SAND: A Fault-Tolerant Streaming Architecture for Network Traffic Analytics

Qin Liu*, John C.S. Lui*, Cheng He†, Lujia Pan†, Wei Fan†, Yunlong Shi†

*The Chinese University of Hong Kong, †Huawei Noah's Ark Lab
{qliu,cslui}@cse.cuhk.edu.hk, {hecheng,panlujia,David.Fanwei,shiyunlong}@huawei.com

***Abstract*—Many long-running network analytics applications impose a high-throughput and high reliability requirements on stream processing systems. However, previous stream processing systems cannot sustain high-speed traffic at the core router level. Furthermore, their fault-tolerant schemes cannot provide strong consistency which is essential for network analytics. In this paper, we present the design and implementation of SAND, a fault-tolerant distributed stream processing system for network analytics. SAND is designed to operate under high-speed network traffic, and it uses a novel checkpointing protocol which can perform failure recovery based on upstream backup and checkpointing. We prove our fault-tolerant scheme provides strong consistency even under multiple node failure. We implement several real-world network analytics applications on SAND, evaluate their performance using network traffic captured from commercial cellular core networks, and demonstrate that SAND can sustain high-speed network traffic and that our fault-tolerant scheme is efficient.**

***Keywords*-stream processing, network analytics, fault-tolerance**

I. Introduction

Stream processing systems are essential for real-time network analytics applications. For instance, telecommunication companies wish to classify network traffic in real-time so that they can perform proper resource allocation on different applications; network administrators wish to detect anomalies in core network traffic as soon as possible so that these traffics can be filtered or rate limited. To enable these real-time network analytics applications, we need a high performance stream processing system that can sustain high-speed network traffic.

However, designing such a system is challenging, because network analytics applications are usually long-running tasks and the stream processing systems must provide high availability for continuous processing. For example, the Policy and Charging Control (PCC) system in cellular networks can be used to manage policy rules and perform real-time charging control [12]. For such application that involves real-time charging, it is necessary to provide correct results after failure recovery.

Although there are a number of stream processing systems available [21, 3, 24, 15], they are not really designed to support network analytics applications and cannot sustain high-speed network traffic at the core router level. Furthermore, previous fault-tolerant approaches such as replication and upstream backup have their limitations. In replication, the system runs several identical copies of jobs on different machines which demands high resource investment. In upstream backup, each node needs to buffer its output data for a long time and suffers from high recovery time. Also, for both approaches, it is difficult to guarantee the consistency of results after failure recovery.

Contributions: In this paper, we present the design and implementation of a stream processing system called SAND which targets for real-time network analytics at the core router level (Section III). We also propose and implement a novel fault-tolerant scheme based on upstream backup and checkpointing. Our checkpointing protocol can guarantee global consistent checkpoints and recovery (Section IV). Finally, we carry out extensive evaluation to show the performance of SAND over other stream processing systems. We experimentally evaluate the fault-tolerant scheme under different failure patterns. We also implement several real-world network analytics applications on SAND and evaluate their sustained throughput using real network traffic collected from commercial cellular networks (Section V).

II. Background and Limitation of Previous Solutions

There are many network analytics which need high performance, real-time, and fault-tolerant stream processing systems. One representative network analytics application is *deep packet inspection* (DPI), which can classify network traffic packets into different application-level protocols. Telecommunication companies are often interested in the distribution of applications in the network traffic because administrators can do proper resource allocation and bandwidth management. Previously, high performance DPI systems were implemented on the Hadoop platform [23]. Hadoop is a *batch processing system*, so results from the system are *non-real-time* and it suffered from high processing latency. There is an urgent need to perform DPI on a real-time stream processing system so network operators can perform real-time resource management.

Many network analytics applications like DPI or network anomaly detection have several common characteristics. Firstly, they need to be executed at the core router level, so the stream processing system needs to sustain and operate at a high-throughput setting (i.e., at Gbps range). Secondly,

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.91

telecommunication administrators wish to process these applications in *real-time* and with *low latency*. This imposes a huge computational constraint on the streaming engine. Lastly, these applications run for a long duration, and this imposes a high reliability and fault-tolerant requirement on the stream processing systems. In case there is a component failure, the stream processing system needs to recover the component without compromising the integrity of the processing results.

To realize a high-throughput, highly fault-tolerant stream processing system, researchers have proposed to use the *continuous operator model* (COM) [21, 3]. Under the COM framework, streaming computation is carried by a set of long-lived operators. Each operator processes input data events and produces output data events that can be further processed by the next operators.

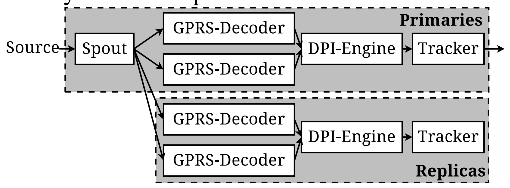


Figure 1: `AppTracker`: a DPI software using the COM.

