Vs. application level buffer size for different message sizes

Table I
CONTRASTING CONTEXT SWITCHES

Mode	Context Switches Per 5 Seconds	
	Mean	Std. Dev.
Batched Processing	4085.2	91.8
Individual Message Processing	89952.4	1086.5

2) Batched Scheduling: Processing multiple stream packets in a single scheduled execution of the stream processor can improve system throughput. This is mostly by amortizing warm-up costs in instruction cache and reduced context switches between worker threads [17]. In NEPTUNE, batch processing is tightly integrated with application level buffering. NEPTUNE schedules processing a set of messages buffered together as a batch in a single scheduled execution. Users need to provide processing logic for a single packet while NEPTUNE transparently manages batched execution.

We evaluated the impact of batching by measuring the number of non-voluntary context switches when batching is enabled and disabled. The NEPTUNE process that was executing the relay processor of the message relay stream processing graph was used for this experiment. We used a modified version of NEPTUNE where batched scheduling and application level message buffering are decoupled from each other, to ensure that the effectiveness of batched processing can be measured without any interference from buffering. Messages of size 50 Bytes were used in the experiment with a buffer size of 1 MB. The results of this experiment are tabulated in Table I. The number of context switches (per 5 seconds) when batched scheduling is disabled is 22 times higher than the number of context switches when batched scheduling is enabled.

3) Object Reuse: Rather than separately and repeatedly create data structures used in serialization and deserialization for individual messages, NEPTUNE creates them once and reuses them for the entire set of buffered messages. This reduces the number short-lived runtime objects at a NEPTUNE process, which in turn reduces the strain on the garbage collector. We measured the time spent on garbage collection with and without object reuse using the same experiment setup as before. Object reuse helped reduce the percentage of time spent by the JVM on garbage collection over the time spent on actual processing from 8.63% to 0.79%.

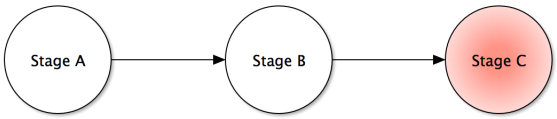

Figure 3. Three-stage stream processing graph used to trigger backpressure. Stream processor at stage C has a variable stream processing rate.

4) Backpressure Management: A stream processing job often involves multiple stages with different stream processing rates. 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 overprovisioned cluster node. 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 cause long and inefficient garbage collection cycles and eventual out of memory errors at the stream processor. Some frameworks employ a fail-fast technique where the senders drop messages during such conditions, which causes loss of messages as well as wasted computation cycles if the dropped messages were already processed upstream [3].

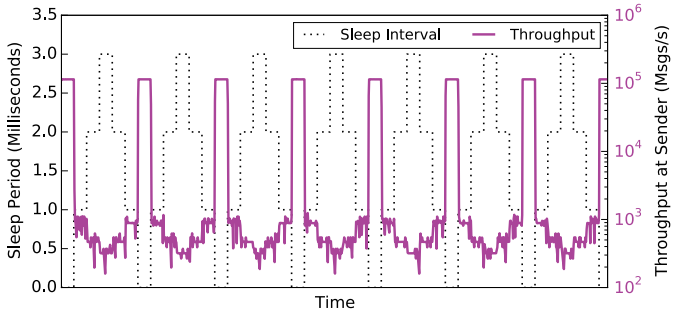

Figure 4. Demonstrating backpressure in Neptune. The throughput at stage A is adjusted based on the data processing rate at stage C. Data processing rate at stage C is varied using a sleep after processing each message.

NEPTUNE uses a backpressure model that leverages the TCP flow control to control the movement of data from upstream operators. For each inbound buffer of an stream processor, we maintain 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 and the buffer usage reaches the low watermark level. Consequently, receive buffers associated with the corresponding TCP connections reach their maximum capacity, narrowing the TCP sliding window. This causes sending buffers at the senders to remain filled. Since NEPTUNE uses shared bounded buffers at IO threads that are handling outbound 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 sufficiently apart from each other to avoid the system oscillating between the two states rapidly.

