THESIS

ACHIEVING HIGH-THROUGHPUT DISTRIBUTED, GRAPH-BASED MULTI-STAGE STREAM PROCESSING

Submitted by

Amila Suriarachchi

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

ABSTRACT

ACHIEVING HIGH-THROUGHPUT DISTRIBUTED, GRAPH-BASED MULTI-STAGE STREAM PROCESSING

Processing complex computations on high volume streaming data in real time is a challenge for many organizational data processing systems. Such systems should produce results with low latency while processing billions of messages daily. In order to address these requirements distributed stream processing systems have been developed. Although high performance is one of the main goals of these systems, there is less attention has been paid for inter node communication performance which is a key aspect to achieve overall system performance. In this thesis we describe a framework for enhancing inter node communication efficiency.

We compare performance of our system with Twitter Storm [1] and Yahoo S4 [2] using an implementation of Pan Tompkins algorithm [3] which is used to detect QRS complexities of an ECG signal using a 2 node graph. Our results show our solution performs 4 times better than other systems. We also use four level node graph which is used to process smart plug data to test the performance of our system for a complex graph. Finally we demonstrate how our system is scalable and resilient to faults.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Multi-stage, distributed stream processing is useful in analyzing data streams generated by programs, sensors, or users. Analysis of click-streams, tweets, and stock-quotes are primary examples where such processing is often performed. Most commonly, in such systems packets encapsulate *tuples* representing values corresponding to a set of variables. One advantage of doing multi-stage processing is that individual stages can be scaled horizontally in response to the load i.e., each stage could have multiple instances.

The stages involved in such stream processing can be fluid. It is possible for stages to be added and removed dynamically. Furthermore, different processing pipelines may share stages. Along the same lines, a data packet may be processed within multiple processing pipelines. Individual stages may transform the packets being processed before forwarding it to other stages. Stream processing systems do not place restrictions on the type of packet transformations and modifications that can be performed.

The rates at which the stream packets arrive place unique strains on such systems. Each packet results in a mix of processing and I/O at each stage. Failure to keep up with the data generation rates result in queue build-ups, followed by overflows, and subsequent process failures. Furthermore, in the case of a multi-stage processing pipeline, the processing is only as fast as the slowest stage in the system.

There are two key aspects in processing stream packets: latency and throughput. Latency corresponds to the end-to-end delay as the packet makes its way through the processing pipeline. This metric is useful in characterizing the how timely the processing is. Throughput is a measure of how many packets can be processed per-second within a pipeline. The throughput represents how well the system can cope with rates of data arrivals. As the number of processing stages increase we expect to achieve higher throughput, though latency may increase a little corresponding to the number of hops within the pipeline.

1.1. Research Challenges

In this thesis we consider the problem of designing a scalable framework for the high throughput processing of data streams. There are several challenges in achieving this.

(1) Continuous data arrivals: In the systems we consider data is continually arriving at high rates. Inability to keep pace with the arrival rates will result in buffer overflows.
(2) Shared communication links: Stages comprising the stream processing pipeline may be distributed over multiple machines, and the links connecting these stages are shared Ethernet LANs. Effective utilization of these links is important for achieving high throughput.
(3) Commodity machines: Individual stages execute on commodity machines that have limited memory (order of a few GB) and processing cores (4). So there are limits to the gains that can be accrued by processing a single packet faster.

1.2. Research Questions

Support for high-throughput stream processing involves accounting for aspects relating to memory and processing at individual stages and also for communications between stages comprising the pipeline. Specific research questions that we explore include:

(1) How can we effectively manage memory consumption during processing? During packet processing, tuples must be extracted and processed. This involves allocation and garbage collection of memory. Given the rate at which packets arrive, operations relating memory management must be managed effectively.

(2) How can we ensure effective management of the processing workloads? Since each packet is processed independently, the framework must allow for multiple packets to be processed concurrently.

(3) How can we reduce latencies involved during processing? Since message passing between stages involves network I/O, we need to ensure that these I/O operations can be effectively interleaved with processing. Inefficiencies in interleaving result in serial processing that may cause queue build-ups.

1.3. Approach Summary

The work described here provides a framework for multistage stream processing. Users are only required to specify the processing graph, connectivity between individual stages, and the processing to be performed by each stage. Our framework manages all aspects relating to memory management, efficiencies in communications, and concurrent processing.

Effective memory management is important to avoid buffer overflows and out-of-memory errors. We achieve this by reusing objects. Our approach precludes the need to create an object for every packet arrival. Given the rates at which data packets arrive, memory management costs relating to managing the object lifecycle (creation, initialization, and garbage collection) would be prohibitive. An additional artifact in creation of objects per packet is thrashing due to page faults as the memory consumption at a stage increases.

Given that the stages are dispersed, message passing between stages comprising the pipeline must not be inefficient. Our effort targets effective utilization of the underlying network in two ways. First, we buffer application messages (messages produced by application logic). This buffering targets minimizing the creation of runtime objects for a large number messages while generating large enough messages to utilize the underlying network. The amount of messages get buffered can be configured to balance the competing pulls of timely delivery and high-throughput transfer. Second, our serialization scheme allows for compact over-the-wire representation of the tuples within the packets. Inefficiencies in serializations schemes result in verbose representations that increase the network footprint of individual packets.

Our framework also focuses on efficiencies in processing these packets. There are two ways in which we accomplish this. First, we rely on thread pools at each stage. Packet processing is performed within threads and the thread-pool is sized so that we balance concurrency gains and thread-switching overheads. Second, we harness non-blocking I/O. Given that packet processing at each stage involves network I/O the use of non-blocking I/O allows us to interleave processing and I/O much more effectively.