To illustrate the concept of COM, let us consider `AppTracker`, a DPI software that we develop on the top of our stream processing system. Figure 1 depicts a software abstraction of `AppTracker`. It is composed of four different types of continuous operators: (1) *Spout*, (2) *GPRS-Decoder*, (3) *DPI-Engine*, and (4) *Tracker*. The Spout reads raw GPRS network traffic packets from an external source and assigns the traffic to different GPRS-Decoders. The GPRS-Decoder decodes GPRS packets, extracts the IP packets and forwards to the DPI-Engine. The DPI-Engine performs application classification on the input data. The Tracker summarizes the distribution of different applications in the network traffic.

Under the COM framework, an operator can be *stateful* or *stateless*. A stateful operator has mutable state that may be changed when it receives an input event, while its output streams depend on the internal state and input streams. For a stateless operator, the output streams do not depend on the internal state. In `AppTracker`, the Spout and GPRS-Decoders are stateless operators, while the DPI-Engines and Tracker are stateful operators. For each input network packet, the GPRS-Decoder decodes the GPRS headers and extracts the IP packets. Computation is done on a per packet basis so one does not need to use "*state*" to keep track of the computation. On the other hand, DPI needs to be carried out on a *flow-level* basis, so the DPI-Engine needs to aggregate IP packets into flows, then perform DPI operation on each flow. Hence the DPI-Engine is a stateful operator and its internal state includes a flow table that stores active flows.

Fault-tolerance in stream processing systems is an important technical challenge that needs to be addressed. In the course of stream processing, it is possible that one or more of these continuous operators may fail. If it is a stateful operator, then when this operator recovers, it is important to restore the internal state and continue with the streaming computation. However, the internal state of each operator potentially depends on the history of the input traffic or states of previous operators, so these internal states cannot be easily recreated by re-processing a small portion of the input stream.

Let us review several fault-tolerant schemes used by existing stream processing systems. The first approach is via *replication* (or active standby) [17]. Under this scheme, the stream processing systems use redundancy for execution. For each operator, it has a primary operator and one or more backup operators (or *replica*). Input data streams are sent to all operators. Consider `AppTracker` in Figure 1, we have primary operators (on the top) and the replica operators (at the bottom). Replication is an expensive fault-tolerant scheme since it at least doubles the resource requirement. Moreover, the replication scheme requires a costly *synchronization* [22, 5]. As shown in Figure 1, DPI-Engine must synchronize with its replica to ensure that they see input events in the same order (we will elaborate why the order of events is important).

Another approach to providing fault-tolerance is via *upstream backup* [17]. Under this scheme, each operator retains a copy of the data events it sent to a downstream operator. The data will only be purged when an acknowledgement is received from the downstream operator indicating that the data events have been processed. For some operators whose internal states only depend on a *subset* of input streams, we can recover their states by replaying the upstream operators' recent output data events. However, this is not applicable to stateful operators. For example, for the DPI-Engine operator, most input data events affect the operator's internal state. Therefore, upstream backup requires large buffer resources and the recovery delay can be significant under a high input traffic rate setting.

To improve the performance of upstream backup, *checkpointing technique* was introduced [11, 13]. Under this scheme, each operator periodically checkpoints its internal state. During recovery, the failed operator resumes from the most recent checkpoint, and only needs to re-process the data events after the last checkpoint. The advantage of this scheme is in reducing the recovery delay. However, the *order* of data events from different input streams is nondeterministic (because of the interleaving of data events within the network). Also, the inter-arrival times of data events are nondeterministic because of processing delays. For operators whose computations depend on the order and inter-arrival times of input, this nondeterminism makes it challenging to provide strong consistency [17] after recovery.

Let us illustrate why the above nondeterminism makes fault-tolerance difficult. Consider a streaming application as

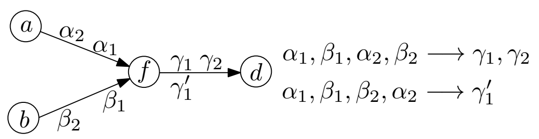

a
b
f
d
α_2
α_1
β_1
β_2
$\gamma_1\ \gamma_2$
γ_1'
$\alpha_1, \beta_1, \alpha_2, \beta_2 \longrightarrow \gamma_1, \gamma_2$
$\alpha_1, \beta_1, \beta_2, \alpha_2 \longrightarrow \gamma_1'$

Figure 2: For operator f, output events depend on the order of input events.

