The Journal of Systems and Software 122 (2016) 553–563

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

SAND: A fault-tolerant streaming architecture for network traffic analytics[☆]

Qin Liu[a], John C.S. Lui[a,*], Cheng He[b], Lujia Pan[b], Wei Fan[b], Yunlong Shi[b]

[a] The Chinese University of Hong Kong, Hong Kong
[b] Huawei Noah's Ark Lab, Hong Kong

ARTICLE INFO

Article history:
Received 14 October 2014
Revised 28 March 2015
Accepted 31 July 2015
Available online 10 August 2015

Keywords:
Stream processing
Network analytics
Fault-tolerance

ABSTRACT

Many long-running network analytics applications (e.g., flow size estimation and heavy traffic detection) impose a high-throughput and high reliability requirements on stream processing systems. However, previous stream processing systems which are designed for higher layer applications cannot sustain high-speed traffic at the core router level. Furthermore, due to the nondeterministic nature of message passing among workers, the fault-tolerant schemes of previous streaming architectures based on the continuous operator model 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, including heavy traffic hitter detection as well as policy and charging control for cellular networks, and we evaluate their performance using network traffic captured from commercial cellular core networks. We demonstrate that SAND can sustain high-speed network traffic and that our fault-tolerant scheme is efficient.

1. 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 instance, mobile operators need to have a scalable and reliable real-time policy and charging control (PCC) system (Finnie, 2011) on the top of stream processing systems. Leveraging PCC systems, operators can provide personalized service plans tailored for subscribers' behaviors, for example, to assist on-the-fly purchasing decisions of various products. Therefore, to provide uninterrupted operation for PCC systems or other critical applications against failures like machine crash, fault-tolerance is a key feature we need to have in distributed stream processing systems. In addition, many customers want *strong consistency* after failure recovery: for critical task like charging service fees, it is necessary to provide correct results even when some machine crashes.

Although there are a number of stream processing systems available[1] (Neumeyer et al., 2010, Huici et al., 2012; Zaharia et al., 2013), they are not really designed to support network analytics applications and cannot sustain high-speed network traffic at the core router level. Most stream processing systems including Storm[1], S4 (Neumeyer et al., 2010), and MillWheel (Akidau and Balikov, 2013) are built on a *continuous operator model*, where streaming computations are carried on a series of stateful operators. Specifically, previous fault-tolerant

[☆] An earlier version of this work was presented at the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014) (Liu et al., 2014). This extended version presents added materials which include the design and implementation of two new complex streaming applications over SAND (see Section 5), and the evaluation of a new application, namely, `SmartPCC` (see Section 6) to demonstrate the utility of SAND.

[*] Corresponding author. Tel.: + 852-2609-8407; fax.: + 852-2603-5024.
E-mail addresses: qliu@cse.cuhk.edu.hk (Q. Liu), cslui@cse.cuhk.edu.hk (J.C.S. Lui), hecheng@huawei.com (C. He), panlujia@huawei.com (L. Pan), David.Fanwei@huawei.com (W. Fan), shiyunlong@huawei.com (Y. Shi).

[1] Storm: distributed & fault-tolerant realtime computation, http://storm.apache.org/

http://dx.doi.org/10.1016/j.jss.2015.07.049
0164-1212/© 2015 Elsevier Inc. All rights reserved.

approaches on the continuous operator model 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. We also propose and implement a novel fault-tolerant scheme based on "*upstream backup*" and "*checkpointing*" over SAND. Our checkpointing protocol can be adapted to all stream processing systems based on the continuous operator model and guarantee strong consistency after failure recovery. To illustrate the utility of SAND, we present several network analytics applications which need real-time processing and demonstrate how to implement these applications on the top of SAND. 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 evaluate the sustained throughput of several real-world network analytics applications using real network traffic collected from commercial cellular networks.

The rest of the paper is organized as follows. Section 2 reviews previous stream processing systems, existing fault-tolerant schemes, and their limitations. Section 3 presents the design of SAND. Our fault-tolerant scheme is described in Section 4. We port several real network analytics applications to SAND in Section 5. We present an evaluation of SAND in Section 6 and conclude in Section 7.

2. Background and related work

2.1. Motivation

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 (Shvachko et al., 2010). 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 Gb/s range). Secondly, 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.

2.2. Model of stream processing

To realize a high-throughput, highly fault-tolerant stream processing system, researchers have proposed to use the *continuous operator model* (COM)[1] (Neumeyer et al., 2010). 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.

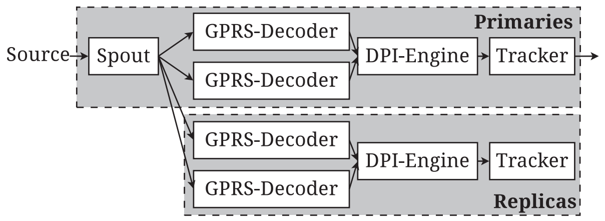

Fig. 1. `AppTracker`: a DPI software using the COM.

To illustrate the concept of COM, let us consider `AppTracker`, a DPI software that we develop on the top of our stream processing system. Fig. 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.