We evaluate the suitability of our methodology by profiling its performance with real applications. This includes real-time ECG processing where we are analyzing waveform data in real-time to perform QRS-complex detection using the well-known Pan-Tompkins [3] algorithm and a multi-stage query to analyze smart plug data. Finally, we also contrast the performance of our system with systems such as Twitter Storm [1] and Yahoo S4 [2] that do not incorporate some of the efficiencies in our system. Our results validate the suitability of our approach with substantially higher throughputs that what can be achieved in these systems. Furthermore, several aspects of our methodology can be incorporated into these aforementioned systems to improve their performance as well.

1.4. Thesis Contributions

In this thesis we have described how to achieve high-throughput, multi-stage distributed stream processing. Our specific contributions include the following:

(1) Our approach identifies aspects that play a key role in achieving high-throughput processing. High-throughput processing is possible by taking a holistic view of the system encompassing memory, computing, and communications efficiencies. This thesis demonstrates how combining object reuse, serialization efficiencies, concurrent processing, buffering application messages, and the use of non-blocking I/O allows us to achieve high throughput.

(2) The per-packet processing latencies that we achieve in our system demonstrate suitability for applications where such timely processing is necessary.

(3) We provide a simple framework for user to express their processing graph. Users only need to specify this graph and the processing that must be performed for each stage. Users are freed from the dealing with concurrent, I/O, and memory efficiency issues.

(4) Our results demonstrate the suitability of our approach. We have contrasted the performance of our system with Twitter Storm [1] and Yahoo S4 [2].

1.5. Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of our system at user level. Chapter 3 illustrates the underlying design of the system. Chapter 4 analyses the experiment results which demonstrate the performance of our solution. Chapter 5 discusses the related work and finally Chapter 6 provides conclusions and future work.

CHAPTER 2

SYSTEM OVERVIEW

Our system comprises a set of processes where each process receives events, processes them, and forwards them to the next stage. We use the term adapters for special processes that receive events from external systems. These processes and adapters form a graph, which represents how a particular event stream is processed within the system. Our current implementation has adapted the notion of topology from Twitter Storm [1] and the notion of cluster from Yahoo S4 [2]. The remainder of this section describes our system from a user perspective. In the next section we provide a detailed explanation about how we have incorporated efficiencies into inter-process communications.

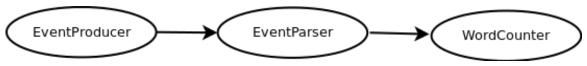


Figure 2.1. An example process graph with 3 nodes to count words.

2.1. Process Graph

A process graph mainly comprises adapters, processes that execute the processing logic, and their interaction patterns. Adapters receive events from external systems and forward them to other processes. Processes process events according to a given logic and emit the generated events either to other processes or to outside systems. Figure 2.1 shows an example process graph with 3 nodes. *EventProducer* reads data from an external system and emits row events to *EventParser*. *Eventparser* processes the event and sends each word as an event. *WordCounter* receives each word and emits the each word count periodically.

2.2. Runtime Graph

When the system is deployed, there can be multiple instances of a particular stage to handle higher loads making the system scalable. Figure 2.2 shows a possible runtime graph for the process graph shown in Figure 2.1. Here each parent process instance sends events to two child process instances. However, in this case the parent process instance has to pick a child process instance to send the message as well. One way of handling this problem is to pick a random node. Another approach is to send messages with same key to particular process instance, using a partitioning function similar to the shuffle phase in MapReduce [4].

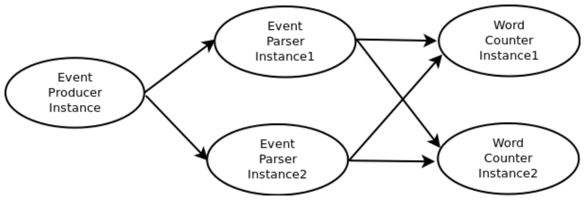


Figure 2.2. Runtime graph for process graph given in Figure 2.1.

2.3. API

In this section, we explain our API that can be used to define custom events, processes and adapters. We provide clear interfaces (Table 2.1) to abstract out all the communication complexities from application developers.

Table 2.1. Interfaces, methods and method parameters of client API

Interface Name	Method	Parameters
Event	getKey	
	serialize	DataOutput
	parse	DataInput
Adapter	start	
	initialize	Container
Processor	onEvent	Event
	initialize	Container

The `Event` interface has a method to get the key for a given event. This key is used in sending messages to next processes as described earlier. `Serialize` method is used to directly convert the event attributes to binary format and `parse` method is used to directly retrieve event attribute values from binary format. When implementing these methods developers can define attribute serialization order in their `serialize` method and use the same order during `parse` method to avoid metadata passing with the binary format. `Processor` interface contains a method called `onEvent` which is invoked by the underlying framework when it receives an event to that process. The `start` method of the `Adapter` interface is invoked by the framework and that can be used to pull events from external systems once the system starts. During initialization both processes and adapters receive the container that can be used to send events to other processes.

2.4. Deployment

As shown in Figure 2.3, a deployment of the system consists of a manager node and a set of worker nodes. Worker nodes are grouped into clusters so that the application users can specify the clusters to which each process needs to get deployed. The Manager node processes applications, deploys them to worker nodes, and initiates the processing by starting the adapters.

2.5. Application Development

An application specifies the runtime graph to process a particular event stream and consists of a set of processes, adapters and events and their interactions. Processes, adapters and events can be developed by implementing the respective interfaces given in Table 2.1. Then the interactions can be specified using a JSON file.

