6-24-2020

Support Efficient, Scalable, and Online Social Spam Detection in System

Hailu Xu
hxu@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

SUPPORT EFFICIENT, SCALABLE, AND ONLINE SOCIAL SPAM DETECTION IN SYSTEM

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Hailu Xu

2020

To: Dean John L. Volakis
College of Engineering and Computing

This dissertation, written by Hailu Xu, and entitled Support Efficient, Scalable, and Online Social Spam Detection in System, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

S. S. Iyengar

Deng Pan

Alex Afanasyev

Gang Quan

Liting Hu, Major Professor

Date of Defense: June 25, 2020

The dissertation of Hailu Xu is approved.

Dean John L. Volakis
College of Engineering and Computing

Andrés G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2020

DEDICATION

I dedicate this dissertation work to my beloved parents.

Without their infinite love, support, patient, and understanding, nothing can be accomplished.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor, Dr. Liting Hu, for her unlimited support and guidance during my Ph.D. journey. Liting guided me into new research areas and taught me a lot of cutting-edge research experience. She encouraged me to create novel ideas, guided me in preparing research draft, and instantly introduced cutting-edge topics to me. She always encouraged me to attend the top-tier conference and supported me to take the opportunity to communicate with other researchers and scientists. Her unlimited support helps me to succeed in my Ph.D. career and create many fantastic research work. She guided me to join the cycle of systems research in computer science and trained me to be a young scientist.

Second, I would like to thank my committees and professors at Florida International University. Many thanks to my committees, Dr. S. S. Iyengar, Dr. Deng Pan, Dr. Alex Afanasyev, and Dr. Gang Quan, for your great encouragement and support during my research work and guided me to deeply explore unsolved problems. Many thanks to Dr. Ning Xie and Dr. Jason Liu, for your great support in my work and guiding me in many details.

Third, I would like to express my thanks to my mentors during my summer intern at Lawrence Livermore National Laboratory, Dr. Chunhua Liao and Dr. Murali Emani. Thank you for your guidance and support in the summer of 2019, which gave me a taste of frontier research across the industry and academia. Also many thanks to Pei-Hung Lin, Tristan Vanderbruggen, Zifan Nan, Weijie Zhou, Anjia Wang, Gleison Souza, and Alok Mishra, we enjoyed a wonderful summer!

Forth, I would like to thank my friends at FIU and Miami, who gave me warm supports and brought colorful life during this special journey. Many thanks to Pinchao Liu and Boyuan Guan, my teammates in Elves Lab. Many thanks to Yekun

Xu, Xiaolong Zhu, Shuai Xu, Wentao Wang, Yiming Liu, Hao Kang, Boyuan Guan, Bingqian Li, Tianyi Wang, Huifeng Zhang, and to my friends that I encountered in this beautiful city. Thank you all for being special parts during this journey.

Finally, from the bottom of my heart, I would like to express my gratitude to my parents and family. My mother, Yufen Lu, who is a great teacher, gives me priceless and infinite love and support, which helps me to be here. My father, Shanyao Xu, who guides me with his actions and loves me unconditionally, supports me in any kind of situation. Their love overwhelms everything I have and is so invaluable that it exceeds anything in the world.

ABSTRACT OF THE DISSERTATION

SUPPORT EFFICIENT, SCALABLE, AND ONLINE SOCIAL SPAM DETECTION IN SYSTEM

by

Hailu Xu

Florida International University, 2020

Miami, Florida

Professor Liting Hu, Major Professor

The broad success of online social networks (OSNs) has created fertile soil for the emergence and fast spread of social spam. Fake news, malicious URL links, fraudulent advertisements, fake reviews, and biased propaganda are bringing serious consequences for both virtual social networks and human life in the real world. Effectively detecting social spam is a hot topic in both academia and industry. However, traditional social spam detection techniques are limited to centralized processing on top of one specific data source, but ignore the social spam correlations of distributed data sources. Moreover, a few research efforts are conducting in integrating the stream system (e.g., Storm, Spark) with the large-scale social spam detection, but they typically ignore the specific details in managing and recovering interim states during the social stream data processing.

We observed that social spammers who aim to advertise their products or post victim links are more frequently spreading malicious posts during a very short period of time. They are quite smart to adapt themselves to old models that were trained based on historical records. Therefore, these bring a question: how can we uncover and defend against these online spam activities in an online and scalable manner?

In this dissertation, we present there systems that support scalable and online social spam detection from streaming social data: (1) the first part introduces

Oases, a *scalable* system that can support large-scale *online* social spam detection, (2) the second part introduces a system named SpamHunter, a novel system supports *efficient* online scalable spam detection in social networks. The system gives novel insights in guaranteeing the efficiency of the modern stream applications by leveraging the spam correlations at scale, and (3) the third part refers to the state recovery during social spam detection, it introduces a customizable state recovery framework that provides fast and scalable state recovery mechanisms for protecting large distributed states in social spam detection applications.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

1.1 Motivation

1.1.1 Social Spam Detection

Online social networks (OSNs) have been an integral part of human life. More and more people are acquiring the latest news, advertisements, social activities, and breaking topics directly from the current popular OSNs such as Facebook, Twitter, and WeChat. For example, a report said that the percentage of US adults who primarily receive news and information from OSNs is as high as 62% [AG17]. However, the openness of widespread OSNs couple with massive spam activities, which are damaging as they cause public panic and social unrest. For example, in February of 2019, social users in Paris watched a lot of photos of kidnappings on Facebook and videos of vans speeding away on Snapchat and Twitter, all of which hinted that the Roma (Gypsies) robbed children with vans in the suburbs of Paris [Bre19]. Although the information proved to be wrong later, they brought serious consequences to the Roma and the whole society: dozens of young men wielding sticks and knives attacked a Roma camp and burned two vans, and tens of people were arrested. Another example is that one latest report said the global enterprise spam filter market was valued approximately USD 849 million in 2018 and is expected to generate around USD 2,675 million by 2026 [zio19]. And it pointed out that the increasing number of social spam is driving the enterprise spam filter market globally.

The unprecedented success of online social networks has created tremendous opportunities for the emergence and rapid spread of spam. By leveraging a large social user, social spam often dominates and influences social life in a short period of time and can reach every corner of the social world. Therefore, quickly detecting spam from large-scale social activities is an urgent need in the current situation.

Furthermore, as our observation, the spammers in the online social networks are not only active on a single platform, but are often active on different social platforms, by simultaneously manipulating dozens or hundreds of fake accounts. Naturally, the information published by these fake accounts is highly similar. This phenomenon has been pointed out by several former studies [XGL+18]. Spammers certainly desire to spread similar posts on different platforms to attract as many people as possible to target on these topics. A case study of social spam posts for multiple different news sites also demonstrates that spam posts show a high degree of similarity in content and topics during the same period of time and will immediately propagate from one site to another [a16]. Therefore, this correlation between cross-platform social spam is a common phenomenon in the current social media world. Although there are not many direct relationships between users, geographic locations, creation purposes, and regions in these various groups or platforms, the spam contents are highly correlated within similar topics during the same period of time.

However, former studies rarely utilized the spam correlations to handle the large-scale social data from distributed data servers. They either focused on the algorithm side to achieve high accuracy in the detection [VBC+14, WP15, VT16, SS16, HBSD17], or the entire processing only targeted on a small size of dataset without the global view from similar data across large-scale data sources [GCL+12, XZJ+16, CWZ+17, XSJ16].

1.1.2 Stateful States in Social Stream Data Processing

Today, we are undergoing a profound transformation with the use of large-scale, diverse, and distributed data sets that allow for data-intensive analytics and decision-making. Stream processing is proposed and popularized as a "technology like Hadoop but can give you results faster", which lets users query a continuous data stream and quickly get results within a very short time period from the time of receiving the data. For that reason, stream processing technology has become a critical building block of many applications, such as making business decisions from marketing streams, identifying spam campaigns from social network streams, predicting tornados and storms from radar streams, and analyzing genomes in different labs and countries to track the sources of a potential epidemic.

Over the last decade, a bloom of stream processing systems has been developed including Storm [ad], Trident [af], Spark Streaming [am], TimeStream [QHS$^+$13], S4 [NRNK10], etc. However, while the progress has been encouraging, the existing efforts have dominantly centered around stateless stream processing, leaving another urgent trend—stateful stream processing—much less explored. A driving need is that the future stream applications need to store and update state along with their processing, and process live data streams in a timely fashion from massive and geo-distributed data sets. Unfortunately, existing systems are mainly designed for low-latency intra-datacenter settings and do not scale well for running stream applications that contain large distributed states, suffering a significantly centralized bottleneck and high latency.

A stream is an unbounded sequence of tuples (e.g., online social network's microblog streams) generated continuously in time. A stream processing system creates a logical topology of stream processing operators, connected in a directed acyclic graph (DAG), processes the tuples of a stream as they flow through the DAG, and outputs the results in a short time. Traditionally, stream processing pipelines are stateless. A new trend is that more complex stream processing pipelines are stateful. For example, a stateful operator maintains the value of state for some of the identified spam accounts so far and updates it with new inputted information, such that the final output should accumulate all results that take into account both historical records and the new input.

However, we are facing significant challenges in managing large distributed states in stream processing systems. First, it is challenging for recovering from simultaneous failures of multiple stream operators for a large number of concurrently running applications. Social stream data processing is by nature long running, and operators refer to it may unexpected fails or lost, which cause state loses. Second, different social stream data processing may have various runtime requirements, e.g., different time sensitivity, deadline requirements, or computation depends, these lead to many kinds of state management in dealing with different social stream data processing.

1.2 Contributions

1.2.1 Scalable and Online Social Spam Detection

We present two systems to support online and scalable social spam detection from separate perspectives. `Oases` shows the system-level design in handling and supporting scalable social spam detection. SpamHunter describes the details of achieving efficient online social spam detection by leveraging spam correlations from geo-distributed sites or servers. We conclude the contributions as follows.

First, in the Chapter 3, an ***online*** spam detection system called `Oases` is presented that defend against real-time spam activities that happen in geo-distributed sites.

Second, in `Oases`, a ***scalable*** DHT-based tree overlay with spam detection related protocols is presented. It uses many progressive aggregation trees for aggregating the properties of spam posts and creating new spam classifiers to actively filter out newest spam.

Third, in the Chapter 4, we present a system named `SpamHunter`, based upon `Oases`, supports efficient online social spam detection in dealing with large-scale social stream data.

Forth, in `SpamHunter`, a peer group communication structure is presented that allows multiple Spiral groups to exchange and utilize the ***spam correlations*** among distributed social data sources.

Finally, comprehensive evaluations of `Oases` and `SpamHunter` performance and functionality on a large cluster using real-world social stream data are presented.

1.2.2 Customizable State Recovery in Social Spam Detection

We next describe the contributions in achieving customizable state recovery in social stream data processing. We make the following contributions in Chapter 5.

First, we show how existing techniques can lead to slow or resource-expensive state recovery in stream applications.

Second, we propose `SR3`, a customizable State Recovery framework that provides fast and scalable failure recovery mechanisms for protecting large distributed states in social spam detection. It does not rely on a central master for recovering the state. The failure recovery process scales to the size of the lost state, offers a significant reduction in failure recovery time and can tolerate multiple simultaneous node failures.

Third, we provide three different failure recovery mechanisms (Sec. 5.4). An important novel aspect of SR3 is that it can host multiple distributed streams and offer each application the recovery mechanism that best fits its requirements. The goal is to cater to the needs of different stateful stream applications (e.g., different stream processing computation models, quality of service requirements, state sizes, and network environments).

Finally, we make a comprehensive evaluation of the scalability, fast recovery and flexibility of the system on a large cluster using real-world stream processing applications' datasets (Sec. 5.5).

1.3 Summary and Roadmap

The rest of this dissertation is organized as follows. We introduce the details of background in Chapter 2, then we describe the `Oases` system in Chapter 3. We next show the design and details of the `SpamHunter` system in Chapter 4. The details of design and implementation of `SR3` system in Chapter 5. Finally, we conclude this dissertation and describe the future work in Chapter 6.

The details of the dissertation are illustrated as follows.

1. **Chapter 2**. Sec. 2.1 shows the previous work that refer to the traditional social spam detection. Sec. 2.2 describes the background about scalable stream data processing. Sec. 2.3 and Sec. 2.4 introduce the state management and recovery in social stream data processing.

2. **Chapter 3**. Sec. 3.1 describes the introduction of this work. Sec. 3.2 shows the details of design and functionality of the system, which include the details

of different functional agent, the tree structures and design benefits. Sec. 3.3 introduces the detailed evaluation of the system performance with real-world social stream data and the runtime overhead. Sec. 3.4 summarizes this work.

3. **Chapter 4**. Sec. 4.1 shows the introduction of the `SpamHunter` system. Sec. 4.2 shows the details of design and functionality of the system, which include the details of overview of workflow, the group management, online data processing, and the group coordination. Sec. 4.3 introduces the detailed evaluation of the system performance with real-world social stream data. Finally, Sec. 4.4 summarizes this work.

4. **Chapter 5**. The introduction of this work is shown in Sec 5.1. Sec. 5.2 defines and describes the problem that is solved in this work. Sec. 5.3 introduces detailed background of this work. Sec. 5.4 introduces the design details, including the workflow, the three types of recovery mechanisms, and mechanism selection. The evaluation part is shown in Sec. 5.5. Finally, Sec. 5.6 concludes this work.

5. **Chapter 6**. In this chapter, Sec. 6.1 concludes the entire dissertation work and the future work is discussed in Sec. 6.2.

CHAPTER 2

BACKGROUND

2.1 Previous Social Spam Detection in Online Social Networks

Offline social data processing. Many former studies had focused on offline methods in analyzing social data [LEC11, SS16, VT16], which utilized a limited historical dataset. Based on the connection between users and user trusts, [TLGP14] identifies victims in Twitter with a dataset that was collected in 10 months. By analyzing several historical datasets, Spade [WIP14] presents that new spam can be detected from one social network across other social networks.

Prior studies classified social spam from various perspectives, either from the view of adult contents [CALS17], user behaviors [ZZPZ16], or from hashtags [SS16], inherent features [WZLP15]. However, these studies still limit in a specific size of historical data and are difficult to catch up the online latest features of social spam.

Centralized social data processing. Former studies normally focused on centralized processing [MK10, TLGP14, VT16, ZNJ$^+$15]. [GCL$^+$12] presents an online spam filtering framework in a central server by using spam campaigns. TopicSketch [XZJ$^+$16] is a real-time framework that combines a sketch-based topic model and a hashing-based dimension reduction to detect bursty topics. Lfun [CWZ$^+$17] is a real-time statistic features-based system which can extract spam from social drifting data. Monarch [TGM$^+$11] utilizes the online URL blacklists to detect URL spam in real-time. The difference between our approach and these studies is that they normally focused on centralized spam analysis, while we focus on social spam detection in distributed manner.

2.2 Distributed Social Data Processing

Recent applications had been cooperated with scalable methods to achieve efficient processing [BML16, KI17, PIP16, SBJM17]. [CLQ11] presents a parallel spam filtering system based upon MapReduce. To mitigate the accuracy degradation by parallel SVM, they augment with ontology semantics. Different with them, we allow data training can be completely implemented in local agents and maintain the desired accuracy. A model within CELAR cloud platform is designed to catalog the distributed, dynamic and redundant cancer data [XJTG15]. ELF [HSAC14] is a decentralized model for the streaming process and supports powerful programming abstraction for batch, iterative and streaming processing. SSTD [ZZW$^+$17] is a dynamic truth discovery scheme which can discover Twitter data truth with scalability. Different from above work, we use the latest data features as the feedback and analyze social spam in a scalable way.

2.3 State Management in Social Stream Data Processing

Existing state management solutions can be divided into three representative categories: *in-memory*, *remote storage*, *in-memory+on-disk*.

Category 1: in-memory. Many industrial stream processing systems either do not support state (Heron [KBF$^+$15], S4 [NRNK10], the early version of Storm [ad]), or they rely on in-memory data structures such as hash tables and hash table variants to store state. For example, Muppet [LLP$^+$12] and Trident [af] store state via hash tables. Spark Streaming [am] enables state computation via Resilient Distributed Datasets (RDDs) [ZCD$^+$12], the core data abstraction from Spark that distributes read-only multiset data items. These techniques rely on a central master for state management that results in a centralized bottleneck and, therefore, may be difficult to scale to large states.

Category 2: remote storage. Some systems such as Millwheel [ABB$^+$13] and Dataflow [ABC$^+$15a] choose to separate state from the application logic. They have the state centralized in a remote storage [ABB$^+$16, CCD$^+$03, ACÇ$^+$03] (e.g., a database management system, HDFS or GFS) shared among applications, periodically checkpointing it for fault tolerance. Using external storage can scale well to large distributed states, but it significantly increases latency in the critical path of stream processing.

Category 3: in-memory+on-disk. A few other systems such as Kafka [ab], Samza [ac, NPP$^+$17], Spark Streaming [am], Flink [aa, CEF$^+$17] try to overcome this issue by using a combination of "soft state" stored in in-memory data structure along with "hard state" persisted in on-disk data store (e.g., RocksDB [al], LevelDB [aj]). However, they sacrifice programming model transparency by requiring programmers to declare and maintain state using built-in data structures (e.g., Spark's RDDs [ZCD$^+$12], Muppet's slates [LLP$^+$12]). The on-disk data store (used by Kafka [ab], Samza [ac], Dataflow [ABC$^+$15a]) incurs large I/O overhead due to well-known high write amplification [DCG$^+$17]. Finally, scaling to large distributed states and recovering from failures in such systems is quite expensive, because when a single node fails, the in-memory state and on-disk state for all dependent nodes must be reset to the last checkpoint, and computation must resume from that point, resulting in significant time and space overhead.

2.4 State Recovery in Social Stream Data Processing

Existing stream processing systems offer failure recovery mainly through the use of three approaches: *replication recovery*, *checkpointing recovery*, and *DStream-based lineage recovery*, which are either not scalable, slow, resource-expensive or incapable to handle multiple failures.

In the replication-based recovery approach, the system maintains a completely separate set of hot failover nodes, which processes the same stream in parallel with the primary set of nodes. The input records are sent to both. When there is a failure or multiple failures in the primary nodes, the system automatically switches over to the secondary set of nodes and the system can continue processing with very little or no disruption. The replication recovery has been used in systems such as Flux [SHB04] and Borealis [BBMS05]. The failover is fast and it can handle multiple failures. However, the replication recovery scheme doubles the hardware requirement.

In checkpoint-based recovery, all nodes periodically checkpoint their states to remote storage such as HDFS or GFS. Each node in the stream pipeline has an in-memory buffer to retains a backup of the data that it has forwarded to the downstream nodes since the last checkpoint. The system also maintains standby nodes. When a primary node fails, a standby node retrieves the latest checkpoint from the persistent storage, and its upstream node essentially replays the backup records serially to this failover node to recreate the lost state. The checkpointing recovery has been used in systems such as TimeStream [QHS +13], Trident [af], Drizzle [VPO +17], and Multilevel Checkpointing [MBMDS10]. It avoids the 2× hardware cost. However, the failover is slower than the replication recovery because it has to retrieve the checkpointed state from the remote storage and replay the buffered data on the last state to recompute the new state.

To achieve both fast recovery and small hardware overhead, the DStream-based lineage recovery was proposed. This approach has been used in Apache Spark-based systems [aa, CEF +17, ZDL +13, SGH +16]. Its key abstraction is the *Discretized Stream* (DStream, for short), a continuous stream of Spark RDDs [ZCD +12]. The most recent state is stored in each node's memory — using RDDs — together with the lineage graph, that is, the graph of deterministic operators used to build RDDs. When nodes fail in the system, instead of preparing a standby node for failover, DStream re-runs the lost tasks in parallel on other reliable nodes in the cluster using the lineage graph. However, the entire recovery processing is linear, that is, the lost tasks need to be executed strictly in line with the original lineage graph. As such, the recovery process may be slow when the lineage graph is long and incur multiple uploads of checkpointed state, incurring substantial network traffic.

Our previous work [LXDS +20] introduced FP4S, a decentralized approach for distributed state recovery based on erasure codes [LXDS +20] that leverages the availability of abundant network bandwidth between the nodes holding fragments of the state. In diverse network environments, the availability of network resources varies dynamically depending on which stream application workflows are active at a given moment; therefore the technique introduced in FP4S is not appropriate for supporting diverse stream workflows.

CHAPTER 3

SUPPORT ONLINE AND SCALABLE SPAM DETECTION

3.1 Introduction

The past few years have seen the rapid rise of Web-based systems incorporating social features, such as online social networks (OSNs) (e.g., Facebook, Twitter). These systems have a common feature that they rely on users as the primary source of posts and enable users to comment on others' posts. Unfortunately, such openness and reliance on users also attract *social spammers*, who advertise commercial spam messages, and disseminate malware [LCW10]. Reports show that nearly 10% of tweets on Twitter are all spam [a15a], and Facebook usually blocks 200 million malicious actions every day [a15b].

We observed that social spammers who aim to advertise their products or post victim links are more frequently spreading malicious posts during a very short period of time. They are quite smart to adapt themselves to spam classifiers which were trained based on historical records. Fig. 3.1 shows the three days' social rumor activities that are extracted from a real-world dataset of the Charlie Hebdo shooting in 2015 [ZLP+16]. X-axis and Y-axis present the time and the number of rumors, respectively. Each peak with a color presents the activities of one specific rumor. We can observe that: (1) once spam post is produced, it spreads in a very short period of time and will soon reach its peak; (2) the content of spam post is always "drifting", and multiple peaks in different colors indicate that the contents of social spam change rapidly.

Besides, recent surveys and research reported that social spam is normally fast changing, and spam activities are usually concentrated in a short period of time [a18, CWZ+17, ZRM15, ZZC+15].

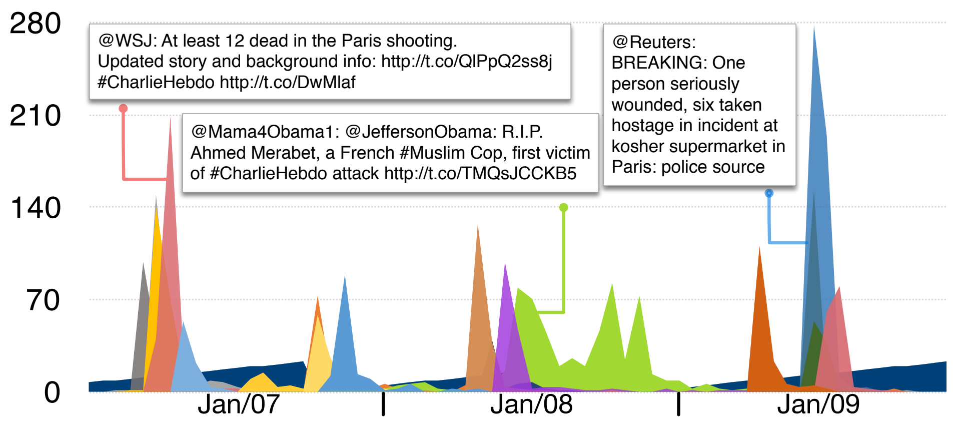


Figure 3.1: Social rumors of the famous Charlie Hebdo shooting in 2015, Twitter [ZLP+16]. We present the activities of 14 rumors in the figure and show several example statements.