We used the setup shown in Figure 3 to simulate a stream processing job with a stream processor with varying performance. The thread of execution for the stream processor at stage C sleeps for some time after processing a stream packet. The sleep interval varies between 0 ms and 3 ms in a cycle that proceeds in steps of 1 ms as illustrated in Figure 4. The backpressure should be propagated to stream source at stage A through the stream processor at stage B. The throughput at the stream source is inversely proportional to the sleep interval at stage C. As can be seen in Figure 4, Stage A controls the emission rate of new packets to be aligned with the processing rate at stage C.

5) Compression: NEPTUNE incorporates support for entropy based dynamic compression. Compression can effectively reduce the volume of data transmitted between two stream computations, especially when dealing with data streams with low entropy. This could increase the effective amount of data (when uncompressed) transmitted within a given time. However, this introduces extra processing overhead at both sending and receiving ends. We have explored if this extra computational overhead can outweigh the gains due to compacted data size. NEPTUNE employs a selective compression scheme that compresses a payload only if its entropy is less than a configurable threshold. To reduce the latency that can be introduced by compression, we used the LZ4 compression algorithm (http://lz4.org) which provides faster compression and decompression with a reasonable compression ratio.

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 presented in DEBS Grand Challenge [19]. Here, sensor readings do not change frequently over time which results in a low entropy when consecutive stream packets are buffered together. To simulate a data stream with higher entropy, we created a synthetic data stream with random binary data with stream packets of the same size as the first dataset. We evaluated how compression impacts throughput, latency and bandwidth consumption of a stream processing job. The results were statistically validated using a Tukey's HSD multiple comparison procedure. There is a clear improvement in performance when the compression is completely disabled for random data (*p-values for individual comparisons* $<$ *0.0001*) whereas there is no strong evidence to support any negative or positive impact of the compression for the sensor readings dataset (*p-values for individual comparisons* $\geq$ *0.1561*). This is due to the significant difference in effective compactions for each dataset. 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.

IV. EVALUATION

A series of evaluations were performed on NEPTUNE to profile its performance, scalability, and other features. Three metrics were used for evaluation: throughput, latency, and bandwidth consumption. We also compared NEPTUNE with a leading open source stream processing system: Apache Storm [2]. Packets were processed in order and exactly-once.

A. EXPERIMENTAL SETUP

An in-house cluster comprising 50 physical machines connected over a 1 Gbps LAN was used for experiments. There were 46 HP DL160 servers (Xeon E5620, 12 GB RAM) and 4 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM).

We have used version 0.9.5 of Storm with reliable message processing feature disabled to ensure that the throughput of Storm is not adversely affected by the additional overhead introduced by acknowledgments. In our experiments, we optimized Storm for high throughput using settings recommended by developers and research literature [20], [21].

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 focused on the underlying communication framework, we used stream processing jobs that were not CPU intensive to minimize interference on the communication layer from the stream processing logic. We have used complex multi-stage stream processing jobs and the associated datasets otherwise.

B. SCALABILITY OF NEPTUNE

A stream processing system should be scalable with respect to the number of concurrent stream processing jobs

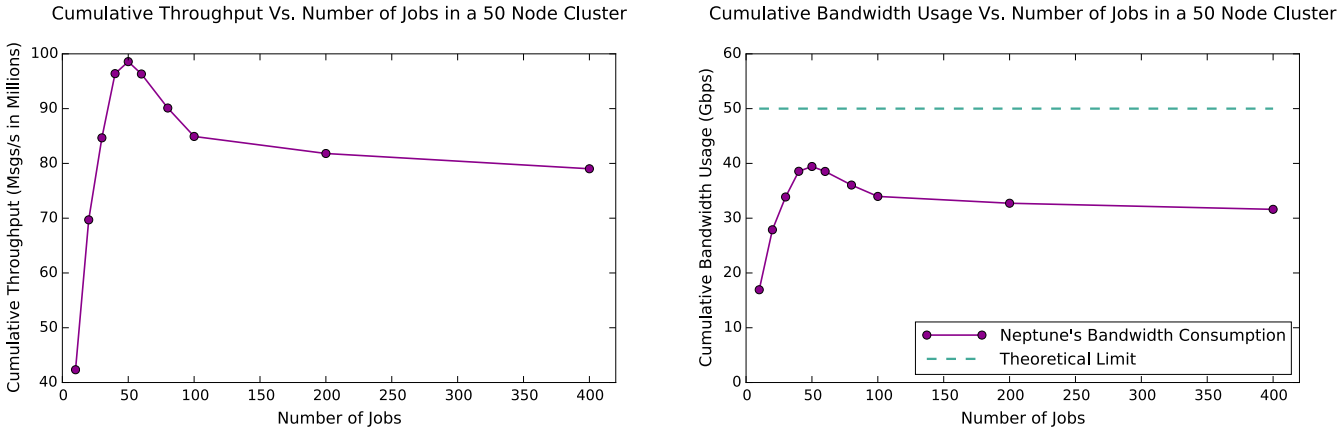

Figure 5. Cumulative throughput and cumulative bandwidth usage with the number of concurrent jobs