shown in Figure 2. Suppose the internal state of operator f is the sum of input data events. When operator f receives an input data event, it updates the sum. If the sum is larger than 10, it emits an output data event with the current sum to the next operator d. Suppose, at checkpoint c, operator f checkpoints its internal state $s_c = 5$ (which means the sum is 5). After checkpoint c, operator f receives data events $\alpha_1 = 2, \beta_1 = 1, \alpha_2 = 6, \beta_2 = 2$ in order from operator a and b. These data events trigger two output events from operator f: $\gamma_1 = 14, \gamma_2 = 16$. Then operator f fails. To recover operator f, we roll it back to state s_c and let operator a and b to replay data events α_1, α_2, β_1, and β_2. Because of the non-deterministic nature of stream processing, the arrival order of replayed data events may be different from the order before failure. Suppose the replayed data events from operator a and b arrive at the operator f in the order of $\alpha_1, \beta_1, \beta_2, \alpha_2$. Then operator f emits a new output event $\gamma_1' = 16$ to operator d. We can see that the output of operator f depends on the arrival order of input events. Note that γ_1, γ_2 and γ_1' are duplicate events triggered by the same input streams in different orders. The order and the number of such duplicate events are nondeterministic in stream processing systems. These duplicate events can cause inconsistency on the state of operator d after recovery, and cannot be handled by simply adding sequence numbers. So, for those applications that require strong consistency, previous systems [11, 13] cannot handle such duplicate events during recovery. For the similar reason, the replication scheme needs synchronization protocols to synchronize the order of input between the primary operators and the replicas, otherwise the primary operators and the replicas may produce different results.

The above discussion indicates that previous stream processing systems are not appropriate for high-speed network analytics. Firstly, network analytics applications require the stream processing system be sustainable at a high traffic rate. Secondly, the semantics of network analytics requires us to have strong consistency. To sustain high-throughput stream processing and provide strong consistency, we design a stream processing system called *SAND*. Let us now discuss SAND's architecture and fault-tolerant scheme.

III. System Design of SAND

In designing SAND, we set two design goals. First, it has to sustain high input traffic rates (i.e., at Gbps range). Our major target applications are network analytics (e.g., DPI and traffic anomaly detection) within the core networks. Note that most open source stream processing systems, S4 [21] and Storm [3], are implemented on Java Virtual Machine (JVM), so it is inefficient to develop and execute non-JVM operators on these two systems. Furthermore, S4 and Storm have heavy processing overheads and they cannot sustain high-speed network traffic (see Section V). SAND is optimized for high-throughput processing. It is implemented in C++ so it can use existing high performance libraries for network analytics. Our second design goal is to be fault-tolerant. We use *upstream backup* and *checkpointing techniques*, and complement these techniques with a novel checkpointing coordination protocol to provide strong consistency.

Figure 3 depicts the architecture of SAND. SAND uses the continuous operator model. A SAND cluster uses the single-master distributed system design and it has two types of nodes: a *coordinator* node and multiple *worker* nodes. Each worker can be viewed as a continuous operator described in Section II. The input to a worker is an input stream, which represents either a sequence of events of data source (e.g., network traffic), or a sequence of events generated by other workers. We name a worker that receives an input stream from an external source as a *source worker*. Otherwise, they are called the *internal workers*.

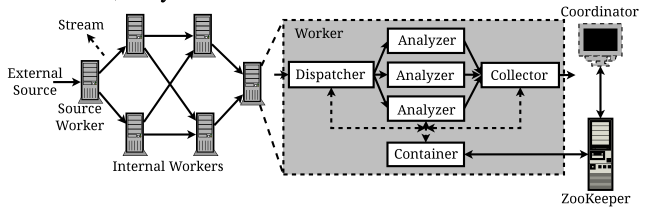


Figure 3: SAND Architecture.

The coordinator is responsible for managing workers and detecting worker failures. It relays control messages like starting a checkpoint and acknowledgement of a checkpoint among workers. All communication between the coordinator and the workers is done through a Zookeeper [16] cluster, which provides reliable distributed coordination service. Additionally, the coordinator is stateless; all states are kept in Zookeeper. So if the coordinator fails, no workers will be affected. Coordinator simply restarts and reconnects to the Zookeeper, which is a replicated service based on a quorum algorithm, so we do not need to consider its failure handling.

Each worker module is responsible for processing a portion of streams. It contains three types of processes: (a) *dispatcher*, (b) *analyzer*, and (c) *collector*. The *dispatcher* receives incoming streams, which can be originated from a data source (e.g., network traffic) or from other workers. The dispatcher decodes the streams and distributes them to one or multiple *analyzers*. Users have the flexibility to decide how to distribute the streams, i.e., they can use built-in load-balancing algorithms (e.g. stateless hashing or join the shortest queue) in SAND, or they can extend the load-balancing library. To utilize the multi-core architecture of modern machines, we can run *multiple analyzer processes* in parallel. Each analyzer is an application dependent analysis process which works on its assigned streams and produces intermediate results. The *collector* aggregates intermediate results from all analyzers to produce the final results, and forwards the results to the next-hop workers for further processing. If a worker has next-hop workers, the collector records the output data events in its *output buffer* of each next-hop worker. This allows the worker to replay its output data events in case there is a failure recovery. In SAND, whenever next-hop workers have performed a checkpointing operation, the system will purge some of the data from its output buffer. We will describe the output data purging operation in the next section.

Usually, the analyzer runs CPU-intensive tasks, and has the most computational load compared to that of the dispatcher or the collector. In SAND, each worker node runs exactly one dispatcher and one collector, but it can potentially run *multiple* analyzers processes. In case the input stream is of high traffic rate (such as in the core router traffic with Gbps rate), the worker can instantiate more analyzer processes to sustain the high processing requirement.