Therefore, *the major challenge for the spam detection system is enabling the update of trained classifiers to keep pace of the collection of spam information promptly, so as to uncover and defend against these social spammers.*

Traditional techniques for discovering evidence of spam and spammers have the following limitations: (1) they mainly focus on analyzing offline historical logs [CALS17, LEC11, VT16, VBC+14, WIP14, XSJ16], limiting the capability to adapt to the new spam emergence and resulting in failing to uncover the most recent spam; and (2) they mainly focus on centralized data sources and ignore the fact that most OSN logs are continuously generated in distributed web servers [LEC11, MK10, TLGP14, VT16, WIP14, ZNJ+15], limiting the capability to take the advantage of continuously processing the distributed data on the fly and resulting in centralized bottleneck and load unbalance.

From the data mining point of view, the spam detection has three major steps: (1) model construction where the spam classifier is created using a training dataset with a specific algorithm, e.g., Random Forest [Bre01]; (2) model test where the test dataset is used to validate the accuracy of the spam classifier; and (3) use well-trained classifier with new social data to get predictions that identify new spam [PZTH].

We propose a novel **O**nline sc**a**lable **s**pam d**e**tection **s**ystem, namely `Oases`. The key idea of `Oases` is to enhance the ***online*** feature and ***scalable*** feature into the general spam detection processing, in which the spam detection model is continuously constructed with the online training dataset and model testing, and the spam detection is performed in a scalable fashion.

`Oases` operates in two phases. The first phase is the spam model construction. We build a distributed hash table (DHT) [RD01] based aggregation tree to feed the distributed data sources into a decentralized peer-to-peer overlay, which consists of a root, branches and many leaves. The root is responsible for disseminating the continuously updated training dataset and test dataset to the branches and leaves. Each leaf node is responsible for spam model construction and spam model test by applying the training dataset to a specific classifier such as Naive Bayes algorithm [Ris01], Random Forest algorithm [Bre01], etc., and testing the spam model using the test dataset. The intermediate results are aggregated by branches to the root for validation and confirmation.

The second phase is the spam model application. Each leaf takes the new coming streaming events from the distributed data sources (e.g., Twitter logs, Facebook logs, etc.), analyzes them using the spam detection model from the first phase, outputs spam and labels. The spam with their labels are then aggregated to the root and reported to the end users.

The novelty of our work lies in that the posts are progressively aggregated for actively filtering out new spam and publishing the training dataset to all distributed leaf agents to update the classifiers in an online and scalable fashion. We believe our system could promptly detect activities of spammers and classify various latest

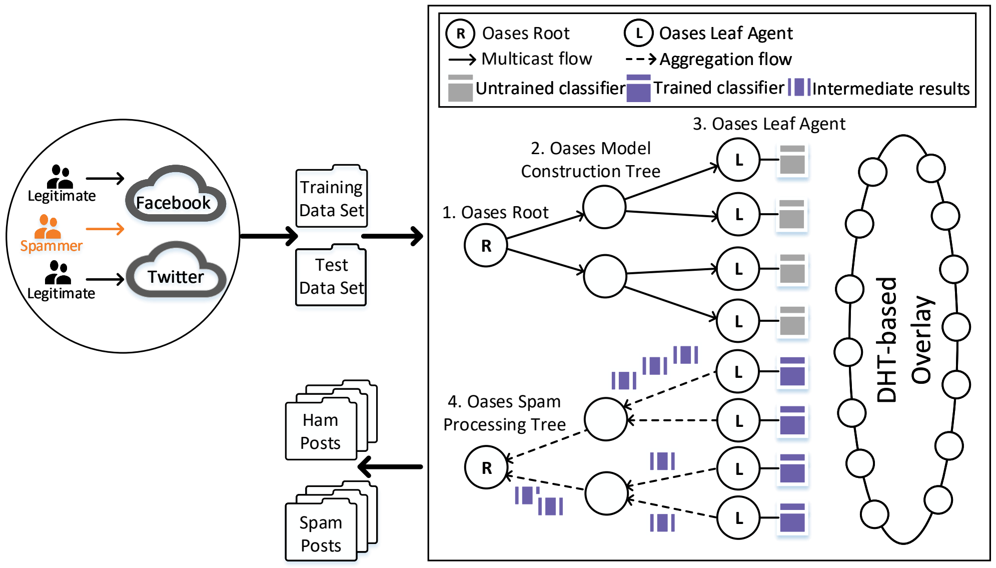


Figure 3.2: Overall framework of Oases system.

spam for social networks.

3.2 Design

In this section, we introduce the Oases system, discuss each functional component of the system, and outline the details of workflows in the Oases system.

3.2.1 Overview

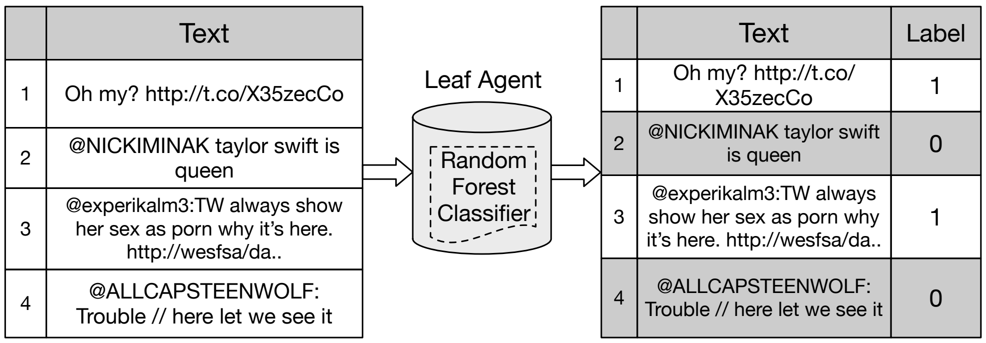
As shown in Fig. 3.2, the Oases system consists of four major components: (1) the Oases root; (2) the Oases model construction tree; (3) the Oases leaf agent; and (4) the Oases spam processing tree.

The first component is the Oases root. The Oases root is responsible for the main control flows on other nodes, e.g., publishing the instructions from the Oases root to branches and leaf agents to start training the classifier, delivering messages to the Oases leaf agents to start classifications, etc. As shown in Fig. 4.1, for the first step, the `Oases` root manually divides the raw dataset into training dataset and test dataset. The training dataset and test dataset are then disseminated by the `Oases` root to all distributed `Oases` leaf agents through the `Oases` model construction tree which is discussed next.

The second component is the `Oases` model construction tree. As shown in Fig. 4.1, for the second step, the `Oases` model construction tree is responsible for creating ***efficient paths*** for the `Oases` root to disseminate the training dataset and test dataset to the `Oases` leaf agents. The key idea is the use of a DHT-based application-level multicast tree [CDKR02], similar to the IP multicast tree [DLL$^+$00], to disseminate copies in a progressively way following the tree path, without maintaining N point-to-point connections for N leaf agents.

The third component is the `Oases` leaf agent. The `Oases` leaf agent is responsible for the training of the spam detection model. As shown in Fig. 4.1, for the third step, the leaf agent applies the received training dataset to the Random Forest algorithm [Bre01] to practice the classifier, and uses the test dataset to enforce the classifier. The trained classifier is used later by the fourth component to do the online spam detection.

The fourth component is the `Oases` spam processing tree. The `Oases` spam processing tree is responsible for orchestrating the distributed `Oases` agents to fulfill the data mining tasks of online spam detection in a scalable fashion. The `Oases` leaf agents are directly connecting to the web servers that generate user activity logs, i.e., tweets, and classify spams out of these logs. As shown in Fig. 4.1, for the fourth step, the workflow of `Oases` spam processing tree is as follows: a scalable aggregation tree "rolls up" the classified results from the `Oases` leaf agents level by level until the results reach the root. For example, if one tree has 7 spam processing agents and each leaf agent classifies 10,000 social data, then after aggregation, the



	Text
1	Oh my? http://t.co/X35zecCo
2	@NICKIMINAK taylor swift is queen
3	@experikalm3:TW always show her sex as porn why it's here. http://wesfsa/da..
4	@ALLCAPSTEENWOLF: Trouble // here let we see it

	Text	Label
1	Oh my? http://t.co/X35zecCo	1
2	@NICKIMINAK taylor swift is queen	0
3	@experikalm3:TW always show her sex as porn why it's here. http://wesfsa/da..	1
4	@ALLCAPSTEENWOLF: Trouble // here let we see it	0

Figure 3.3: Sample of the data classification in the `Oases` leaf agent.

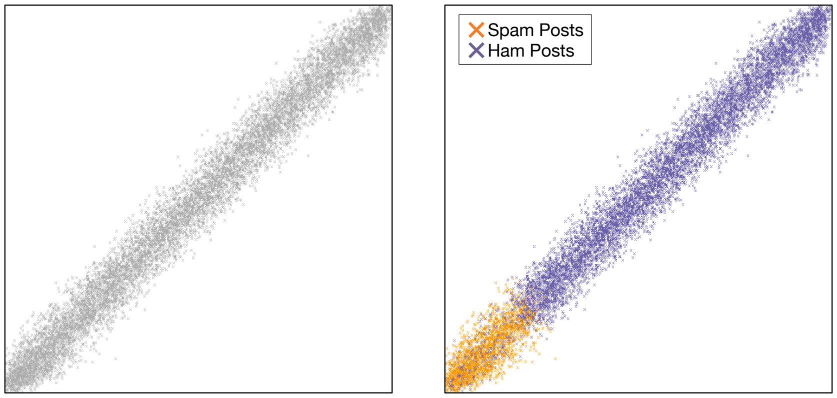
root agent receives 70,000 classified results.

3.2.2 Root

The `Oases` root is responsible for the main control flows of the whole system, including (1) dividing the raw dataset into training dataset and test dataset; (2) publishing the datasets from the `Oases` root to branches and leaf agents to start training the classifier; and (3) aggregating the spam detection intermediate results from the `Oases` leaf agents to the root.

The `Oases` root uses a DHT-based hierarchical tree as the main channel for disseminating datasets and instructions. The DHT-based hierarchical trees are built as follows:

1. Step 1: constructing a peer-to-peer overlay leveraging Pastry [RD01]. Each `Oases` node is assigned a unique, 128-bit nodeId in a circular nodeId space ranging from $0 \sim 2^{128} - 1$. All nodes' nodeIds are uniformly distributed. Given a message and a key, the message can be guaranteed to be routed to the node with the nodeId numerically closest to that key, within $\lceil log_{2^b} N \rceil$ steps, where b is a base with a normal value 4.



(a) Unclassified data. (b) Classified data with labels

Figure 3.4: Visualizaiton of the data classification in the `Oases` leaf agent. (a) shows the original dataset which has no labels. (b) shows the dataset with labels predicted by the model.

2. Step 2: building a multi-cast tree leveraging Scribe [CDKR02] (more details can be found in Section 3.2.5). Any node in the overlay can create a group with a groupId which is the *hash* (SHA-1) the group's name concatenated with its creator's name. Other nodes can join the group by routing a JOIN message towards the groupId. The node which its nodeId is most near to the groupId serves as the root. The tree multicasts a message to all members of the group within $O(logN)$ hops.

3. Step 3: enhancing the aggregation function on branches. The `Oases` root and middle-level nodes jointly implement (1) the aggregation flow and (2) the control flow. For example, for the social spam detection application, batches of social logs are parsed as a map from hashed data contents (ID) to the classified tags (*labels*), i.e., (*DDA2*, 1) and (*F7B5*, 0) in the `Oases` leaf agent (here the classified tag 1 means the data is classified as spam, tag 0 represents non-spam, and we use shortened hashes to indicate ID). Then the aggregation tree that progressively 'rolls up' and reduces those *ID-label* pairs from the distributed leaf agents to the root (more details can be found in Section 3.2.6). Besides,

when necessary, the Oases root can multicast to its workers within the group, to notify them to empty their sliding windows and/or synchronously start a new batch.

3.2.3 Leaf Agent

The Oases leaf agent is responsible for the local data processing task by executing the root's instructions. Data processing task in the leaf agent consists of two roles: (1) local data classification and; (2) local online social spam detection.

The local data classification is the first role of the leaf agent. Each leaf agent trains a classifier by using the training dataset. Then it uses the test dataset to examine the accuracy of the trained model. Besides, the leaf agent updates its trained model periodically with the new delivered training and test datasets from the root. This allows the trained model to detect spam efficiently with the latest spam features.

Fig. 3.3 shows the processing of data classification in a leaf agent. Original social data is normalized in dataset without labels. After the classification via the trained model, each instance of the original data acquires a classified label, which identifies spam or not.

Fig. 3.4 shows the visualization of data classification in one leaf agent. Fig. 3.4a shows the original dataset without predicted labels. After the classification, this dataset is classified as two groups, as shown in Fig. 3.4b, where the purple and orange color represent *Ham* (non-spam) and *Spam*, respectively.

The local online social spam detection is the second role of the leaf agent. In Oases, each leaf agent connects to a web server so as to collect the online social streaming data from this server. Then the leaf agent completes the online data

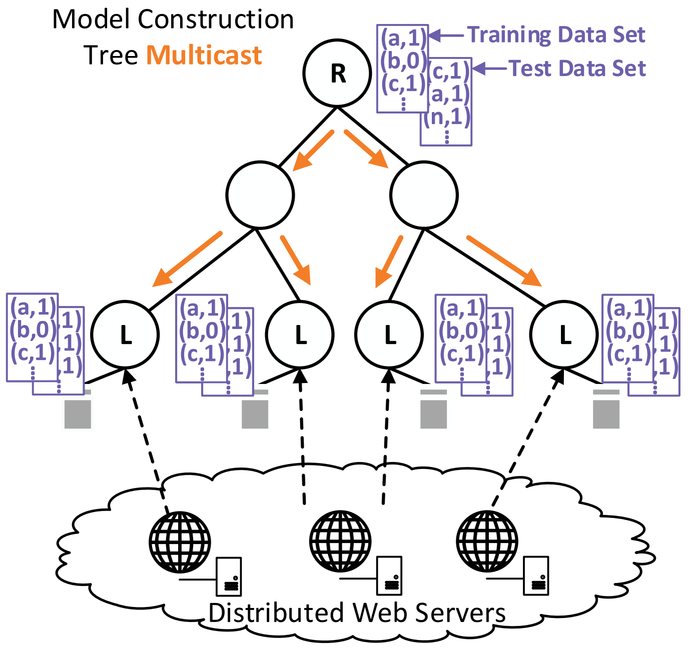


Figure 3.5: `Oases` Model Construction Tree.

analysis upon streaming data flow with the trained model and produces classified results. Finally, all leaf agents collaborate to shuffle the classified results to the upper layer via the spam processing tree. More details can be found in Section 3.2.6.

3.2.4 The Classified Algorithm

We next introduce the details of the classic algorithm that be implemented, Random Forest algorithm [Bre01], in our training and test processing.

Why Random Forest? Random Forest algorithm is a classic data mining algorithm and had been implemented with graceful performances in various works of social spam detection [WZLP15, XSJ16, ZZC$^+$15]. Random Forest constructs a fixed number of decision trees for training during the training processing and results in one final decision which is determined from multiple individual trees. This algorithm is derived from decision tree learning and tree bagging.

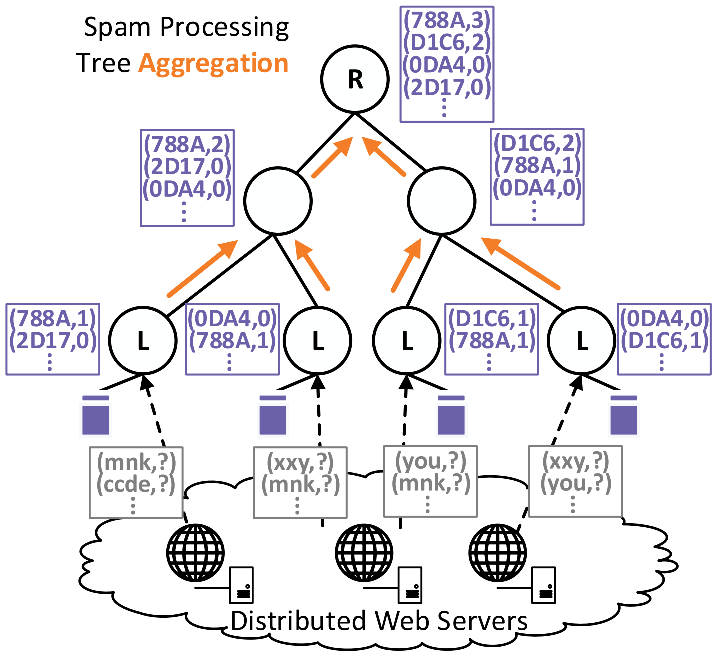


Figure 3.6: Aggregation in the spam processing tree.

In the training process, the classifier in each leaf agent receives the training dataset from the root agent, then randomly samples N cases to create a subset of the data. The subset usually about 66% of the total set. One subset of the samples creates one decision tree. That is repeatedly to choose some different small subset of attributes at random and creates all decision trees. When leaf agent starts the test processing, trained classifier puts the test dataset into the forest. Then it runs down all trees of the forest. The classification result is the majority vote among all decision trees.

3.2.5 Model Construction Tree

The `Oases` model construction tree is responsible for creating ***efficient paths*** for the `Oases` root to disseminate the training dataset and test dataset to the `Oases` leaf agents. Here we use an example to illustrate the `Oases` model construction tree. The sample scenario is presented in Fig. 3.5. Assume there are 7 nodes in the `Oases` system. The node with nodeId numerically closest to the *topicId* acts as the rendezvous point for the associated multicast tree. For example, if *hash*(model) equals to 0088, the node with same identifier or closest identifier like 0087 or 0089 will be the root of the model tree. The `Oases` model construction tree is shown in Fig. 3.5. The tree is rooted at the rendezvous point and the other nodes subscribe to this tree. The `Oases` root multicasts the training and test datasets to all leaf agents in $O(logN)$ hops. In the figure, (a, 1) means that the post "a" is a spam post and (b, 0) means that the post "b" is a non-spam post. Then those leaf agents are triggered to apply the received dataset to the local classifier to complete the model training and test processing using the Random Forest algorithm [Bre01].

3.2.6 Spam Processing Tree

The `Oases` spam processing tree is responsible for coordinating distributed leaf agents to accomplish the online spam detection globally. In this section, we use a sample scenario to present the workflow of the `Oases` spam processing tree.

As shown in Fig. 3.6, the leaf agent processes the online social streaming data with its trained classifier. For instance, after processing, original data (*mnk*, ?), in which the question mark means it hasn't been classified, is detected as spam and marked as (*mnk*, 1). Then the leaf agent sends the hashed content, e.g., *hash*(*mnk*) = *788A* and its label formatting as (*788A*, 1), to the upper layer.

Further, the spam processing tree progressively rolls up and reduces those ID-label pairs from the distributed leaf agents to the root. For example, ⟨(*788A*, 1), (*2D17*, 0)...⟩, ⟨(*0DA4*, 0), (*788A*, 1)⟩, are reduced as ⟨(*788A*, 2), (*2D17*, 0), (*0DA4*, 0)...⟩ in the branches of tree. And then those pairs are reduced as ⟨(*788A*, 3), (*D1C6*, 2), (*2D17*, 0), (*0DA4*, 0)...⟩ to the root as the final results. The value of labels indicates the number of leaf agents which detect this data as spam, e.g., (*788A*, 3) represents that there are 3 leaf agents classifying the data "*mnk*" as a spam post. Those hashed IDs with larger values in labels indicate the higher possibility as spam posts.

3.2.7 Self-adjustable Tree

`Oases` supports self-tuning in the tree structure level. By manipulating the parameter n of the tree fan-out, with achieving 2^n fan-outs per agent, it can format different trees.

The design of this feature is to support multiple targets in spam processing. For example, when an application is high latency sensitivity, it can modify the tree depth by adjusting the value of tree fan-out. Assuming there are 10^b agents in `Oases`, the default depth of the tree is $log_{2^b}N$, where b = 4. By changing the default fan-out from 2^4 to 2^5, the average depth of the tree is reduced from 5 to 4. So root-to-leaf data transfer can achieve lower latency by across fewer layers.

When an application desires a good failure recovery, `Oases` can increase the depth of trees by reducing the tree fan-out. Using the same example above, `Oases` can change the fan-out from 32 to 16, resulting in that a tree's depth increases from 4 to 5. A deeper tree can benefit the agent's failure recovery. This depends on the mechanism for failure recovery in `Oases`: once a child fails to receive a heartbeat message, it suspects its parent failed, and this agent will route the JOIN message to the group's identifier. `Oases` then sends the message to a new parent to repair the tree. When a tree has a small fan-out and a large depth, the failure of one agent can only affect the performances of the following sub-agents, while fewer sub-agents can reduce this failure effects.

3.2.8 Benefits and Design Rationale

In this section, we discuss why `Oases` has the online and scalability benefits and the rationale behind the design.

Online. `Oases` enables the progressive aggregation of the properties of the spam posts for creating new spam classifiers to actively filter out new spam posts and update the classifiers to all distributed data process agents. That ensures the spam classifiers to always keep pace with the latest social spam, and identify new spam with high efficiency.

Exploring DHTs for Scalability. The `Oases` model construction tree and spam processing tree are self-organizing and self-repairing, and can be easily expanded in a distributed manner. The use of DHT guarantees that the cost of multicast and aggregation can be fulfilled within $O(logN)$ hops. Moreover, multiple groups (e.g., model construction tree and spam processing tree) are supported in one single overlay, which means that the overhead of maintaining a complex overlay can be amortized over all groups' spinning trees [RD01]. Specifically, all agents in overlay are viewed as equal peers, so, each agent can be a root, parent, leaf agent or any combination of the above, which leads to well balance of the computation loads.

Handling Nodes' Failures. The `Oases` system uses leaf sets to handle node failure [CDKR02]. Each node maintains a leaf set. The leaf set is the set of l nodes which nodeIds that are numerically closest to the present nodeId, with $l/2$ larger and $l/2$ smaller. A typical value of l is nearly $8\lceil log\ {}_{2^b}N\rceil$, where N is the total number of nodes in the system. Neighboring nodes in node' leaf set exchange keep-alive

Table 3.1: Results of spam classification with multiple classifiers.