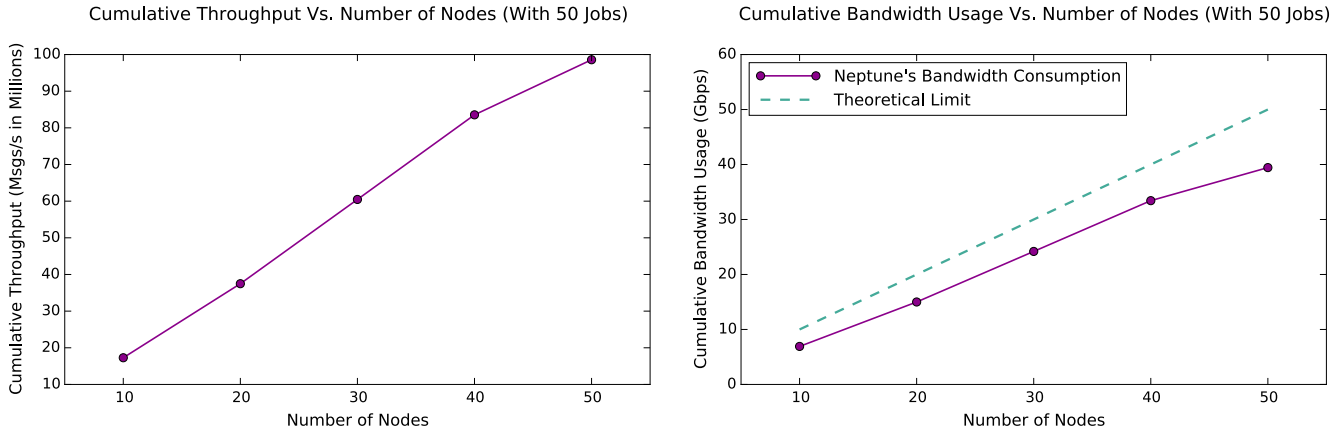

Figure 6. Cumulative throughput and cumulative bandwidth usage with the number of nodes in the cluster

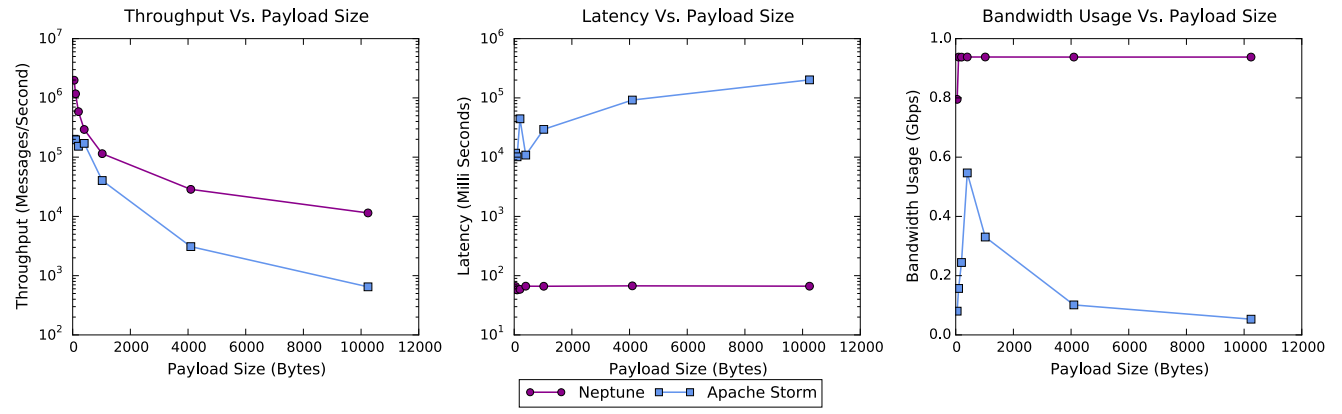

Figure 7. Throughput, end-to-end latency and bandwidth usage Vs. message size in Neptune and Storm

as well as the complexity of stream processing jobs. Complexity of a stream processing job can have many aspects: number of processing stages, complexity of the processing logic embedded in stream processors, level of parallelism and complexity of partitioning schemes.