2.3. Existing fault-tolerant schemes

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) (Hwang et al., 2005). 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 Fig. 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* (Balazinska et al., 2008; Shah et al., 2004). As shown in Fig. 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* (Hwang et al., 2005). 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 (Fernandez et al., 2013; Gu et al., 2009). 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 (Hwang et al., 2005) after recovery.

2.4. Challenge due to nondeterminism

Let us illustrate why the above nondeterminism makes fault-tolerance difficult. Consider a streaming application as shown in Fig. 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 $\alpha_1, \alpha_2, \beta_1$, and β_2. Because of the nondeterministic 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, the upstream backup scheme with checkpointing cannot handle such duplicate events during recovery due to the above nondeterminism. 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.

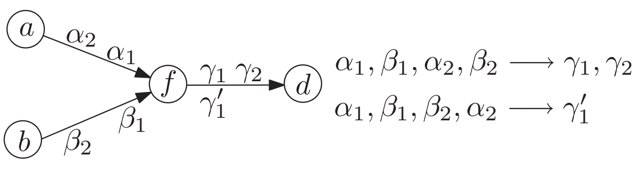
a
b
f
d
α_2
α_1
β_1
β_2
$\gamma_1\ \gamma_2$
γ_1'
$\alpha_1, \beta_1, \alpha_2, \beta_2 \longrightarrow \gamma_1, \gamma_2$
$\alpha_1, \beta_1, \beta_2, \alpha_2 \longrightarrow \gamma_1'$

Fig. 2. For operator *f*, output events depend on the order of input events.

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 discuss SAND's architecture and fault-tolerant scheme in the following sections.

2.5. Related work

There have been extensive studies on distributed streaming architectures for real-time processing, and we highlight some examples here. Like our system, S4 (Neumeyer et al., 2010), Storm[1], Flume[2] and Tigon[3] are based on the COM framework. They treat streams as a sequence of events and they are handled by different processing operators. These systems ensure reliable message delivery but they cannot provide strong consistency after recovery. Several fault-tolerant schemes on the COM framework have been described in the previous section.

Fernandez et al. (2013) assign each data event a key and define the processing state of an operator as a set of key/value pairs. This adjustment on the COM framework allows a stream processing system to partition the state or checkpoint of an operator and scale it out or recover it in a distributed manner. This is orthogonal to our approach described in Section 4 and could be used to reduce recovery times. However, their system only considers the case with one operator failure. Since each operator stores checkpointed state on one of its upstream operators, their system cannot recover if two consecutive operators fail. In contrast, our system can recover even under multiple node failure.

There are also efforts to perform streaming computation by pipelining batch computations. MapReduce Online (Condie and Sears, 2010) supports continuous processing within and across different MapReduce jobs. Spark Streaming (Zaharia et al., 2013) adds support for stream processing to Spark (Zaharia et al., 2012), an in-memory cluster computing system. Spark Streaming decomposes computing jobs in small timescales and stores computing state in a fault-tolerant distributed memory abstraction called Resilient Distributed Datasets (RDDs). RDDs are checkpointed to disks periodically and can be recovered in parallel, therefore Spark Streaming provides reliable fault-tolerance. Trident (Marz, 2014) provides a functional API similar to Spark Streaming, while it stores state in a replicated database to provide fault-tolerance, which is more expansive than RDDs. Both Spark Streaming and Trident use the micro-batching model which allows them to treat a stream as a sequence of small batches. While the micro-batching model makes the fault-tolerance easier, both systems suffer from higher latency as compared to other systems that are based on the continuous operator model (Zaharia et al., 2013). Moreover, both systems only support certain pre-defined operators (e.g., map, join, and aggregate) and are not as extensive and scalable as our system.

Note that above architectures are designed for general purposes. In the context of network traffic processing, cSAMP (Sekar et al., 2008) distributes processing across routers and uses a centralized node for global optimization. RouteBricks (Dobrescu et al., 2009) employs a high-speed extensible software-based router by distributing

[2] Apache Flume. http://flume.apache.org.

[3] Tigon – 100% open source low latency, high-throughput streaming technology, http://tigon.io/.

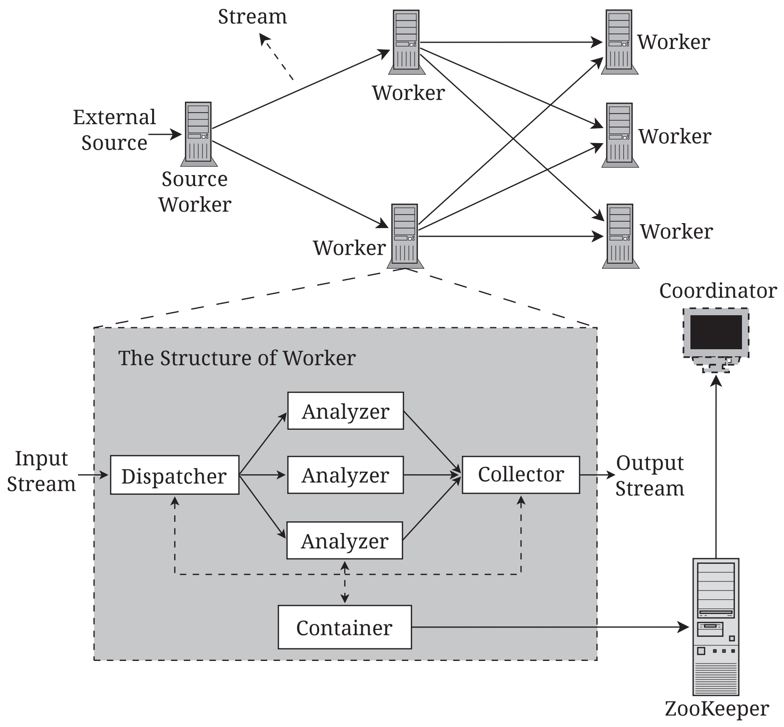

Fig. 3. SAND architecture.