Classifiers	**Accuracy**	**F1**	**FPR**
Random Forest	**94.8%**	**0.962**	**0.26**
SVM	94.5%	0.937	0.446
KNN	91%	0.911	0.374
Logistic	92%	0.908	0.303
Naive Bayes	86%	0.871	0.477

messages periodically. An agent is presumed as a failure if it is unresponsive for a period. Then those members in the left set of failed node's leaf set are notified and they update their leaf sets. Once the node recovers, it will contact the node in its last known leaf set, obtains their current leaf sets, updates its own leaf set and then notifies the members in the new leaf set of its recovery.

3.3 Evaluation

We evaluate the `Oases` system with the real-world online social network streaming data. Experimental evaluations answer the following questions:

- What are the spam detection accuracy rates of the `Oases` system (Sec. 3.3.2)?
- What are the performances of data shuffling, processing and delivery latency in `Oases` (Sec. 3.3.3 & Sec. 3.3.4)?
- What is the overhead and resource consumption of the system at runtime (Sec. 3.3.5)?

3.3.1 Testbed and Application Scenarios

Experiments are conducted on a testbed of 800 agents hosted by 16 servers running on Linux. Each server has a QEMU Virtual CPU with 3.4GHz processor, 4G of

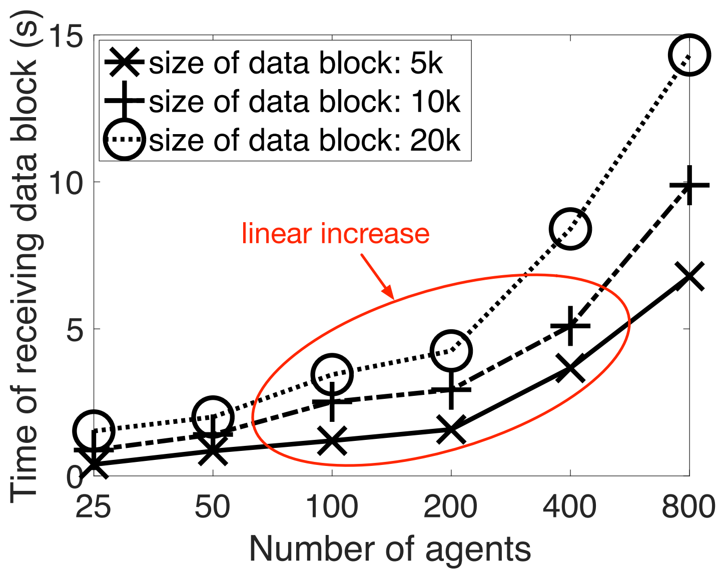


Figure 3.7: Time of the leaf agents receiving data blocks from root with various size of data blocks and different number of agents.

memory and 30 GB hard drives. The system was implemented in Java by using Java SE Development Kit 7 in x64, version 1.7.

`Oases`'s functionality is evaluated by running an online social data application. Nearly 3,000,000 tweets from Twitter streaming API had been collected and evaluated via our system from 12.2016 to 02.2017. The application's purpose is to identify social spam, such as posts used by malicious links or contents to draw users' clicks and spread malware. We use the straightforward content features (URL, words, etc.) to predict labels. `Oases` uses test dataset to examine the model which is trained with training dataset. The application is implemented to produce predicted labels from online data streams via the `Oases` leaf agents.

3.3.2 Spam Classification Results

We evaluate the spam classification results of `Oases` and compare the performance with several popular classifiers. Evaluations rely on a sample dataset which consists

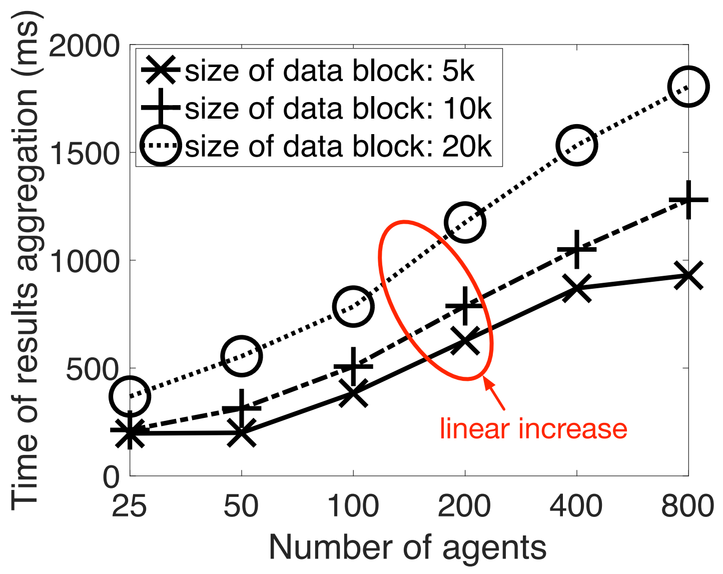


Figure 3.8: Time of shuffling data from leaf agents to upper layer in the tree.

of 50,000 posts (37465 posts are *Ham* and 12535 posts are *Spam*). The results are shown in Table 3.1. Random Forest is default implemented in `Oases` and other classifiers include K-Nearest Neighbor (KNN), Support Vector Machine (SVM), Logistic Regression, and Naive Bayes. F1-score (F-Measure) responses for an important factor in measuring the classification performance. The results show that Random Forest achieves promising performance with the F-measure up to 96.2% and the accuracy up to 94.8%. Combined with these key indicators, Random Forest achieves the best performance among all classifiers.

3.3.3 Data Shuffling Time and Data Processing Time

The `Oases` model construction tree, as the efficient paths for dataset distribution, directly influences the local data processing time. On the other hand, after the data processing, the classified intermediate results propagate from distributed leaf agents, to the upper layer for aggregation, until they reach the `Oases` root. The aggregation tree, as the structure for shuffling classified results from the leaves to

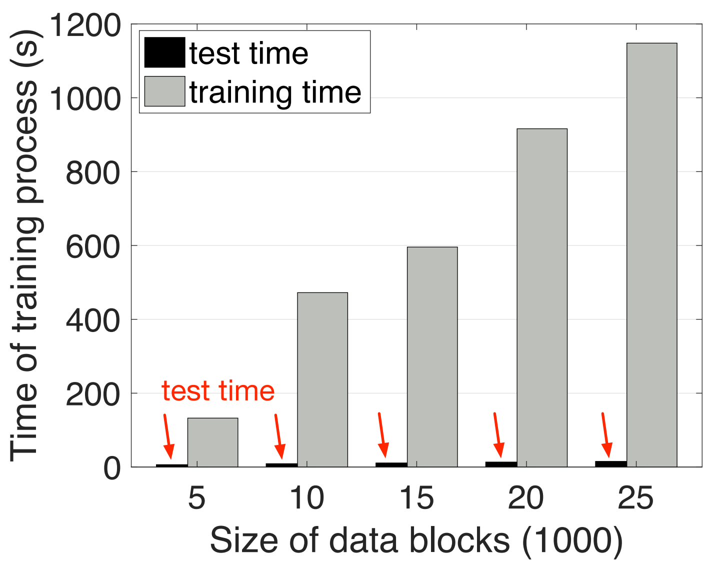


Figure 3.9: Training and test time in data processing with different size of data blocks.

the root, directly influences total online spam processing latency. Therefore, we first report the time of the leaf agents receiving data blocks (datasets) from the root in Fig. 3.7, and then report the time of intermediate results aggregating from the leaves to the root in Fig. 3.8. Finally, we show the data processing time of each leaf agent in Fig. 3.9.

Data Shuffling Time. Here we classify the data shuffling time into two parts: (1) the time of the leaf agents receiving data blocks from the root; and (2) the time of results aggregating from the leaf agents to the root. The number of the `Oases` agents varies from 25 to 800. Simultaneously, various sizes of data blocks are used for evaluation.

Fig. 3.7 and Fig. 3.8 show that, when the system uses the same datasets but with a different number of agents, the time of delivery and reception linearly increases, rather than fold increases. This is because that the linear increment of the delivery or reception time is strictly determined by the tree depth $O(logN)$, which further reflects that the tree topology in the overall performance exhibits a very

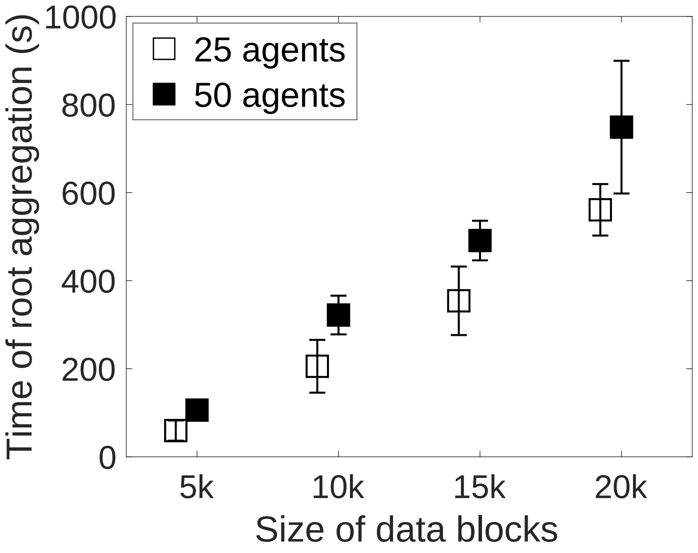


Figure 3.10: Average latency of root agent aggregates whole results in one process cycle.

good balance.

Data Processing Time. Fig. 3.9 shows the time of model training and test processing in one agent with various sizes of data blocks. Result shows that with the increment of the size of data blocks, the training processing time also increasing rapidly, especially when the data block has $20k$ and $25k$ posts. It indicates that over-large size of data blocks can be the bottleneck of the whole system when considering the size reaches to $25k$ with the training time up to 1150s. Costly time in the training processing will cause the whole system looks like in "busy-waiting" - though the leaf agent is working on the training processing, the root cannot get any useful results in a long time. Therefore, the choice of a suitable size of the data block can promote the best performance of the system.

3.3.4 Aggregation Latency and Self-adjustable Tree

Performance impact due to large data blocks. After the hashed ID-label pairs are shuffled to the upper level, the `Oases` root actively aggregates the data

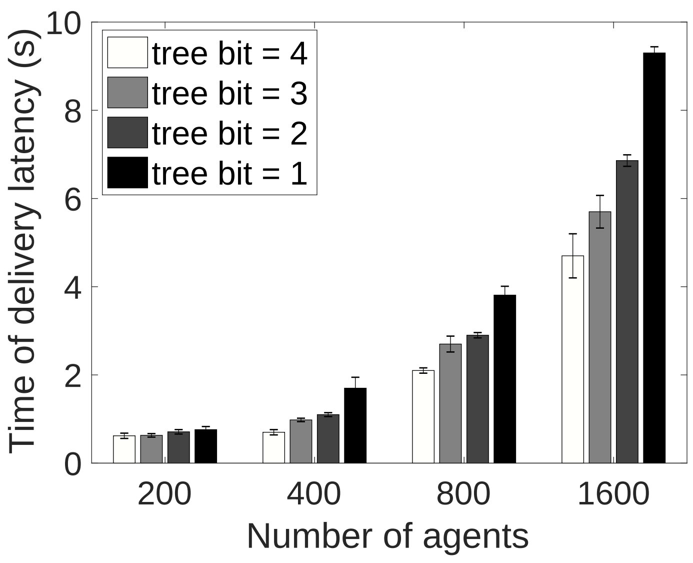


Figure 3.11: Average data delivery latency across the different trees.

Table 3.2: The runtime overheads of `Oases`. It represents the CPU, Memory, I/O, and context switch overheads.

VSD	CPU	Memory	I/O	C-switch
	%used	*%used*	*wtps*	*cswsh/s*
5k	47.6%	38.0%	2.23	322.72
10k	51.1%	43.8%	3.20	280.69
15k	51.0%	44.2%	2.78	304.57
20k	50.9%	45.1%	1.80	314.06

VSD: various size of data blocks.
wtps: write transactions per second.
cswsh/s: context switches per second.

stream into the final result pool. Although the `Oases` architecture ensures that the aggregation processing can be completed with $logN$ hops. However, many factors may impact the performance of the root aggregation, such the size of the data blocks, the network bandwidth, and the traffic interference.

As shown in Fig. 3.10, the size of the data blocks (training and test datasets) has an important effect on the latency. In the case of reasonable dataset size, e.g., $5k$, the average latency is nearly 100 seconds. However, when using large data blocks

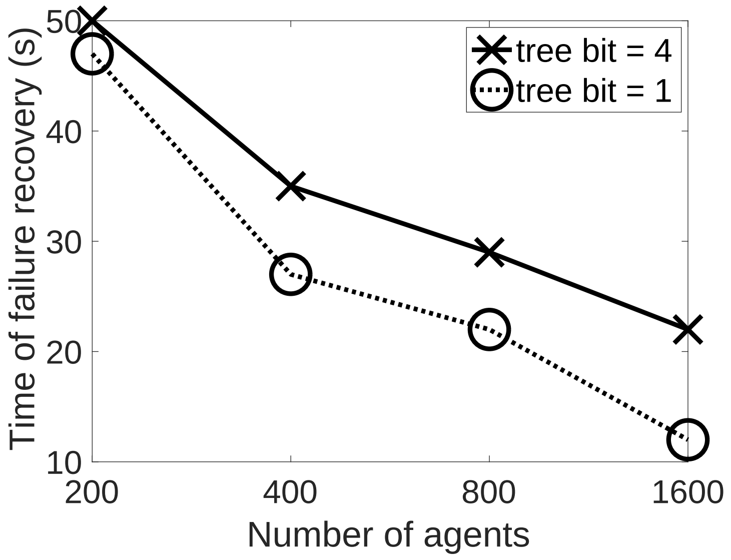


Figure 3.12: Average fault recovery latency of failed agents in different trees.

(e.g., $20k$), the latency grows much faster than the size increment of the data block.

We believe that the increased latency indicates that the system has reached a limited overload when processing with extra large data blocks. In this case, some agents are still active but other agents may be blocked to wait for the server's resources. In addition, oversized dataset exacerbates the burden of each leaf agent during the training and testing processing that causes the overload even further overload.

Tree Structure Adjustment. Fig. 3.11 and Fig. 3.12 represent the performances of delivery latency and recovery latency with different tree structures. In Fig. 3.11, *tree bit* decides the tree fan-out of each agent. For example, when *tree bit* = 4, the fan-out is 16, which means each agent has 16 following agents. Results in Fig. 3.11 show that delivery latency increases when the tree layer increases (small *tree bit*), in which delivering a data block from the leaf to the root need to cross more layers.

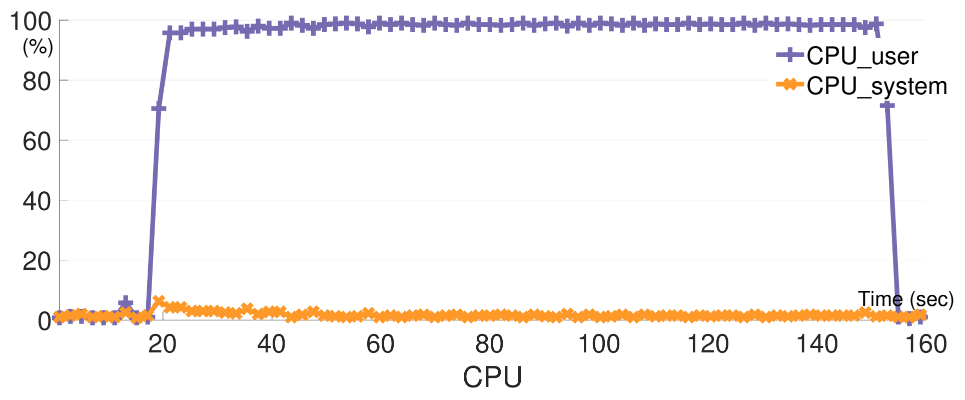


Figure 3.13: CPU utilization in one server.

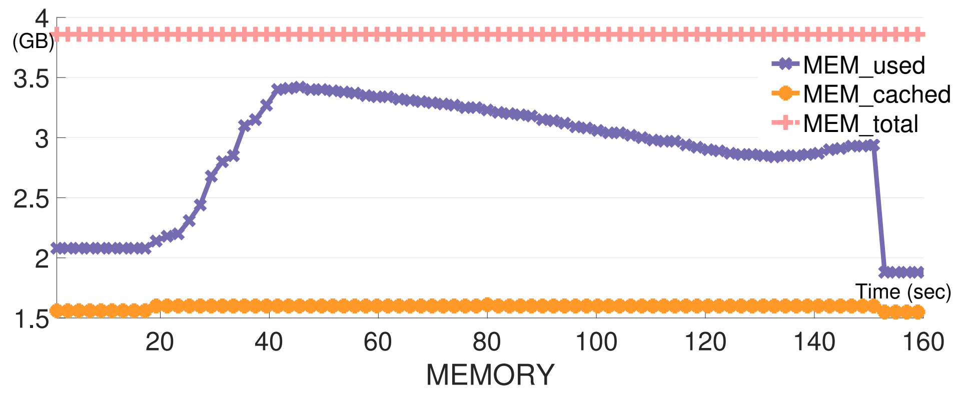


Figure 3.14: Memory utilization in one server.

3.3.5 Runtime Overhead

Table. 3.2 shows the runtime overhead of `Oases`. As the result shows, the `Oases` system has similar overheads in the utilization of CPU, memory, I/O, and context switches when dealing with the data blocks of different sizes. This is because the `Oases` system uses a decentralized architecture to distribute the management load evenly over the distributed servers, and the hierarchical tree structure facilitates communication across multiple agents and servers.

We also evaluate the server's resource consumption, with each server supporting five leaf agents at the same time. As shown in Fig. 3.13 and Fig. 3.14, the processing cycle time is close to 140 seconds, with the CPU and memory utilization reaching a higher level from 15s to 155s. In addition, the processing performance is quite consistent with the former results.

3.4 Summary

In this chapter, we present the online scalable spam detection system (`Oases`), a distributed and scalable system which detecting the social network spam in an online fashion. By periodically updating the trained classifier through a decentralized DHT-based tree overlay, `Oases` can effectively harvest and uncover deceptive online spam posts from social communities. Besides, `Oases` actively filters out new spam and updates the classifiers to all distributed leaf agents in a scalable way. Our large-scale experiments using real-world Twitter data demonstrate scalability, attractive load-balancing, and graceful efficiency in online spam detection.

CHAPTER 4

ACHIEVE EFFICIENT SPAM DETECTION BY EXPLOITING SAPM CORRELATIONS

4.1 Introduction

Online social networks (OSNs) have been an integral part of human life. More and more people are acquiring the latest news, advertisements, social activities, and breaking topics directly from the current popular OSNs such as Facebook, Twitter, and WeChat. For example, a report said that the percentage of US adults who primarily receive news and information from OSNs is as high as 62% [AG17]. However, the openness of widespread OSNs couple with massive spam activities, which are damaging as they cause public panic and social unrest. For example, in February of 2019, social users in Paris watched a lot of photos of kidnappings on Facebook and videos of vans speeding away on Snapchat and Twitter, all of which hinted that the Roma (Gypsies) robbed children with vans in the suburbs of Paris [Bre19]. Although the information proved to be wrong later, they brought serious consequences to the Roma and the whole society: dozens of young men wielding sticks and knives attacked a Roma camp and burned two vans, and tens of people were arrested. Another example is that one latest report said the global enterprise spam filter market was valued approximately USD 849 million in 2018 and is expected to generate around USD 2,675 million by 2026 [zio19]. And it pointed out that the increasing number of social spam is driving the enterprise spam filter market globally.

The unprecedented success of online social networks has created tremendous opportunities for the emergence and rapid spread of spam. By leveraging a large social user, social spam often dominates and influences social life in a short period of time and can reach every corner of the social world. Therefore, quickly detecting spam from large-scale social activities is an urgent need in the current situation.

Furthermore, as our observation, the spammers in the online social networks are not only active on a single platform, but are often active on different social platforms, by simultaneously manipulating dozens or hundreds of fake accounts. Naturally, the information published by these fake accounts is highly similar. This phenomenon has been pointed out by several former studies [XGL$^+$18]. Spammers certainly desire to spread similar posts on different platforms to attract as many people as possible to target on these topics. A case study of social spam posts for multiple different news sites also demonstrates that spam posts show a high degree of similarity in content and topics during the same period of time and will immediately propagate from one site to another [a16]. Therefore, this correlation between cross-platform social spam is a common phenomenon in the current social media world. Although there are not many direct relationships between users, geographic locations, creation purposes, and regions in these various groups or platforms, the spam contents are highly correlated within similar topics during the same period of time.

However, former studies rarely utilized the spam correlations to handle the large-scale social data from distributed data servers. They either focused on the algorithm side to achieve high accuracy in the detection [VBC$^+$14, WP15, VT16, SS16, HBSD17], or the entire processing only targeted on a small size of dataset without the global view from similar data across large-scale data sources [GCL$^+$12, XZJ$^+$16, CWZ$^+$17, XSJ16]. In this paper, to explore the efficient method in dealing with large-scale social data sources, we present a new social spam detection system, named **SpamHunter**, to take advantage of the spam correlation among distributed data sources for efficient large-scale social spam detection. SpamHunter implements multiple groups, where each group contains a DHT-based functional tree that jointly

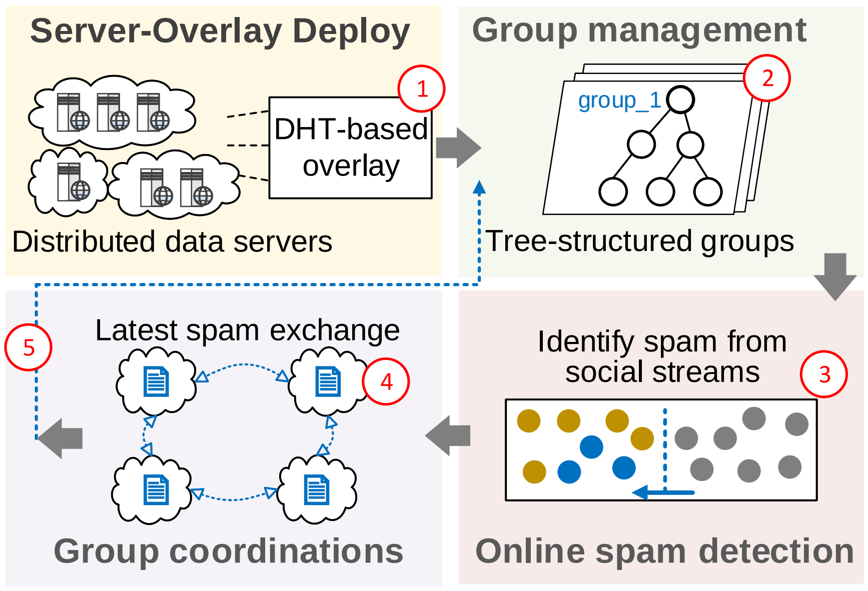


Figure 4.1: The overview of the system design.

connecting multiple data sources (e.g., servers, datasets, etc.) to share the spam correlations (e.g., updated spam features) in a distributed manner. The DHT-based functional trees response to data delivery, spam identification, and correlation exchanges. Besides, the group-level coordination ensures multiple groups or clusters can instantly exchange and share the correlated features during the processing, that is, they collectively leverage the latest spam correlations to enhance the performance of spam detection.