To manage the processing power of a worker, each worker runs a *container* daemon that spawns or stops the dispatcher, analyzer, and collector processes in the worker. The container also serves as a communication interface between the worker and the coordinator. Thus, the coordinator can manage the resources of each worker through the worker's container. For instance, the coordinator can inform a container to create more analyzers to increase the processing power.

To provide high-speed communications between the dispatcher and each analyzer, and those between each analyzer and the collector, SAND uses *lock-free ring buffers* [18] to avoid lock contention which may degrade system performance. Workers communicate via ZeroMQ [2], which is a messaging library optimized for high-throughput communication. We use ZeroMQ sockets instead of TCP sockets because (1) it provides reliable connection; (2) it batches small messages into a single unit to avoid expensive system calls; (3) it queues messages in a separate I/O thread, so sending and receiving operations are asynchronous.

It is important for us to emphasize that we only define the components of a worker as an *abstraction*. Developers can easily extend the detailed functionalities of the dispatcher, analyzer, and collector processes, as well as define the formats of the streams being processed and the messages exchanged among the workers and the coordinator.

To illustrate the mapping, consider `AppTracker` in Figure 1. Each operator can be directly mapped to a worker in a SAND cluster. Since GPRS decoding is a CPU-intensive task, GPRS-Decoder can have multiple analyzers which means it can use multiple cores in a single machine. If decoding demands more CPU resources, we can start multiple GPRS-Decoders in multiple machines. Workers which are not CPU intensive (e.g. Spout and Tracker) can be placed on the same machine.

IV. Fault-Tolerance & Checkpoint Coordination

SAND adopts a combination of upstream backup and checkpointing to achieve a balance between run-time overhead and recovery delay. Furthermore, we have designed and implemented a novel checkpointing protocol to provide strong consistency (Section IV-A) and failure recovery (Section IV-B), and finally, we prove its correctness (Section IV-C).

For ease of presentation, we assume that the dispatcher and collector in each worker are stateless, and we only need to checkpoint the state of analyzers. SAND does not need a priori knowledge of the internal state of an analyzer. Instead, it uses the two user-defined functions `export` and `import` in analyzers to complete a strong consistent checkpointing. All checkpoint data are written to HDFS [23], which performs replication for data reliability. The implementation of checkpointing uses the *copy-on-write* semantics of the `fork` system call. In `fork`, it creates a new process by duplicating the calling process. When a worker starts a checkpoint, each analyzer process calls `fork`, and creates one child process which is an exact copy of parent. The parent analyzer then resumes with the normal processing. The child analyzer writes the internal state using the `export` function to HDFS, then sends an acknowledgement to the coordinator, and finally exits. Note that the parent process does not have to wait for checkpointing to complete. During recovery, analyzer processes call the `import` function to fetch the checkpointed data from HDFS. In what follows, we formally describe the checkpointing protocol.

A. *Checkpointing Protocol*

To perform the real-time streaming computation, we assume that all workers in a SAND cluster can be mapped to a directed acyclic graph. Let W be the set of all workers. For each worker w in W, we define $\mathcal{U}_w$ as the set of *upstream workers* of w, and each workers in $\mathcal{U}_w$ generates data events as direct input to w. We define $\mathcal{D}_w$ as a set of *downstream workers* of w, and the input of workers in $\mathcal{D}_w$ is derived from w. Note that $\mathcal{D}_w$ includes the next-hop workers of w, as their next-hop workers, and so on. Let V be a set of workers. We define $\mathcal{U}_V = \cup_{w \in V} \mathcal{U}_w$ and $\mathcal{D}_V = \cup_{w \in V} \mathcal{D}_w$. To illustrate the notation, consider an example in Figure 4, for the worker e, we have $\mathcal{U}_e = \{c\}$ and $\mathcal{D}_e = \{g, h\}$. If $V = \{c, e\}$, then $\mathcal{U}_V = \{a, c\}$ and $\mathcal{D}_V = \{e, f, g, h\}$.

We propose a protocol to coordinate checkpointing operation on each worker in SAND. Note that the checkpointing protocol is similar to the Chandy-Lamport snapshot algorithm [7]. The main difference is that we apply it on stream processing systems and prove that it provides strong consistency under the COM framework. We allow workers to perform checkpoint at different time but the system can create a *global consistent checkpoint* from all workers' checkpoints. We assign a sequence number to each global checkpoint. The coordinator starts the new global checkpoint with a sequence number c by emitting special "*checkpointing messages*" to all *source* workers. When a source worker receives the checkpointing message, it emits special events called "*anchor events*" to all its next-hop workers. An anchor event indicates that the internal state depending on input events which arrived before this anchor event should be checkpointed. So when a worker receives the anchor events from *all* of its upstream workers, it checkpoints its internal state and emits anchor events to its next-hop workers. The checkpointing protocol is stated as follows:

