NATIONAL AND KAPODESTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSC THESIS

An Efficient Decentralized Streaming Model

Evangelos Grigoriou

Supervisors: **Alexios Delis,** Professor NKUA

ATHENS

OCTOBER 2019

ΕΘΝΙΚΟ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ

An Efficient Decentralized Streaming Model

Ευάγγελος Γρηγορίου

Επιβλέποντες: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ
ΟΚΤΩΒΡΙΟΣ 2019

BSC THESIS

An Efficient Decentralized Streaming Model

Evangelos Grigoriou
S.N.: 1115201000027

SUPERVISORS: **Alexios Delis,** Professor NKUA

ΠΤΥΧΙΑΚΗ

An Efficient Decentralized Streaming Model

Ευάγγελος Γρηγορίου
Α.Μ.: 1115201000027

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξιος Δελής**, Καθηγητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Πρόσφατα, όλο και μεγαλύτερα ποσά δεδομένων παράγονται από διάφορες πηγές. Τα πλαίσια λογισμικού ροής για Μεγάλα Δεδομένα βοηθούν στην αποθήκευση, ανάλυση και στην απόσπαση χρήσιμων πληροφοριών, από τέτοιου είδους δεδομένα που παράγονται συνεχώς. Υπάρχουν αρκετά τέτοια πλαίσια λογισμικού ροής, όπως το Apache Storm, το Apache Spark και το Apache Flume.

Στην παρούσα πτυχιακή εργασία παρουσιάζουμε ένα μοντέλο αποκεντρωμένης επεξεργασίας ροής. Χρησιμοποιεί ένα πρωτόκολλο DHT για να επιτευχθεί μία αρχιτεκτονική πολλών αφέντων-πολλών εργατών και σε κάθε εργασία να αντεθεί ο δικός της αφέντης.

Για κάθε εργασία δημιουργούνται όμοιες ομάδες χρησιμοποιώντας τις ιδιότητες δρομολόγησης του συστήματος, με αποτέλεσμα τον σχηματισμό ενός ιεραρχικού δέντρου, αποτελούμενο από κόμβους μου συμμετέχουν στο δίκτυο. Η ρίζα αυτού του δέντρου ενεργεί ως ο αφέντης της ομάδας και είναι υπεύθυνος για τον συγχρονισμό των μελών της ομάδας.

Ο κάθε κόμβος καταναλώνει ζωντανά αρχεία καταγραφής δεδομένων, τα οποία αναλύονται σε μικρές παρτίδες και αποθηκεύονται σε μία δομή δεδομένων που χρησιμοποιεί την μνήμη αποδοτικά. Οι κόμβοι συγκεντρώνουν τα τοπικά τους δεδομένα και τα αποτελέσματα ανεβαίνουν προς τα πάνω στο δέντρο.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία Ροής

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: μεγάλα δεδομένα, συντονισμός εργασιών, δέντρο με χρήση buffer και συμπίεσης, κατανεμημένα συστήματα , αρχιτεκτονική πολλών αφεντών

ABSTRACT

Recently, increasingly large amounts of data are generated from a variety of sources. Streaming frameworks for Big Data applications help to store, analyze and extract useful information from such continuously generated data. There are several existing streaming frameworks, like Apache Storm, Apache Spark and Apache Flume.

In this thesis, we present a decentralized stream processing model. It uses a DHT protocol to achieve a many masters-many workers architecture and assign each job its own master.

Even groups are created for each job by utilizing the system's routing properties, resulting in a hierarchical tree formation, consisted of agents that are participating in the network. The root of this tree acts as the master of the group and is responsible for synchronizing the group's members.

Each agent consumes live data logs, which are parsed into mini batches and stored in a memory efficient data structure. The agents aggregate their local data and the results are rolled up the the aggregation tree.

SUBJECT AREA: Stream Processing

KEYWORDS: big data, cross job coordination, compressed buffer tree, distributed systems, many masters architecture

ACKNOWLEDGEMENTS

We would like to thank our supervisors, Mr. Alexis Delis for his excellent guidance and assistance in the preparation of this thesis.

CONTENTS

FIGURES LIST

1. INTRODUCTION

The importance of stream data processing these days is undeniable. Big companies and industries are becoming increasingly interested in investigating streaming systems for good reasons. Among these reasons are the following:

- Timely insights into their data are vital and streaming is a good way to achieve low latency.
- The datasets are enormous and can be of any size. It is easier to analyze and gain useful information from such data with systems that are designed for never ending streams.
- Processing data as they arrive spreads workloads out more evenly over time, yielding more consistent and predictable consumption of resources.

The management of such data offers plenty of benefits businesses, including more effective marketing, generating business-critical decisions, fraud detection and overall better customer service. Therefore, streaming systems, which could be defined as a process engine designed for infinite datasets, are mainly used for such purposes.

In this thesis, we implement and evaluate some components of a streaming system model, based on a presentment on 2014 USENIX Annual Technical Conference by Liting Hu[1]. Specifically, we implement an efficient tree data structure, which is used for storage and aggregation of the collected records. Furthermore, a DHT protocol responsible for network routing is implemented, as well as a decentralized publish/ subscribe system, enabling multicast group communication.

1.1 Three Vs of Big Data

Big Data as a term means large sets of structured and unstructured data. But that is a little vague, since input data could originate from various sources, like social networks, audio streams or bank transactions. In order to clarify and make it easier to understand the nature of the data, Big data can be defined by three characteristics, *volume*, *velocity* and *variety*[2].

- **Volume**

  Volume is the amount of the data to be processed (gigabytes, terabytes etc). Volume could refer for example to the amount of data generated through websites or online applications. With the vast increase of Internet use in the past years, volumes of data can reach unprecedented heights.

- **Velocity**

  Velocity is the speed that information moves through the system. Data may be flowing into the system from multiple sources. Some data might come in real-time, while other might come in batches.

- **Variety**

  Variety refers to the different types of data that may occur. Data can be collected from server logs, social media feeds or other sources, with the type and format of data varying significantly. Images, videos or audio recordings are ingested alongside text or pdf files.