4.2 Design

4.2.1 Overview

Figure 4.1 shows the designed architecture of the SpamHunter system. SpamHunter is built upon a peer-to-peer DHT-based Pastry overlay [RD01]. The overlay is utilized to orchestrate large-scale distributed social servers. As shown in the first step of Figure 4.1, these data servers can be grouped by various kinds of features (e.g., geo-location, topic tags, or institutions), and the large amount of servers are connected to the DHT-based overlay. In the second step, SpamHunter creates a functional tree upon Pastry for each group, where nodes jointly route around a specific key (see details in subsection 4.2.2). The functional tree for each group will respond to the primary workload during processing, for example, data dissemination, spam detection, and results aggregation.

In the third step, SpamHunter deploys online social spam detection within the group management. In each group, SpamHunter manages the functional tree to fulfill the online social data processing. The root of the tree is responsible for the data/model dissemination and in charge of the entire workflow. The distributed leaf nodes will complete the processing of spam detection by following the root's instructions by coordinating the classified models. The root of the tree also aggregates the identified results from the following nodes, updates the spam dataset, and extracts the latest spam. Furthermore, as shown in the fourth step of Figure 4.1, after the online spam detection, multiple groups in SpamHunter will periodically exchange and share the latest spam with others, so that all groups have a global view of the newest social spam and then utilize the correlated new spam in the continuing processing, as shown in the fifth step of the Figure 4.1.

Next, we will introduce the system's functionality and implementations details. We first introduce the deployment of SpamHunter and the group management. Then we introduce the online social spam processing. Finally, we propose the group coordination and communication in enhancing the detection performance by leveraging the large-scale spam correlations.

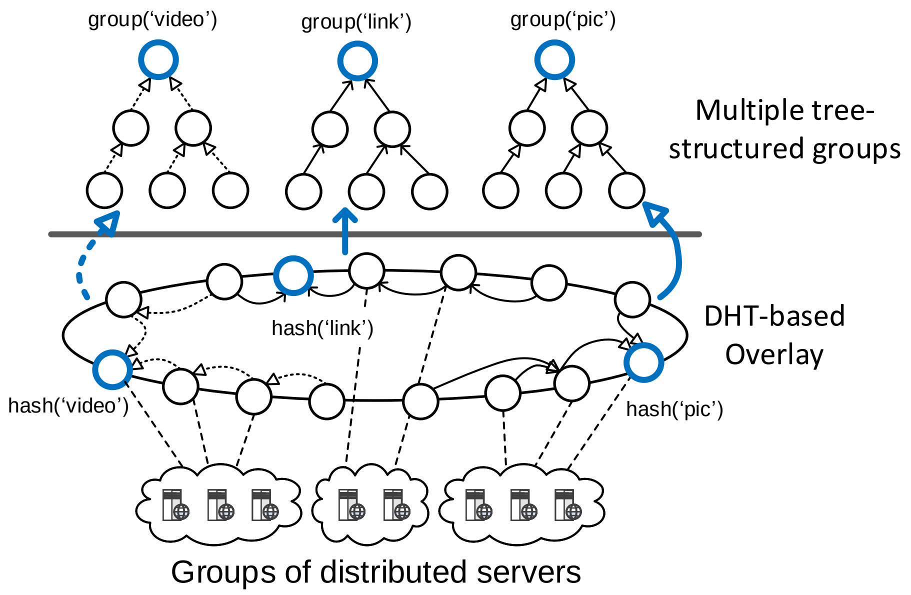


Figure 4.2: The server-overlay structure and group management.

4.2.2 Group Management

We first present the details of the overlay in SpamHunter. SpamHunter is built upon the peer-to-peer Pastry overlay [RD01], where each node has a unique 128-bit nodeId with a nodeId space ranging from $0 \sim 2^{128} - 1$. Note that all nodeIds are evenly distributed, so that the deployment of nodes can be flexibly scaled to a large amount of instances. The message is the main link between nodes: nodes can route messages towards a specific key, for example, the key can be a target nodeId, a groupId, or a specific topic concatenates with a groupId. With the targeted key, messages can be routed to the node which nodeId is numerically closest to the key in $\lceil log_{2^b} N \rceil$ steps, where the default value of b is 4.

By leveraging Scribe [CDKR02], each node in SpamHunter can create a group by a groupId. Typically, the groupId is obtained by hashing (SHA-1) the name of the group with the name of its creator. Other nodes can randomly join a group by routing a JOIN message towards the groupId as the key, which enables flexible group membership. The nodeId of the rendezvous node in the group is closest in value to the groupId. Each group constitutes a functional tree which creates valid paths for the root to communicate with multiple layer nodes. The key idea is the use of a DHT-based application-level multicast tree [CDKR02] to propagate data/model replicas through the tree path, which has the advantage of not maintaining N point-to-point connections for N leaf nodes. For example, assuming there are 7 nodes jointly work as group "*video*", if $hash$(video + creator name) equals to $EA34$, the node whose nodeId is closest to it, such like $EA34$ or $EA35$, will serve as the root of the functional tree. The other six nodes will then subscribe to this tree and follow the root node. Due to the tree structure, the tree root can multicast the messages, instructions, or models to all leaf nodes in $O(logN)$ hops.

SpamHunter Tree's Functions. SpamHunter creates multiple groups to support the scalability of social data processing. Each group constitutes as a functional tree, where the spam detection is fulfilled in this tree. As shown in Figure 4.3, the group's functional tree mainly has four functions: *spam detection*, *aggregate function*, *spam extract*, and *external/inner tunnel*. We next present the details of these functions.

The *spam detection* is fulfilled by the coordination of the root and leaf nodes. In the group's tree, the tree root is in charge of the workflow of spam identification via the *inner tunnel* in root and leaf. The root of the tree will build a spam detection model by training the model using the training data set and then testing the model using the test data set. In addition, it manages the processing workflow by propagating instructions and models through the functional tree to the following branches and leaf nodes. By following the instructions of the tree root, the leaf nodes complete the pre-data processing and spam identification with the model. Details are presented in subsection 4.2.3.

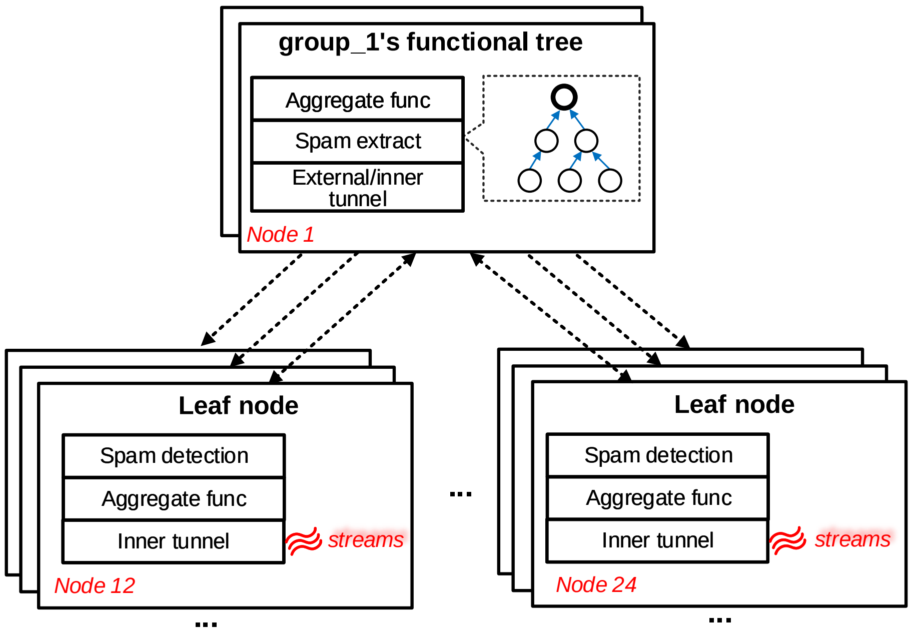


Figure 4.3: The DHT-based functional tree.

The SpamHunter functional tree supports *aggregate function* during the processing to collect the interim results after spam detection. The tree branches and middle-level nodes are able to jointly work with the tree root to fulfill the aggregation. We next use an example to present it. After the local social spam classifications in the leaf nodes, batches of social logs are parsed as mappings from the content (*posts*) to categorical tag (*labels*), i.e., (*post_1*, 0) and (*post_2*, 1) in leaf nodes (here the tag 1 represents spam and the tag 0 represents non-spam). The leaf node will first filter out the identified spam data, (i.e., the social data has been detected as spam and marked with label 1), then sends the paired instances, e.g., (*post_i*, 1), to the upper layer via the *deliver tunnel*.

The third function supported in SpamHunter tree is the *spam extract*. After results aggregation, the tree root will accumulate the latest spam posts from the collected interim results and identify the prospected posts which are most highly be spam. For example, in a specific case, when 6 servers' interim results notify that the social post *post_k* as spam post, after the aggregation, the root will acquire the final votes for this post as (*post_k*, 6). The root node will extract this new identified spam post and join this post into the new training dataset, a set of data with identified spam and ham post which is used for creating spam models. After the default batch size, the root node will generate a new dataset consisting of latest spam posts and then periodically create a new spam model upon this dataset. After that, the tree root will disseminate the newly trained model to all following nodes in the continuing processing. Besides, when necessary, the tree root can multicast to its nodes within the group, to notify them to empty their sliding windows and/or synchronously start a new batch [XHL+18].

As shown in Figure 4.4, after the date processing in the tree-level, the *external tunnel* ensures multiple groups' roots exchanging and sharing the latest spam posts at the runtime, which means that each group can leverage spam correlation to enhance its own processing and get better performance. As mentioned earlier, in order to allow distributed data servers to obtain a global view of spam information, SpamHunter allows the root of the group to send its aggregated spam to the roots of other peer groups. Details of data delivery among groups can be seen in Section 4.2.2. Each group's root can periodically update and exchange its extracted spam data with other groups.

SpamHunter ensures the entire spam detection in flexible processing granularity, including both globally large-scale data processing and locally distributed data processing. To support multiple processing targets, such as high latency sensitivity or good failure recovery, the functional tree is able to self-tune at the tree structure level by adjusting the tree fan-out element n, with achieving 2^n fan-outs per node.

Specifically, when the latency is the prime target of the users' defined applications, SpamHunter can customize the depth of functional tree by adjusting the fan-out element n. For instance, when 10^b (i.e., b = 4) nodes exist in the system, the original depth of the functional tree is $log_{2^b}(10^b)$. By adjusting the fan-out element from 4 (2^4) to 5 (2^5), the average depth of the tree can be pruned from 5 to 4. Consequently, the overall latency of root-to-leaf transmission will obtain 20% decrement.

In another case, when the application is defined to require a strong failure recovery, SpamHunter can tune to construct deeper functional trees to ensure this. Using the same example above mentioned, SpamHunter can change the fan-out parameter from 5 (2^5) to 4 (2^4), resulting in that the average tree's depth increases from 4 to 5. A deeper tree can achieve more robust performance when multiple nodes' fail. This depends on the mechanism for failure recovery in SpamHunter: once a child fails to receive a heartbeat message, it suspects its parent failed, and this node will route the JOIN message towards its belonging groupId. SpamHunter then sends the message to a new parent to repair the tree. When a tree has a small fan-out and a large depth, the failure of one node can only affect the performances of the following sub-agents, which fewer sub-agents (smaller fan-out) can reduce the deficiency due to parent node's failure.

4.2.3 Online Social Spam Detection

The SpamHunter leaf node is responsible for two functionality: (1) social raw log collection and normalization and (2) local spam detection. Each leaf node collects the social network logs (i.e., social posts, images, news, Tweets, and so on) from the distributed web servers. The leaf node can utilize the openly APIs (e.g., Twitter API, Facebook API) to collect the online/real-time streaming social logs. The logs will be collected from scripts and saved to the local server, which can be utilized next by the

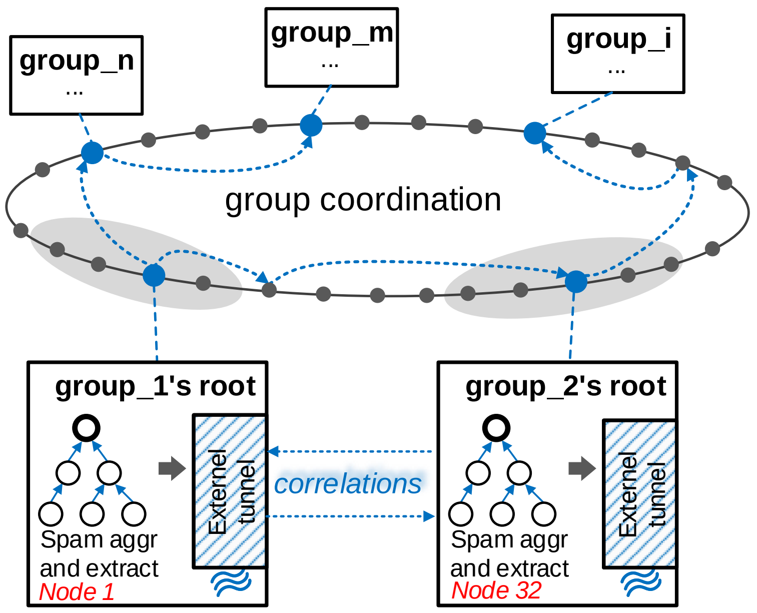


Figure 4.4: The group coordination in the system. Group_1's root shares the spam correlations with group_2's root. Besides, the group_1's root communicates with other groups' roots via the overlay.

leaf node that connected to this server. Furthermore, the leaf node will pre-process and normalize the raw social logs into the same formatted separate set. The majority of posts contain URLs, typically, to confuse the malicious URLs, spammers will add white spaces and unicode characters into them [GHW+10]. This is a simple but effective way to bypass the filters that blacklist URLs only by simple string matching. Inspired by [GHW+10], we de-confuse URLs by removing whitespace padding and normalizing the encoded characters (e.g., "subsexvideo%26ip%3Dauto%26click%3D1" becomes "subsexvideo&ip=auto&click=1"). For social contents, specifically, we remove punctuation, tokenize each word, and remove stopwords. We extract the tf-idf values of the terms in each document. The tf-idf weight of the term represents the frequency at which the term appears throughout the document [ZHC07].

After the data collection, SpamHunter leaf node will first normalize the original data into unified formats. The SpamHunter leaf node extracts the posts' contents from the JSON formatted log. Then it divides these social data into same sized datasets for the local online spam detection. Next, the leaf node will utilize the trained spam model which is disseminated from SpamHunter root to complete the local data processing task. Original data which consists of unprocessed social logs without identified labels. After the spam classification with the trained model, an identified label will be created to each instance of the original social logs (here 1 presents spam and 0 presents non-spam). Besides, the SpamHunter leaf node will facilitate completing the results aggregation flow by sending intermediate results to the upper layers. Note that the leaf nodes will instantly process the collected social logs without long latency. Besides, they will follow the root's instructions to clean its slides and start new batches with the updated spam model.

The online social spam detection is completed by coordinating both tree-level and group-level. The tree-level processing has been presented before, we next introduce the group-level coordination in online processing, which primarily relies on group communications.

4.2.4 Group Coordination

The group communications in SpamHunter are responsible for the main function of spam correlated model update and data exchange among the entire detection. We fulfill the group communication by implementing diffusion broadcasting group. We now present the details of this functional component. SpamHunter group provides two major functions: *multicast* and *anycast*. *Multicast* is used to construct a hierarchical functional tree, which acts as a fundamental frame for scalability in SpamHunter. *multicast* allows messages or instructions can be delivered to all the members in one group. As presented before, any nodes can create a group in the overlay; and other nodes can flexibly join the group and then *multicast* messages from the rendezvous point to all member of the group along the functional tree.

Anycast can be used for group communications and model transmissions among multiple groups. It is implemented by the distributed depth-first search (DFS). Each node in the overlay (may in/out of group k) can *anycast* to the group k by routing a message towards the group k's groupId [RD01]. The convergence of local routing in Pastry guarantees that this message can highly reach a group member near the sender's nodeId. *Anycast* can also be used to serve the communications between multiple groups, such as exchanging the updated dataset and exploring the spam correlations among them.

SpamHunter supports group-level communications to allow multiple groups to exchange their updated models to enhance the final performances. Once a group finishes its whole processing in the leaf nodes, the root node aggregates the results that contain the newest spam information and then updates its model. Further, root nodes in groups exchange the updated models by disseminating their updated models to other peer-groups. Then all groups own the newest models from other groups and can utilize the new models in the continuing processing.

SpamHunter originally supports star group that allows each root of one group *anycasts* the updated model to other groups. Given a graph G with N nodes, one root needs to send $n - 1$ messages during one round time. In this case, SpamHunter group has to send out $m = 1/2 \times n \times (n-1)$ messages which takes $O(n^2)$ time. To diminish the group communication latency, we design diffusion group in SpamHunter. We now present the details of this type of group communication.

The diffusion group communication lies in: each root node in a group holds a table where contains the model version and the original group, which denotes as a <*groupID*, *versionNum*>. The root within updated model in one group randomly chooses two other groups to disseminate the model with the new version number. The root of these two groups will check its model table to see if the current received model is the latest one. If the *versionNum* is larger than the value in the table, they will save the model and update the model table with the new version number. If not, they will return a message to the original group to notify they already own the newest one. These two roots will act as new propagators and begin to deliver the latest model to other groups. Finally, all groups' roots will receive the updated model and update their model tables. It's easy to refer that the number of rounds to propagate a single update model to all groups is $O(logn)$, where n is the number of groups in SpamHunter.

4.3 Evaluation

The experimental evaluation of the system is carried out with online real-world streaming data from social media. We utilize the data which is collected from Twitter streaming APIs [XHL+18]. We manually labeled the dataset for examining the performance of spam detection. These data were labeled based on the posts' URL, content, and Twitter official identifications. The dataset contains 60,000 posts which including of 43,897 ham posts and 26,100 spam posts. The application's purpose is to identify social spam posts, which produces predicted labels from online data streams via the system. Note that near 1'000,000 posts are used for evaluating the scalability of the system.

Experiments are conducted on a testbed of 10,000 nodes hosted by 10 servers. Each server has a 3.4GHz CPU, 4G of memory and 30 GB hard drives. Our evaluations mainly answer the following questions:

- What are the performances of system metrics in the scale, such as the delivery and aggregation latency, functional tree construction latency, and runtime overhead (Sec. 4.3.1)?
- What are the performances of group coordination and online spam detection by utilizing the spam correlations (Sec. 4.3.2)?

4.3.1 System Performances with Scalability

SpamHunter achieves effective spam detection in large-scale online social data sources, therefore, scalability is a major part of the overall evaluation. To evaluate SpamHunter, we deploy the system with the nodes ranging from 1,000 to 10,000, which consists of ten groups (functional trees) in the cluster of servers.

SpamHunter implements functional tree to complete the local distributed social spam detection, which means that the functionality of the tree directly affects the final performance of the process. We first look at the tree paths in the group. The functional tree responses for the message routing, delivery, and communications between multiple layers of nodes, therefore, the average hops (steps) among node communication should affect the performances within scaling to a large amount of data sources. The evaluated average hops are shown in Figure 4.5. From this figure, we can see that the average hops between multiple layers of nodes are consistent when the system scales to large amount of nodes. The typical hops among the functional tree are around 2 to 3. This demonstrates that SpamHunter can flexibly and conveniently support large-scale data sources and servers, and can guarantee the total communication between servers/instances in a relatively small distance, which indicates the low latency in handling node interactions and communications.

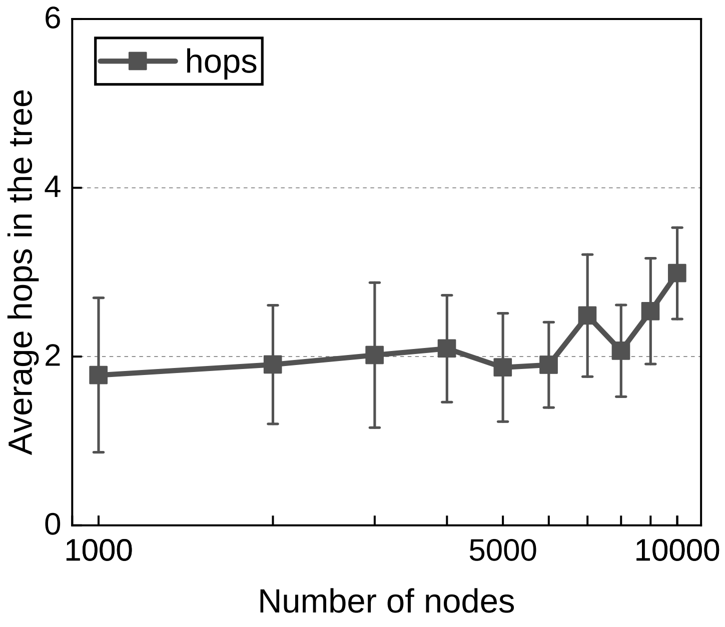


Figure 4.5: Model delivery latency.

The communication latency among SpamHunter is mainly from two parts: the delivery latency from the root and the aggregations latency from the leaf. The delivery latency refers to the root node of the functional tree disseminating the messages, data, spam model, and instructions to all following nodes. The aggregation latency generally refers to the root node aggregates the interim results (i.e., identified social spam posts) from the leaf nodes. The results of these two kinds of latency are shown in the Figure 4.6. From this figure, we can observe that when scaling the nodes to a large scale (up to 10,000), the latency is slowly increased with a few hundreds of milliseconds. This is reasonable since a large amount of nodes will cause part of delay in the message delivery and results aggregation. The difference between these two kinds of latency is usually from the delivered data size, for example, the delivered spam model is up to several megabytes, which causes the delivery latency is higher than the result aggregation latency.