traffic across multiple multi-core machines. Gigascope (Cranor et al., 2003) is a stream processing system for network monitoring that provides a programmable architecture via SQL-like query language. Blockmon (Huici et al., 2012) schedules CPUs and communication channels to achieve high-performance message passing. Our work complements the above studies by also providing strong consistency for critical network analytics applications like real-time charging.

3. System design of SAND

In designing SAND, we set two design goals. First, it has to sustain high input traffic rates (i.e., at Gb/s 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 (Neumeyer et al., 2010) and Storm[1], are implemented on Java Virtual Machine (JVM), so it is inefficient to develop and execute non-JVM operators on these two systems. Furthermore, Storm has high processing overheads and it cannot sustain high-speed network traffic (please refer to Section 6), while the performance of S4 is even much lower than Storm (Zaharia et al., 2013). 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.

SAND uses the continuous operator model for two reasons: (1) in network analytics, packets are processed one by one which is similar to data events in the continuous operator model, so it is natural for users to develop new network analytics applications under the continuous operator model; (2) the micro-batching model (Zaharia et al., 2013) suffers higher latency which is undesirable in many time critical applications like intrusion detection. Fig. 3 depicts the architecture of SAND. 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 2. 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*.

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 (Hunt et al., 2010) 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

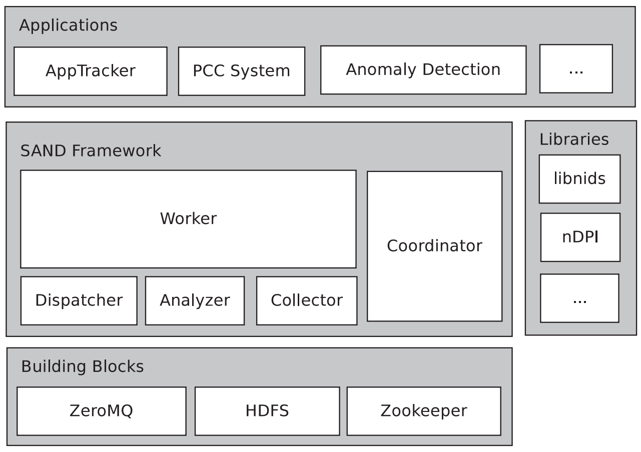

Fig. 4. Technology stack of SAND.

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 buffers. 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 Gb/s 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* (Lamport, 1977) to avoid lock contention which may degrade system performance. Workers communicate via ZeroMQ[4], which is a messaging library optimized for high-throughput communication. We use ZeroMQ sockets instead of TCP sockets because (1) it provides more reliable connections on top of raw TCP sockets; (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 Fig. 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.

In Fig. 4, we summarize the technologies and libraries used in SAND. As described previously, we connect all workers using ZeroMQ and the coordinator manages each worker through a Zookeeper cluster. For our fault-tolerant scheme, the workers need to store checkpoints to a distributed file system, HDFS (Shvachko et al., 2010). We will elaborate this in the next section.

Users can build various network analytics applications on top of SAND. Currently, we also provide some basic workers including *Spout*, which reads network traffic packets from a network interface or a file, and *Decoder* which decodes TCP/IP packets. The APIs of SAND are simple and flexible. Usually, to build a worker, users only need to implement the `process()` method of the analyzer and define state variables in the analyzer. Then, `process()` is invoked for each incoming data event of the worker. In `process()`, users can handle each data event, modify internal state, and use the `emit()` method to send new data events to other workers. To build a new worker, users can also employ existing C++ libraries in the analyzer. For example, in the Decoder, we perform IP defragmentation and TCP stream assembly using libnids[5]; to implement a worker that performs DPI, we can use an open source library like nDPI[6].

4. Fault-tolerance and 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 4.1) and failure recovery (Section 4.2). Finally, we prove its correctness (Section 4.3) and discuss its limitations (Section 4.4).

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 (Shvachko et al., 2010), 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.

4.1. 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 Fig. 5, 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. The checkpointing protocol is similar to the

[4] Distributed Computing made Simple – zeromq, http://zeromq.org/.

[5] libnids, http://libnids.sourceforge.net/

[6] nDPI – Open and Extensible LGPLv3 Deep Packet Inspection Library. http://www.ntop.org/products/ndpi/.