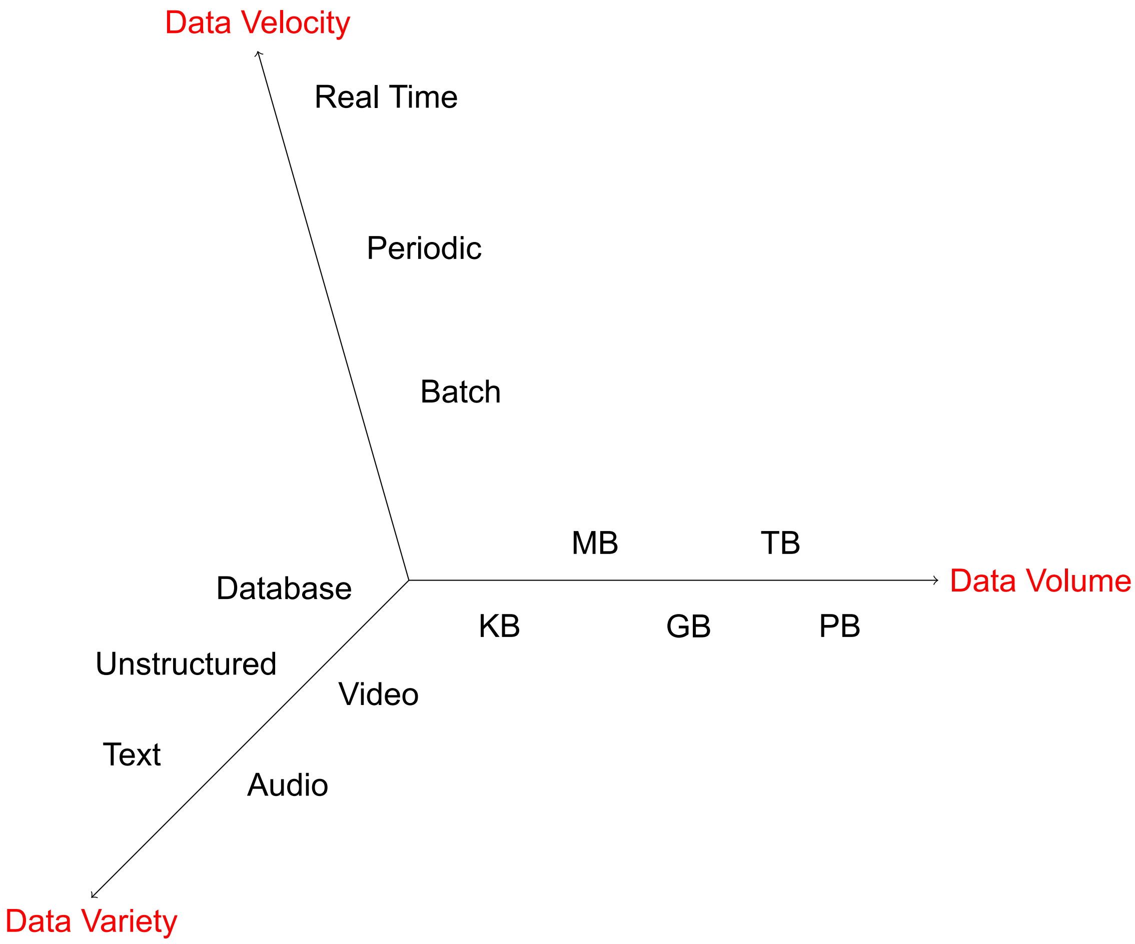

Figure 1: 3Vs of Big Data.

With these three attributes taken in consideration, it is easier to select the appropriate analytic and algorithmic tools to extract meaningful information.

1.2 Contributions

In this thesis we make the following contributions:

- A memory efficient data structure able to aggregate and store a large volume of past data.
- A decentralized many masters-many workers architecture, with each job having its own respective master.
- Capability of cross job coordination. The intermediate results of one job may be requested and used by a different job.

2. DATA PROCESSING

In this section we describe some methods of data streaming and give an example of concurrent stream job execution, that could prove to be challenging for streaming systems.

2.1 Types of data processing

There are many types of data processing, but the most common types are Batch Processing and Stream Processing.

2.1.1 Batch Processing

Batch processing is the processing of a significant amount of data, that have been stored over a period of time. It is useful in situations where real time analytics results are not necessary, since it requires a large amount of time compared with stream processing.

2.1.2 Stream Processing

Stream processing refers to real-time processing of continuously generated data. Such data may originate from various sources like website visits, customer transactions or event sensors. Some other names for stream processing are real-time analytics, event processing and streaming analytics.

It provides the capability of getting instant results and it is useful for tasks like fraud detection.

2.2 Application Example

Some job examples of a streaming system follow, that use as input user activity of an e-commerce company.

2.2.1 Diverse Job Execution

1. The first job calculates item popularity by collecting user clicks, and lists the *topK* items. The results can be used for placing discounts on those popular items. Batch processing may be more suitable for this job, since clicks may be gathered for a long period of time.
2. The second job collects item purchases and links together items that are usually bought together. It can be used to recommend items to users, that are similar to the items that the user has bought. Fast results are needed for this application and a stream processing model is preferable.
3. The third job queries whether an item is popular. The results of the *topK* application are needed for this.

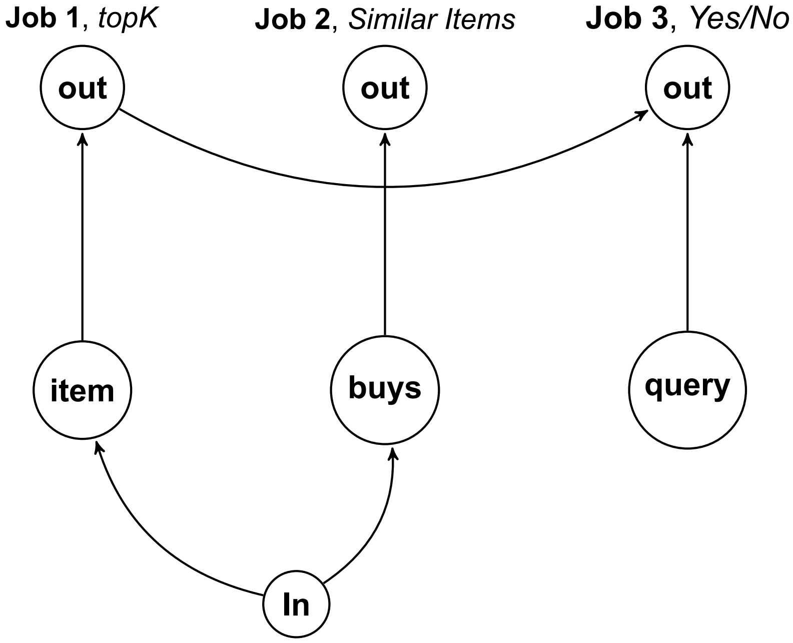

Figure 2: Streaming application example.

2.2.2 Streaming System Challenges

The above application comes with some challenges for streaming systems. First, historical records are needed to determine the topK items in a given time span, for example the past hour. An efficient data structure is required for this purpose, able to store a great amount of data.

Second, a high number of diverse jobs may be running simultaneously. With a single master responsible for coordination, potential bottleneck is created.

Third, communication across different jobs may also be necessary, since the results of a running job may be needed for another application.

2.3 Solutions

1. **Efficient Data Structure**

   For the purpose of storing incoming data, an in-memory data structured is used called compressed buffer tree[3] (CBT). In order to achieve high throughput and be able to store considerable volumes of data, CBT uses buffering and compression techniques.

   Hashtable implementations are another option, but the CBT uses less memory and provides high throughput.

2. **Many Masters/Many Workers Architecture**

   A DHT-based overlay is used. It is capable of hosting multiple jobs running simultaneously. Each job is assigned its own master and set of workers, with the master being responsible for coordinating and controlling its respective workers.

   Every master can easily communicate with a master responsible for another job. A single message is required to be sent. The message delivery is handled in a decentralized manner.

3. COMPRESSED BUFFER TREE

As we mentioned before, each agent uses an in-memory data structure called Compressed Buffer Tree (CBT). Agents have direct data access and locally parse live data logs into key-value pairs, which are stored and aggregated in the CBT. In this section we describe CBT's implementation and how it uses buffering and compression to improve performance and storage capacity.