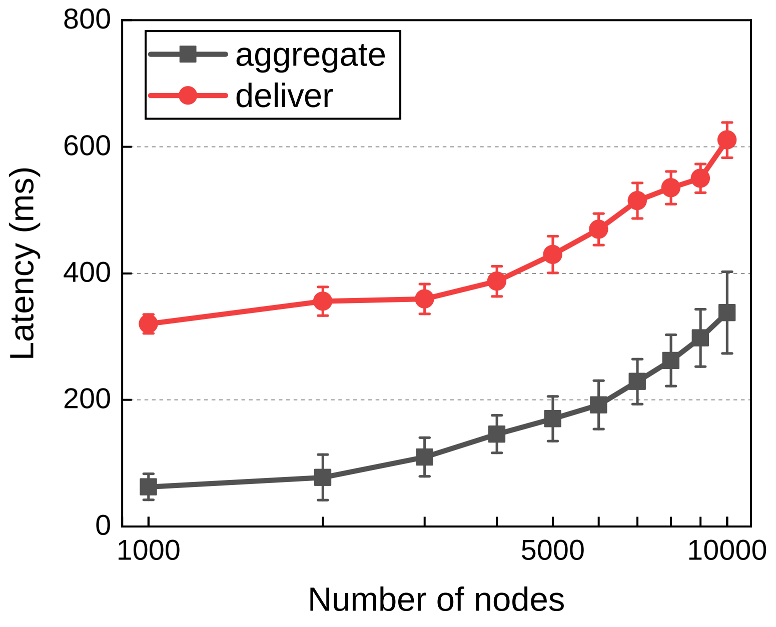


Figure 4.6: Latency of deliver and aggregate.

Figure 4.7: Results of spam detection.

Model	F1	Precision	Recall
RF	0.951	0.951	0.951
SVM	0.942	0.945	0.944
RT	0.927	0.928	0.927
Logistic	0.859	0.866	0.855

Moreover, we evaluate the latency in constructing functional trees with a different number of nodes. As shown in Figure 4.8, when the nodes scale from 1,000 to 10,000 (note that these are ten trees here, for each tree, the nodes scale from 100 to 1,000), the construction latency is linearly increased with the nodes' increment. The latency is usually from the hash of nodeId and the joining of the overlay. Note that the functional tree only needs to be built once at the beginning, and it will not cause other latency during the data processing.

Further, we evaluate the runtime overhead of the system in deploying the functionality. Results of the CPU and memory utilization are shown in Figure 4.9. The values of utilization present the overhead in one server and here they leave out the

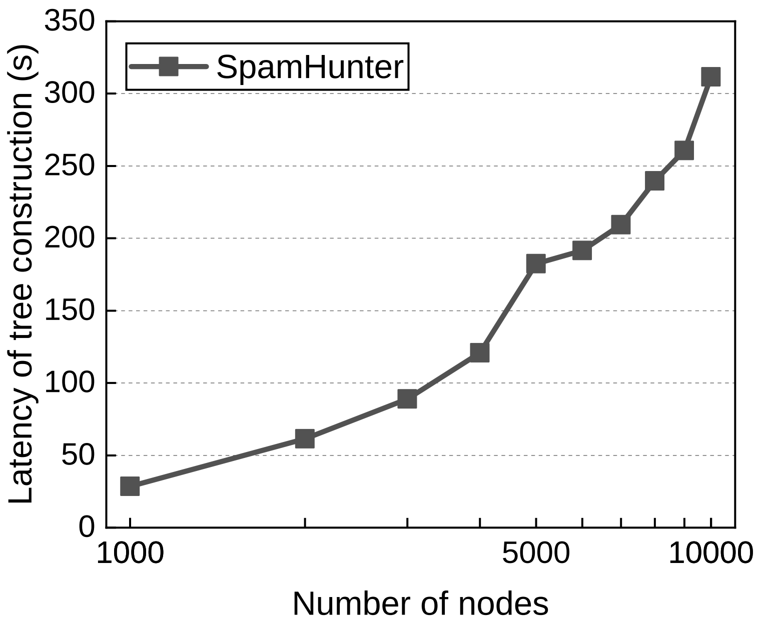


Figure 4.8: Average latency in creating functional tree with different nodes.

processing of spam detection since the data processing will periodically cost lots of computations and will make the overhead confusing in evaluating the functional tree's performances. From this figure, we can see that with the number of nodes scales to 10,000, the runtime overhead linearly increases by 16% in CPU and 7% in memory. It presents that SpamHunter achieves relatively lightweight overhead in guaranteeing the tree and group functions at runtime. And it can be beneficial for the large-scale data processing in the future.

4.3.2 Group Coordination and Spam Detection

SpamHunter supports multiple groups to exchange and share the latest spam posts with each other. In this subsection, we present the experimental results of group coordination and the performance of spam detection.

Figure 4.10 shows the latency of communication among multiple groups' roots. Here we use two sizes of dataset, with 1,000 and 5,000 posts separately. From

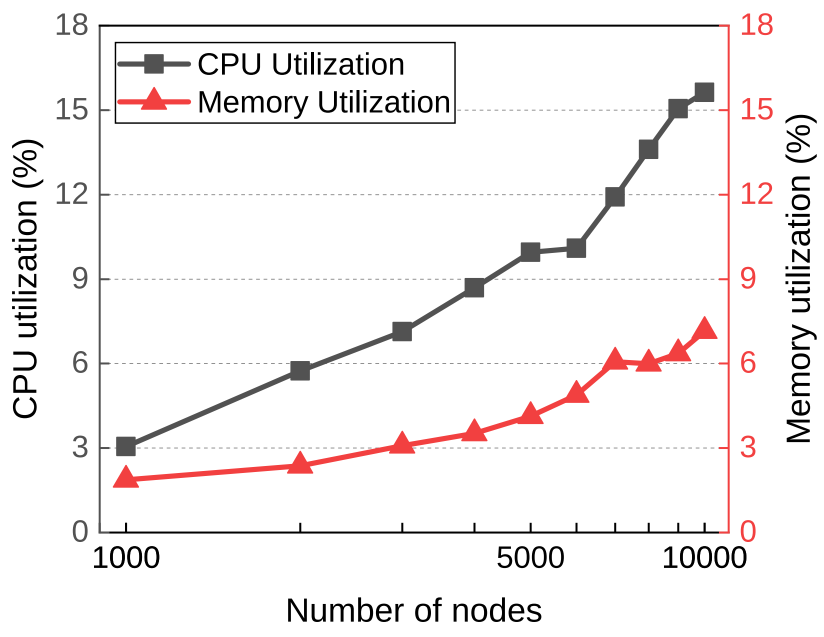


Figure 4.9: RUntime overhead of CPU and memory utilization.

the figure, we can see that the latency of communication is consistent with the increment of groups from 10 to 100. When deployed with a large number of groups, for instance, 100 groups, the average latency has linear increment. In general, the latency mainly depends on the size of the delivered data. When the root shares a large size of spam posts, it will incur longer latency.

Table 4.7 presents the performances of spam detection in SpamHunter. We implement several classical algorithms such like RF (Random Forest), SVM (Support Vector Machine), RT (Random Tree), and Logistic in the detection with the labeled dataset. We present the major parameters of the performance including F1, Precision, and Recall, where the F1 score responses for an important factor in measuring the performance. From the table we can see the Random Forest (RF) achieves the best performance with the F1 score near 95%. This presents that SpamHunter can achieve good performances in dealing with the online real-world social spam data.

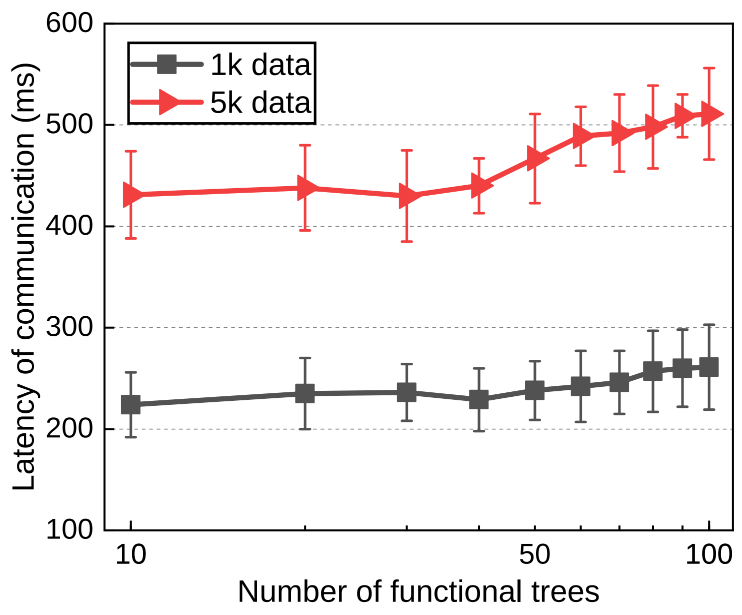


Figure 4.10: Group communication latency.

4.4 Summary

Social spam has become an inevitable part of the current social world. Various garbage activities surround people and cause huge negative impacts on both virtual and real life. In this chapter, we present an online social spam detection system, named SpamHunter [XHLG19], which leverages the spam correlations among large-scale distributed data sources to enable efficient spam detection in a scalable manner. SpamHunter supports multiple groups to manage social data from various topics, areas, and geo-location. Each group forms a functional tree that guarantees flexible management across a large number of data servers/instances. Moreover, group coordination in SpamHunter allows multiple groups to exchange and share spam correlations from distributed data sources, enabling efficient processing with the latest social spam from online data streams.

CHAPTER 5

CUSTMOIZABLE STATE RECOVERY FOR SOCAIL STREAM DATA PROCESSING

5.1 Introduction

Stream processing (e.g., social spam detection) is proposed and popularized as a "technology like Hadoop but can give you results faster" [Per18], which lets users query a continuous data stream and get results shortly after receiving the data. Stream processing technology has become a critical building block of many scientific applications, such as predicting tornadoes and storms from radar streams, real-time imaging of cement hardening from x-ray beam data [BGK+17], and analyzing nanometer-scale dynamics of materials using x-ray photon correlation spectroscopy [Sut16]. Upcoming frameworks to accelerated discovery in material sciences [TBH+19] will require distributed stream services in their workflows.

While in the early days stream operators were used for simple computations which are *stateless*, such as `filter`, `sort`, today's stream operators are capable of powering more complex computations and evaluating more complex scientific logic which are *stateful*, such as `mapWithState`. This requires today's stream processing systems to offer "*state handling*" – i.e., operators that can remember past input and use it to influence the processing of upcoming input.

However, stream processing applications may be highly dynamic due to factors such as variable data rates, network congestions, and application-specific data source characteristics. Stream processing applications are also often subject to instabilities and failures, where multiple streaming operators may fail at the same time, resulting in severe state loss that may break or hinder the progress of scientific application workflows.

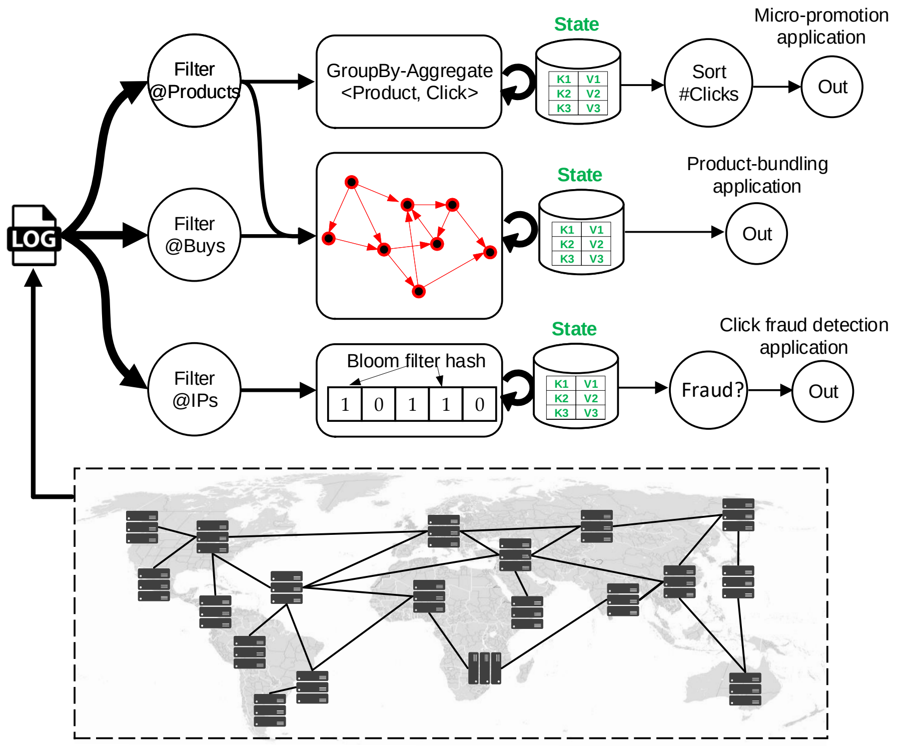


Figure 5.1: Real-world examples of stateful stream processing.

In this paper, we explore customizable state recovery mechanisms for protecting large distributed states in stream processing systems, in order to cater the needs for different stream processing applications that have different stream processing computation models, state sizes, and network environments.

Figure 5.1 shows the real-world examples of stateful stream processing. When we are shopping at e-commerce websites, our user activities (e.g., *clicks*, *likes*, *buys*, *reviews*) are going to be continuously logged by these sites. On the backend, many stateful stream applications are concurrently running on top of these user activity streams to create insights and make business decisions. For example, Figure 5.1 top is a "*micro-promotion*" application, which analyzes the live page views of its products, `groupby-aggregates` them, and then *sorts* them to find the top-k products with the most clicks to apply discount. Here the "state" is the stored knowledge base of key-value pairs consisting of product names and corresponding clicks. Figure 5.1 middle is a "*product-bundling*" application, which extracts users' buys, creates graphs of vertices and edges to get an idea of what products are usually purchased together, then makes online recommendations such as "*you like this, you may also like that*". Here the "state" is the stored knowledge base of connected graphs consisting of product names and bundlings. Figure 5.1 bottom is a "*click fraud-detection*" application, which identifies ad clicks as fraudulent by deploying a space-efficient probabilistic data structure like a Bloom filter [ag] to memorize the IP addresses or the cookies of previous clicks, and comparing them to the new coming click stream to detect duplicate clicks in a short time. Here the "state" is the stored knowledge base in the Bloom filter.

However, we are facing significant challenges in managing these large distributed states in stream processing systems.

- *Challenge 1: how to recover from simultaneous failures of multiple stream operators for a large number of concurrently running applications?* Streaming computations are, by nature, long-running. Their workloads, as well as the runtime environment, may change in unpredictable ways. A stream computation is usually represented as a logical directed acyclic graph (DAG), where vertices denote operators and edges denote data dependencies between them. This means that if one operator fails and loses state, the dependent operators may also fail and lose their states. What makes it particularly challenging is that many stream processing applications run concurrently on the same HPC infrastructure and consume the same data source. We need to be able to recover lost state for large numbers of concurrently running applications on the same HPC infrastructure.

- *Challenge 2: how to customize the failure recovery mechanism for different types of stream processing applications?* For example, Spark Streaming based systems [aa, CEF$^+$17, ZDL$^+$13, SGH$^+$16] treat streaming computations as a series of batch computations, whereas Storm based systems [ad, af, KBF$^+$15] treat streaming computations as a dataflow graph in which vertices asynchronously process incoming records. The state size for batch applications is usually large, whereas the state size for stream applications are usually small. Some applications run on an HPC infrastructure that has abundant uploading bandwidths, whereas some applications run on an HPC infrastructure that has bandwidth constraints [GAB$^+$15, PACT18]. Therefore, different stateful stream processing applications need different state recovery mechanisms that best meet their needs.

Over the last decade, there has been a boom of stream processing systems, including Storm [ad], Trident [af], Spark Streaming [am], Borealis [AAB$^+$05], TimeStream [QHS$^+$13], and S4 [NRNK10]. However, there is a lack of fast and scalable failure recovery mechanisms for protecting the large distributed states for these systems. The reasons are as follows: (1) they mostly inherit MapReduce's "single master/many workers" architecture, where the central master is responsible for all scheduling activities. As such, they do not scale well to a large number of concurrently running applications due to the inherent centralization bottleneck; (2) these systems offer failure recovery mainly through three approaches: replication recovery [SHB04, BBMS05], checkpointing recovery [ad, af, QHS$^+$13] and DStream-based lineage recovery [aa, CEF$^+$17, ZDL$^+$13, SGH$^+$16], which are either slow, resource-expensive or fail to handle multiple simultaneous failures. Replication recovery adds high hardware cost because multiple copies must concurrently run on distinct nodes for failover. In distributed streaming, checkpointing recovery is known to be prohibitively expensive, leading users in many domains to disable this feature [MMI$^+$13, ABB$^+$16, PD10, PLGC15, GXD$^+$14]. The third approach, DStream-based lineage recovery, is slow when the lineage graph is long (i.e., the streaming involves long sequences of operators) and falls short in handling multiple simultaneously failures; and (3) these systems are limited to a fixed computation model, e.g., asynchronous stream processing like Storm [ad], synchronous mini-batch processing like Spark [am], and they do not have customizable state recovery mechanisms.

5.2 Problem Statement

We follow a generic stream query model [ABC$^+$15b, CKE$^+$15, LFQ$^+$16, MMI$^+$13, ZDL$^+$13]. A stream processing application's query is a directed acyclic graph (DAG) that specifies the dataflow, denoted as $Q = (V, E)$. DAGs can be implemented via many execution models, such as the partition/aggregate model which scales out by partitioning tasks into many sub-tasks (e.g., Dryad [IBY$^+$07]), the sequential/dependent model in which streams are processed sequentially and subsequent streams depend on the results of previous ones (e.g., Storm [ad]), and the hybrid model with sequential/dependent and partition/aggregate components (e.g., Spark Streaming [am], Naiad [MMI$^+$13]).

A vertex $v \in V$ corresponds to a stream operator f_v that consumes input streams i from its predecessor (upstream) vertices and produces output streams o to its successor (downstream) vertices ($o = f_v$(i)). Each edge $e \in E$ represents a data flow between two vertices. The stream operator f_v can be *stateless* or *stateful*. A stateless operator consumes one input record at a time and outputs each result based solely on that last input record. A stateful operator maintains state that captures characteristics of some of the records processed so far and updates it with each new input, such that the output takes into account both historical records and the new input. Stateless operators are easy to recover because, by definition, input records are handled independently, and upon failure we can simply start a new operator instance. In contrast, stateful operators are much more difficult to recover.

The problem is: *how to achieve a scalable and fast failure recovery framework that protects large distributed states for concurrently running applications deploying diverse execution models?* These applications run concurrently on a shared distributed environment. Their operators are stateful. The applications comprise several DAGs, deploy diverse execution models, and vary on their requirements of CPU, memory, and network bandwidth.

5.3 Background

For maintaining and recovering state, our solution leverages peer-to-peer (P2P) overlay networks, more specifically, the Distributed Hashtable(DHT)-based consistent ring overlay with routing. The primary purpose of the P2P model (e.g., Pastry [RD01], Chord [SMK$^+$01]) is to enable all nodes to work collaboratively to deliver a specific service. In such model, all nodes have similar roles, both serving and requesting services. For example, in BitTorrent [Coh03], if someone downloads some file, the file is downloaded to her computer in bits and parts that come from many other computers in the system that already have that file. At the same time, the file is also sent (uploaded) from her computer to others who ask for it. Similarly to BitTorrent, where many machines work collaboratively to download and upload files, our solution enables all distributed nodes to work collaboratively to achieve state management, relieving the task scheduler (often implemented as a centralized service) from handling state. For this purpose, we leverage the following three data structures from DHT-based consistent ring overlays:

- *Routing table*: The routing table consists of node characteristics (*IP address*, *Node Id*) organized in rows by the length of common prefixes in the representation of a *Node Id*. When routing a message to `nodeId`, a node forwards it to the node in its routing table with the longest prefix in common with `nodeId`. State are associated with keys and nodes are responsible for a range of keys. In a system where N nodes store state, it is guaranteed that queries can be routed to the appropriate `nodeId` within $O(logN)$ hops. We use the routing table for locating state and in the line-structured recovery mechanism (Section 5.4.3).
- *Leaf set*: The leaf set for a node is a fixed number of nodes that have the numerically closest `nodeIds` to that node. This assists nodes in routing messages and in rebuilding routing tables when nodes fail. We use the leaf set for the star-structured recovery mechanism (Section 5.4.2).
- *Multicast*: Any node in the overlay can create a communication group; other nodes can join the group and then multicast message to all members of the group. Multicast messages are disseminated through a multicast tree. We use multicast for constructing in the tree-structured recovery (Section 5.4.4).

5.4 Design

In this section, we introduce the SR3 framework, which includes the system overview (Sec. 5.4.1), the star-structured recovery mechanism (Sec. 5.4.2), the line-structured recovery mechanism (Sec. 5.4.3), the tree-structured recovery mechanism (Sec. 5.4.4), and how SR3 determines which mechanism to use (Sec. 5.4.5).

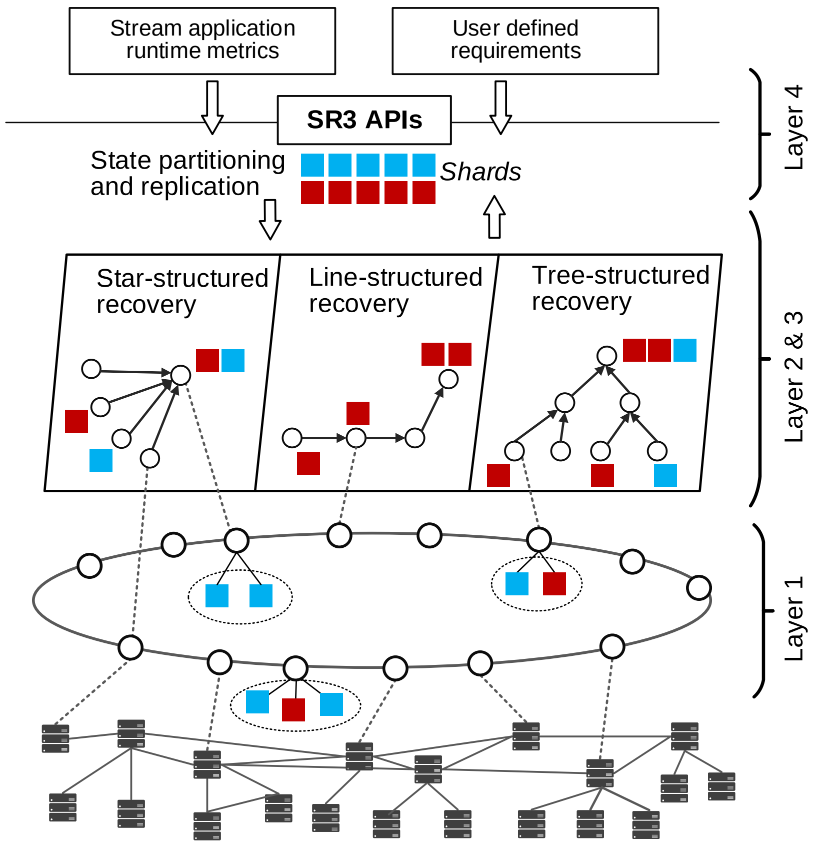


Figure 5.2: The overview of SR3 design.