1) Suppose the checkpointing interval is T seconds which can be set by the user. Every T seconds, the coordinator increments the sequence number and sends checkpointing messages to all source workers to start a new global checkpoint with sequence number c.
2) When a source worker receives the checkpointing message from the coordinator, it emits *anchor events* to all of its next-hop workers. The checkpointing messages and anchor events contain c to indicate that they are initiated by checkpoint sequence number c.
3) Each worker w uses a boolean array `flag` to track upstream workers from which anchor events have not arrived. For each upstream worker $u \in \mathcal{U}_w$, `flag[u]` is initialized to false which means w has not received the anchor event from u. When worker w receives an event E from an upstream worker u, it checks:
   a) if `flag[u]` is false and E is a data event, worker w processes the data event E normally;
   b) if `flag[u]` is false and E is an anchor event, w sets `flag[u]` to true. If for each $v \in \mathcal{U}_w$, `flag[v]` is true, w emits anchor events to all of its next-hop workers, resets every element of `flag` to false, and finally starts the checkpoint procedure by forking analyzer processes:
      - The children analyzers checkpoint the internal state of w, which we denote as s_c^w, to HDFS. Then the children processes send an acknowledgement to the coordinator via the container.
      - Concurrently, the parent analyzers can process input events normally.
   c) if `flag[u]` is true, w buffers E and postpones the processing of E until w starts checkpoint c. If E is an anchor event, it must be initiated by some checkpoint d where $d > c$. Since it will be buffered, it will not affect checkpoint c.
4) When the coordinator receives acknowledgements of checkpoint c from all workers, it means the global checkpoint c completes.

To illustrate the protocol, let us consider the example in Figure 2 again. In this case, the two source workers are a and b. Suppose after the coordinator starts the global checkpoint c, worker a emits an anchor event μ_a before α_1 and worker b emits anchor event μ_b before β_1. Suppose the arrival order of input events to worker f is $\mu_a, \alpha_1, \mu_b, \beta_1, \alpha_2, \beta_2$. According to our protocol, when worker f receives α_1, it will buffer α_1 and postpone the processing of α_1 until it receives μ_b. When μ_b arrives, f emits an anchor event to worker d. This implies that worker d receives this anchor event and checkpoints its internal state before γ_1 arrives. Now if worker f fails, we can rollback worker f and d to checkpoint c, and then replay α_1, α_2, β_1, and β_2. Note that the state of worker d at checkpoint c does not depend on data events from worker f after checkpoint c. Unlike the previous fault-tolerant schemes described in Section II, when worker d processes duplicate events like γ_1', it will not cause inconsistent state. Our checkpointing algorithm possesses the following property.

Property IV.1. *For each worker w, the internal state s_c^w only depends on the data events from all upstream workers of w before checkpoint c.*

Proof: For each checkpoint c and each worker w, before worker w checkpoints the state s_c^w, suppose worker w receives a data event E which was produced by an upstream worker u after checkpoint c. Our protocol ensures that worker w has already received the anchor event from u and `flag[u]` was set to true. So data event E will be buffered and will not affect the state s_c^w. ■

One important technical note is that at each checkpoint, a worker needs to buffer some input events until anchor events from all upstream workers arrive. The arrival times of anchor events are decided by the processing delays of upstream workers, or the workloads of upstream workers. As long as the workloads at workers are balanced, then the system does not need to buffer too many input events. We will present our evaluation on the runtime overhead caused by buffering in Section V.

Note that each worker records its output data events locally in an *output buffer*. This allows a worker to replay its output events when any of its next-hop workers fails. We cannot store these events to HDFS because of the latency incurs in HDFS. When the coordinator detects that global checkpoint c is finished, it informs every worker to delete output events before checkpoint c in its output buffer and delete all data associated with checkpoint $c-1$ on HDFS. The output buffers need to store all data events between two consecutive checkpoints. So the checkpointing interval cannot be too long, otherwise the output buffers may consume too much memory. We will present out performance evaluation results on the impact of checkpointing interval in Section V.

B. Failure Recovery

Next we introduce how to recover from failures. First of all, the container of each worker maintains heartbeats with the Zookeeper cluster by creating an ephemeral node on Zookeeper. When a worker fails unexpectedly, the Zookeeper will notify the coordinator. When the coordinator detects worker failures, suppose c is the largest sequence number of available global checkpoints on HDFS. It means that for each worker w, state s_c^w is available on HDFS. Let F be the set of failed internal workers and each worker in F has at least one upstream worker. When a worker w fails, we need to recover its internal state by resuming it to the latest checkpoint. However, as described in Section II, the failure of worker w will also affect the order of input events of $\mathcal{D}_w$. If it is not handled correctly, workers in $\mathcal{D}_w$ will generate inconsistent state. This implies we must also rollback the workers in $\mathcal{D}_w$. Define the *rollback set* of F as $\mathcal{R}_F = F \cup \mathcal{D}_F$, and the *replay set* of F is $\mathcal{P}_F = \mathcal{U}_{\mathcal{R}_F} - \mathcal{R}_F$. During failure recovery, for each worker w in $\mathcal{R}_F$, the coordinator restarts worker w and rolls it back to state s_c^w. Then the workers in $\mathcal{P}_F$ replay the data events in their output buffers. As an example shown in Figure 4, suppose the failed workers are $F = \{d, e\}$. The downstream workers of failed workers are $\mathcal{D}_F = \{g, h\}$. During recovery, we rollback workers in $\mathcal{R}_F = F \cup \mathcal{D}_F = \{d, e, g, h\}$, and replay data events in the output buffers of $\mathcal{P}_F = \{b, c, f\}$. We do not consider the failure of a source worker (like a in Figure 4), because the external data source cannot replay lost data events. Thus when a fails, data losses may happen.