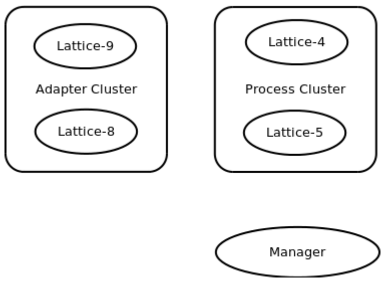


FIGURE 2.3. Deployment view of the system with 6 worker nodes into 3 clusters and a Manager node.

```
{
    "processors": [
        {
            "name": "parser",
            "className": "edu.colostate.cs.sample.EventParser",
            "cluster": "parser",
            "instances": 2,
            "eventType": "edu.colostate.cs.sample.Word",
            "receivers": [
                {
                    "name": "producer",
                    "type": "random"
                }
            ]
        },
        {
            "name": "counter",
            "className": "edu.colostate.cs.sample.WordCounter",
            "cluster": "count",
            "instances": 2,
            "receivers": [
                {
                    "name": "parser",
                    "type": "key"
                }
            ]
        }
    ],
    "adapters": [
        {
            "name": "producer",
            "className": "edu.colostate.cs.sample.EventProducer",
            "cluster": "adapter",
            "instances": 1,
            "eventType": "edu.colostate.cs.sample.Line"
        }
    ]
}
```

Listing 1: Configuration for runtime graph given in Figure 2.2

As shown in Listing 1, the *instances* parameter can be used to specify the required number of instances to be deployed. The *receivers* parameter is used to specify from which process it expects to receive messages and how the sending process should distribute the events among instances. The *key* type indicates that the events with the same key must be received by the same instance. Finally users can deploy the application to manager which processes them and deploys to workers.

CHAPTER 3

SYSTEM DESIGN

In this section we describe key aspects of our system design which increase the throughput of inter node communication and underlying framework implementation details. First of all it is worth to note that even with one TCP connection, it is possible to saturate the LAN network link usage, if the sending and receiving processes send and receive byte arrays without any delay even with 256 bytes array messages. However serializing java object messages to byte arrays and deserializing byte arrays back to java objects takes a considerable amount of time. Hence a key aspect of improving the system performance is to reduce this message conversion delay. We use multitasking and bulk message serializing to solve this problem.

In this design, we use thread pools both at the client side and the server side to process messages while sharing a single TCP connection. Firstly, both thread pools access the underlying connection resource only to send and receive byte arrays making message serialization and deserialization process parallel. Secondly, it buffers the messages and performs a batch serialization. This ensures large enough binary messages to fully utilize the underlying network and also minimize the number of runtime objects created for message conversion. Figure 3.1 shows the design involved in sending a message from one process to another.

Let's assume that the client side process requires to send a set of messages to server side process. In order to send these messages, first the client process threads send these messages to its *ElementContainer*. *ElementContainer* holds all the streams (this is an abstraction of a link in the process graph) to which these messages need to be sent and it pushes each message to all the streams. *Stream* then decides the target node and buffers messages for a configured

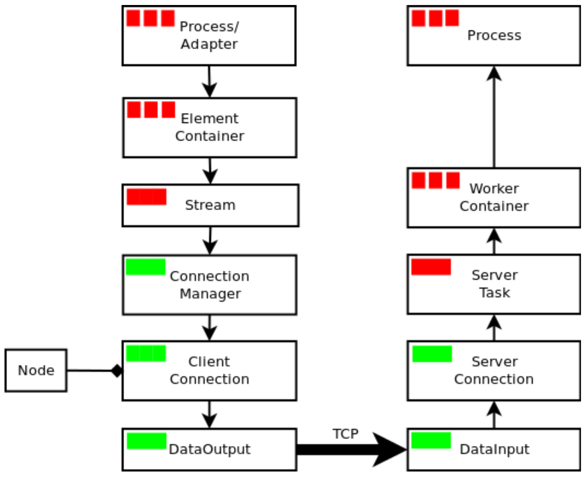


Figure 3.1. Communication between two Process

amount of size. Then it passes the buffered message list along the target node details to *ConnectionManager* which holds *ClientConnections* for each node. *ConnectionManager* serializes the message list and sends the byte array to correct *ClientConnection* using the target node. *ClientConnection* uses the *DataOutput* which wraps the TCP connection to send the message to server side.

At the server side there is a set of *ServerTasks* that read received messages from a pool of *DataInput* streams available in the *ServerConnection* (we register these connections during creation time) and send them to appropriate processes. First a server task acquires a *DataInput* stream and reads the binary message and release it. Then it deserializes the binary message using the sending process id to identify the event type. After that it passes this message to the *WorkerContainer* which holds all processes. Finally *WorkerContainer* dispatches this message to correct process using receiving process id. Message ordering happens at the process level if required.

3.1. Application message processing

We use the the *DataOutputStream* with *ByteArrayOutputStream* to serialize application messages to binary format in batches and *DataInputStream* with *ByteArrayInputStream* to

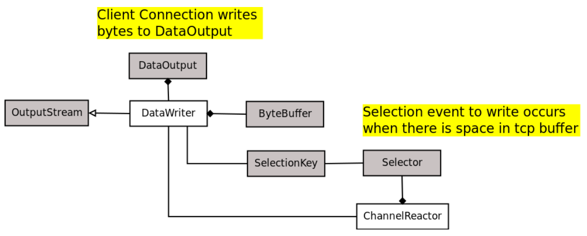

Client Connection writes bytes to DataOutput
DataOutput
OutputStream
DataWriter
ByteBuffer
Selection event to write occurs when there is space in tcp buffer
SelectionKey
Selector
ChannelReactor

FIGURE 3.2. OutputStream interface implementation with java blocking API

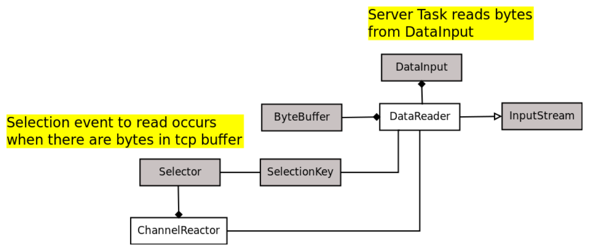

Server Task reads bytes from DataInput
DataInput
ByteBuffer
DataReader
InputStream
Selection event to read occurs when there are bytes in tcp buffer
Selector
SelectionKey
ChannelReactor

FIGURE 3.3. InputStream interface implementation with java blocking API

deserialize from the binary format. The first four bytes of the binary format are used to specify the number of messages in the batch. Deserializer uses this information to read the exact amount of messages from the binary message. As mentioned earlier, we minimize these object creation by batch processing messages. We also experimented with Kryo [5], which provides an efficient binary format in terms of number of bytes used but it results in a dropping throughput compared to the above mechanism.