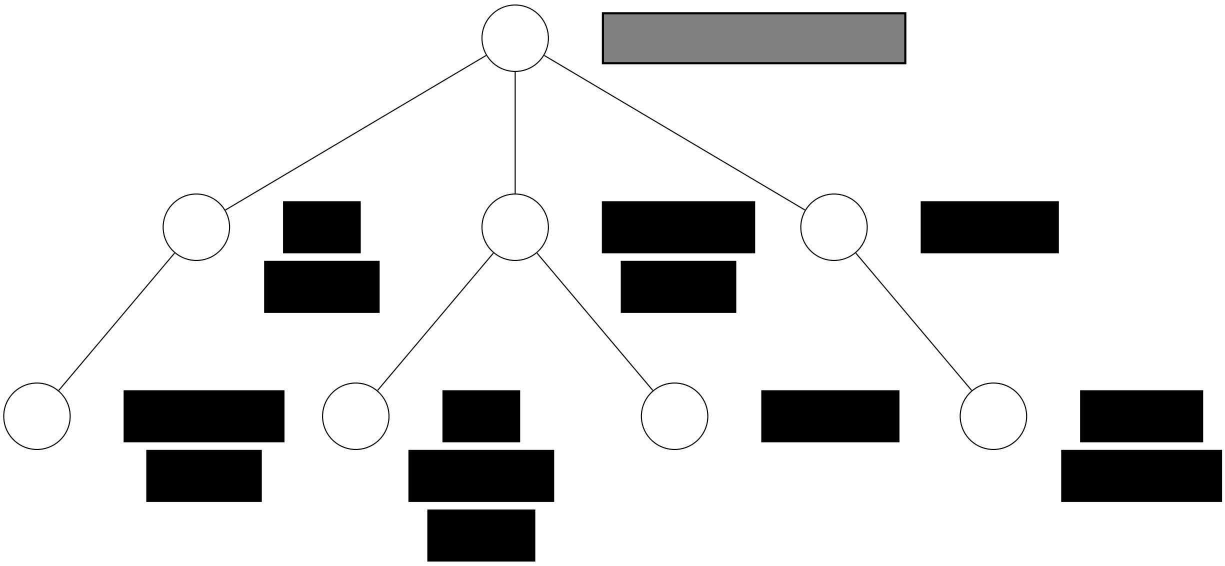

Figure 3: Compressed buffer tree.

3.1 Overview

CBT uses an (a-b)[4] tree with each node augmented by a large memory buffer[5]. An (a-b) tree is a kind of balanced search tree, which has the following properties:

- 2 ≤ a ≤ (b+1)/2
- Every internal node, except the root, has at least *a* and at most *b* children.
- All leaves have the same distance to the root.

The root is always uncompressed, while the buffers of the other nodes are always compressed. The internal nodes' buffers may contain a varying size and number of compressed fragments. Figure 3 shows a hypothetical state of a CBT. The root's buffer is displayed with gray color and internal nodes' fragments are displayed with black color. For buffer compression/decompression we use Snappy [6] by Google, a library intended for high speed performance.

When an item is inserted, the insertion is not immediately performed. Instead the item is appended to the root's buffer. When the root reaches a given threshold, for example half the size of the buffer, then an empty (Figure 4) operation is performed to the root. By delaying the insertions the number of compressions/decompressions are reduced and I/O performance is optimized.

```
procedure empty(N)
    if isRoot(N) then
        if isLeaf(N) then
            splitRoot()
        else
            spillRoot()
    else
        if isLeaf(N) then
            splitLeaf()
        else
            spillNode()
```

Figure 4: Empty operation on a node.

CBT is also used as an aggregator for the stored data. It merges the inserted key-value pairs with the existing ones in memory.

3.2 Aggregation

Each new key-value pair inserted is represented as a structure termed *Partial Aggregated Object* (PAO). During aggregation given two PAOs that share the same key, their values are merged forming a new PAO with the same key and the new value. For example, in a word count application, PAOs (*and, 1*), (*the, 3*), (*or, 5*), (*and, 3*), (*the, 2*), will be merged resulting (*and, 4*), (*the, 5*), (*or, 5*).

3.3 CBT Operations

CBT consists of three main operations, *insert*, *finalize* and *empty*. Finalize aggregates all the PAOs that are stored in the tree's levels and returns the final results. Empty clears the tree's buffers.

3.3.1 Insertion

After a PAO has been created for the inserted key-value pair, the tuple (*hash, size, serializedPAO*) is created and it is appended to the root. Hash is the hash result of the key and size is the size of the serialized PAO. The key is hashed because integer comparison is faster than string comparison.

When the root is full, first the tuples in the buffer are sorted by hash using the non-comparative sorting algorithm, radix sort. Then the existing PAOs that have the same key are aggregated and merged into a single PAO. Finally the buffer is pushed to the lower levels of the tree.

If the root is a leaf then the buffer is split in two creating a new tree level. Otherwise, the buffer is split and copied into the children nodes, based on the hash ranges of the children, and afterwards compressed. If an internal node becomes full by this process, a similar procedure, as when the root is full, occurs. If a leaf becomes full, the buffer is split and a new leaf node is created.

Since the internal and leaf nodes' buffers are always compressed, they must be decompressed before sorting and aggregating them.

The CBT always maintains an (a,b) tree's properties, and thus when one node's children exceed the maximum designated number, the node is further split.

3.3.2 Finalize

PAOs with the same key may reside in different levels of the tree. In order to get the final aggregated results, starting from the root, the PAOs are pushed to the next tree level in the same way the insertion is performed. This is repeated until all PAOs reside in the leafs, resulting in all duplicate PAOs being merged.

3.3.3 Empty

Empty clears the CBT's buffers and can be used when there is the need for a new batch to be processed.

3.4 Alternatives

3.4.1 Metis

Metis[7] is a MapReduce library for multi core architectures. Metis uses a hash table combined with a B+tree to store intermediate results. It works well for many workloads if the keys hash uniformly, but encounters some performance issues with workloads that have few repeated keys.

3.4.2 SparseHash

SparseHash[8] is a memory efficient hash table implementation by Google. It uses sparse arrays and is time efficient for good hash functions.

3.5 CBT Benefits

The main advantage of the CBT is the use of buffering for insertions. With the use of a simple (a,b) tree, the incoming key-value pairs traverse the tree and insertion is performed immediately. Read and write operations are increased and performance declines.

By keeping intermediate results in a compressed form, use of memory is reduced. Compressions and decompressions may add to the overall compute work, but this way it is possible to store a greater amount of data.

4. DHT-BASED OVERLAY

All agents are structured into a peer-to-peer overlay based on the Pastry[9] distributed hash table. Scribe[10], a multicast infrastructure, is built on top of Pastry to enable agent communication. With the combination of these two software applications, a decentralized many-master infrastructure is achieved, able to run diverse jobs concurrently.

4.1 Pastry Overview

Pastry is a decentralized network designed for various peer-to-peer applications like file sharing. Given a message and a destination key, Pastry routes the message to the appropriate node efficiently. The expected number of steps the routing procedure requires is $O(logn)$, where n is the number of nodes in the network.

