A Configurable and Executable Model of Spark Streaming on Apache YARN

Jia-Chun Lin*

Department of Informatics,
University of Oslo, 0315 Oslo, Norway
Email: kellylin1219@gmail.com
*Corresponding author

Ming-Chang Lee

Department of Information Security and Communication Technology,
Norwegian University of Science and Technology (NTNU), 2802 Gjøvik, Norway
Email: ming.c.lee@ntnu.no

Ingrid Chieh Yu

Department of Informatics,
University of Oslo, 0315 Oslo, Norway
Email: ingridcy@ifi.uio.no

Einar Broch Johnsen

Department of Informatics,
University of Oslo, 0315 Oslo, Norway
Email: einarj@ ifi.uio.no

Abstract: Streams of data are produced today at an unprecedented scale. Efficient and stable processing of these streams requires a careful interplay between the parameters of the streaming application and of the underlying stream processing framework. Today, finding these parameters happens by trial and error on the complex, deployed framework. This paper shows that high-level models can help determine these parameters by predicting and comparing the performance of streaming applications running on stream processing frameworks with different configurations. To demonstrate this approach, this paper considers Spark Streaming, a widely used framework to leverage data streams on the fly and provide real-time stream processing. Technically, we develop a configurable and executable model to simulate both the streaming applications and the underlying Spark stream processing framework. Furthermore, we model the deployment of Spark Streaming on Apache YARN, which is a popular open-source distributed software framework for big data processing. We show that the developed model provides a satisfactory accuracy for predicting performance by means of empirical validation.

Keywords: Modeling; Simulation; Spark Streaming; Apache YARN; batch processing; stream processing; ABS

Reference to this paper should be made as follows: Lin, J.-C., Lee, M.-C., Yu, I.C., and Johnsen, E.B. (2018) 'A Configurable and Executable Model of Spark Streaming on Apache YARN', *Int. J. Ad Hoc and Ubiquitous Computing*, Vol. X, No. Y, pp.000–000.

Biographical notes: Jia-Chun Lin is currently a postdoctoral research fellow at Department of Informatics, University of Oslo, and will soon work at Norwegian University of Science and Technology (NTNU), Norway. Her research interests include parallel and distributed computing, cloud computing, Fog computing, Internet of Things, big data analytics, data mining, and machine learning.

Ming-Chang Lee is currently a postdoctoral research fellow at Norwegian University of Science and Technology (NTNU), Norway. His research interests include cloud computing, big data, deep learning, intelligent systems, HPC, Security, and Privacy.

Ingrid Chieh Yu is an associate professor at Department of Informatics, University of Oslo. Her research interests include specifications and formal modeling, and methodology for the design and analysis of concurrent, distributed, context-dependent and evolving systems.

Einar Broch Johnsen is a full professor at Department of Informatics, University of Oslo. His research interests include programming models and methodology; program specification and modeling; formal methods and associated theory; lightweight analysis, type systems, testing; as well as deductive verification and formal logic.

1 Introduction

Streams of data are produced today at an unprecedented scale. The leverage of this data requires stream processing solutions which can scale transparently to large amounts of data and complex workflows, and process the data on the fly. State-of-the-art stream processing frameworks include Spark Streaming (Spark Streaming), Apache Storm, AWS Kinesis Streams, IBM InfoSphere Streams, and TIBCO StreamBase. Achieving stable and efficient processing for a streaming application requires a careful interplay between the configurations of the underlying stream processing framework and the streaming application itself. Inappropriate configurations and deployment decisions may lead to wasteful resource overprovisioning or poor performance, both of which may substantially increase costs.

Using model-based analyses, appropriate configurations and deployment decisions can be explored and compared "in the laboratory" (Hähnle and Johnsen, 2015), thereby helping users to predict the performance of an application before the application is deployed. Our goal is to give users an easy-to-use support for such analyses based on a highly configurable and executable model. For this purpose, the configurable model needs to be faithful to the real systems in the sense that the model-based analysis of an application's deployment complies with the performance of its actual deployment.