5.4.1 The SR3 Overview

Figure 5.2 shows the overview of the SR3 system. It consists of several layers as follows.

Layer 1: DHT-based overlay. In our system, we introduce a new abstract concept called "node" to facilitate state management. Each stream operator is associated with a node. The association is unrelated to where operators execute; operators at the same vertex may be associated with different nodes. Each node is randomly assigned a globally unique identifier known as the "`nodeId`" in a large circular node ID space (e.g., 0-2^{128}). We organize these nodes into a P2P overlay network. The overlay is self-organizing and self-repairing.

Layer 2: State partitioning and replication. The node's state is stored in an in-memory hashtable data structure. Periodically, we divide each node's state into m shards, each of which is then replicated to n replicas and distributed to peer nodes. The peer nodes are preferably chosen as to enable high bandwidth communication. The parameters of m and n are determined by the adopted recovery mechanism (we offer three alternatives) and application characteristics. Our design ensures that when a failure happens, different sets of available shards can reconstruct failed state in parallel, thereby reducing the failure recovery time while tolerate multiple simultaneous node failures.

Layer 3: State recovery. Applications differ in state sizes, execution models and QoS requirements such as latency and throughput. Some applications, such as simulations that can adjust to data errors, can tolerate lower accuracy in exchange for efficiency in accessing state and quick recovery other tasks, such as state visualization for application debugging, cannot. We design three state recovery mechanisms to satisfy the needs from different applications. SR3 tracks user-defined requirements (e.g., latency sensitivity) and the application's characteristics (e.g., size of the state) to select the most appropriate mechanism (see Sec. 5.4.2, Sec. 5.4.3 and Sec. 5.4.4 for more details).

Layer 4: SR3 API. SR3 is currently integrated into Apache Storm [ad]. We provide a high-level API that exposes to users configuration parameters and enables SR3's portability to other stream processing systems.

5.4.2 The Star-structured Recovery Mechanism

Figure 5.3 shows a straightforward implementation of star-structured recovery mechanism. Each node has a routing table and a leaf set. In this example, the state of

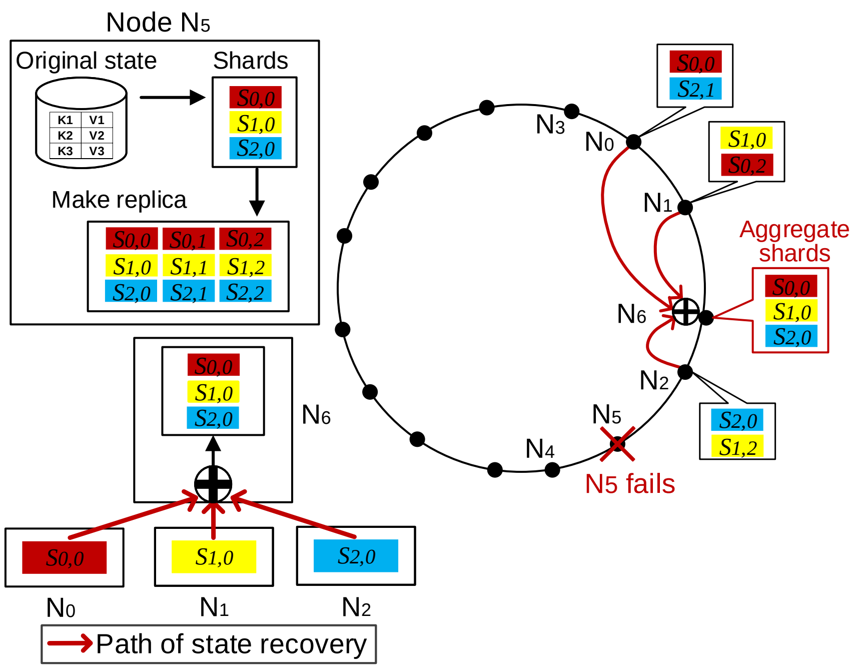


Figure 5.3: The star-structured recovery process.

node N5 is divided into 3 shards and each shard has two replicas. They are distributed to the leaf set nodes to ensure that the original state can be reconstructed from 3 shards of the 9 total shards. As shown in Figure 5.3, the nine shards $s_{0,0}$, $s_{0,1}$, ..., $s_{2,2}$ are stored in N_0, N_1, ..., N_5 respectively. When N_5 fails, N_0, N_1, and N_2 upload $s_{0,0}$, $s_{1,0}$, and $s_{2,0}$ to N_6 to recompute the state of N_5.

The benefits are: (1) the recovery process is fast. Different nodes from non-overlapping leaf set nodes can work in parallel to recompute the lost state, which is much faster than retrieving the state from the remote storage (e.g., HDFS). (2) We achieve data locality because the leaf set contains nodes that are geographically close to the original node (e.g., within the same rack in the same site) that have abundant upload bandwidth.

5.4.3 The Line-structured Recovery Mechanism

The star-structured recovery works fine when the state is small. However, when the state is large, the replacing node needs to do all the downloading and reconstructing work, suffering a centralized bottleneck that increases the recovery latency, which we aim to avoid. We design the line-structured state recovery to fix this issue, where shards are transmitted and combined through a line covering the replacing node and all providing nodes. As shown in Figure 5.4, the nine shards $s_{0,0}$, $s_{0,1}$, ..., $s_{2,2}$ are stored in N_0, N_1, ..., N_5 respectively. When N_5 fails, N_3 uploads $s_{2,0}$ to N_0. N_0 merges $s_{2,0}$ with $s_{1,0}$, reconstructs it, and then uploads the result to N_1. N_1 merges the result with $s_{0,0}$, reconstructs it, and uploads the final result to N_6 to replace of N_5. The benefit is that, the downloading and computing load are well balanced among all involved nodes which helps recover large state. However, it can only recover one node at a time. When recovering multiple node failures, it may incur multiple times of network traffic and recovery time. Besides, the line-structured recovery disregards the bandwidth asymmetry in cloud environment.

5.4.4 The Tree-structured Recovery Mechanism

We design a shard-based parallel recovery mechanism to tolerate multiple node failures, where shards are transmitted and combined through a spanning tree covering the replacing node and all providing nodes. This spanning tree is built on top of a scalable application-level multicast infrastructure, called Scribe [CDKR02]. The key idea is to divide the state into many shards (e.g., based on key ranges), and use different sets of available replicas of shards scattered across leaf set nodes to reconstruct unavailable shards in parallel. By doing this, all nodes storing available shards can work as providing nodes and each of them only needs to participate in

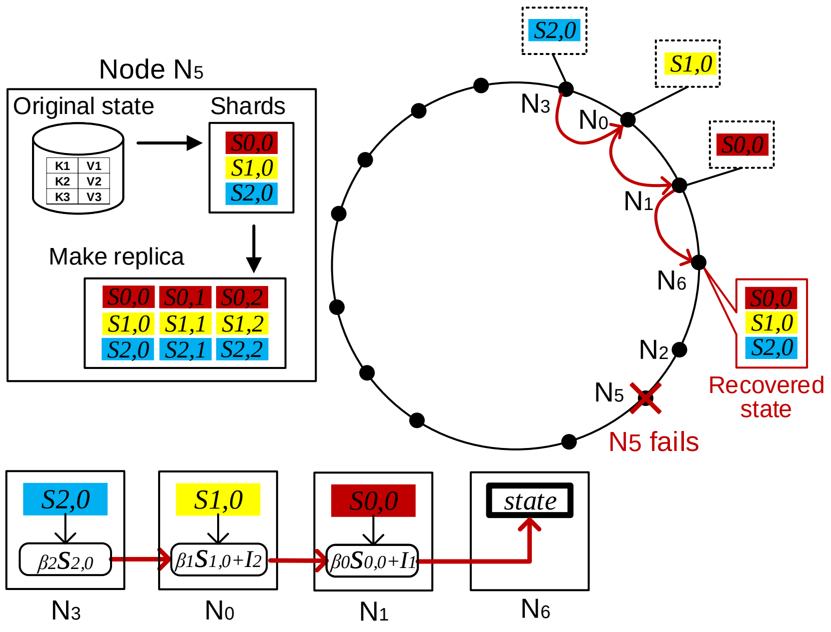


Figure 5.4: The line-structured recovery process.

the recovery of some unavailable shards. This means a providing node only needs to upload some of the shards it stores. Thus, the amount of data a providing node uploads is reduced in a way that respects bandwidth asymmetry. The downloading and computing load are well balanced among all involved nodes without any centralized bottleneck.

Figure 5.5 & Figure 5.6 show the recovery process from a single failure and two failures in the tree-structured mechanism. N_6 and N_7 are the replacing nodes for recovering the state when N_4 and N_5 fail. We can see that the state is divided into 3 shards, s_0, s_1 and s_2. Each shard is further divided into 3 sub-shards and the replication factor is two. So for one shard, it has total 6 sub-shards. For example, $s_{2,0,1}$ denotes the second replica of the first sub-shard in s_2, and $s_{2,1,0}$ denotes the second replica of the second sub-shard in s_2. In the tree-structured recovery process, the providing node only needs to upload 3 out of the 6 total sub-shards to reconstruct each shard. The recovery from multiple failures is similar with the recovery from a

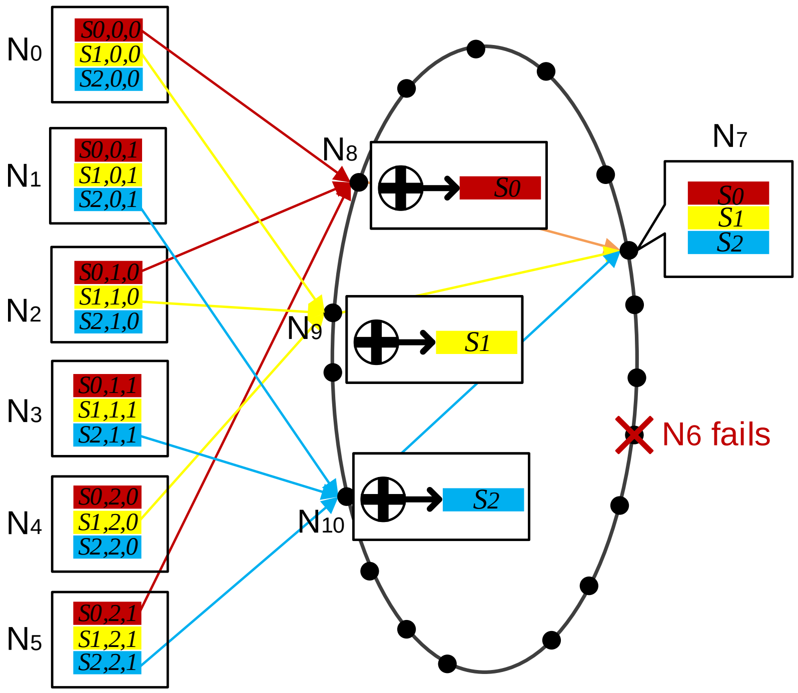


Figure 5.5: The tree-structured recovery process for a single failure.

single failure. The difference is that every reconstructing node needs to reconstruct multiple shards and sends them to multiple replacing nodes.

5.4.5 Mechanism Selection

Which mechanism to use? Determining the optimal state recovery mechanism is difficult since it needs to consider various factors and application specifics. Thus, we rely on a heuristic approach that adapts mechanism based on (1) state sizes, (2) application QoS requirements, (3) network environments (e.g., bandwidth bottleneck), and (4) computation models (e.g., synchronous micro-batch processing model or asynchronous stream processing model) .

Figure 5.7 shows how we determine which mechanism to use. In the case of stateless operator failures, it will simply resume the whole execution pipeline since

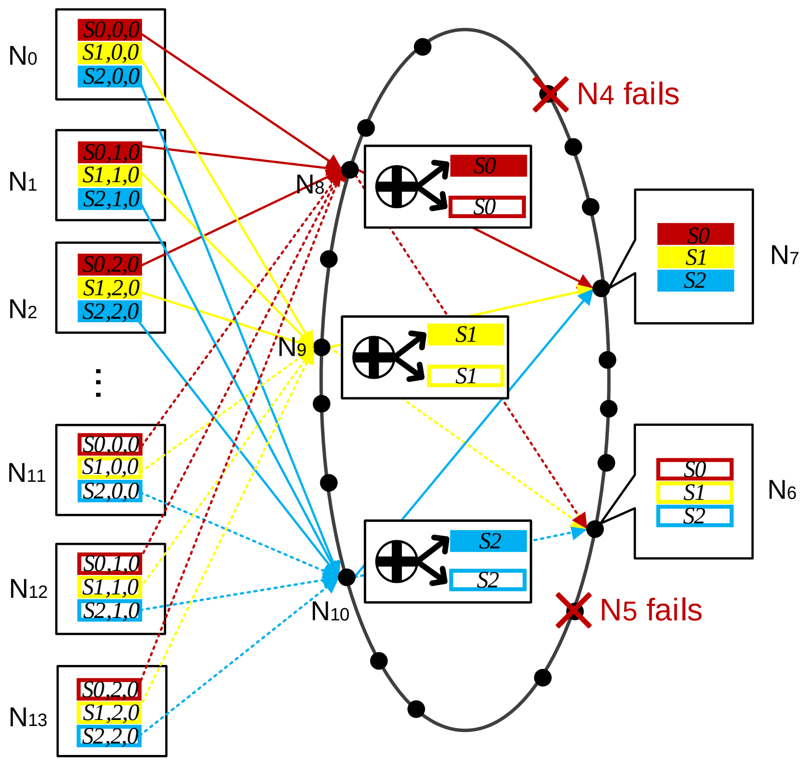


Figure 5.6: The tree-structured recovery process for two failures.

there is no overhead for recovering states. In the case of stateful operator failures, SR3's state recovery mechanisms may not always outperform the traditional checkpointing recovery if the state size is too small or if the application can tolerate the checkpointing overhead of writing state to the remote storage. Thus, we use SR3 only with stateful operators for (1) applications that have strict QoS requirements for low recovery latency and (2) high probability of simultaneous failures that will involve large distributed states.

This information about application's QoS requirements and state size is typically available as part of the job submission information. If the state size is small, we choose star recovery in priority. On the other hand, if the state size is large, we further consider if the execution is constrained by the network bottleneck. In the case of abundant bandwidth, we choose line-structured recovery in priority by

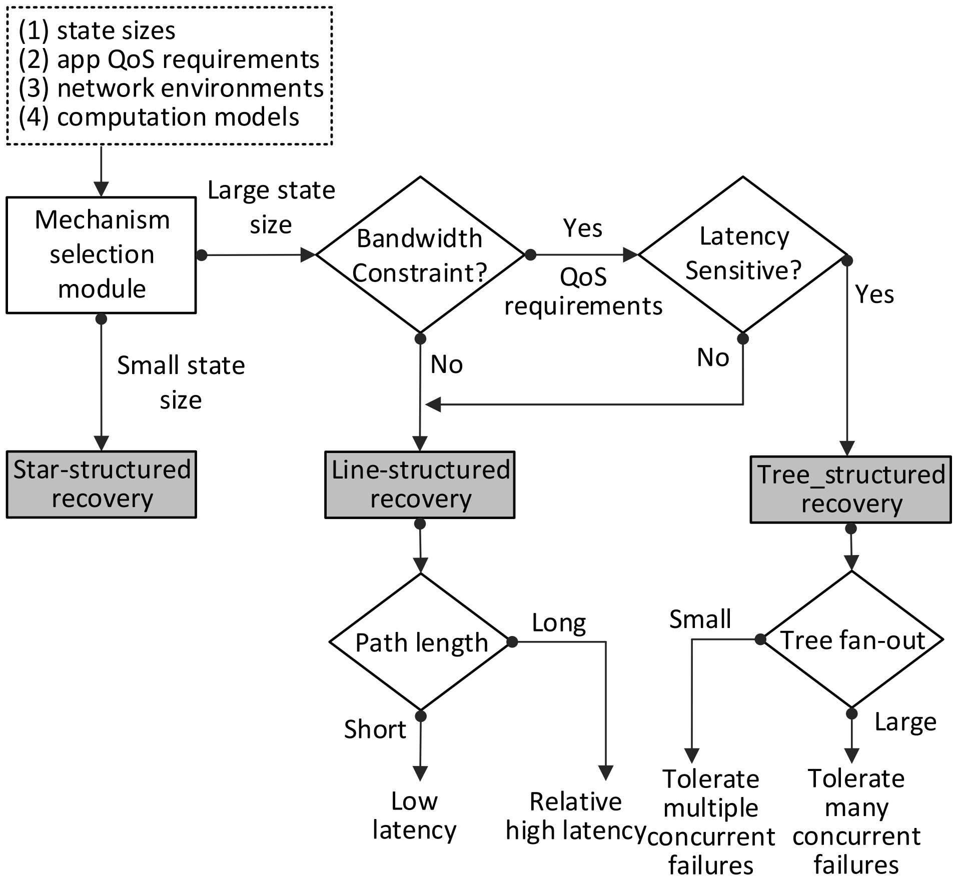


Figure 5.7: Determining which state recovery mechanism.

adjusting the recovery path length to deal with different sizes of state and latency requirements. In the case of limited bandwidth, we further consider application's QoS requirements. If it is latency insensitive, we still choose line-structured recovery in priority. Otherwise, we choose tree-structured recovery in priority by adjusting the tree fan-out and the depth of each branch to deal with different sizes of state, latency requirements, and concurrent failures that occur at the same time.

5.5 Evaluation

We evaluate SR3 on emulation testbed in a distributed network environment. We explore its performance for a variety of real-world stream processing applications. Our evaluation answers the following questions:

- Does SR3 improve the state save and recovery performance when deploying different stream applications with various sizes of states?
- Does SR3 support flexible state recovery in handling various sizes of states with different network environments?
- Does SR3 scale with the number of concurrently running stream applications?
- What is the runtime overhead of SR3?

5.5.1 Setup

Evaluation deployment. Emulation experiments are conducted on a testbed of 50 virtual machines (VMs) running Linux 3.10.0, all connected via Gigabit Ethernet. Each virtual machine has 4 cores and 8GB of RAM, and 60GB disk. Specifically, to evaluate SR3's scalability, we use one JVM to emulate an SR3 node and emulate up to totally 5,000 SR3 nodes in our testbed. Linux VMs are equipped with LANs with high bandwidth diversity set by traffic control.

Baseline. We used Apache Storm [ad] as the stream processing engine baseline. We use Apache Storm 2.0.0 [ae] configured with 10 TaskManagers, each with 4 slots (maximum parallelism per operator = 36). We use Pastry 2.1 [ak] configured with leaf set size of 24 and transport buffer size of 32MB.

Benchmark and applications. To demonstrate generality across diverse computations and streaming operators, we evaluate SR3 in state recovery with the real-

Table 5.1: Real-world application's dataset.

Application	**Dataset**	**Size**
Bargain Index	Google Finance [ai]	>1TB
Word Count	Wikimedia Dumps [an]	9GB
Traffic Monitoring	Dublin Bus Traces [ah]	4GB

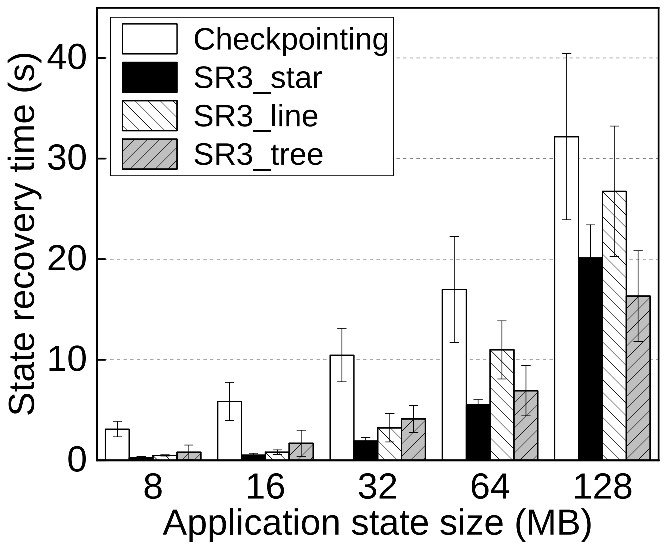


Figure 5.8: The state recovery time by varying the size of state with no bandwidth constraint.

world stream applications (see Table 5.1). These stream applications contain various representative streaming operators: stateless streaming transformations (e.g., `map`, `filter`), stateful operators (e.g., `incremental join`), and various window operators (e.g., sliding window, tumbling window and session window).

We compare SR3 with a state-of-the-art failure recovery solution: the checkpointing recovery approach commonly used in TimeStream [QHS+13], Storm [ad], and Trident [af]. We choose the checkpointing recovery approach as the baseline approach because of two reasons: (1) the replication recovery already costs 2× hardware, and (2) the DStream-based lineage recovery approach is not generalized for users. Because DStream-based lineage recovery sacrifices programming model transparency by forcing programmers to manage state using RDDs [ZCD+12].

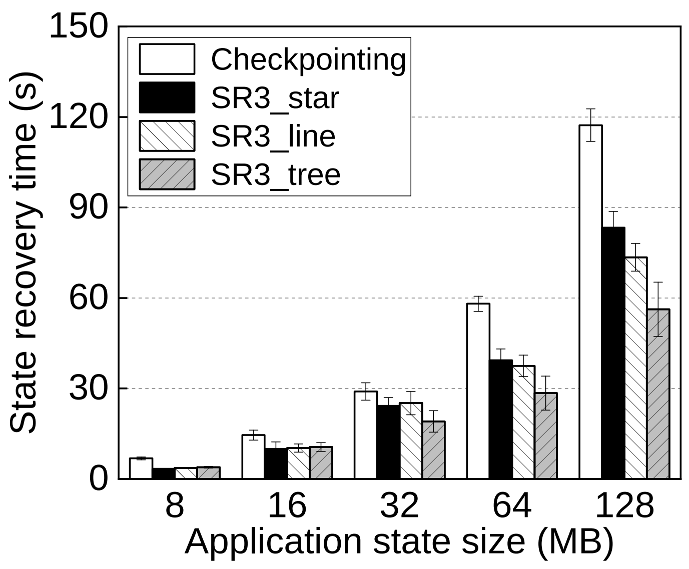


Figure 5.9: The state recovery time by varying the size of state with bandwidth constraint.