4.1.1 Design

Each node is assigned 128-bit nodeId. The nodeId is assigned the moment a node joins the system ranging from 0 to $2^{128} - 1$. NodeIds could be generated by hashing the IP address of the node. For this purpose we use the MD5[11] hash function, a non-reversible encryption algorithm, ensuring uniform distribution of nodeIds. SHA-1[12] could also be used, a cryptographic hash function that can process a message to produce to produce a condensed representation called a message digest. The ids are treated of as sequences of digits in base 2^b, where typically $b = 4$, so these digits are hexadecimal.

To support the routing procedure, each node maintains three separate tables. A *routing table*, a *leaf set* and a *neighborhood set*.

- **Leaf Set**

  The Leaf set consists of the numerically closest Ids. Half of these Ids, are smaller than the self Id of the node and the other half are greater.

- **Routing Table**

  The Routing table is a two dimensional table, with $128/b$ rows and 2^b columns. The Ids at row n of the routing table share the same n first digits, with the $n + 1$ digit having a different value.

- **Neighborhood Set**

  The Neighborhood set contains the Ids of the nodes that are closest in terms of network locality.

The size of the Leaf set and the Neighborhood set are usually 2^b or $2x2^b$. Figure 5 demonstrates an example of the three tables with $b = 2$. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId.

Leaf Set	**NodeId 31323201**			
	SMALLER		LARGER	
	20133102	22310022	31333012	33002112
	21022001	23301201	32001022	33221001

Routing Table				
	-0-1022103	-1-0123311	-2-2331010	3
	3-0-112103	1	3-2-332011	3-3-221031
	31-0-00211	31-1-12332	31-2-21001	3
	313-0-0230	313-1-1302	2	313-3-0212
	3132-0-322	3132-1-000	3132-2-121	3
	31323-0-01		2	31323-3-11
	0	313232-1-0		
		1		

Neighborhood set				
	21022130	10221210	12331203	33201233
	20200230	22301130	20331203	31313321

Figure 5: Example of a Pastry node state.

Chord[13] is a network protocol similar to Pastry, but instead of using prefix routing it uses consistent hashing[14]. Each node has a finger table which contains the successor and predecessor of the node.

4.1.2 Routing Procedure

The routing procedure is responsible for delivering a given message to the node responsible for. When a message with a key *D* arrives, the node executes the following steps:

1. It checks its Leaf set to find the numerically closest node to the key. If the key is covered by the Leaf set, then the message is forwarded to the appropriate node.
2. If the key is not covered by the Leaf set, then the node checks its Routing table. The message is forwarded to the node that shares a common prefix with the key by at least one more digit.
3. Finally, if no such entry exists in the Routing table, then the message is forwarded to the node that shares the same amount of digits with the key as the local node, and is numerically closer to the key.

4.1.3 Pastry Operations

Pastry's basic operations are the following:

- **join**

  The node joins the network. If no network exists then it creates one. If a node wishes to join the network, then a node that is already in the network must be made known to the joining node.

- **route**

  Given a message and a key, it routes the message to the node with numerically closest nodeId to the key within the network.

- **forward**

  It forwards the message to the next node with nodeId numerically closest to the key.

- **deliver**

  It is called when the message has arrived at the designated node.

4.2 Scribe Overview

Scribe is a group communication and event notification system built on top of Pastry. It provides an efficient application-level multicast.

4.2.1 Design

Scribe consists of a network of Pastry nodes. With the Scribe application, these nodes can be managed in groups associated with a specific topic. Each topic has a unique topicId, which is the hash of the topic's textual name. Again, the MD5 function is used for hashing, to ensure uniform distribution of topics across Pastry nodes.

By creating topics and joining node, multicast trees are formed that are responsible for publishing events. Each topic can only have a single multicast tree. A new topic may be created by any node and others may join. Nodes can be subscribed to more than one topic and are able to multicast messages, even to groups that are not part of.

4.2.2 Scribe Messages

The possible messages in Scribe are CREATE, SUBSCRIBE and PUBLISH. By routing these messages with Pastry, a Scribe node is able to create topics, subscribe to topics and publish events. The procedures for these operations are described below.

- **CREATE**

  To create a topic, Scribe asks pastry to route a CREATE message. First the name of the topic is hashed to generate the topicId. Then the pastry delivers the message to the node whose nodeId is numerically closer to the topicId. This node becomes the rendezvous point and acts as the root of the multicast tree. Subsequently the root adds the topic to its topic set.

- **SUBSCRIBE**

  When a node wishes to subscribe to a topic, it asks Pastry to route a SUBSCRIBE message. The message is routed towards the rendezvous point of the topic. The nodes encountered through the process are either subscribed to the topic or not subscribed. If a node is not a subscriber, it adds the topic to its topic set and adds the source node to its children set. The node is now a forwarder of the topic. If a node is already a forwarder, it simply adds the source node to its children set and the forwarding procedure is terminated.

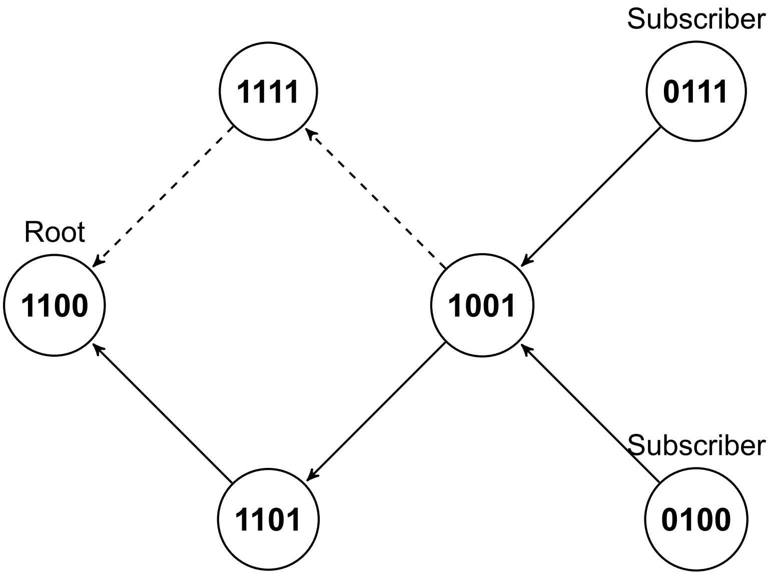

Figure 6: Example of Subscription mechanism.

Figure 6 showcases two subscription cases. First, node 0111 wishes to subscribe and a SUBSCRIBE message is routed to node 1001, then node 1101 and finally the messaged is delivered to node 1100, which is the root of the topic. The subscription of node 0111 causes nodes 1001 and 1101 to become forwarders for the topic and each add the preceding node to its children table. When node 0100 wishes to subscribe, the SUBSCRIBE message is routed to node 1001, but since it is already a forwarder, it adds node 0100 to its table and the message is terminated.