We have used a two stage stream processing graph to measure the scalability of NEPTUNE as it helped us to create a setup where there is data flow between every pair of nodes in the cluster. Figure 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

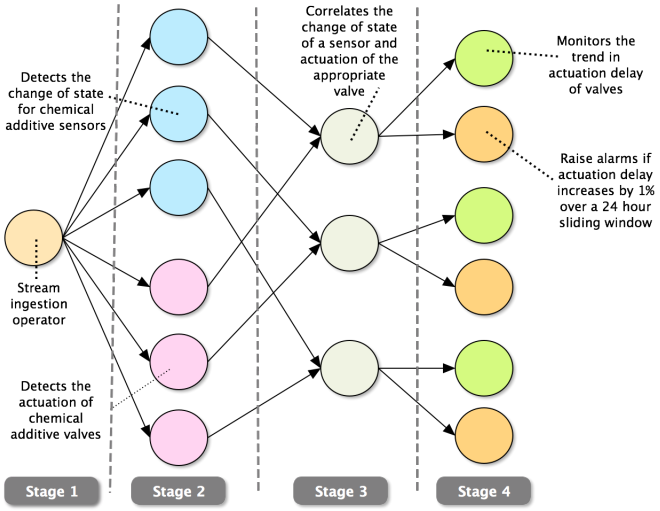

Figure 8. Multi-stage stream processing graph for monitoring a manufacturing equipment

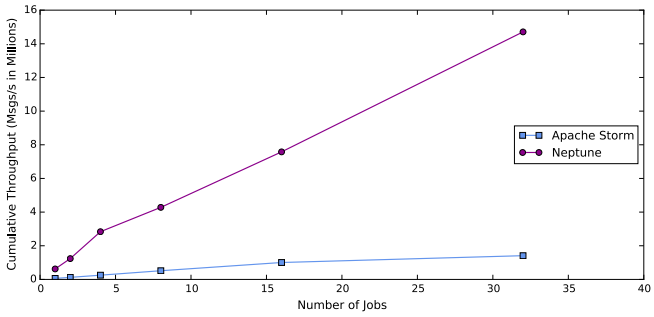

Figure 9. Cumulative throughput Vs. number of concurrent jobs for the manufacturing equipment monitoring use case

is equal to 50. This phase of the plot corresponds to an adequate provisioning of resources. Beyond this point, when the number of jobs increased further, the cluster reaches an overprovisioned 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 and varied the number of machines in the cluster. Figure 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 when the cluster size is further increased.

C. Contrasting Neptune and Apache Storm

We compared the performance of communication modules of NEPTUNE with Apache Storm using the message relay setup depicted in Figure 1. The evaluation metrics: throughput, latency and bandwidth usage were recorded by varying the message size from 50 bytes to 10 KB.

As illustrated in Figure 7, the results of this experiment show that NEPTUNE outperforms Storm in all three metrics. The latency observed with 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 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 a small wait period is introduced between emitting tuples, 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. This issue has been discussed in other Storm performance benchmarks [3], [21].

NEPTUNE was evaluated with a real-world stream processing application involving the monitoring of manufacturing equipment by processing data streams generated by sensors attached to the equipment in real time [19] as illustrated in Figure 8. The system ingests a continuous stream of readings captured by sensors. For this particular use case, we used 6 different data fields and the timestamp out of 66 different data fields available in a single reading. Three of these sensor readings correspond to the states of three chemical additive sensors whereas the other three readings capture the states of the corresponding valves. When the state of a sensor changes, the valves actuate resulting in a change of its state. The objective of the job is to monitor the delay between the sensor state change and actuation of the corresponding valve over a 24-hour time window.