In this paper, we present a highly configurable model of Spark Streaming based on Real-Time ABS (Bjørk et al., 2013; Johnsen et al., 2015). Spark Streaming is a widely used stream processing framework due to its scalability, efficiency, resilience, fault tolerance, and compatibility with several different cluster/cloud platforms, including Spark standalone clusters, Apache YARN (Vavilapalli et al., 2013), Amazon EC2 (Amazon EC2), and Apache Mesos (Amazon Mesos). Real-Time ABS is a formal executable language for modeling distributed systems and deployed virtualized software. Our model SSP (which stands for Spark Streaming Processing) models Spark streaming applications, the underlying Spark stream processing framework, and the execution of the applications on the framework. SSP allows users to not only specify the workflow of their streaming applications, but also to configure crucial parameters for the underlying stream processing framework. We further model the deployment of Spark Streaming on Apache YARN (Vavilapalli et al., 2013), a popular open-source distributed software framework for big data processing, by integrating SSP with a modeling framework for YARN (Lin et al., 2016).

To validate the faithfulness of the proposed model, we compare performance as simulated using our model with performance as observed on Spark Streaming on Apache YARN to see how well our model can simulate and reflect streaming applications under different scenarios. The main contributions of this paper can be summarized as follows:

- Formalization: SSP is a formal executable model targeting a state-of-the-art stream processing framework, i.e., Spark Streaming. We show how Spark Streaming can be deployed on a popular distributed big-data processing framework, i.e., YARN, from a modeling perspective.
- Configurable modeling framework: The SSP model we proposed in this paper can be applied to when users need to make configuration decisions for their streaming applications. SSP enables users to configure stream processing workflows and experiment with different crucial parameters for the underlying stream processing framework at the abstraction level of the model. By means of simulations, users can easily compare how different parameter configurations affect the performance of their streaming applications.
- Evaluation: The proposed model is validated through several scenarios in which dynamic data arrival patterns and different parameter configurations are considered. The results suggest that our model provides a satisfactory modeling accuracy compared with Spark Streaming on YARN.

Paper overview. Section 2 introduces Spark Streaming and Real-Time ABS. Section 3 presents SSP. Section 4 shows how to model the deployment of Spark Streaming on YARN by extending SSP. Sections 5, 6, and 7 discusses model validation, surveys related work, and concludes the paper, respectively.

2 Background

In this section, we provide background about Spark Streaming and Real-Time ABS.

2.1 Spark Streaming

Spark Streaming (Spark Streaming) is developed as part of the Apache Spark tool suite (Spark: Lightning-fast cluster computing) for high-throughput and fault-tolerant data stream processing. Spark Streaming uses the discretized stream model (i.e., resilient distributed dataset or RDD for short) (Zaharia et al., 2012) to structure stream processing as a series of stateless, deterministic batch computations on small time intervals (Zaharia et al., 2013). Each Spark streaming application is a long-running and non-terminating stream processing service with a Spark driver as its master. Each Spark driver is responsible for initiating the reception of continuously flowing input data, periodically dividing the received data into batches based on a pre-configured batch interval, and managing the processing of each batch. A batch may contain zero or more data items depending on the data arrival frequency and the batch interval. A batch with no data is called an empty batch. Otherwise, it is called a non-empty batch.

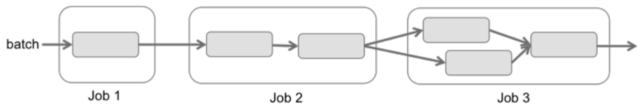

Figure 1. An example of batch processing workflow.

In Spark Streaming, each batch is processed by the workflow of a streaming application defined by the application designer. In this paper, the workflow is called batch processing workflow. In general, a batch processing workflow consists of a series of sequential jobs. Each job, which is triggered by an action, may have several sequential or parallel stages. For example, the workflow shown in Figure 1 comprises 3 jobs where job 1 has one stage, job 2 has two sequential stages, and job 3 have both sequential and parallel stages. Each Spark driver also controls the maximum number of jobs that are allowed to be concurrently processed, which has a default value of one (Apache Spark JobScheduler).

Spark Streaming can be deployed on different clusters or on the cloud, including Spark standalone clusters (Spark Streaming), Apache YARN (Vavilapalli et al., 2013), Amazon EC2 (Amazon EC2), and Apache Mesos (Apache Mesos). After it is deployed, the Spark driver of a streaming application can request a set of worker nodes (which may be physical machines, virtual machines, or containers) to start the application and process each batch on these workers according to the corresponding workflow.