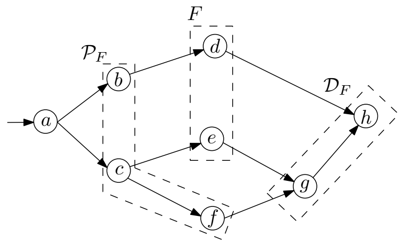


Figure 4: An example of our failure recovery process.

C. Proof of Consistency

As described in Section II, because of the unpredicted interleaving of input data events, running the same worker for multiple times cannot produce a unique output result even given the same input streams. Nevertheless, we only need to ensure that the output of a recovered worker is one of the possible results without failures. We define consistency as following.

Definition IV.2. A worker is *consistent* after recovery if and only if the output data events and internal state of the worker are one of the possible results when running the worker without failures.

Next, we prove that all workers are consistent after failure recovery in a SAND cluster. Let W denote the set of all workers in a cluster and assume W is a directed acyclic graph (DAG). Let $\mathcal{G}_F$ be $W - \mathcal{R}_F$, which is a set of workers which do not need to be rolled back. In other words, we partition W into two disjoint sets $\mathcal{G}_F$ and $\mathcal{R}_F$. For example, in Figure 4, we have $F = \{d, e\}$, $\mathcal{G}_F = \{a, b, c, f\}$, and $\mathcal{R}_F = \{d, e, g, h\}$. Now we prove workers in the two sets are consistent after recovery.

Theorem IV.3. *Workers in $\mathcal{G}_F$ are consistent after recovery.*

Proof: Since the workers in $\mathcal{G}_F$ do not fail and they do not need to roll back during recovery, hence they are not affected during the recovery process, and they are consistent after the completion of the recovery. ∎

Theorem IV.4. *Workers in $\mathcal{R}_F$ are consistent after recovery.*

Proof: Suppose the checkpoint c is the latest available global consistent checkpoint. For each worker w in $\mathcal{R}_F$, worker w is rolled back to s_c^w during recovery. By Theorem IV.3, we only need to prove the following claim: for $V \subset \mathcal{R}_F$, if workers in $W - V$ are consistent, workers in V are also consistent after recovery.

Supposing $|V| = n$, we shall prove the above claim using induction on n. When $n = 1$, let $V = \{w\}$. The upstream workers of w must belong to $W - V$. Since workers in $W - V$ are consistent, they either replay or reproduce all input events of worker w after the checkpoint c and before failure. So worker w can re-process input data events after checkpoint c. By Property IV.1, s_c^w only depends on the input events before the checkpoint c. So the replayed or reproduced data events do not compromise the state of worker w during recovery. Hence worker w is consistent after recovery.

Suppose the statement hold for $n = k$. Consider the case $n = k + 1$. Since W is a DAG, the subgraph V is also a DAG. So we can find a worker $w \in V$ with no incoming edges, which means all upstream workers of w are in $W - V$. Hence worker w is consistent after recovery. By inductive hypothesis, workers in $V - \{w\}$ are consistent after recovery. So workers in V are also consistent after recovery and the result follows by induction. ∎

By Theorem IV.3 and Theorem IV.4, all workers in the SAND cluster are consistent after recovery.

V. Evaluation and Applications

In this section, we present the performance evaluation of SAND. In particular, we show its sustainability under high input traffic, its scalability and fault-tolerant capability. We also implement and compare four different heavy hitter detection algorithms to show the extensibility of SAND.

Experiment 1 (Sustainability under High Input Traffic): First, we compare SAND with two open source stream processing systems, Storm [3] and Blockmon [15], and see how these stream processing systems stack up when they are subjected under high traffic rate. We implement an application called packet counter and install it on Storm, Blockmon and SAND. The packet counter application reads network packets, decodes the TCP/IP header of each packet and counts total number and size of packets. We use this packet counter application to demonstrate the overheads of different stream processing systems. In SAND, we use two workers to implement the packet decoder: one source worker for reading network packets and one worker for counting packets. We implement similar functionality in Storm (using one spout and one bolt) and Blockmon (using two blocks).

We install SAND, Storm and Blockmon in our testbed. Our testbed is a quad-core 3.10 GHz machine with 4GB RAM. We collect a packet header trace from CAIDA [4]. The trace lasts for 21 minutes in the PCAP format, and contains 331 million packets accounting for a total of 143GB of traffic. To analyze the performance of the systems at the peak traffic rate, we load the trace file into memory, and have the systems process the packet headers as fast as possible. Since the memory is limited, we only load the first 2GB of the trace file lasting for 90 seconds. Loading the trace file into memory enables us to eliminate the overhead due to disk read, and hence the performance bottleneck should lie within the stream processing systems. The measurement is repeated for 10 times and we average the results.