3.2. Binary Message Communication

Once the application messages are converted to binary format, binary messages need to be send to the server side. At the client side we use *DataOutput* to write the message length with the message, and at the server side we use *DataInput* to read messages according to the length. *DataOutput* and *DataInpu*t interfaces requires *OutputStream* and *InputStream* objects to write and read data. We implement those interfaces with the java non-blocking I/O API as given bellow. This implementation allow as to reuse the underlying *ByteBuffer* objects.

At the client side (Figure 3.2), *DataWriter* (implementation of OutputStream) contains a *ByteBuffer*. When a higher layer method writes data to the *DataOuput*, *DataWriter* put those bytes into its *ByteBuffer*. When a selection event occurs, *Datawriter* writes its *ByteBuffer* to the underlying *SocketChannel*. At server side (Figure 3.3), *DataReader* contains a *ByteBuffer* similar to client side. When a selection event occurs *DataReader* input bytes from the *SocketChannel* to its *ByteBuffer*. When a *ServerTask* reads data through *DataInput*, it reads data from the *ByteBuffer* and passes to higher layer.

3.3. Load balancing

Load balancing is the primary means of achieving data parallelism in stream processing. Each stage of the processing graph can be deployed on several nodes and can balance the load for the next stage. In our framework, load balancing happens at the *Stream* level specific to the stream. *KeyStream* balances the load by distributing keys among the nodes. *RandomStream* balances messages randomly among next level of nodes. As mentioned above, buffering of the messages happens at the *Stream* level. Further, users can plug in any custom stream with a specified partitioning function as well.

3.4. Fault tolerance

Fault tolerance is the ability of a system to operate in the presence of failures. These failures can be node failures or network failures and requirements to operate are depend on the system. In our framework, we provide fault resilient or ability to balance the load for existing nodes in the presence of node failures. If a receiving node fails then there will be a failure at the TCP connection and we close such connections and remove nodes from the *Streams* (Stream objects hold the receiving node details). This prevents further routing messages to failure nodes.

CHAPTER 4

PERFORMANCE BENCHMARKS

We conducted several performance benchmarks to measure the throughput and scalability of our system. All these tests were performed on a LAN with a network bandwidth of 1 Gbps. All nodes are Intel(R) Xeon(R) 2.4GHz, 4 core duo machines with 16 GB of memory. We used Apache S4 0.6.0 version and Apache storm 0.9.2 version. Sample code used for all tests can be found here [6].

4.1. Effect of message buffer size and thread pool

For this experiment, we used the graph shown in Figure 4.1 to process ECG signal data. The *EventProducer* reads a file containing over 7500000 ECG records and pushes events with multiple threads. *EventReceiver* receives the ECG events, processes them and calculates heart rate interval periodically using Pan Tompkins algorithm [3].

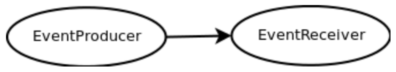


Figure 4.1. ECG Process Graph

We measured the throughput and latency between two nodes for different thread pool and message buffer sizes to understand the effect of those parameters. As shown in figure 4.2 throughput increases clearly with the message buffer size and increase with thread pool size for larger message buffer sizes. Therefore we can conclude both make a positive impact on the throughput. As shown in Figure 4.3 latency also tend to increase with the message buffer size. However lower thread pool sizes causes higher latencies compared to higher thread pool sizes. We measured the latency as the time gap between processing the message at the *EventReceiver* and reading the record at the *EventProducer*. Therefore this includes the time it spends at the message buffer as well. Lower thread pool sizes reduces the message processing speed and hence can cause messages to wait longer times in the queue increasing the latency.

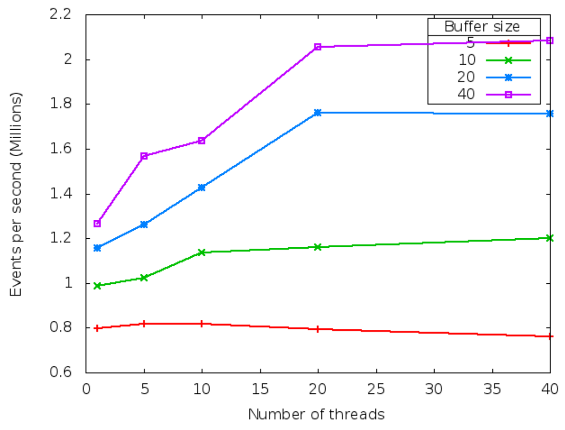


FIGURE 4.2. Throughput variation with the thread pool size and message buffer size.

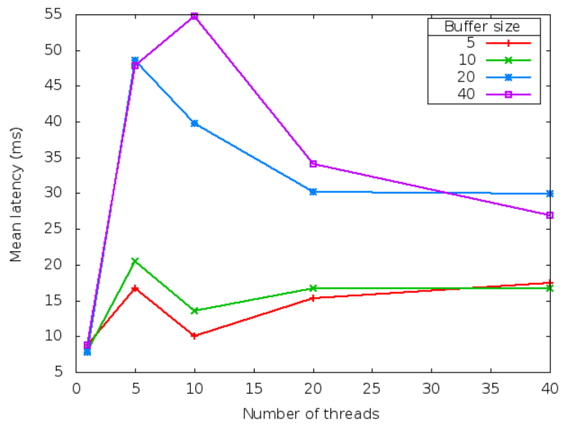


FIGURE 4.3. Latency variation with the thread pool size and message buffer size.