2.2 Real-Time ABS

In this subsection, we introduce Real-Time ABS and explain why we chose this language to model Spark Streaming. Real-Time ABS (Bjørk et al., 2013; Johnsen et al., 2015) is a formal, executable, object-oriented language for modeling distributed systems and deployed virtualized software. This language combines functional and imperative programming styles with a Java-like syntax and a formal semantics. The first reason we chose this language is because it supports concurrent object groups. A concurrent object group is a group of objects. Concurrent object groups execute in parallel and communicate by asynchronous method calls and futures (Johnsen et al., 2011). Objects execute processes that stem from method calls. At any time, at most one process in a concurrent object group is active, whereas inactive processes are suspended and stored in a queue waiting to be executed on an object of the group. This cooperative scheduling of processes allows active and reactive behaviors to be combined in the object groups. Internal computations in an object are captured in a simple functional language based on user-defined algebraic data types and functions.

The second reason we chose this language is that communication and synchronization in Real-Time ABS are decoupled. Communication is based on asynchronous method calls on the form $f = o!m(e)$ where f is a future variable, o an object expression, m a method name, and e the parameter values for the method invocation. These calls are non-blocking: After calling $f = o!m(e)$, the caller may proceed with its execution without blocking on the method reply. Synchronization is controlled by operations on futures. The statement **await** f? releases the processor while waiting for a reply, allowing other processes to execute. When the reply arrives, the suspended process becomes enabled and the execution may resume. The return value from the method call is retrieved by the expression f.**get**, which blocks all execution in the object until the return value is available. The syntactic sugar x = **await** $o!m(e)$ encodes the standard pattern $f = o!m(e)$; **await** f?; $x = f$.**get**.

The third reason is because Real-Time ABS supports the timed behavior of concurrent objects, which is captured by a maximal progress semantics (Bjørk et al., 2013). Execution time can be specified explicitly by means of duration statements, or be implicit in terms of observations on the executing model. The statement **duration** (e_1; e_2) will cause time to advance between the best case e_1 and the worst case e_2 execution time, blocking all execution in the concurrent object group until time has advanced. The statement **await duration** **(**e_1;e_2**)** will suspend the process until time has advanced beyond e_1, allowing other processes to be scheduled.

The last reason is that Real-Time ABS is able to model deployment by introducing a separation of concerns between the resource cost of executing a task and the resource capacity of the location where the task executes (Johnsen et al., 2015). A resource cost expression e can be associated with a statement s by a cost annotation [Cost: e] s. This allows the execution time of the statement s to depend on the location where it is executed. To model resource-constrained deployment architectures, Real-Time ABS uses deployment components to model locations with given resource specifications. A number of concurrent objects can be deployed on each deployment component. In other words, each deployment component has its own execution speed, which determines the performance of the objects running on it. A deployment component is created by the statement x= **new** DeploymentComponent (descriptor, capacity), where x is typed by the DC interface, descriptor is a descriptor for the purpose of monitoring, and capacity specifies the resource capacity of the deployment component. Using the DC annotation on the statement of creating an object, the object can be deployed on the corresponding deployment component.

3 The SSP Model

In this section, we introduce how the SSP model is defined and written in Real-Time ABS to model Spark stream applications and the underlying processing framework.

3.1 Modeling Spark Stream Applications

A batch is represented by a Real-Time ABS data type with an identifier bID and an associated size bSize. In datatype definitions, the parameter names of a constructor become accessor functions; e.g., bSize(Batch(1,5)) reduces to 5. We define a Boolean function isEmptyBatch to recognize empty batches:

```
type BatchID = Int;
data Batch = Batch(BatchID bID, Int bSize);
def Bool isEmptyBatch(Batch batch)=(bSize(batch)==0);
```

Recall that a batch processing workflow in Spark Streaming consists of a series of jobs and each job has several stages. In this paper, we model such workflow from the job level. by means of two datatypes Workflow and JobInfo. The former defines a batch processing workflow as a list of jobs. The latter defines a job as an unique identifier and a list of constraints constr for executing the job.

```
type JobID=String;
data Workflow=Workflow(List<JobInfo> jobs);
data JobInfo=JobInfo(JobID jID,List<JobID> constr);
```