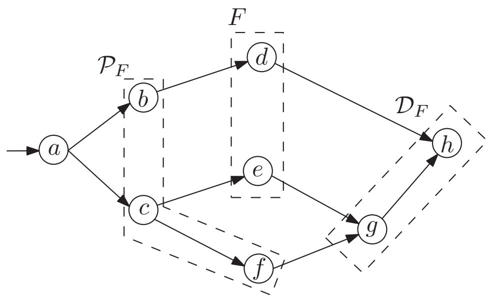

Fig. 5. An example of our failure recovery process.

Chandy–Lamport snapshot algorithm (Chandy and Lamport, 1985). 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 Fig. 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 μ_a, α_1, μ_b, β_1, α_2, β_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 2, when worker d processes duplicate events like γ_1', it will not cause inconsistent state. Our checkpointing algorithm possesses the following property.

Property 4.1. *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. Suppose worker u sends a data event E to worker w after worker u starts the checkpoint procedure of checkpoint c. When data event E arrives worker w, worker w has already received the anchor event from u and `flag[u]` was set to true. Term 3(c) of our protocol ensures that 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 6.

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.

4.2. Failure recovery

Next, we introduce how to recover the internal state of affected workers when some workers fail. 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, the Zookeeper will discover the failure because it will not receive the heartbeat signal from that worker, and so it 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 2, 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 as $\mathcal{P}_F = \mathcal{U}_{\mathcal{R}_F} - \mathcal{R}_F$. 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 Fig. 5, 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 $\mathcal{P}_F = \{b, c, f\}$.

4.3. Proof of consistency

As described in Section 2, 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 4.2. 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 Fig. 5, 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 4.3. *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 4.4. *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. Next, we 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 4.1, 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 4.3, workers in $W - \mathcal{R}_F = \mathcal{G}_F$ are consistent. Thus we complete the proof by the above claim. □

By Theorems 4.3 and 4.4, all workers in the SAND cluster are consistent after recovery.

4.4. Discussion and limitations

In this section, we discuss the limitations of SAND, in particular, we state the two failure scenarios from which it cannot recover consistently. First, our fault-tolerant approach cannot recover from the failure in source workers. Note that the data source of a source worker (e.g., a in Fig. 5) is typically external to the stream processing system, hence SAND has no control over it so there is no guarantee that the external data source can replay data during failure recovery. Thus when a source worker fails, the SAND can only recover its internal state from the most recent checkpoint, and data losses may happen. We believe this is a reasonable assumption since any streaming architecture has no control over the external source.

Secondly, if an output buffer is overloaded, data losses may happen because the dropped data events cannot be replayed during failure recovery. Let us consider how we can reduce the probability of buffer overflow. Consider the output buffers of the worker w. During normal processing, the output buffers of worker w need to store all data events generated by worker w between two consecutive checkpoints. When one of downstream workers of w fails, because input streams keep coming in, worker w's output buffers also need to store these newly generated data events during the recovery time. This implies that the size of worker w's output buffers must be larger than the size of data events generated during the checkpointing interval and the recovery time. In other words, we have the following relationship:

$$\text{size of output buffers} \geq (\text{checkpointing interval} \\ + \text{recovery time}) \times \text{throughput}.$$

In Experiment 3 of Section 6, we show that, in the worst case, the recovery time is roughly equal to the checkpointing interval. The reason is that most of the recovery time is used to process replayed data events which are limited by the checkpointing interval. Hence, we have:

$$\text{size of output buffers} \geq 2 \times \text{checkpointing interval} \times \text{throughput}.$$

Hence, once we know the available main memory for output buffers and the desired throughput, we can choose a proper checkpointing interval accordingly to avoid overloading output buffers.

5. Network analytics applications

In this section, we present two real network streaming and analytic applications using the COM framework and show how to use SAND to perform the traffic analysis.

5.1. Policy and Charging Control in cellular networks

In recent years, Policy and Charging Control (PCC) is receiving a lot of attention due to its functionalities in 3G or 4G mobile broadband networks. PCC can play a variety of roles (Finnie, 2011), e.g., it can be used to guarantee bandwidth for key applications, or to assure fair usage of network bandwidth, or to ensure appropriate online charging based on user subscription data. In essence, PCC is a software node which can aggregate information from cellular core networks and other sources in real-time. We design a new application called `SmartPCC` as an extended PCC service. `SmartPCC` helps mobile operators to provide better user experience by reducing users worry of potential huge mobile data bill. For example, when a user overuses his mobile data traffic because he has been subscribing to video streaming services, his mobile phone bill will be high. This form of overuse is quite common and it creates a lot of grievances among mobile users, which in turn generates a lot of complaints to the mobile operators. `SmartPCC` monitors quota balance and network traffic usage of a user in real-time. When a user's behaviors trigger some processing rule (e.g., quota exceeded), `SmartPCC` sends signals to an external component called the Real-Time Decision (RTD). Then the RTD component takes actions like sending alert to the user. For example, when the user is about to run out of mobile data quota, `SmartPCC` notifies the RTD component to send a warning SMS to the user about impending condition of exceeding mobile quota. Or when the user exceeds mobile data quota, `SmartPCC` notifies the RTD component to provide temporary free additional quota for the user, so the user can have some time to decide if he needs to purchase additional data plan for the video streaming service. We design

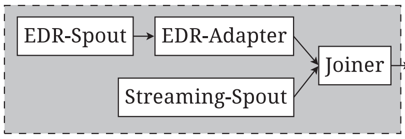

Fig. 6. SmartPCC: a real-world example of streaming applications.

a SmartPCC over the COM framework and it is illustrated in Fig. 6. Furthermore, SmartPCC is implemented on the top of SAND.