Metrics. We focus on the performance metrics such as latency of state save and recovery. The latency measurement is separated by the state save and recovery. The latency is evaluated by deploying different size of state shards and various size of states of stream application. To evaluate the scalability of SR3, we measure how much state shards are distributed in each node with deploying different stream applications. To the runtime overhead of SR3, we focus on the CPU and memory utilizations during the state recovery.

5.5.2 SR3 vs Checkpointing Recovery

We compare the state recovery time of SR3 with Storm by varying the size of state with no bandwidth constraint. As Figure 5.8 shows, SR3 generally achieves 35.5% $\sim$ 65% less state recovery time compared to Storm. More specifically, when a state is relatively small (<32MB), the star-structured recovery mechanism achieves the fastest recovery. Line-structured recovery and tree-structured recovery take a little

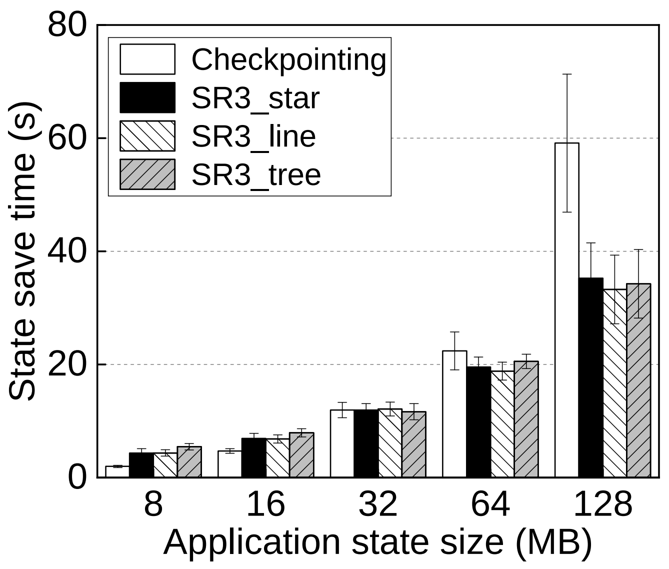


Figure 5.10: State save time by varying the size of state.

longer due to the introduction of redundant calculations in their state recovery paths. When the state grows larger than a threshold (e.g., 64MB), line-structured recovery leads to the longest recovery time due to the longest lineage path. On the contrary, since tree-structured recovery has many paths for recovering at the same time in parallel, the time is reduced.

Figure 5.9 shows the state recovery time comparison of SR3 with Storm under bandwidth constraint. Note that the upload bandwidth is limited to 100Mb/s per server. Results show that SR3 generally achieves 29.8% $\sim$ 42.5% less state recovery time compared to Storm. More specifically, when the state is relatively large ($>$ 64MB), due to the constraint of the upload bandwidth, the star-structured recovery has a centralized bottleneck because all traffic flows to a single node, which leads to the slowest state recovery. On the contrary, the line-structured recovery and tree-structured recovery avoid this bottleneck, and thus are much faster. However, when the state becomes extremely large, the tree-structured recovery performs the best because it has many paths to recover state at the same time in parallel. This gives

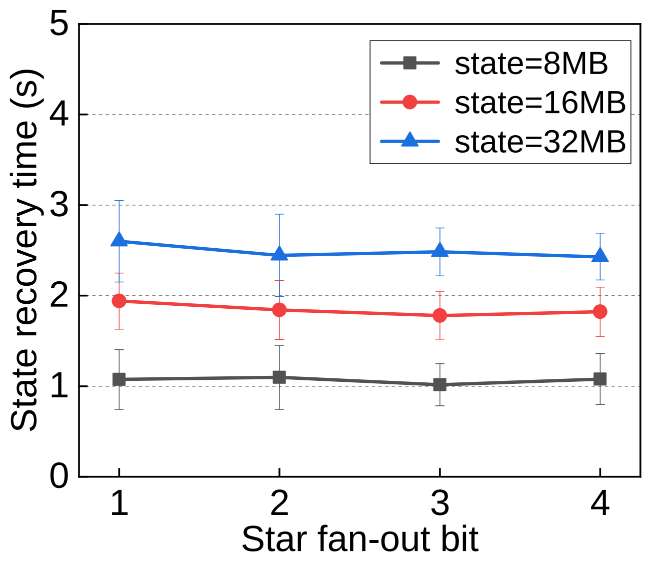


Figure 5.11: The state recovery time by varying star fan-out bit in SR3 star-structured recovery.

us insight that we should decide which mechanism to use based on the application characteristics, the network environment, and the size of state.

Figure 5.10 shows the state saving time comparison of SR3 with Storm. The state saving cost includes the time cost for dividing the state into shards, replicating each state, and then writing the shards into leaf set nodes. We write them into the leaf set nodes serially to enable a fair comparison with the checkpointing recovery. We can see that for a small state (<64MB), it takes more time for SR3 to save the state, while for large state (>64MB), it takes less time for SR3 to save the state. This is because, for small state, the overhead of partitioning and replication is not negligible compared to the total time. However, in the case of large state, many nodes in the leaf set take part in the partitioning and replication that balance the workload.

Figure 5.11 shows the state recovery time by varying star fan-out bit in SR3 star-structured recovery. Results show that the state recovery time does not change much as the star fan-out changes. This is because the depth of the star structure

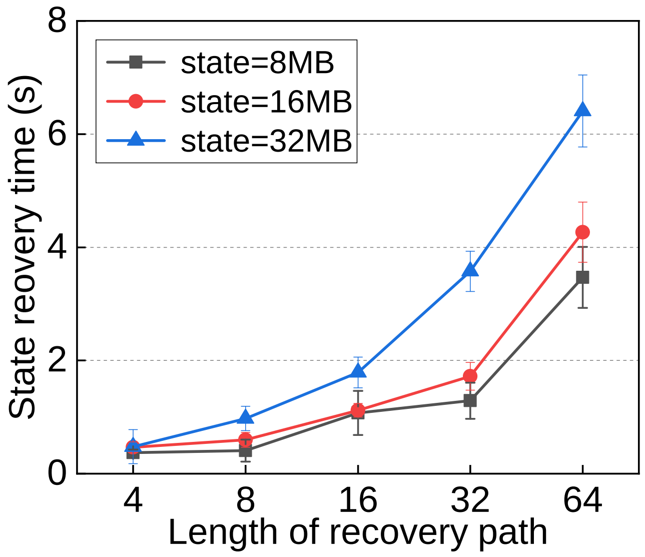


Figure 5.12: The state recovery time by varying the path length in the SR3 line-structured recovery.

always equals to one and thus the latency is only related to the state size and the transmission speed. However, in extreme cases, e.g., very large state size, increasing fan-out will share the pressure on bandwidth and significantly reduce latency.

Figure 5.12 shows the state recovery time by varying the path length in the SR3 line-structured recovery. Results show that the state recovery time increases as the path length increases. This is because the longer the path, the more stages of the computation required, and the higher the latency. However, when the state is too large to be finished within one or two stages, we need a longer path that has many stages to distribute the computation.

Figure 5.13 shows the state recovery time by varying the branch length in SR3 tree-structured recovery. Similar to Figure 5.12, given the same state size, the state recovery time increases as the branch length increases. This is because the longer the branch, the more stages of the computation required, and the higher the latency.

Figure 5.14 shows the state recovery time by varying the tree fan-out in SR3 tree-structured recovery. Note that the tree fan-out n determines the fan-out of each node

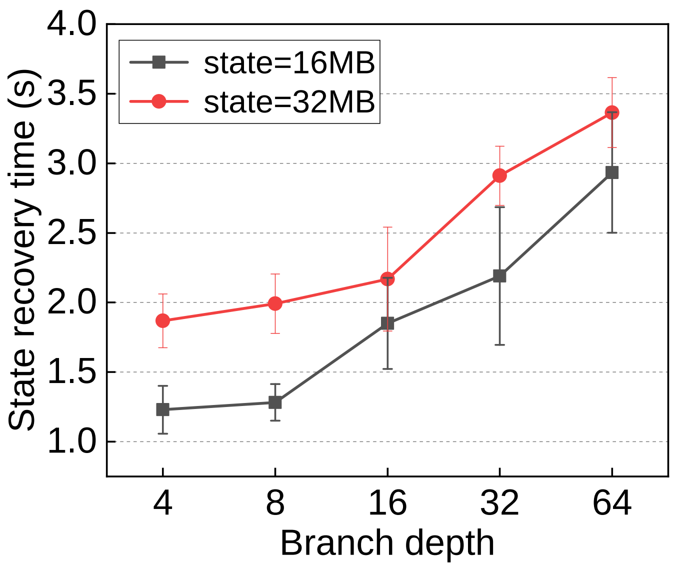


Figure 5.13: The state recovery time by varying the branch depth in SR3 tree-structured recovery.

with 2^n. Given the same state size, when the tree has larger fan-out bit, the depth of the tree will be less and the recovery involves fewer layers, which introduces lower latency for recovering the original state. In addition, larger fan-out trees can tolerate more concurrent node failures or shard loss. Therefore, we should choose different tree structures for different applications based on their latency requirements and fault tolerance requirements.

Failure tolerance is evaluated with methods that use human intervention. To cause simultaneous failures, we deliberately remove some shards of application's state in some nodes to evaluate how fast SR3 can recover the state. Figure 5.15 shows the average recovery time for different number of simultaneous failures in the tree-structured recovery. The two curves show that the recovery time slightly increase with increasing number of shards failures. This is because, when a shard fails, the tree-structured mechanism can quickly retrieve the relevant shards from its leaf set and rebuild the failed shard, and the tree architecture can evenly distribute

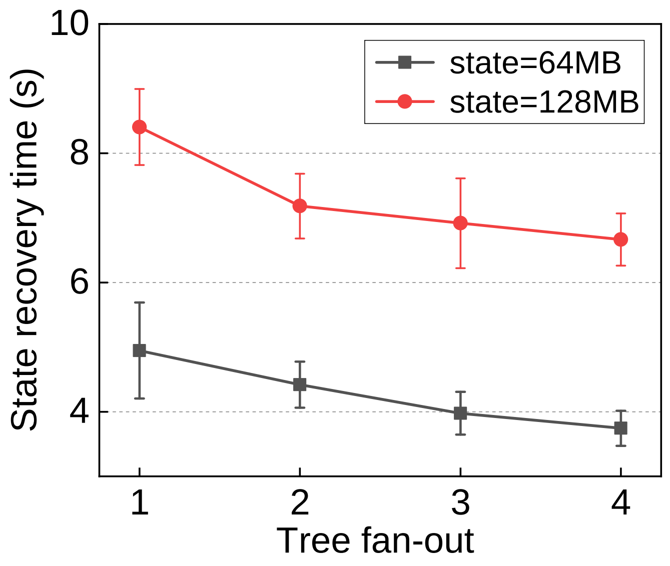


Figure 5.14: The state recovery time by varying the tree fan-out in SR3 tree-structured recovery.

the recovery overhead for recovering multiple simultaneous failures.

5.5.3 Load Balance

SR3 has attractive load balance property because it distributes state across all nodes in the overlay, which is especially beneficial when deploying a large number of concurrent applications. We evaluate SR3's load balance by deploying 500 stream processing applications and 1,000 stream processing applications on 5,000 Pastry nodes, respectively. The replication factor is set to be two. The state for each application is 32 MB, and the size for each shard is 512KB. As shown in Figure 5.16, each node has around 25 shards (red dash line) when deploying 500 applications. As shown in Figure 5.17, each node has around 40 shards (red dash line) when deploying 1,000 applications. This is because the P2P model of SR3's star-structured recovery, line-structured recovery and tree-structured recovery ensures that all peers can participate in the state saving process and the state recovery process.

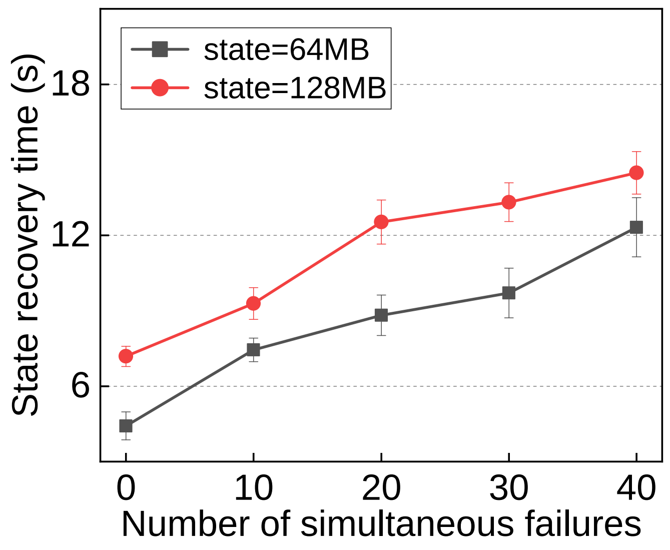


Figure 5.15: State recovery time with different number of failures.

Figure 5.18 shows the normal probability of the number of shards stored per node. Results show that when deploying 500 applications, around 95% nodes store less than 50 shards (25MB), and around 95% nodes store less than 100 shards (50MB) when deploying 1,000 applications. This demonstrates that the large volume of states from concurrently running applications are almost evenly distributed in the overlay with no centralized bottleneck. This demonstrates that SR3 achieves good load balance when recovering state for large numbers of concurrently running applications.

5.5.4 Overhead Analysis

We evaluate SR3 runtime overhead and compare them with Storm's checkpointing approach.

CPU overhead. Figure 5.19 shows the per-node CPU runtime overhead comparison of SR3's three state recovery mechanisms with Storm's checkpointing approach. The CPU overhead of SR3 is around 26.8% $\sim$ 44.3% less than the check-

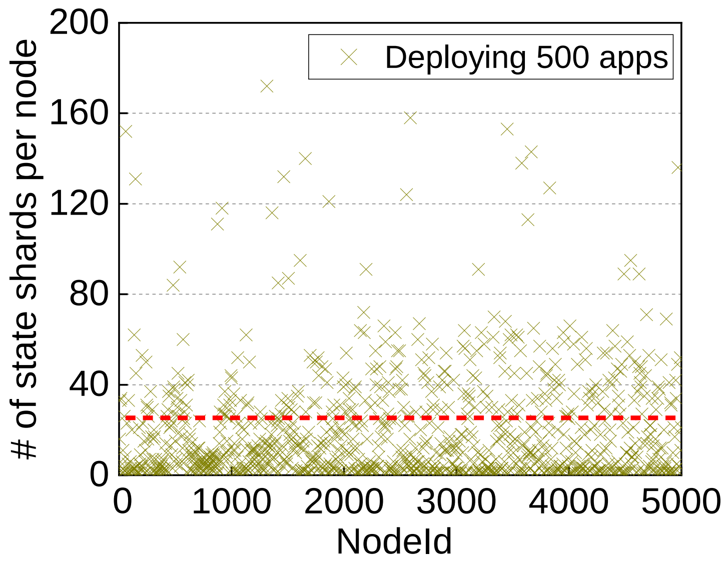


Figure 5.16: The distribution of state among the overlay when deploying 500 applications.

pointing recovery. This is because SR3 evenly distributes the recovery load across many peer nodes which reduces the per-node overhead, while the checkpointing recovery only relies on one or several centralized nodes for recovery.

Memory overhead. Figure 5.20 shows the per-node memory runtime overhead comparison of SR3's three state recovery mechanisms with Storm's checkpointing approach. The memory overhead of SR3 is around 30.9% $\sim$ 35.6% less than the checkpointing recovery. This is because checkpointing recovery involves a coordination service such as Zookeeper that needs to continuously maintain connections with all other nodes while SR3 avoids it.

Network overhead. Figure 5.21 shows the additional network traffic imposed by SR3 with varying the number of nodes without managing any state (showing purely the maintenance overhead). Results show that the number of bytes sent per node increase only linearly, with an exponential increase in the number of nodes. This is because most network traffics are ping-pong messages used for maintaining the overlay and routing (e.g., initialization and keep alive). So in most cases, each

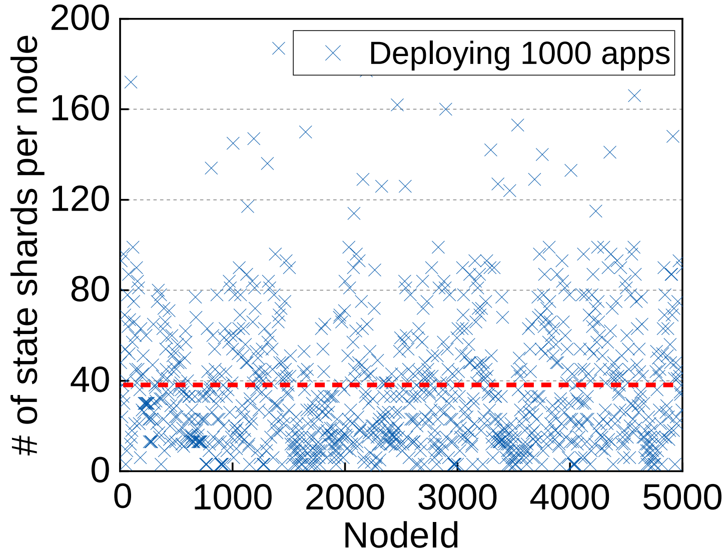


Figure 5.17: The distribution of state among the overlay when deploying 1,000 applications.

node pings to a limited set of nodes in the leaf set.

5.6 Summary

In this chapter, we have described and evaluated SR3, a state recovery framework that provides fast and scalable failure recovery mechanisms for protecting large distributed states in stream processing systems. Unlike existing failure recovery approaches in modern stream processing systems, which rely on the central master to perform replication recovery, checkpointing recovery, or DStream-based lineage recovery, SR3 introduces a distributed state recovery framework by leveraging DHT-based consistent ring overlay and routings. SR3 provides three different mechanisms to cater the needs for different stream applications that have diverse computation models and sizes of state.

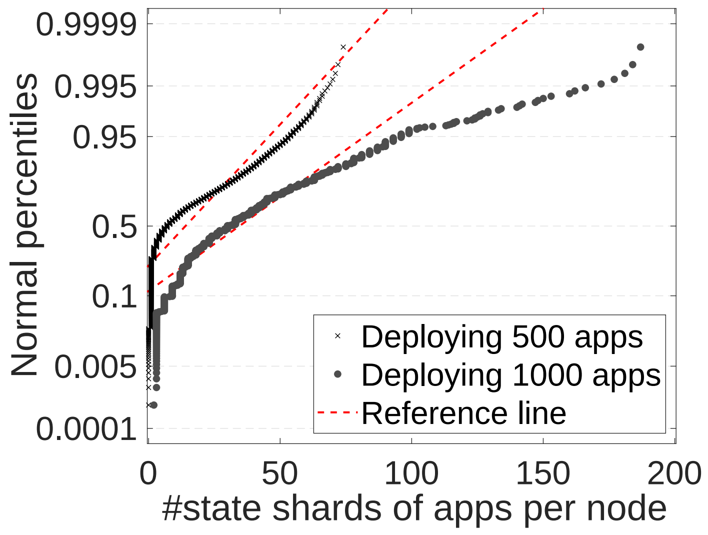


Figure 5.18: Normal probability of the number of shards per node.

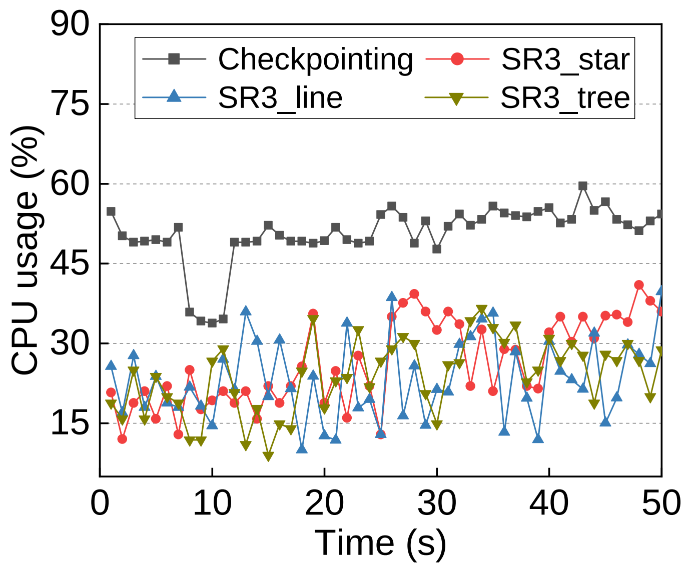


Figure 5.19: The runtime CPU overhead.

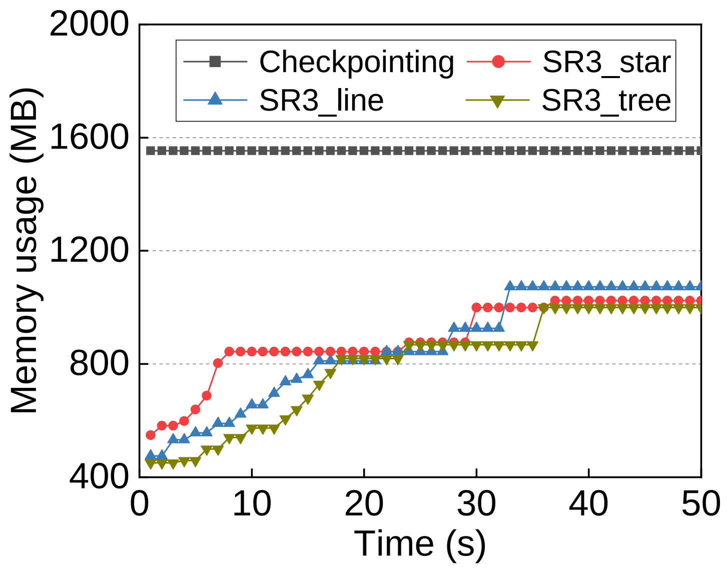


Figure 5.20: The runtime memory overhead.