- **PUBLISH**

  A PUBLISH message is routed to the rendezvous point of the topic, which disseminates the events. It notifies its children and they in turn do the same until every subscribed node has been notified. The IP of the rendezvous point can be cached as an optimization and the publishing node needs only to contact the rendezvous point directly.

Scribe implements the *forward* (Figure 7) and *deliver* (Figure 8) methods to achieve the above operations. The following variables are used in the pseudocode: *topicSet* is the set of topics that the local node is aware of, *msg.source* is the source node of the message, *msg.type* is the type of the message and *topic* is the topicId of the topic.

```
procedure forward(msg, topic)
    if msg.type = SUBSCRIBE then
        if ! isSet(topic) then
            topicSet.add(topic)
        addChild(topic, msg.source)
        msg.setSource(selfInfo)
```

Figure 7: Scribe implementation of forward.

```
procedure deliver(msg, topic)
    if msg.type = CREATE then
        topicSet.add(topic)
    else if msg.type = SUBSCRIBE then
        topic.addChild(msg.source)
    else if msg.type = PUBLISH then
        sendChildren(msg)
        if isSubscribed(topic) then
            invokeEventHandler(msg)
```

Figure 8: Scribe implementation of deliver.

4.3 Benefits

Using the DHT, efficient aggregation trees can be built that guarantee multicasts with only $O(logN)$ hops. Multicasts are more efficient, with each node of the tree sending messages only to its children, instead of a single node notifying all the members of the group.

Additionally, many independent groups can be supported which have the capability to communicate with each other by publishing messages, making job interaction possible.

5. SYSTEM IMPLEMENTATION

5.1 Group Creation

Unlike other streaming systems with static assignments of nodes to act as masters vs. workers, all agents are treated equally. As we have described in Section 3, they are structured into a P2P overlay, in which each agent has a unique nodeId and can act as master or worker for multiple jobs.

The first step, is to create a group of agents for every job. Using Scribe's software, a CREATE message is routed using the job's id as the key. JobId is the hash of the job's textual name. The message eventually arrives at the destination node with nodeId numerically closer to jobId and is set as the job's master.

All other nodes wishing to join the group must then route a SUBSCRIBE message using jobId as the key. The unions of all messages' paths are registered to construct the group, in which the internal node, as the forwarder, maintains a children table for the group containing an entry (IP address and jobId) for each child.

Since a hash function is used for generating jobId, even distribution of groups across all agents is ensured. A single node is rarely the master of more than one job, avoiding potential bottleneck.

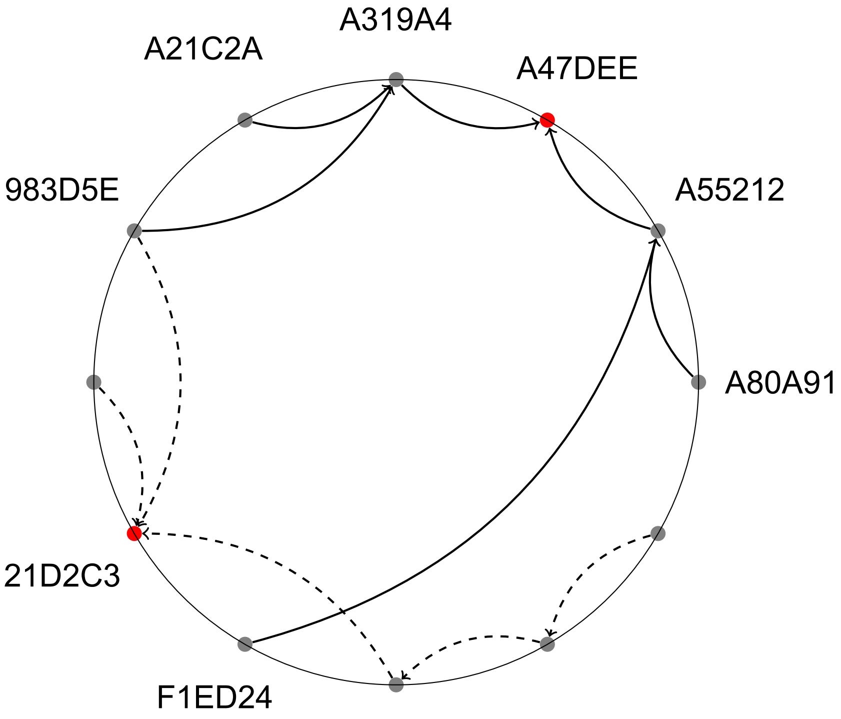

Figure 9: Group creation.

5.2 Master / Worker

Each job can only have a single master. Every other agent in the group acts as a job worker. Consequently, an agent can only act as a master, a worker or both.

5.2.1 Job Master

Job master is the aggregation's tree root. It is responsible for synchronizing the workers and collecting the globally aggregated results. This is achieved by routing a PUBLISH message through the system, which is guaranteed to notify all of the group's agents.

Job master is also responsible for interacting with other job masters, when the results of its job are required for a different job. It can actually provide the results to any node that requests it.

5.2.2 Job Worker

A job worker has the following responsibilities:

- **Input stream parsing**.

  Adopting an 'in-situ'[15] approach to data access, each agent consumes live logs and parses them into key-value pairs, which are inserted in the CBT. CBT resides in local agent's memory. As an example, we use Apache Flink[16] for log collection. Flink's Twitter Streaming API[17] provides access to the stream of tweets made available by Twitter, with a built-in TwitterSource class.

- **Local Aggregation**

  Each key-value pair is represented as a partial aggregation object (PAO) . New PAOs are inserted into and accumulated in the CBT. When requested, new and past PAOs are aggregated and returned.

Job Worker
Stream Observer
Stream Parser
PAO Execution
P2P Socket

Figure 10: Worker components.

5.3 Global Aggregation

Scribe's multicast trees are enhanced to also support aggregation functions.

Specifically, when agents periodically send their updates for map results towards the root, all intermediate nodes in the path aggregate the datasets collected from their children, applying the aggregation functions along the entire path. Aggregation, therefore, occurs in $O(log_{2^b}N)$ hops.

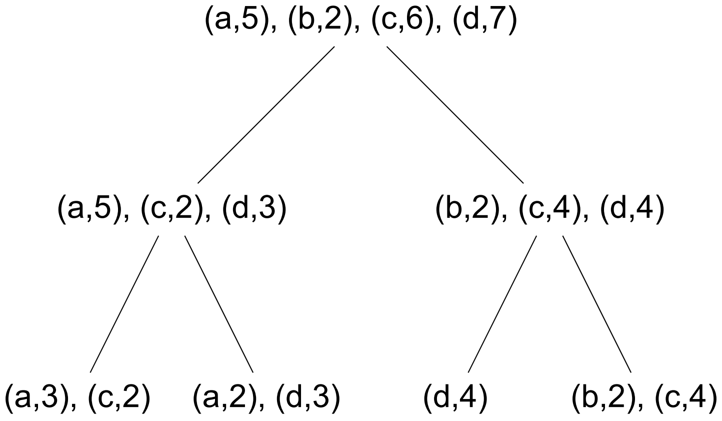

Figure 11: Example of key-value pair aggregation.