SmartPCC receives two different types of events from the cellular core network:

- *Event Data Record (EDR):* When users access the Internet through cellular networks, the accounting component in the network periodically reports the usage of users' data plans to SmartPCC. The content of an EDR includes timestamp, user's MSISDN, the name of the data plan, quota limit, quota usage, etc. By analyzing EDRs, SmartPCC can perform real-time monitor on the quota usage of each user.
- *Streaming event:* When users subscribe to a video streaming service in cellular networks, the probe device in the cellular network monitors video streaming behaviors and periodically sends streaming events to SmartPCC. The content of a streaming event includes timestamp, user's MSISDN, address of the video, etc. By analyzing streaming events, SmartPCC can determine whether a user is using video streaming service.

Next, we elaborate the functionality of each worker in SmartPCC under the COM framework. As shown in Fig. 6, SmartPCC is composed of four workers: (1) *EDR-Spout*, (2) *Streaming-Spout*, (3) *EDR-Adapter*, and (4) *Joiner*. The EDR-Spout and Streaming-Spout read EDRs and streaming events from external sources respectively. The EDR-Adapter receives EDRs from the EDR-Spout, filters out EDRs whose quota usage is less than 90% of the quota limit, and dispatches EDRs to the Joiner. The core component of SmartPCC is the Joiner that receives EDRs and streaming events from the EDR-Adapter and Streaming-Spout as input, respectively. The Joiner parses all the input events and caches each EDR in a 15-min sliding time window format. When a streaming event is received, the Joiner correlates it with cached EDRs which have the same user's MSISDN. The Joiner has three processing rules:

1. If the Joiner receives a streaming event and the user's quota usage is equal to or larger than 90% of the quota limit, the Joiner sends signal *a* with the user's MSISDN to the RTD component. After receiving signal *a*, the RTD component sends an SMS to alert the user that he is about to run out of quota.
2. If the Joiner receives a streaming event and the user has run out of quota already, the Joiner sends signal *b* to the RTD component. After receiving signal *b*, the RTD component sends another SMS to the user and provides the user with some temporary free additional quota.
3. If the Joiner receives a streaming event and the user has run out of free quota provided by the mobile operator, the Joiner sends signal *c* to the RTD component. After receiving signal *c*, the RTD component sends the final SMS alert to the user as a final warning that the user has run out of quota and needs to pay extra charge so to continue to use the mobile data service.

For SmartPCC, it is critical to provide strong consistency after failure recovery, because the output of SmartPCC has a great impact on the purchasing decisions of users and the correctness of output is important for the user experience. In Section 6 Experiment 4, we will present the experimental results of SmartPCC.

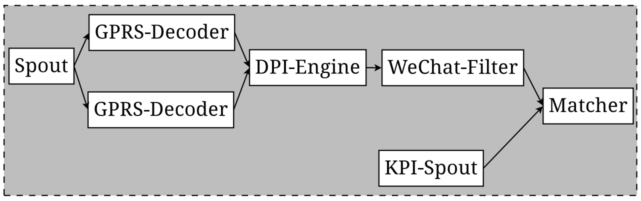

Fig. 7. FlowControl.

5.2. WeChat flow control

FlowControl is another example of extended PCC services which can help mobile operators to prioritize key applications (e.g., voice call) in cellular networks. WeChat[7] is one of the most popular mobile text and voice messaging communication services in China. Voice messages sent through WeChat incur high volume of data transmission and occupy many wireless channel resources in 2G networks, this translates to a negative impact on the quality of voice call. Mobile operators want to assure the quality of voice calls by limiting the bandwidth of data transmission. We consider FlowControl, which monitors network traffic incurred by WeChat voice messages and the rate of voice call channel of each Base Station Controller (BSC). When the rate of voice call channel of a BSC is high, FlowControl notifies the BSC to limit the rate of data transmission so to assure the quality of voice calls.

As shown in Fig. 7, one can compose FlowControl using the COM framework. Furthermore, FlowControl is more complex than previous applications and it reuses three workers of AppTracker described in Section 2. In SAND, we provide many useful workers for packet capture, packet decoding, DPI, filtering, and window join. Users can exploit existed workers to compose their own applications. The *Spout* captures network traffic packets from the GPRS core network. Then the *DPI-Engine* classifies decoded packets into application-level protocols. The *WeChat-Filter* filters out irrelevant packets and generates *WeChat events*. The content of a WeChat event includes timestamp, WeChat voice message size, user's MSISDN, Location Area Code (LAC), and Cell ID. The user's MSISDN, LAC, and Cell ID are extracted from the GPRS header of each packet by the *GPRS-Decoders*. BSCs periodically send BSC Key Performance Indicator (KPI) events to the *BSC-KPI-Spout*. The content of a BSC KPI event includes timestamp, BSC ID, and the rate of voice call channel traffic.