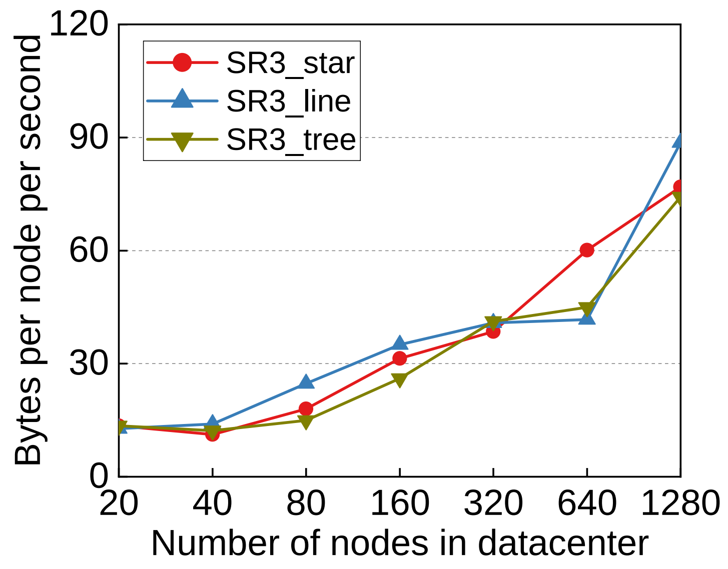


Figure 5.21: The network traffic overhead per node.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

How should we design computing platform for supporting the next-generation social stream data processing? This dissertation shows a possible solution perspective: a distributed and easy-to-use system, with defined processing abstract, instantly collaborates with the online data information, can provide useful knowledge that people need and promising performances in functionality.

In this dissertation, three research works are introduced from different perspectives of social stream data processing. More specifically, they mainly focus on the system-level design in supporting scalable, online, and efficient social spam detection.

Specifically, `Oases` can effectively harvest and uncover deceptive online spam posts from social communities, which is fulfilled by periodically updating the trained classifier through a decentralized DHT-based tree overlay. Besides, to guarantee online social spam detection, `Oases` actively filters out new spam and updates the classifiers to all distributed leaf agents in a scalable way.

In `SpamHunter`, it leverages the spam correlations among large-scale distributed data sources to enable efficient spam detection in a scalable manner. `SpamHunter` supports multiple groups to manage social data from various topics, areas, and geo-location. Each group forms a functional tree that guarantees flexible management across a large number of data servers/instances. Moreover, group coordination allows multiple groups to exchange and share spam correlations from distributed data sources, enabling efficient processing with the latest social spam from online data streams.

SR3 is a state recovery framework that provides fast and scalable failure recovery mechanisms for protecting large distributed states in the social stream data processing/applications. SR3 introduces a distributed state recovery framework by leveraging DHT-based consistent ring overlay and routings. SR3 provides three different mechanisms to cater the needs for different stream applications that have diverse computation models and sizes of state.

In conclusion, the first work, a system named `Oases`, is introduced to support online and scalable social spam detection in the system. The second work, a system named `SpamHunter`, is introduced to achieve efficient social spam detection by leveraging the online spam correlations from large-scale social stream data. The third work, `SR3`, is introduced to guarantee customizable state recovery in social stream data processing. They correspond to specific but interrelated functions of next-generation streaming data processing/applications, which can be used to support more different streaming applications in different environments.

6.2 Lessons Learned

The importance of system support. Most of the previous research work focused on context, format, the platforms, or they explored useful information from public information (such as accounts, users, or community activities). To a certain extent, these can be very useful for detecting and identifying spam activities (such as spam activities, social robots, or fake news). However, they did not consider how to fully integrate the system to achieve the entire work. Most previous works did not distinguish the differences between different platforms. For example, we can identify and detect social spam activities from streaming systems, distributed systems, centralized systems, batch systems, and parallel systems. However, how to combine different applications with different platforms to make full use of the entire system Performance has not received sufficient attention.

We witness that the modern systems such as Apache Spark [am], Storm [ad], and Kafaka [ab] have greatly improved the performance of deployed applications and many companies had transferred their computation and workload into these systems. This is certain that more and more large scale computation and applications should be allocated with appropriate system supports to ensure better performance and create more profits.

The views from data. Undoubtedly, data, is the key to social applications. The era of big data has brought a lot of excellent research work, and has continued to have a profound impact on our daily lives. In my research projects, extracting the appropriate social data for the system is a very important part, and it greatly affects the overall performance. Not only is the context of the collected data, but we should also pay more attention to data validity. Further, the pattern of the collected data will determine the main methods applied in the system. For example, data with contexts requires natural language processing models, and data composed of spatial and social relationships requires graph-based models. Therefore, the design of the system should consider the overall format of the data and the application to adapt to the data deployed in the system.

Not limited to a single data source. Results from research projects provide a valuable hint for the social data application, that is, the data source should not be limited to a single source. Unlike previous work that only focused on a single data source or platform, SpamHunter's results show that social data from multiple sources or platforms can greatly improve overall performance. This is because multiple data sources can provide a comprehensive view of valuable information, such as spam activity, and can improve performance.

Moreover, the research innovation of this project is not limited to social spam detection. It can be combined with various streaming applications and other perspectives of big data processing. The core of the entire study is focused on supporting scalable, efficient, and online streaming data applications, which means it can be extended to any other streaming applications that pursue high throughput, scalable processing, and high performance. In the next step, it is an interesting and promising realization to seamlessly migrate our system to more scenarios.

We learn to know that the next-generation stream system should flexibly support many different kinds of big data applications with minor or no modifications. Besides, it still comprehensively includes diverse functionalities to accommodate many kinds of applications with different requirements. Portability, flexibility, and availability should be considered in the design of system architecture and functionality.

Optimize the system bottleneck. One interesting lesson is how to carefully design the system architecture and avoid the bottleneck in deploying with the distributed system. This is an important part when designing systems to support new applications and scenarios.

Diverse systems may lead to different bottlenecks when faced with data applications such as social rumor detection, data trends analysis, and product review analysis. For example, when deploying the data application in a centralized server or platform, the computational overhead and processing latency of the server dominate the overall performance; when deploying the application in a large-scale distributed system, the data consistency and process management are key parameters for evaluation.

6.3 Future Work

Many open questions and challenges are needed further consideration and research efforts in supporting scalable, efficient, and online social stream data applications, as shown in following:

1. **System-level design**. Additional implementation steps can be developed, e.g., to implement new specification/configuration APIs for end users, achieve high-availability by exploring checkpointing/failover approaches, reduce run-time overhead, all with goals of achieving both good performance and high resource efficiency for large-scale online spam detection. Besides, how to integrate the system performance with the spam detection model is an interesting questions, e.g., balance the model costs and the system overhead, tolerate the failed data processing in distributed servers, recover processing failures in each server, etc.

   An interesting question for future work is how to recovery the state from stragglers in the social stream data processing. Stragglers are slow nodes. Stragglers are inevitable in large clusters. The root causes for stragglers can be disk failures, CPU contention, memory pressure, network congestion, or other internal factors such as unfair input partitioning. Left unchecked, stragglers will cause serious problems such as state inconsistency. We plan to explore speculation approach to address this challenge, in which speculative backup copies of slow tasks could be run in DHT's leaf set nodes.

2. **Exploiting deep neural networks.** Deep neural networks (DNNs) have been implemented in rumor and fake news detection without the traditionally tedious and time-consuming feature engineering [MGM$^+$16], and this leads to promising applications in the social spam detection areas. In the future, we

will explore the roles of DNNs in the large-scale online social spam detection and try to use the different characteristics of social data (e.g., time series, temporal patterns, propagation characteristic, etc) to enhance the detection accuracy and performances.

3. **Detecting social spam across various data formats, platforms, sources, and languages.** The various kinds of social networks bring the prosperity of the social life. Interesting, though various social networks or platforms have tremendous differences in the data/content formats, organization of accounts, or languages, they have significant correlations and similarities in the specific topics, news, headlines, pictures, and videos. Therefore, how to use these kinds of correlations and similarities in the next-generation social spam detection system desires more research efforts in the future. Besides, system-level design in dealing with large-scale online data streams from distributed sources needs solid and well-rounded implementations and leaves many open problems to be solved.

We hope that the continuous research experience of system support in streaming data applications can help us solve these problems and challenges. We also wish more research efforts can be conducted to design innovative next-generation big data stream systems.

BIBLIOGRAPHY

[aa] Apache flink. http://flink.apache.org/.

[ab] Apache kafka. http://kafka.apache.org/.

[ac] Apache samza. http://samza.apache.org/.

[ad] Apache storm. http://storm.apache.org/.

[ae] Apache storm 2.0.0. https://storm.apache.org/2019/05/30/storm200-released.html.

[af] Apache trident. http://storm.apache.org/releases/current/Trident-tutorial.html.

[ag] Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter.

[ah] Dublin bus gps sample data from dublin city council. https://data.gov.ie/dataset/.

[ai] Google finance data api. http://finance.google.com/finance/feeds/.

[aj] Leveldb. https://github.com/google/leveldb/.

[ak] Pastry. https://www.freepastry.org/FreePastry/.

[al] Rocksdb. `http://rocksdb.org/`.

[am] Spark streaming. https://spark.apache.org/streaming/.

[an] Wikimedia dumps. https://dumps.wikimedia.org/.

[a15a] Almost 10% of twitter is spam, 2015. `https://www.fastcompany.com/3044485/almost-10-of-twitter-is-spam\%20from\%20your\%20cite`.

[a15b] Social media today predictions for 2018, 2015. http://www.sileo.com/social-spam/.

[a16] Getting real about fake news, 2016. https://www.kaggle.com/mrisdal/fake-news/home.

[a18] Avoiding social spam hackers on facebook and twitter, 2018. https://www.socialmediatoday.com/news/social-media-today-predictions-for-2018-infographic/513179/.

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[ABB+16] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[ABC+15a] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel

Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[ABC$^+$15b] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015.

[ACÇ$^+$03] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[AG17] Hunt Allcott and Matthew Gentzkow. Social media and fake news in the 2016 election. *Journal of Economic Perspectives*, pages 211–36, 2017.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24. ACM, 2005.

[BGK$^+$17] Tekin Bicer, Doga Gursoy, Rajkumar Kettimuthu, Ian T Foster, Bin Ren, Vincent De Andrede, and Francesco De Carlo. Real-time data analysis and autonomous steering of synchrotron light source experiments. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 59–68. IEEE, 2017.

[BML16] Janki Bhimani, Ningfang Mi, and Miriam Leeser. Performance prediction techniques for scalable large data processing in distributed mpi systems. In *Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International*, pages 1–2, 2016.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1), 2001.

[Bre19] Aurelien Breeden. Child abduction rumors lead to violence against roma in france. `https://www.nytimes.com/2019/03/28/world/europe/roma-kidnap-rumors-france.html`, March 2019.

[CALS17] Mauro Coletto, Luca Maria Aiello, Claudio Lucchese, and Fabrizio Silvestri. Adult content consumption in online social networks. *Social Network Analysis and Mining*, 7(1):28, 2017.

[CCD$^+$03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Cidr*, volume 2, page 4, 2003.

[CDKR02] Miguel Castro, Peter Druschel, A-M Kermarrec, and Antony IT Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[CEF$^+$17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[CKE$^+$15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[CLQ11] Godwin Caruana, Maozhen Li, and Man Qi. A mapreduce based parallel svm for large scale spam filtering. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, pages 2659–2662, 2011.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[CWZ$^+$17] Chao Chen, Yu Wang, Jun Zhang, Yang Xiang, Wanlei Zhou, and Geyong Min. Statistical features-based real-time detection of drifted twitter spam. *IEEE Transactions on Information Forensics and Security*, 12(4):914–925, 2017.

[DCG$^+$17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[DLL$^+$00] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE network*, 14(1):78–88, 2000.

[GAB$^+$15] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the i/o of hpc applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[GCL$^+$12] Hongyu Gao, Yan Chen, Kathy Lee, Diana Palsetia, and Alok N Choudhary. Towards online spam filtering in social networks. In *NDSS*, volume 12, pages 1–16, 2012.

[GHW$^+$10] Hongyu Gao, Jun Hu, Christo Wilson, Zhichun Li, Yan Chen, and Ben Y Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 35–47. ACM, 2010.

[GXD$^+$14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th Symposium on Operating Systems Design and Implementation*, pages 599–613, 2014.

[HBSD17] Torsten Hoefler, Amnon Barak, Amnon Shiloh, and Zvi Drezner. Corrected gossip algorithms for fast reliable broadcast on unreliable systems. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 357–366, 2017.

[HSAC14] Liting Hu, Karsten Schwan, Hrishikesh Amur, and Xin Chen. Elf: Efficient lightweight fast stream processing at scale. In *USENIX Annual Technical Conference*, pages 25–36, 2014.

[IBY$^+$07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[KI17] Imrul Kayes and Adriana Iamnitchi. Privacy and security in online social networks: A survey. *Online Social Networks and Media*, pages 1–21, 2017.

[LCW10] Kyumin Lee, James Caverlee, and Steve Webb. Uncovering social spammers: social honeypots+ machine learning. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 435–442. ACM, 2010.

[LEC11] Kyumin Lee, Brian David Eoff, and James Caverlee. Seven months with the devils: A long-term study of content polluters on twitter. In *ICWSM*, 2011.

[LFQ+16] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 439–453, Berkeley, CA, USA, 2016. USENIX Association.

[LLP+12] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, 2012.

[LXDS+20] Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. Fp4s: Fragment-based parallel state recovery for stateful stream applications. In *2020 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2020.

[MBMDS10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[MGM+16] Jing Ma, Wei Gao, Prasenjit Mitra, Sejeong Kwon, Bernard J Jansen, Kam-Fai Wong, and Meeyoung Cha. Detecting rumors from microblogs with recurrent neural networks. In *IJCAI*, pages 3818–3824, 2016.

[MK10] Michael Mathioudakis and Nick Koudas. Twittermonitor: trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[NPP$^+$17] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177. IEEE, 2010.

[PACT18] Tapasya Patki, Emre Ates, Ayse Coskun, and J Thiagarajan. Understanding simultaneous impact of network qos and power on hpc application performance. In *Computational Reproducibility at Exascale (CRE'18), Supercomputing Workshop*, 2018.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 251–264, 2010.

[Per18] Shrinath Perera. A gentle introduction to stream processing. https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97, April 2018. Accessed in April 2020.

[PIP16] Daniel Pop, Gabriel Iuhasz, and Dana Petcu. Distributed platforms and cloud services: Enabling machine learning for big data. In *Data Science and Big Data Computing*, pages 139–159. Springer, 2016.

[PLGC15] Mayank Pundir, Luke M Leslie, Indranil Gupta, and Roy H Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 195–208. ACM, 2015.

[PZTH] Jian Pei, Bin Zhou, Zhaohui Tang, and Dylan Huang. Data mining techniques for web spam detection. *Simon Fras University Microsoft Ad Center.*

[QHS$^+$13] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[Ris01] Irina Rish. An empirical study of the naive bayes classifier. In *IJCAI workshop on empirical methods in AI*, pages 41–46, 2001.

[SBJM17] Shweta Salaria, Kevin Brown, Hideyuki Jitsumoto, and Satoshi Matsuoka. Evaluation of hpc-big data applications using cloud platforms. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1053–1061, 2017.

[SGH$^+$16] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 6. ACM, 2016.

[SHB04] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[SS16] Surendra Sedhai and Aixin Sun. Effect of spam on hashtag recommendation for tweets. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 97–98, 2016.

[Sut16] Mark Sutton. Streaming data analysis tools to study structural dynamics of materials, August 2016.

[TBH$^+$19] Anjana Talapatra, Shahin Boluki, Pejman Honarmandi, Alexandros Solomou, Guang Zhao, Seyede Fatemeh Ghoreishi, Abhilash Molkeri, Douglas Allaire, Ankit Srivastava, Xiaoning Qian, et al. Experiment design frameworks for accelerated discovery of targeted materials across scales. *Frontiers in Materials*, 6:82, 2019.

[TGM$^+$11] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time url spam filtering service. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 447–462, 2011.

[TLGP14] Kurt Thomas, Frank Li, Chris Grier, and Vern Paxson. Consequences of connectivity: Characterizing account hijacking on twitter. In *Pro-*

ceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.

[VBC$^+$14] Bimal Viswanath, Muhammad Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Towards detecting anomalous user behavior in online social networks. In *USENIX Security Symposium*, pages 223–238, 2014.

[VPO$^+$17] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389, 2017.

[VT16] Courtland VanDam and Pang-Ning Tan. Detecting hashtag hijacking from twitter. In *Proceedings of the 8th ACM Conference on Web Science*, pages 370–371. ACM, 2016.

[WIP14] De Wang, Danesh Irani, and Calton Pu. Spade: a social-spam analytics and detection framework. *Social Network Analysis and Mining*, 4(1), 2014.

[WP15] De Wang and Calton Pu. Bean: a behavior analysis approach of url spam filtering in twitter. In *Information Reuse and Integration (IRI), 2015 IEEE International Conference on*, pages 403–410. IEEE, 2015.

[WZLP15] Bo Wang, Arkaitz Zubiaga, Maria Liakata, and Rob Procter. Making the most of tweet-inherent features for social spam detection on twitter. *arXiv preprint arXiv:1503.07405*, 2015.

[XGL$^+$18] Hailu Xu, Boyuan Guan, Pinchao Liu, William Escudero, and Liting Hu. Harnessing the nature of spam in scalable online social spam detection. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3733–3736. IEEE, 2018.

[XHL$^+$18] Hailu Xu, Liting Hu, Pinchao Liu, Yao Xiao, Wentao Wang, Jai Dayal, Qingyang Wang, and Yuzhe Tang. Oases: An online scalable spam detection system for social networks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 98–105, 2018.

[XHLG19] Hailu Xu, Liting Hu, Pinchao Liu, and Boyuan Guan. Exploiting the spam correlations in scalable online social spam detection. In *International Conference on Cloud Computing*, pages 146–160. Springer, 2019.

[XJTG15] Wei Xing, Wei Jie, Dimitrios Tsoumakos, and Moustafa Ghanem. A network approach for managing and processing big cancer data in clouds. *Cluster Computing*, 18(3):1285–1294, 2015.

[XSJ16] Hailu Xu, Weiqing Sun, and Ahmad Javaid. Efficient spam detection across online social networks. In *Big Data Analysis (ICBDA), 2016 IEEE International Conference on*, pages 1–6, 2016.

[XZJ$^+$16] Wei Xie, Feida Zhu, Jing Jiang, Ee-Peng Lim, and Ke Wang. Topicsketch: Real-time bursty topic detection from twitter. *IEEE Transactions on Knowledge and Data Engineering*, pages 2216–2229, 2016.

[ZCD$^+$12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion 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*, pages 2–2. USENIX Association, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[ZHC07] Yue Zhang, Jason I Hong, and Lorrie F Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international conference on World Wide Web*, pages 639–648, 2007.

[zio19] Global enterprise spam filter market, 2019. `https://www.zionmarketresearch.com/report/enterprise-spam-filter-market`.

[ZLP+16] Arkaitz Zubiaga, Maria Liakata, Rob Procter, Geraldine Wong Sak Hoi, and Peter Tolmie. Analysing how people orient to and spread rumours in social media by looking at conversational threads. *PloS one*, 11(3), 2016.

[ZNJ+15] Xiang Zhu, Yuanping Nie, Songchang Jin, Aiping Li, and Yan Jia. Spammer detection on online social networks based on logistic regression. In *International Conference on Web-Age Information Management*, pages 29–40. Springer, 2015.

[ZRM15] Zhe Zhao, Paul Resnick, and Qiaozhu Mei. Enquiring minds: Early detection of rumors in social media from enquiry posts. In *Proceedings* *of the 24th International Conference on World Wide Web*, pages 1395–1405, 2015.

[ZZC$^+$15] Xianghan Zheng, Zhipeng Zeng, Zheyi Chen, Yuanlong Yu, and Chunming Rong. Detecting spammers on social networks. *Neurocomputing*, 159:27–34, 2015.

[ZZPZ16] Rongda Zhu, Aston Zhang, Jian Peng, and Chengxiang Zhai. Exploiting temporal divergence of topic distributions for event detection. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 164–171. IEEE, 2016.

[ZZW$^+$17] Daniel Yue Zhang, Chao Zheng, Dong Wang, Doug Thain, Chao Huang, Xin Mu, and Greg Madey. Towards scalable and dynamic social sensing using a distributed computing framework. In *The 37th IEEE international conference on distributed computing systems (ICDCS).*, pages 966–976, 2017.

VITA

HAILU XU

	Born, Laiwu, Shangdong Province, China
2010-2014	B.S., Computer Science and Engineering North China Electric Power University Baoding, China
2014-2016	M.S., Computer Science, Department of Electrical Engineering and Computer Science University of Toledo Toledo, Ohio
Jan 2015 – May 2016	Research Assistant Transportation Systems Research Laboratory Ohio Department of Transportation, Federal Highway Administration University of Toledo
Aug 2016-May 2019	Research Assistant Experimental and Virtualized Systems (ELVES) Research Lab School of Computing & Information Sciences Florida International University
2016-2020	Ph.D., Computer Science, School of Computing & Information Sciences Florida International University Miami, Florida, USA
May 2019 – Aug 2019	Research Summer Internship Lawerence Livermore National Laboratory California, USA

PUBLICATIONS

1. Hailu Xu, Liting Hu, Pinchao Liu, and Boyuan Guan, "Exploiting the Spam Correlations in Scalable Online Social Spam Detection", *In Proceedings of the 2019 International Conference on Cloud Computing (CLOUD)*, June 2019.

2. Hailu Xu, Murali Emani, Pei-Hung Lin, Liting Hu, and Chunhua Liao, "Machine Learning Guided Optimal Use of GPU Unified Memory", *MCHPC '19: Workshop on Memory Centric High Performance Computing, in conjunction with SC 19*, 2019.

3. Hailu Xu, Liting Hu, Pinchao Liu, Yao Xiao, Wentao Wang, Jai Dayal, Qingyang Wang and Yuzhe Tang, "Oases: An Online Scalable Spam Detection System for Social Networks", *2018 IEEE International Conference on Cloud Computing (IEEE CLOUD)*, June 2018.

4. Hailu Xu, Boyuan Guan, Pinchao Liu, William Escudero, and Liting Hu. "Harnessing the Nature of Spam in Scalable Online Social Spam Detection", *2018 IEEE Big Data workshop on Big Social Media Data Management and Analysis, in conjunction with IEEE Big Data*, 2018.

5. Pinchao Liu, Hailu Xu, Dilma Da Silva, Qingyang Wang, Sarker Tanzir Ahmed, and Liting Hu. "FP4S: Fragment-based Parallel State Recovery for Stateful Stream Applications", *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2020).*

6. Boyuan Guan, Liting Hu, Pinchao Liu, Hailu Xu, Jennifer Fu, Qingyang Wang, "dpSmart: A Flexible Group Based Recommendation Framework for Digital Repository Systems", *In Proceedings of the 2019 International Congress on Big Data (Big Data Congress)*, July 2019.

7. Pinchao Liu, Adnan Maruf, Farzana Beente Yusuf, Labiba Jahan, Hailu Xu, Boyuan Guan, Liting Hu, and Sitharama S. Iyengar, "Towards Adaptive Replication for Hot/Cold Blocks in HDFS using MemCached", *In Proceedings of 2019 International Conference on Data Intelligence and Security (ICDIS 2019), June 2019.*