Figure 11 illustrates an aggregation example. Starting from the root, an aggregation message is emanated through the tree. When a node receives the message, it replicates the message to its children and waits until all of them respond with their results. The results are merged and sent back to the node's parent. In the end the root has gathered all of the aggregated key-value pairs of the tree.

6. EVALUATION

In this section, we evaluate CBT's throughput, as well as the system's aggregation speed. In order to evaluate CBT's throughput we measured the time required to insert 1,000,000 records with and without the use of flushing, and for the evaluation of the system we measured the time required to complete a topK application with varying number of nodes participating in the network.

All the tests were conducted on PCs with 16GB Ram and 4 cores.

6.1 CBT Evaluation

In this part, the time measurements of inserting key value pairs in the CBT are presented, as well as the time measurements for flushing the CBT. We evaluated the insertion time in the CBT without the use of flushing and with the CBT flushing every 50,000 records. Since the PAOs are pushed to the bottom of the tree, the insertion time with the use of flushing is expected to be lower. On both occasions, buffers with size of 30MB and 30KB were used, considering that the higher the size of the buffer, the less operations are required.

We also measured the time for flushing the CBT for different number of key-value pairs inserted, as well as when the flushing occurs every 50,000 inserted records.

- **Time measurements for insertion without flushing**. Figure 12 displays the time required for inserting 1,000,000 records without ever flushing the CBT. In the first graph we allocate 30 MBs for the buffers, while in the second 30 KBs are allocated.

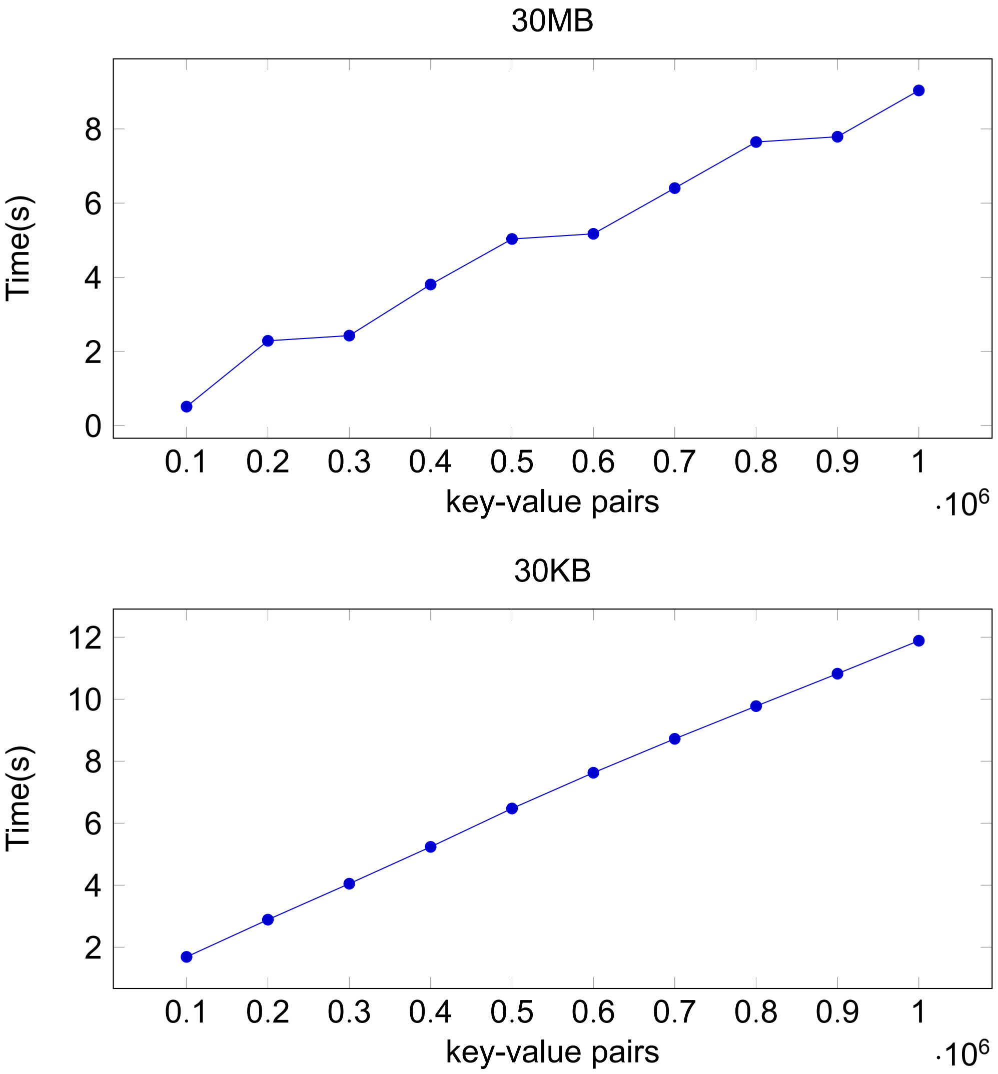

Figure 12: Insertion times without flushing.

- **Time measurements for insertion with flushing**. Figure 13 displays the time required for inserting 1,000,000 records while flushing the CBT every 50,000 records inserted. The same space for the buffers is allocated.