The *Matcher* receives WeChat events and BSC KPI events from the WeChat-Filter and BSC-KPI-Spout, respectively. The Matcher stores the rate of voice call channel of each BSC extracted from BSC KPI events. For each WeChat event, the Matcher first uses its LAC and Cell ID to locate the BSC that rigger this event. Then the Matcher calculates the total volume of WeChat voice messages sent from this BSC in the last 5 min. For each BSC, if the rate of voice call channel exceeds thresholds t_1 and the network traffic incurred by WeChat voice messages exceeds thresholds t_2, the Matcher notifies the BSC to limit the rate of data transmission.

5.3. Real-time heavy hitter detection

Real-time identification of significant patterns in network traffic, such as TCP/IP flows which contribute large volumes of network traffic (heavy hitters), or those flows which introduce large change in volumes of network traffic (heavy changers) (Estan and Varghese, 2003), is critical for anomaly detection. For example, a flow that accounts for more than 1% of the total traffic may suggest over-usage

[7] WeChat is nothing like WhatsApp – and that makes it even more valuable, http://qz.com/179007/wechat-is-nothing-like-whatsapp-and-that-makes-it-even-more-valuable/.

of network bandwidth. However, as the Internet continues to grow, identifying heavy hitter/heavy changer becomes challenging due to both the computational overheads and memory requirements. To improve scalability, anomaly detection needs to be performed on distributed stream processing systems. However, previous techniques are mainly studied and evaluated in a single-processor or single machine setting. In this work, We implement four state-of-the-art heavy hitter and heavy changer detection algorithms on the top of SAND. These algorithms are: *Combinational Group Testing* (CGT) (Cormode and Muthukrishnan, 2005), *SeqHash* (Bu et al., 2010), *Fast Sketch* (Liu et al., 2012), and *LD-Sketch* (Huang and Lee, 2014).

We implement these four algorithms in a single worker as shown in Fig. 3. For each algorithm, the dispatcher reads raw packets and forwards them to selected analyzers. Each analyzer updates its internal data structure (e.g., sketch) when it receives a packet. The collector summarizes and outputs anomalies. Note that CGT, SeqHash, and Fast Sketch are designed for the single-processor setting, and their accuracy decrease as the number of analyzers increase. On the other hand, LD-Sketch is designed for distributed systems and can exploit the distributed nature to improve detection accuracy by aggregating detection results from multiple analyzers (Huang and Lee, 2014). In Section 6 Experiment 4, we will present the experimental results of these heavy hitter and heavy changer detection algorithms.

6. Performance evaluation of SAND

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 of SAND under high input traffic)**.** First, we compare SAND with two open source stream processing systems, Storm[1] and Blockmon (Huici et al., 2012), 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 4 GB RAM. We collect a packet header trace from CAIDA[8]. The trace lasts for 21 min in the PCAP format, and contains 331 million packets accounting for a total of 143 GB 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 2 GB of the trace file and replay the loaded chunk for 90 s. 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 1. SAND achieves the highest throughput of 31 Gb/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

Table 1
Performance of Storm, Blockmon and SAND.

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

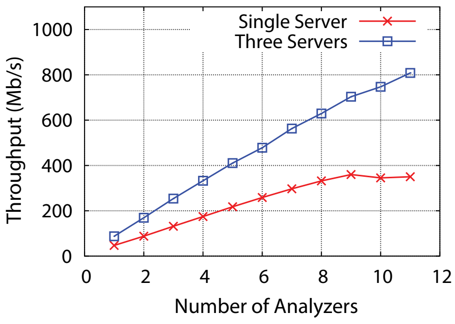

Fig. 8. Scalability of SAND.

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.

Experiment 2 (Scalability of SAND)**.** We use `AppTracker` described in Section 2 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 1 Gb/s LAN. Each server has 16 physical 2.10 GHz CPU cores and 94 GB RAM. We run our evaluation on a 2-h network trace (32 GB), 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 2, 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 Fig. 8, 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 Fig. 8, 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. Note that in the case of three servers, S2 and S3 are dedicated for GPRS-Decoder and SAND can scale up until we run eleven analyzers in each GPRS-Decoder. While in the case of single server, the throughput of SAND reaches the maximum

Table 2
Result of `AppTracker`.

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

[8] The CAIDA Anonymized Internet Traces 2009 Dataset, http://www.caida.org/data/passive/passive_2009_dataset.xml.

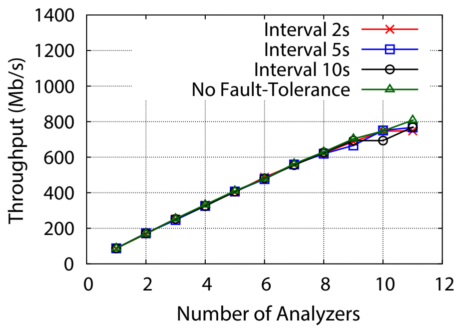

Fig. 9. Overheads of fault-tolerant scheme.