The throughput results of three systems are presented in Table I. SAND achieves the highest throughput of 31Gb/s, which shows that it can process packets at the core routers level. Furthermore, the achieved throughput of SAND is 3.7 times and 37.4 times as compared to Blockmon and Storm respectively. The main reason why Storm has poor performance is because it was implemented in Java, and it is mainly used to perform analytics for higher layer applications (e.g., web analytics) but not for network traffic. Blockmon is implemented in C++, but we noticed that its implementation of internal communication channels is not efficient which causes the performance degradation.

Table I: Performance of Storm, Blockmon and SAND.

Streaming System	Packets/s	Payload Rate	Header Rate
Storm	260K	840Mb/s	81.15Mb/s
Blockmon	2.7M	8.4Gb/s	844.9Mb/s
SAND	9.6M	31.4Gb/s	3031.7Mb/s

Experiment 2 (Scalability): We use `AppTracker` described in Section II to demonstrate the scalability of SAND. Our testbed is a cluster of three Linux 3.2 servers: S1, S2 and S3. The three servers are connected by 1Gbps LAN. Each server has 16 physical 2.10 GHz CPU cores and 94GB RAM. We run our evaluation on a 2-hour network trace (32GB), which is collected from a commercial GPRS core network in China in 2013. The raw IP packets with full payload are captured (without sampling) from the GPRS core network and stored in the PCAP format. We also deploy the Zookeeper and HDFS service on the three servers. As an example, in Table II, we show the top 5 applications in our trace obtained from the output of `AppTracker`.

We then evaluate the throughput of the overall system. First, we run four workers on a single server. In our implementation, the overall performance is bounded by GPRS-Decoder, so we vary the number of analyzers of GPRS-Decoder. As shown in Figure 5a, the throughput scales up linearly as we add analyzers. Second, we run `AppTracker` on three servers. We run *Spout*, *DPI-Engine* and *Tracker* on S1, and two *GPRS-Decoders* on S2 and S3. Again, we vary the number of analyzers of GPRS-Decoders. As shown in Figure 5a, the throughput also scales up linearly as we add analyzers. The result shows that (1) SAND can scale up by parallelizing the computation of a worker on multiple CPU cores in a single server; (2) it can also scale out by running parallel workers on multiple servers.

Table II: Result of `AppTracker`.

Application	Distribution
HTTP	15.60%
Sina Weibo	4.13%
QQ	2.56%
DNS	2.34%
HTTP in QQ	2.17%

Experiment 3 (Fault-Tolerance): In this experiment, we deploy `AppTracker` on three servers as in Experiment 2. We evaluate the throughput of SAND using different checkpointing intervals. As shown in Figure 5b, the overheads of our checkpointing protocol is negligible.

Next, we evaluate the recovery time of SAND after different failure scenarios. We also set the number of analyzers in each of the two GPRS-Decoders as nine. The result is presented in Figure 6. First, we set the checkpointing interval as $T = 5$ seconds. Then we terminate both the GPRS-Decoder and DPI-Engine processes at time t_2, t_3 and t_5 respectively. In this scenario, SAND can recover from failures in about few seconds. Secondly, we set the checkpointing interval as $T = 10$ seconds. We terminate the Tracker at t_1. In this scenario, SAND can recover in 3 seconds. However, when we kill the GPRS-Decoder at t_4, it takes around 11 seconds to recover. The recovery time is composed of three parts (1) the time for the coordinator to detect failures; (2) the time to restart and roll back failed workers; (3) the time for those workers to process replayed data events. Usually, a larger checkpointing interval increases the time to process replayed events in upstream workers' output buffer, so we can see in Figure 6 that the recovery time at t_4 is longer than t_2, t_3 and t_5.

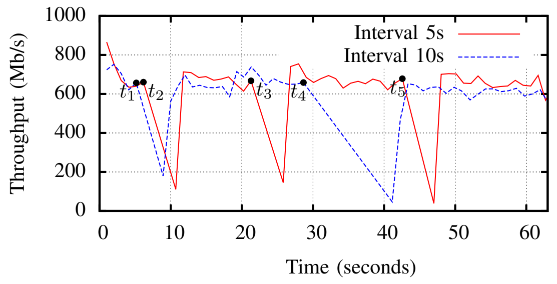
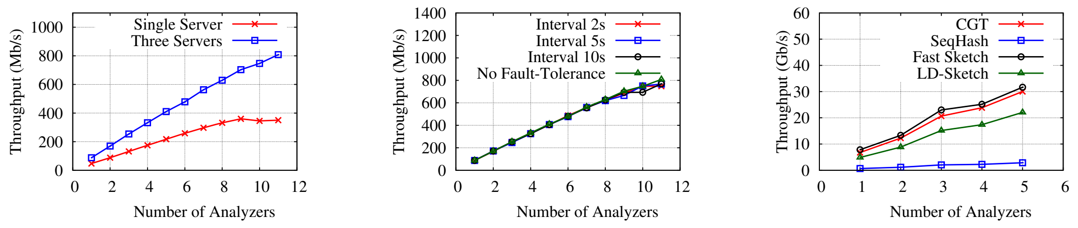


Figure 6: Recovery Time.