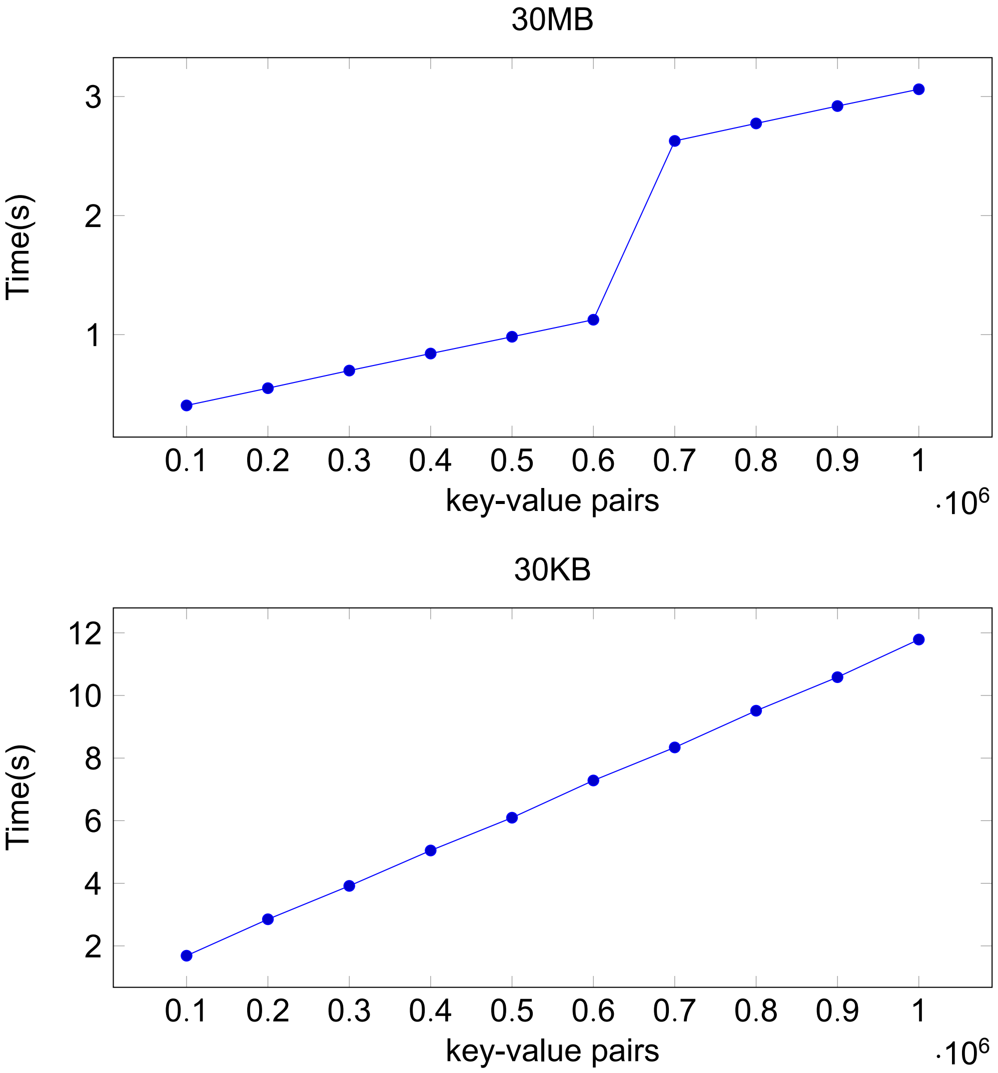

Figure 13: Insertion times with flushing.

- **Time measurement for flushing**. The first graph presents the time needed when flushing occurs every 50,000 records and the second one presents the time needed for flushing records in total.

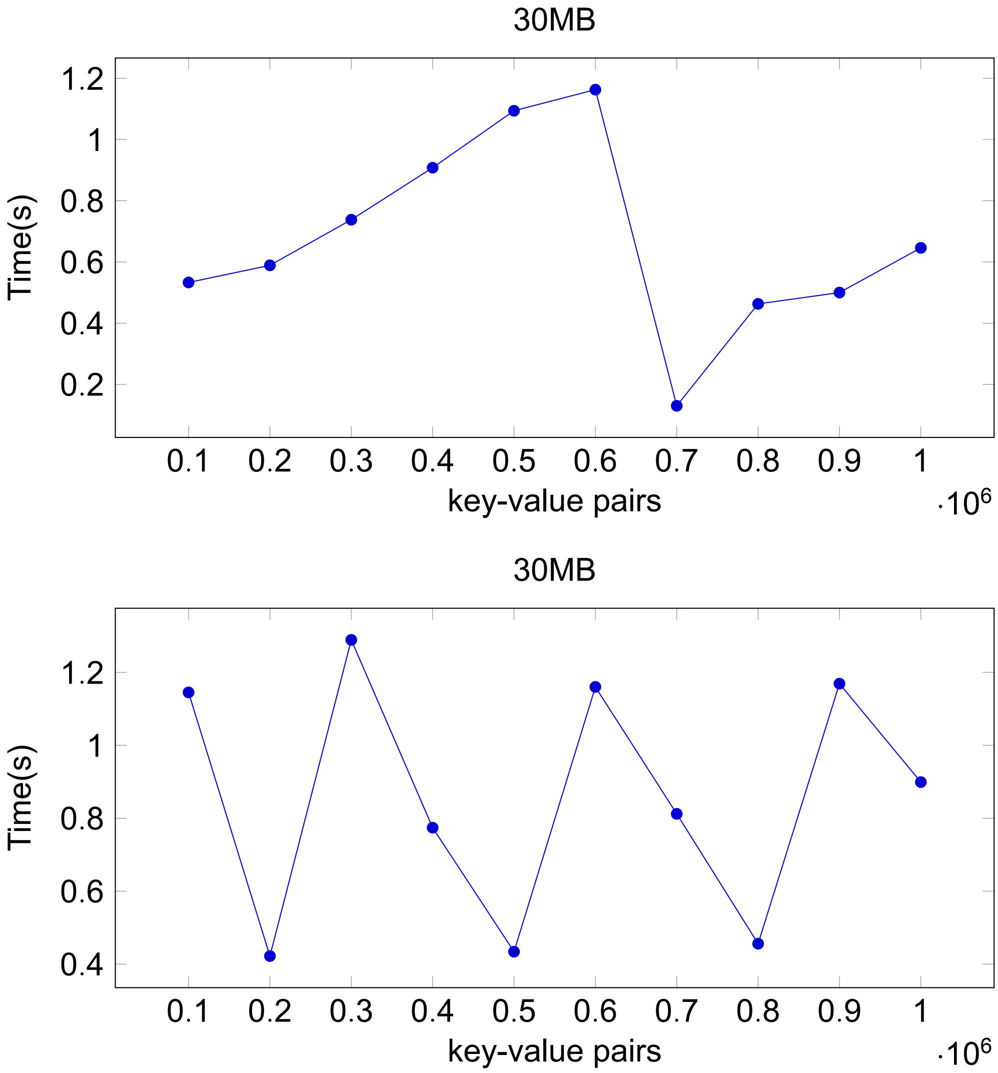

Figure 14: Flushing times.

6.2 System Evaluation

In this section, we present the measurements for a topK application. Each node has in before 2,000,000 records stored in its CBT. First, we measured the total time required to complete a topK application (Figure 15). Then, during a topK operation, we measured solely the time required for the for the all the key-value pairs to reach the root of the aggregation tree from the moment an aggregation message is published (Figure 16). Finally, we evaluated the average time of data transmission from on agent to another (Figure 17).

The tests were conducted with 2, 4, 6 and 8 nodes participating.

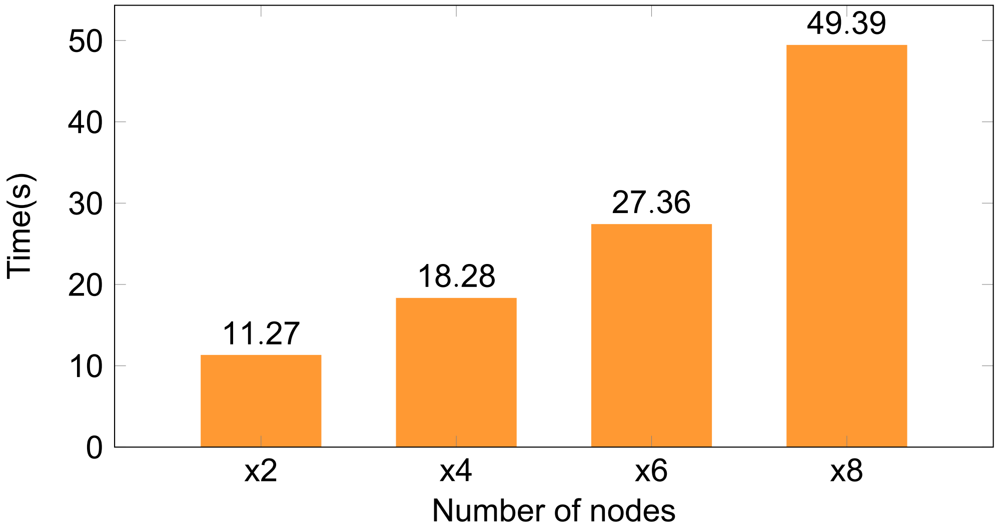

Figure 15: TopK measurements.