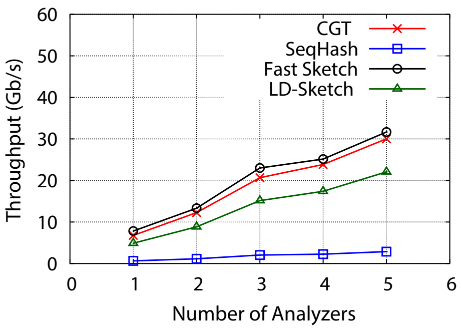

Fig. 10. Heavy hitter detection algorithm comparison.

for nine analyzers in GPRS-Decoder, because the server must reserve some computation resource for other workers.

Experiment 3 (Fault-tolerance of SAND)**.** 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 Fig. 9, 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 Fig. 11. First, we set the checkpointing interval as $T = 5$ s. 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 $T = 10$ s. We terminate the Tracker at t_1. In this scenario, SAND can recover in 3 s. However, when we kill the GPRS-Decoder at t_4, it takes around 11 s 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 Fig. 11 that the recovery time at t_4 is longer than t_2, t_3 and t_5.

Experiment 4 (Using SAND for real-time heavy hitter detection)**.** We use the four heavy hitter detection algorithms described in Section 5.3 to show how to implement and scale up anomaly detection algorithms on SAND. Our testbed is a multi-core server with 12 physical 2.93 GHz CPU cores and 50 GB 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 600 GB of traffic.

Fig. 10 shows the throughput of the four heavy hitter detection algorithms using multiple analyzers. One can observe that the throughput increases almost linearly as the number of analyzers grows. When using five analyzers, Fast Sketch can reach over 30 Gb/s of throughput. This shows that using SAND, we can scale up the throughput of different traffic anomaly detection algorithms in SAND via parallelization.

Experiment 5 (Using SAND for Policy and Charging Control in cellular networks)**.** We use SmartPCC described in Section 5.1 to demonstrate the extensibility of SAND. Our testbed is S1 in Experiment 2. To drive the experiment, we generate a synthetical trace of EDRs and streaming events on the disk. Similar to previous experiments, in order to eliminate the overhead due to disk read, we load the trace file to the memory at the beginning. SmartPCC on SAND can handle 1.3 million EDRs per second and 0.9 million streaming events per second (2.2 million tuples per second in total). Without stream processing capabilities in the current cellular networks, operators can only implement and deploy PCC systems on specialized hardwares which are not scalable and only have processing throughput with less than 300,000 tuples per second. Furthermore, we cannot implement complex applications like SmartPCC or FlowControl on the top of these specialized hardwares, because they are not extensible like SAND.

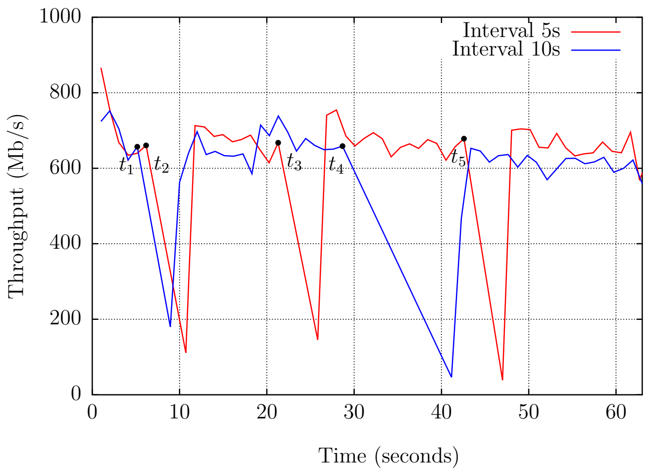

Fig. 11. Recovery time.

7. Conclusion

We have presented SAND, a new distributed stream processing system for network analytics on core networks. SAND is a general purpose stream processing system on which one can build various real-time services. We demonstrated how one can build new services such as heavy network traffic detectors as well as policy and charging control for large scale cellular networks. SAND can sustain high-speed network traffic and provide reliable fault-tolerance. SAND uses a novel checkpointing protocol to provide strong consistency of processing results even when the system experiences multiple node failure. We demonstrated that SAND can operate at core routers level, and the fault-tolerance approach has low overheads and can recover from failure in order of seconds.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.jss.2015.07.049

References

Akidau, T., Balikov, A., 2013. MillWheel: fault-tolerant stream processing at Internet scale. PVLDB 6 (11), 1033–1044.

Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M., 2008. Fault-tolerance in the borealis distributed stream processing system. ACM Trans. Database Syst 33 (1) Article 3.

Bu, T., Cao, J., Chen, A., Lee, P.P., 2010. Sequential hashing: a flexible approach for unveiling significant patterns in high speed networks. Comput. Netw. 54 (18), 3309–3326.

Chandy, K.M., Lamport, L., 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3 (1), 63–75.

Condie, T., Sears, R., 2010. MapReduce Online. In: Proceedings of Conference on Networked Systems Design and Implementation, NSDI.

Cormode, G., Muthukrishnan, S., 2005. What's new: finding significant differences in network data streams. IEEE/ACM Trans. Netw. 13 (6), 1219–1232.

Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V., 2003. Gigascope: a stream database for network applications. In: Proceedings of ACM International Conference on Management of Data, SIGMOD. ACM, pp. 647–651.

Dobrescu, M., Egi, N., Argyraki, K., Chun, B.G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., Ratnasamy, S., 2009. RouteBricks: exploiting parallelism to scale software routers. In: Proceedings of ACM Symposium on Operating Systems Principles, SOSP, vol. 9. CiteSeer.