Experiment 4 (Real-Time Heavy Hitter Detection): Real-time characterization of traffic anomalies, e.g., heavy hitters and heavy changers [10], is critical for network operations. Characterizing traffic anomalies in real-time is challenging.



(a) Scalability of SAND. (b) Fault-Tolerant Scheme Overheads (c) Comparing Hitter Detection Alg

Figure 5: Experimental Results for 2, 3 and 4.

A scalable anomaly detection algorithm needs to be performed on parallel streaming systems. Previous techniques are mainly studied and evaluated in a single-processor setting. We implement four state-of-the-art heavy hitter and heavy changer detection algorithms on the top of SAND: *Combinational Group Testing*(CGT) [9], *SeqHash* [6], *Fast Sketch* [19], and *LD-Sketch* [14]. For each algorithm in a single worker, the dispatcher reads raw packets and forwards them to selected analyzers. The analyzer updates its internal data structure (sketch) when it receives a packet. The collector summarizes and outputs anomalies.

Our testbed is a multi-core server with 12 physical 2.93GHz CPU cores and 50GB RAM. We run our evaluation on real IP packet header traces which we collected on December 2010 from a commercial 3G UMTS network in mainland China. The traces contain 1.1 billion packets that account for a total of around 600GB of traffic. Figure 5c shows the throughput of the four heavy hitter detection algorithms using multiple analyzers. We see that the throughput increases almost linearly as the number of analyzers grows. This shows we can scale up the throughput of different traffic anomaly detection algorithms in SAND via parallelization.

VI. Related Work and Conclusion

There have been extensive studies on distributed streaming architectures for real-time processing, and we highlight some examples here. MapReduce Online [8] supports continuous processing within and across different MapReduce jobs. S4 [21], Storm [3], and Flume [1] are based on the COM framework. They treat streams as a sequence of events and they are handled by different processing elements. These systems ensure reliable message delivery but they cannot provide strong consistency after recovery. Blockmon [15] schedules CPUs and communication channels to achieve high-performance message passing. D-Stream [24] and Strom Trident [20] provide reliable fault tolerance by decomposing computing jobs in small timescales. However, since they do not use the COM framework, they are not extensible as our system and suffer from higher latency.

We present SAND, a new system for distributed stream processing for network analytics. SAND can sustain high-speed network traffic and provides reliable fault tolerance and to ensure strong consistency of processing results. We demonstrated that SAND can operate at core routers level and can recover from failure in order of seconds.

Acknowledgment

We thank our shepherd, Prof. Yair Amir, for his helpful and insightful comments and suggestions.

References

[1] Apache Flume. http://flume.apache.org.
[2] Distributed Computing made Simple - zeromq. http://zeromq.org/.
[3] Storm:Distributed & fault-tolerant realtime computation http://storm-project.net/.
[4] The CAIDA Anonymized Internet Traces 2009 Dataset. http://www.caida.org/data/passive/passive_2009_dataset.xml.
[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 2008.
[6] T. Bu, J. Cao, A. Chen, and P. P. Lee. Sequential hashing: A flexible approach for unveiling significant patterns in high speed networks. *Computer Networks*, 54(18):3309–3326, 2010.
[7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.
[8] T. Condie and R. Sears. MapReduce Online. In *NSDI*, 2010.
[9] G. Cormode and S. Muthukrishnan. What's new: finding significant differences in network data streams. *IEEE/ACM Transactions on Networking (TON)*, 13(6):1219–1232, 2005.
[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.
[11] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736, 2013.
[12] G. Finnie. Mobile broadband & the rise of policy: Technology review & forecast. Technical report, Heavy Reading, 2011.
[13] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. An empirical study of high availability in stream processing systems. In *Middleware*, 2009.
[14] Q. Huang and P. P. C. Lee. LD-Sketch: A Distributed Sketching Design for Accurate and Scalable Anomaly Detection in Network Data Streams. In *INFOCOM*, 2014.
[15] F. Huici, A. Di Pietro, B. Trammell, J. M. Gomez Hidalgo, D. Martinez Ruiz, and N. d'Heureuse. Blockmon: A high-performance composable network traffic measurement system. *SIGCOMM CCR*, 42(4):79–80, 2012.
[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.
[17] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. B. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790, 2005.
[18] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 1977.
[19] Y. Liu, W. Chen, and Y. Guan. A fast sketch for aggregate queries over high-speed network traffic. In *INFOCOM*, 2012.
[20] N. Marz. Trident tutorial. https://github.com/nathanmarz/storm/wiki/Trident-tutorial.
[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.
[22] M. A. Shah, J. M. Hellerstein, and E. A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD*, 2004.
[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, pages 1–10, 2010.
[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, 2013.