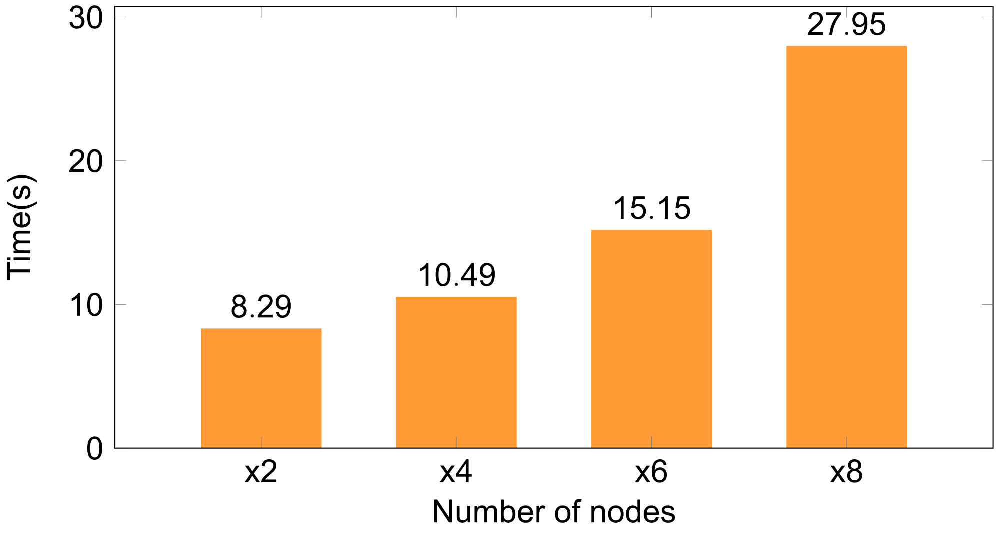

Figure 16: Roll up times.

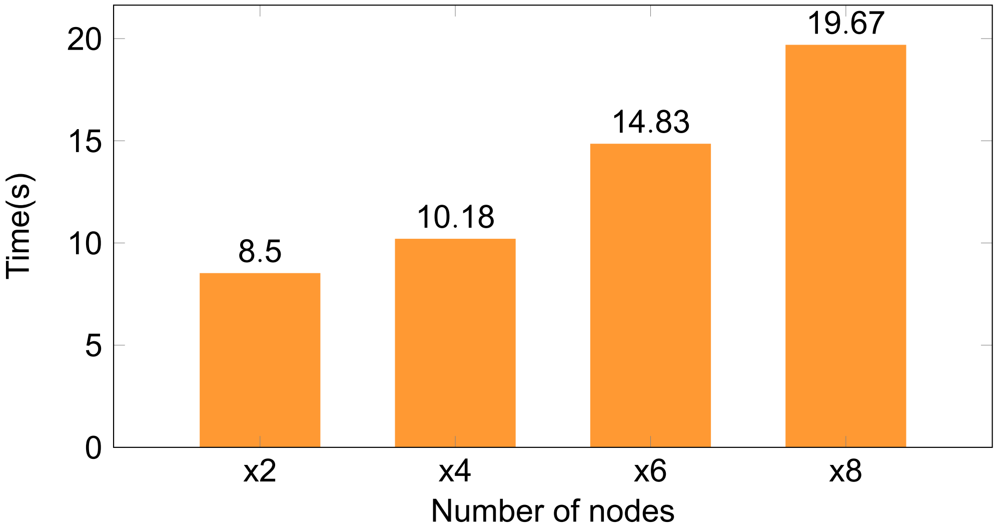

Figure 17: Average time of data transmission.

7. CONCLUSION

In this thesis, we have described and implemented a decentralized streaming system able to run concurrently diverse jobs. It utilizes Pastry’s routing protocol and Scribe’s multicast trees to achieve a many masters many workers architecture.

Streams are parsed int mini-batches and are stored and aggregated in memory efficient compressed buffer trees. The aggregated results of each node in the network are rolled up the aggregation tree that is formed, until they become available to the tree’s root. Coordination among different jobs also becomes available by routing simple messages through the network.

Finally, the system’s performance can be further improved by optimizing the compressed buffer tree’s throughput with the use of threads and multiple root buffers. Optimizations in the way the key-value pairs are aggregated through the system are also possible. The capability to detect node failures by periodically sending keep-alive messages is another implementation extension.

ABBREVIATIONS, ACRONYMS

CBT	Compressed Buffer Tree
PAO	Partial Aggregated Objdect
DHT	Distributed Hash Table
MD5	Message Digest 5
SHA-1	Secure Hash Function 1
P2P	Peer to Peer

REFERENCES

[1] L.Hu, K.Schwan, H. Amurm, X. Chen ”ELF: efficient lightweight fast stream processing at scale”. Application Delivery Strategies, USENIX ATC’14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference.

[2] D.Laney, ”3D Data Management: Controlling Data Volume, Velocity, and Variety”. Application Delivery Strategies, Meta Group 2001.

[3] H. Amur, W. Richter, D. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, E. Zawadzki. ”Memory-Efficient GroupBy-Aggregate using Compressed Buffer Trees”. In SoCC 2013.

[4] D. Cormer, ”The ubiquitous B-tree”, ACM Computing Surveys, 11(2):121-137, 1979.

[5] L Arge. ”The Buffer Tree: A Technique for Designing Batched External Data Structures ”. Algorithmica , 37(1):1–24, 2003.

[6] ”Snappy”. https://github.com/google/snappy.

[7] Y. Mao, R. Morris, F. Kaashoek. ”Optimizing MapReduce for Multicore Architectures”. MIT-CSAIL-TR-2010-020.

[8] ”SparseHash”. https://github.com/sparsehash.

[9] A. Rowstron, P. Druschel. ”Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”. 18th IFIP/ACM Int ernational Conference on Distributed Systems Platforms (Middleware 2001).

[10] A. Rowstron, A. Kermarrec, M. Castro, P. Druschel. ”SCRIBE : The design of a large-scale event notification infrastructure”. Networked Group Communication 2001.

[11] R. Rivest. ”The MD5 Message-Digest Algorithm”. MIT Laboratory for Computer Science and RSA Data Security 1992.

[12] D. Eastlake, P. Jones. ”US Secure Hash Algorithm 1 (SHA1)”. Cisco Systems 2001.

[13] I. Stoica, R. Morris, F. Kaashoek, H. Balakrishnan, ”Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”, Proceedings of the 2001 ACM SIGCOMM Conference 2001 .

[14] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, R. Panigrahy. ”Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web”. In Proceedings of the 29th Annual ACM Symposium on Theory of Computing.

[15] D. Logothetis, C. Trezzo, K. Webb, K. Yocum. ”In-situ MapReduce for Log Processing”. USENIX annual technical conference 2011.

[16] ”Flink”. https://flink.apache.org.

[17] ”Twitter Streaming API”. https://developer.twitter.com/en/docs.