Multiple instances of the stream processing job were executed simultaneously and the cumulative throughput at the stream source was measured across a 50 node cluster for different number of concurrent jobs. As depicted by Figure 9 both systems scale linearly with the number of concurrent jobs. But the throughput is higher in NEPTUNE. With 32 jobs, NEPTUNE's throughput is 8 times higher than Storm.

We also measured cluster wide resource consumption by both systems. Due to a scheduling constraint in Storm that dedicates a Storm worker process to a single job, the maximum number of jobs we could run at a given time was 50 using a cluster of 50 workers. Figure 10 depicts the cluster-wide CPU consumption and memory consumption by both systems. Please note that the CPU usage shows the cumulative value over 8 virtual cores. Memory usage is the amount of memory consumed by the system as a percentage of total available memory. NEPTUNEs CPU consumption is consistently lower compared to the CPU consumption of Storm across all 50 nodes (*p-value for the one tailed t-test* < 0.0001) The high CPU consumption in Storm is due to 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]. NEPTUNE uses a simplified 2-tier thread model, which results in less overhead and reduced queue contention. With respect to memory consumption, there is no noticeable difference between the systems (*p-value for the two-tailed t-test* $= 0.0863$).

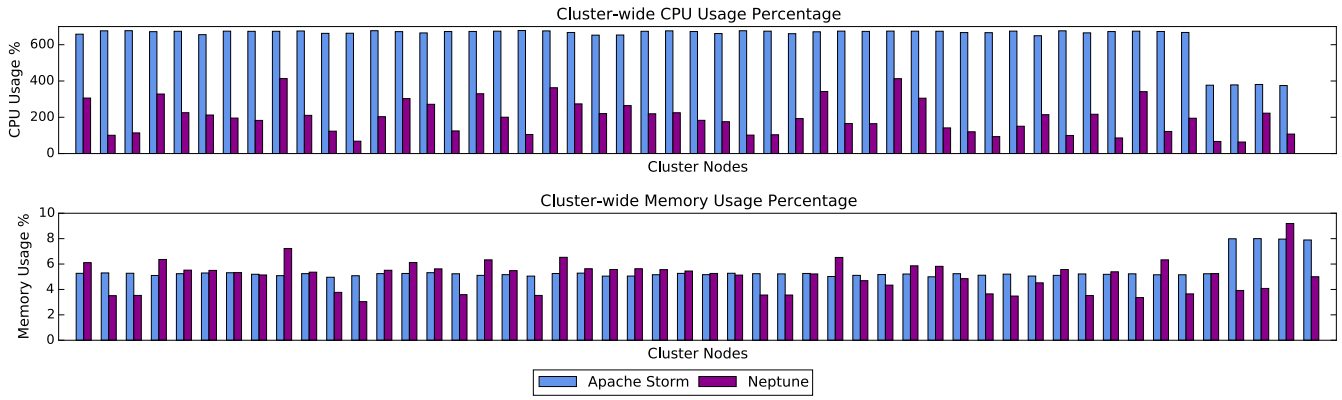

Figure 10. Average cluster-wide resource consumption by Storm and NEPTUNE

V. RELATED WORK

Streaming Databases or Data Stream Management Systems [4], [5], [22] were one of the initial efforts that extended relational database concepts 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) [23] through which users can express processing logic. Though query languages constrain the capabilities of the system compared to a general purpose programming language, they simplify the generation and optimization of query plans.

Most recent attempts [2], [3], [6]–[9], [24]–[26] in stream processing let users express processing logic using imperative languages as opposed to queries. S4 [6] models a stream processing job as a set of *processing elements*. Each 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. Stream Processing Core (SPC) [24] also uses processing elements as the basic unit of computation. The processing elements define their input and output streams through a publish/subscribe model similar to Granules' internal messaging layer. Communication in SPC is abstracted to the *data fabric* layer that supports communications based on pointers, shared memory, or network transport.

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 generate intermediate streams if necessary. Spouts and Bolts form a topology, that is deployed in a cluster of Storm workers. Storm provides different levels of reliable processing guarantees: *at most once* and *at least once* through an acknowledgment scheme. Through the use of Trident abstraction layer, Storm can be extended to provide *exactly once* at the expense of throughput. Heron [3] improves Storm's execution model and resource utilization.