4.2. LOAD BALANCING

We used the same graph (Figure 4.1) used in our previous experiment with 1,2 and 4 *EventReciver* nodes with one *EventSender* node for this experiment. Since a thread pool size of 40 and a message buffer size of 40 gives highest throughput, we used these values with our solution. Further, we compared our system with Yahoo S4 [2] and Twitter Storm [1]

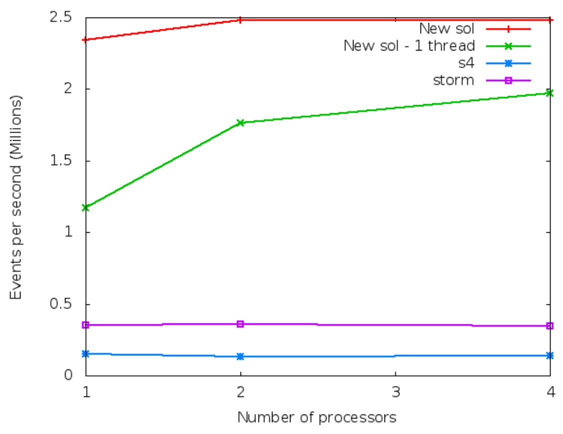


FIGURE 4.4. Throughput comparison of S4, Storm and our solution.

with the same graph. Twitter Storm [1] places tasks arbitrarily within the available nodes. Therefore we used 1 task for *EventSender* and 1,2 and 4 tasks for *EventReceivers* to place them in different nodes. This makes Twitter Storm [1] execute only one thread to process data in each node. In order to compare this situation, we measured our system performance with 1 thread and a 10-message buffer size as well.

Figure 4.4 shows the throughput of the systems. All systems keep initial throughput when increasing nodes. But our solution outperforms the other systems. Figure 4.5 shows the network bandwidth at each node (measured using atop linux command). For each framework the sending node consumes the same amount of network bandwidth since it sends all messages to other nodes. For the receiving side each receiving node consumes the corresponding portion of the bandwidth according to the number of receiving nodes. For an example 4-node configuration, each receiving node consumes only one-fourth of the sending node bandwidth. Compared to the other systems our solution harnesses all the available bandwidth.

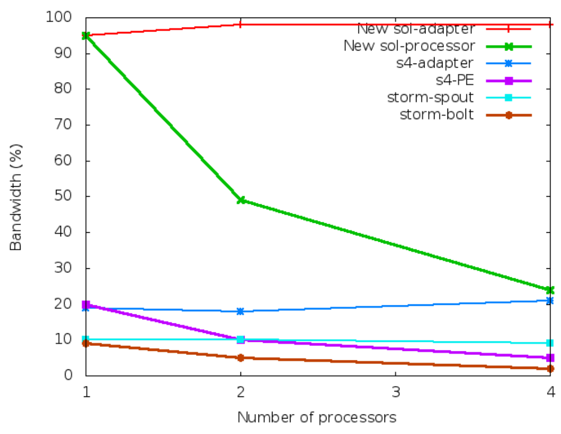


Figure 4.5. Network usage at each node. For multiple receiving nodes it is the network utilization at a one node.

4.3. Multistage Graph Processing

We performed a multi-stage graph test for our system using the 3-level graph shown in Figure 4.6. The data as well as the processing logic were obtained from the Grand Challenge problem at 8th ACM International Conference on Distributed Event Based Systems [7]. This project explores the possibility of using stream processing to process data generated in the smart-grid domain. The data is generated by sensors attached to a device called a smart plug which is plugged as an intermediate layer between a power outlet and the plug of a home appliance. The sensors capture various measures related to power consumption of each connected device and send these measurements continuously to a central processing system. Each household contains several such smart plugs that will report sensor readings once in every second. At the central processing system, these data should be processed in order to forecast load and generate load statistics for real time demand management. We implemented the first part of this project that predicts the load using an algorithm. This algorithm divides the events to fixed intervals called time slices and uses average value of time slices to predict the average of feature time slices. We used the publicly available dataset to generate the events.

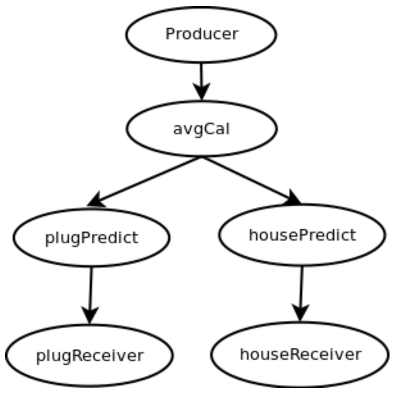
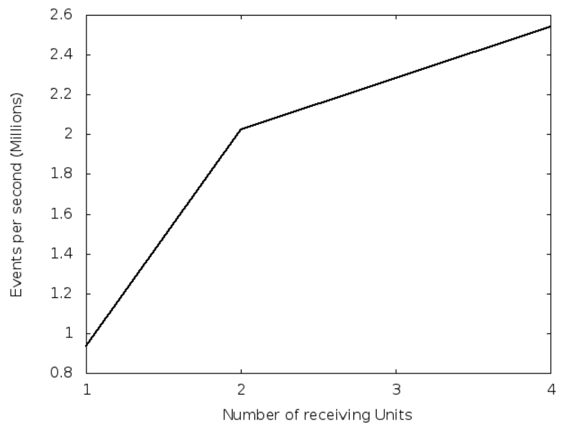