Estan, C., Varghese, G., 2003. New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. ACM Trans. Comput. Syst. 21 (3), 270–313.

Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P., 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In: Proceedings of ACM International Conference on Management of Data, SIGMOD, pp. 725–736.

Finnie, G. (2011). Mobile Broadband & the Rise of Policy: Technology Review & Forecast. Technical Report. Heavy Reading.

Gu, Y., Zhang, Z., Ye, F., Yang, H., Kim, M., Lei, H., Liu, Z., 2009. An empirical study of high availability in stream processing systems. In: Middleware.

Huang, Q., Lee, P.P.C., 2014. LD-Sketch: a distributed sketching design for accurate and scalable anomaly detection in network data streams. In: Proceedings of IEEE Conference on Computer Communications, INFOCOM.

Huici, F., Di Pietro, A., Trammell, B., Gomez Hidalgo, J.M., Martinez Ruiz, D., d'Heureuse, N., 2012. Blockmon: a high-performance composable network traffic measurement system. SIGCOMM CCR 42 (4), 79–80.

Hunt, P., Konar, M., Junqueira, F.P., Reed, B., 2010. ZooKeeper: wait-free coordination for Internet-scale systems. In: Proceedings of USENIX Annual Technical Conference.

Hwang, J.-H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., Zdonik, S.B., 2005. High-availability algorithms for distributed stream processing. In: Proceedings of IEEE International Conference on Data Engineering, ICDE, pp. 779–790.

Lamport, L., 1977. Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng 3 (1), 63–75.

Liu, Q., Lui, J.C., He, C., Pan, L., Fan, W., Shi, Y., 2014. Sand: a fault-tolerant streaming architecture for network traffic analytics. In: Proceedings of DSN.

Liu, Y., Chen, W., Guan, Y., 2012. A fast sketch for aggregate queries over high-speed network traffic. In: Proceedings of IEEE Conference on Computer Communications, INFOCOM.

Marz, N. (2014) Trident tutorial. https://storm.apache.org/documentation/Trident-tutorial.html. Accessed in 2015 August 15.

Neumeyer, L., Robbins, B., Nair, A., Kesari, A., 2010. S4: Distributed stream computing platform. In: Proceedings of IEEE International Conference on Data Mining Workshops, ICDMW.

Sekar, V., Reiter, M.K., Willinger, W., Zhang, H., Kompella, R.R., Andersen, D.G., 2008. cSamp: a system for network-wide flow monitoring. In: Proceedings of Conference on Symposium on Networked Systems Design & Implementation, NSDI, 8, pp. 233–246.

Shah, M.A., Hellerstein, J.M., Brewer, E.A., 2004. Highly-available, fault-tolerant, parallel dataflows. In: Proceedings of ACM International Conference on Management of Data, SIGMOD.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The Hadoop distributed file system. In: Proceedings of Symposium on Mass Storage Systems and Technologies, MSST, pp. 1–10.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I., 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of Conference on Symposium on Networked Systems Design & Implementation, NSDI.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I., 2013. Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of ACM Symposium on Operating Systems Principles, SOSP.

Qin Liu is a Ph.D. student in the Department of Computer Science & Engineering at the Chinese University of Hong Kong starting from 2012 under the supervison of Prof. John C.S. Lui. His research interests are mainly in distributed systems, networking, and algorithms.

John C.S. Lui is currently a full professor in the Department of Computer Science & Engineering at The Chinese University of Hong Kong. He received his Ph.D. in Computer Science from UCLA. After his graduation, he joined the IBM Almaden Research Laboratory/San Jose Laboratory and participated in various research and development projects on file systems and parallel I/O architectures. His current research interests are in Internet, network sciences with large data implications (e.g., online social networks), machine learning on large data analytics, network/system security (e.g., cloud security, mobile security), network economics, cloud computing, large scale distributed systems and performance evaluation theory. John has been serving in the editorial board of IEEE/ACM Transactions on Networking, IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, Journal of Performance Evaluation, Journal of Network Science and International Journal of Network Security. John received various departmental teaching awards and the CUHK Vice-Chancellor's Exemplary Teaching Award. John also received the CUHK Faculty of Engineering Research Excellence Award (2011–2012). John is a co-recipient of the best paper award in the IFIP WG 7.3 Performance 2005, IEEE/IFIP NOMS 2006, and SIMPLEX 2013. He is an elected member of the IFIP WG 7.3, Fellow of ACM, Fellow of IEEE, Senior Research Fellow of the Croucher Foundation and is currently the chair of the ACM SIGMETRICS.

Cheng He is currently a research manager at Huawei Noah's Ark Lab in China. His research interests are in computer networks, wireless networks, and distributed system.

Lujia Pan is currently a senior researcher at Huawei Noah's Ark Lab in China. Her research interests are in computer and communication networks.

Wei Fan is the Director/Deputy Head of Baidu Research Big Data Lab in Sunnyvale, CA. This work was done when he was with Huawei Noah's Ark Lab.

Yunlong Shi is currently a senior engineer at Huawei in China. His research interests are in computer networks, wireless networks, and distributed system.