Apache Samza [8] relies on a message broker system called Kafka (kafka.apache.org) for ingesting streams into the system and for inter-task communication. A stream is modeled as a Kafka topic, which is partitioned and distributed across the broker network. The partition for an event published to a given topic is based on the event key. A topic partition corresponds to a stream partition and is processed in a separate Samza task.

Apache Flink [9] provides support for both stream and batch processing in a single framework. Flink supports *exactly-once* semantics, backpressure and state checkpointing for stream operators. Similar to Flink, Spark Streaming [25] also uses a unified programming model for batch and stream processing. Spark Streaming discretizes streams into a series of small batches based on time intervals ($\sim$500 ms) before processing. Spark runtime treats these discretized streams as *resilient distributed datasets (RDDs)* [27] and processes them as regular RDDs created during batch processing. RDDs are kept in-memory, but replicated to multiple nodes in the cluster to provide fault tolerance. Optimizing a stream processing job in Spark for high throughput requires partitioning streams at larger time intervals, which results in latencies in the order of seconds. In NEPTUNE, we are able to achieve high throughput while maintaining sub-second latencies. Google's Millwheel [26] provides fault-tolerant stream processing framework through exactly-once delivery, check-pointing and idempotency built into the framework. Millwheel provides in-order deliveries of streaming data using a watermarking approach on event timestamps. Millwheel is designed for low latency processing rather than high throughput.

VI. 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.

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

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.

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 93.7% 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 10 KB packets) for the 99% of the packets was less than 87.8 ms 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.

Future work will target developing algorithms for fault tolerant processing while reducing overheads that often accompany such schemes. We also plan to add support for a dynamic deployment model that leverages the available capabilities of cluster nodes, properties of the stream processing graph, and the data arrival patterns of data streams.

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

REFERENCES

[1] J. Gubbi *et al.*, "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 Storm," https://storm.apache.org, 2014.
[3] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proceedings of the SIGMOD*. ACM, 2015, pp. 239–250.
[4] A. Arasu *et al.*, "Stream: The stanford data stream management system," *Book chapter*, 2004.
[5] D. J. Abadi *et al.*, "Aurora: a new model and architecture for data stream management," *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.
[6] L. Neumeyer *et al.*, "S4: Distributed stream computing platform," in *ICDMW 2010*. IEEE, 2010, pp. 170–177.
[7] W. Lam *et al.*, "Muppet: Mapreduce-style processing of fast data," *Proceedings of the VLDB*, vol. 5, no. 12, pp. 1814–1825, 2012.
[8] "Apache Samza," http://samza.apache.org, 2014.
[9] "Apache Flink," https://flink.apache.org/index.html, 2015.
[10] M. Zaharia *et al.*, "Spark: cluster computing with working sets," in *Proceedings of the USENIX HotCloud*, vol. 10, 2010.
[11] M. Isard *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.
[12] J. Dean *et al.*, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[13] The Netty project, "Netty Project," http://netty.io, 2015.
[14] S. Pallickara *et al.*, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *IEEE CLUSTER*. IEEE, 2009, pp. 1–10.
[15] S. Kamburugamuve *et al.*, "A framework for real time processing of sensor data in the cloud," *Journal of Sensors*, vol. 2015, 2015.
[16] M. Sstrik, "ZeroMQ," http://aosabook.org/en/zeromq.html.
[17] M. Hirzel *et al.*, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.
[18] S. Fu *et al.*, "An experimental study towards understanding data delivery performance over a wsn link," *arXiv preprint arXiv:1411.5210*, 2014.
[19] "6^{th} acm debs - grand challenge," http://www.csw.inf.fu-berlin.de/debs2012/grandchallenge.html.
[20] 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.
[21] Z. Nabi *et al.*, "Of streams and storms," *IBM White Paper*, 2014.
[22] M. A. Hammad *et al.*, "Nile: A query processing engine for data streams," in *Proceedings of ICDE*. IEEE, 2004, p. 851.
[23] A. Arasu *et al.*, "The cql continuous query language: semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.
[24] L. Amini *et al.*, "Spc: A distributed, scalable platform for data mining," in *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. ACM, 2006, pp. 27–37.
[25] M. Zaharia *et al.*, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX HotCloud*, 2012.
[26] T. Akidau *et al.*, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB*, vol. 6, no. 11, pp. 1033–1044, 2013.
[27] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the USENIX NSDI*, 2012, pp. 2–2.