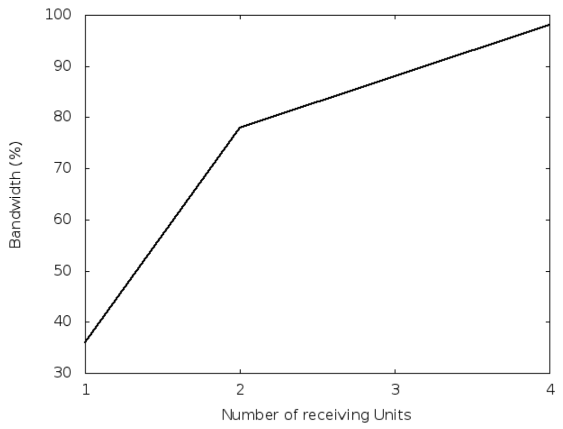
FIGURE 4.6. Multilevel Node Graph

As shown in Figure 4.6, the *Producer* node reads the data file and emits events to the *avgCal* node which calculates the last time slice average and sends the same event to both *plugPredict* and *housePredict* processors. Both *plugPredict* and *housePredict* processes predict the next values and send events to receivers. The original problem only requires us to send those prediction events in 30s intervals. But we used a prediction event for each message to observe how the system performs under high load.

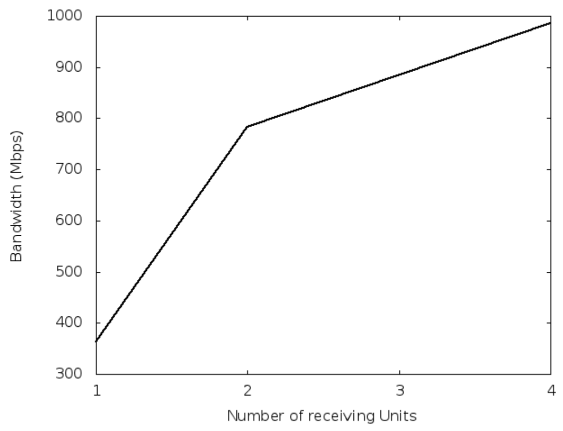
As in the earlier case, we conducted our experiments using one producer and incrementing the other processing nodes by 1, 2 and 4 times to measure the throughput increase at the producer. We ran each node in a separate machine so that our receiver configurations used 5, 10, 15 machines respectively. All nodes were configured to use both thread pool and message buffer size of 40. We measured the throughput, network bandwidth and the CPU load average at the producer to examine the scalability of the system. Since the one receiver unit throughput is greater than that of the Twitter Storm [1] one node scenario, we only used our system for this experiment. As shown in Figure 4.7 system scales up until it fully utilized the available network bandwidth.



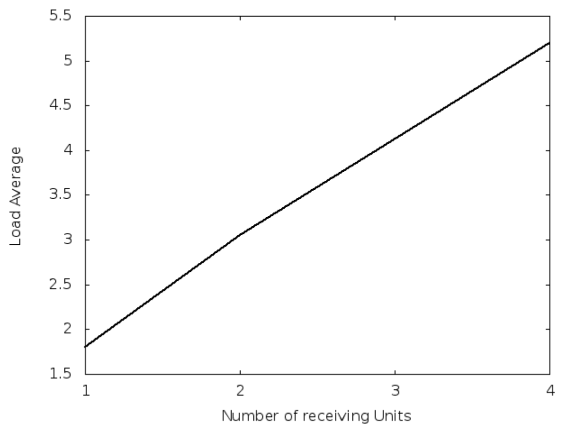
(A) Throughput of the System



(B) Network Usage (%) at Sender



(C) Network Usage Bandwidth at Sender



(D) CPU Load Average at Sender

FIGURE 4.7. Performance of Multistage Graph processing.

4.4. Scalability and Latency

We analyze the scalability of our framework using cumulative throughput similar to other experiments found in research literature [8] [9]. In this experiment, a given set of nodes communicate directly with each other using TCP connections. We used same message type (of size 46 bytes in serialized format) that we used for our earlier experiment. The cumulative throughput we report is for application-level message rate excluding the TCP overhead; all communications are through the network. For all experiments we use a 20

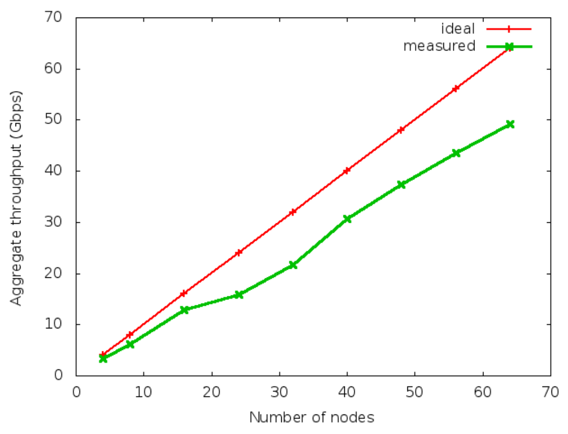


FIGURE 4.8. All to All exchange Throughput

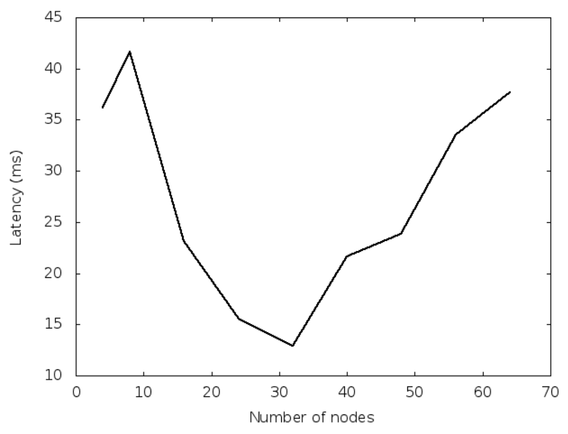


FIGURE 4.9. All to All exchange Latency

message buffer. Since each node is capable of sending and receiving 1Gbps of data, the ideal throughput is same as number of nodes times 1Gbps.

As shown in Figure 4.8 system linearly scales with the number of nodes. Figure 4.9 shows the mean message latency for all messages. Our latency corresponds to message latency for data messages. It is worth noting that the latency can be reduced by reducing the buffer size, but this would reduce the throughput.

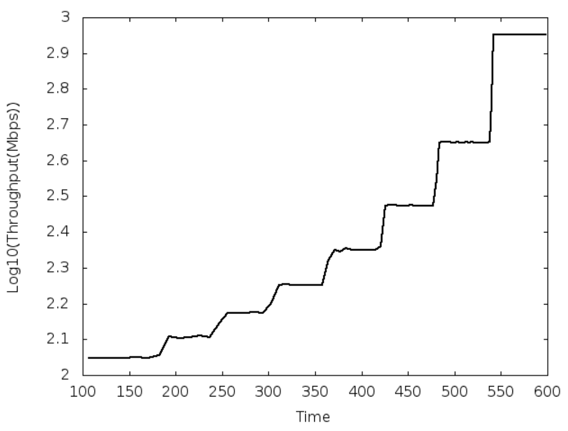


FIGURE 4.10. Receiver throughput variation at the last node.

4.5. FAULT TOLERANCE

For this experiment we analyzed the resilience of the system in the presence of node failures. Initially we had a load balanced system where the event producer sends events to 8 receiving nodes. The producer produced messages at the rate of around 2500000 and each receiving node receive messages at a rate around 320000 ($\sim$ 2500000 / 8). Then we shutdown one process by one process within 1 minute interval. This increased the throughput of each receiving node. Figure 4.10 shows the throughput received at the last node to shutdown with the time. As depicted in the Figure 4.10 when a node fails, the system automatically removes the node from its configurations and routes the messages to other nodes and balances the load among existing nodes.

CHAPTER 5

RELATED WORK

Much of the inspiration for stream processing systems can be traced back to data stream management systems (DSMS). These systems support a window based query language. Most of these languages are derived from SQL, for instance STREAM [10] defines a language called *Continuous Query Language(CQL)* which has similar syntactic characteristics as SQL. Aurora [11], STREAM [10] and Nile [12] are some of the prominent implementations of DSMSs. Complex event processing(CEP) systems such as Esper [13], Siddhi [14], Cayuga [15] also share many characteristics with DSMSs except for their use cases. CEP systems are capable of handling multiple incoming event streams(which are similar to the input streams) and identifying patterns across streams which are of interest to the end users. However these systems do not support user defined logic and hence not suitable to implement complex logic based on machine learning techniques.

MapReduce [4] is a widely used technique to process batch data. Apache Hadoop [16] is a widely used implementation of MapReduce. In a MapReduce environment a problem is partitioned into smaller parts identified by a key and process parallely. Although these systems support user defined functions, data processing flow of these systems are fixed. Ciel [17] address this problem by dynamically generating data flow graph. Since all these systems communicate through file system they inherently not suitable for real time processing [18].

Stream Processing Core (SPC) [19], Yahoo S4 [2] and Twitter Storm [1] address the above issue by supporting direct communication. These systems are based on the Actor model [20] and each system has a notion of processing element or computation. Processing Elements are used to perform user defined logic on the receiving events and emit newly generated events to other processing elements. Muppet [18] introduces a programing style called MapUpdate to overcome issues with MapReduce [4]. Further it provides a detailed explanation about why MapReduce [4] is not suitable for fast data processing. In addition to basic communication these systems provide fault tolerance features such as reliable message delivery and state preservation. However most of these systems do not focus on improving inter node communication performance.

Spark [21] introduces its Resilient Distributed Datasets RDDs [22] to achieve better fault tolerance by storing the operations and replaying them instead of replicating data itself or use checkpointing to store data periodically. Spark Streaming [9] extends this concept to distributed stream processing with the concept of discretized events. Spark streaming [9] processes data as batches while storing them as RDDs [22] to achieve higher throughput. This batch processing may introduce latencies which are not acceptable for some applications. Further it is not clear how to implement complex event processing algorithms such as we use our benchmarks with Spark Streaming. MillWheel [23] is the stream processing system used at Google to process web queries. MillWheel [23] provides reliable message processing as well as state persistence for its graph based computations. Unlike our system, MillWheel treats messages as binary messages leaving message parsing and serialization to the application layer. Naiad [8] is a streaming system which aims to provide high throughput as of batch processors while providing low latencies suitable for stream processors. In order to mark the boundaries of event streams it uses the concept of epoch label.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis we have described our approach to achieving high-throughput multi-stage stream processing. We have used application message buffering to improve serialization and deserialization process and thread pools to improve parallelism.

We have validated the soundness of our methodology using empirical benchmarks that contrast its performance with well-known systems such as Twitter Storm [1] and Yahoo S4[2]. Our approach allows cumulative throughputs that significantly outperform these aforementioned systems. In a two stage setting we are able to achieve a processing throughput of 2.3 million messages per-second and a network utilization of 95% (950Mbps/1Gbps) even with one receiver. In a four-stage setting we are able to achieve a processing throughput of 2.5 million messages per-second and a network utilization of 98% (982Mbps/1Gbps) with four receiving units.

Our future work will mainly target two areas: reliable message delivery and compression. We will focus on implementing a reliable message delivery protocol to support at least one guarantee on top of our communication framework. The work on compression will target reducing the overthe-wire footprints of the packets. We will explore the use of compression algorithms while ensuring that the gains in message size reduction do not result in unacceptable latency overheads and also that the throughput improves.

BIBLIOGRAPHY

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, ACM, 2014.

[2] 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.

[3] B.-U. Kohler, C. Hennig, and R. Orglmeister, "The principles of software qrs detection," *Engineering in Medicine and Biology Magazine, IEEE*, vol. 21, no. 1, pp. 42–57, 2002.

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

[5] "Kryo : Fast, efficient java serialization and cloning." `https://code.google.com/p/kryo/`.

[6] "Solution code." https://github.com/amilaSuriarachchi.

[7] Z. Jerzak and H. Ziekow, "The debs 2014 grand challenge," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 266–269, ACM, 2014.

[8] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, ACM, 2013.

[9] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438, ACM, 2013.

[10] 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.

[11] 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 JournalThe International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[12] 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.

[13] "Esper." http://esper.codehaus.org/.

[14] 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.

[15] 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.

[16] "Apache hadoop." http://hadoop.apache.org/.

[17] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: A universal execution engine for distributed data-flow computing.,"

[18] 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.

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

[20] G. A. Agha, "Actors: a model of concurrent computation in distributed systems," 1985.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets,"

[22] 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.

[23] 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.

APPENDIX A

SAMPLE PROCESS IMPLEMENTATION

This section describes a sample implementation using our framework for the scenario defined in Chapter 2. The first step of implementing a process graph is to determine the information passes between each edge. These information can be represented as events and implemented by extending the existing abstract *Event* class. We can define the event between the *EventProducer* and *EventParser* as a *Line* which stores the line as a string. Similarly event between the *EventParser* and *WordCounter* can be defined as *Word* which stores the word as string.

```java
public class Line extends Event {
    private String line;

    public Line(String line) {
        this.line = line;
    }

    public Object getKey() {
        return this.line;
    }

    public void serialize(DataOutput dataOutput) throws MessageProcessingException {
        try {
            dataOutput.writeUTF(this.line);
        } catch (IOException e) {
            throw new MessageProcessingException("Can not write message ");
        }
    }

    public void parse(DataInput dataInput) throws MessageProcessingException {
        try {
            this.line = dataInput.readUTF();
        } catch (IOException e) {
            throw new MessageProcessingException("Can not read the message ");
        }
    }

    public String getLine() {
        return line;
    }
}
```

Listing 2: Implementation of *Line* Event

As shown in Listing 2 the *Line* event contains a string field called line to store the line details. `Serialize` method is used to write the line detail to dataInput and `parse` method is used to store the value back. Listing 3 shows a similar implementation of *Word* event.

```java
 public class Word extends Event {

    private String word;

    public Word(String word) {
        this.word = word;
    }

    public Object getKey() {
        return this.word;
    }

    public void serialize(DataOutput dataOutput) throws MessageProcessingException {
        try {
            dataOutput.writeUTF(this.word);
        } catch (IOException e) {
            throw new MessageProcessingException("Can not write the message ", e);
        }
    }

    public void parse(DataInput dataInput) throws MessageProcessingException {
        try {
            this.word = dataInput.readUTF();
        } catch (IOException e) {
            throw new MessageProcessingException("Can not read the message ", e);
        }
    }

    public String getWord() {
        return word;
    }

    public void setWord(String word) {
        this.word = word;
    }
}
```

Listing 3: Implementation of *Word* Event

After defining the events, we can implement adapters which reads events from external sources and passes them to the system. Listing 4 shows a simple implementation of *Event-Producer*. At the initialization underlying framework invokes the `initialize` method and provides the *Container* object and parameters. *Container* object is an implementation of the underlying *ElementContainer* which should be used to emit events from the *Adapter*. Parameters can be used to obtain any user defined value from the deployment descriptor.

After initializing all adapters and processes container invokes the start method to start the adapter. At this method users can start multiple threads and start sending messages. In this example it simply sends a hard coded message in an infinite loop.

```java
public class EventProducer implements Adaptor {

    private Container container;

    public void start() {
        while (true) {
            try {
                this.container.emit(new Line("Test word to process"));
            } catch (MessageProcessingException e) {
            }
        }
    }

    public void initialise(Container container, Map<String, String> parameters) {
        this.container = container;
    }
}
```

Listing 4: Implementation of *EventProducer* Class

Finally processes which process the events generated from the adapters can be implemented. Listing 5 shows the *EventParser* process. As in adapters, all processes get *Containter* object through `initialize` method. When an event received for this process, the `onEvent` method is called with the received event. This event can be cast to real event type and process accordingly. In this case the receive event is cast to *Line* type to get the line send from the *EventProducer*. *EventParser* splits this line and emits individual words to be processed at *WordCounter*. As shown in Listing 6, *WordCounter* counts each word.

```java
public class EventParser implements Processor {

    private Container container;

    public void onEvent(Event event) {
        String line = ((Line) event).getLine();
        for (String word : line.split(" ")) {
            try {
                this.container.emit(new Word(word));
            } catch (MessageProcessingException e) {
            }
        }
    }

    public void initialise(Container container, Map<String, String> parameters) {
        this.container = container;
    }
}
```

Listing 5: Implementation of *EventParser* Class

```java
public class WordCounter implements Processor {

    private Map<Object, Integer> countMap = new HashMap<Object, Integer>();

    public void onEvent(Event event) {
        Word wordCount = (Word) event;
        Integer count = this.countMap.get(wordCount.getWord());
        this.countMap.put(wordCount.getWord(), count + 1);
    }

    public void initialise(Container container, Map<String, String> parameters) {

    }
}
```

Listing 6: Implementation of *WordCounter* Class