For instance, the batch processing workflow depicted in Figure 1 has three jobs, so the workflow corresponds to the ABS term Workflow [JobInfo(J1,Nil), JobInfo(J2,list["J1"]), JobInfo(J3,list["J2"])]. To guarantee that every job in a batch processing is executed in accordance with its constraints, we define a Boolean function check:

```
def Bool check <A>(List<A> constr, List<A> fin)=
  case constr {
    Nil => True;
    Cons(cs1,rest) => case element(cs1,fin)
    { True => check(rest,fin);
      False => False;};};
```

For each unprocessed job, this function checks recursively to see if all its constraints are included in fin, which is a list of completed jobs of the batch processing. A job can be executed only when all its constraints have been resolved.

To model the processing time of a batch, users might achieve it at the stage level if they know the cost of executing each stage for each job of a batch processing. Although such approach is fine-grained, it takes a lot of efforts for users to determine stage costs. Therefore, in this paper, SSP focuses on job-level modeling. In other words, users only need to specify a cost for executing each job of a batch processing. Since the time spent by a job to process a non-empty batch and an empty batch are different in Spark Streaming, our model allows users to define two cost approximation function: jobCostNonEmptyBatch and jobCostEmptyBatch. The former assigns an execution cost expression c_i to each job J_i (where i = 1, 2, …, n) that processes an non-empty batch. The latter assigns an execution cost expression to each job that processes an empty batch.

```
def Rat jobCostNonEmptyBatch(JobID jID) =
  case jobID{"J1"=>c1(); "J2"=>c2(); ... ;}

def Rat jobCostEmptyBatch(JobID jID) =
  case jobID{"J1"=>e1(); "J2"=>e2(); ... ;}
```

Note that users can specify cost expressions at an appropriate level of precision based on their preferences. Cost expressions may be derived using cost analysis (e.g., SACO (Albert et al., 2014)) or represent an estimated or average execution time. Later in this paper, we will show how we assign such costs.

3.2 Modeling the Spark Stream Processing Framework

The architecture of SSP, shown in Figure 2, consists of a main block for a user to configure his/her streaming application and the underlying Spark stream processing framework, a class SparkDriver to model the Spark driver of the streaming application, and a class Worker to model worker nodes. Here we assume that a set of worker nodes is available to the stream processing framework. The deployment of Spark Streaming to make such worker nodes available will be modeled in Section 4.

Before launching SSP, users are required to configure the following parameters for their streaming applications:

- the specification of the batch processing workflow
- the execution cost for each job of the batch processing workflow
- the maximum number of worker nodes used to run the application (denoted by num)
- the resource specification of each worker node (denoted by rs)
- data inter-arrival pattern
- batch interval (denoted by bi)
- the maximum number of jobs allowed to execute concurrently (denoted by conJobs)

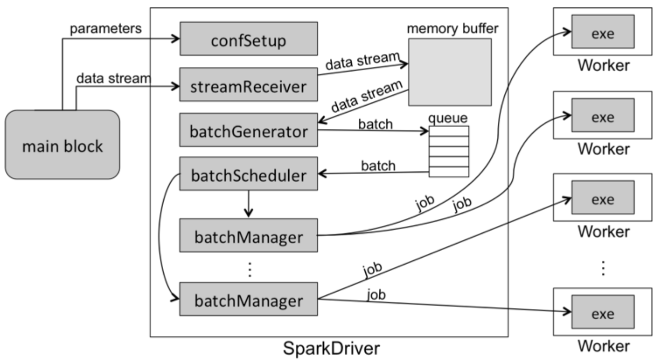

Figure 2. The architecture of SSP.

The SparkDriver interface has the following five methods: confSetup, streamReceiver, batchGenerator, batchScheduler, and batchManager.

```
interface SparkDriver {
  Bool confSetup(Int num, RSpec rs, Int bi, Int conJobs);
  Unit streamReceiver(Int sizeStream);
  Unit batchGenerator();
  Unit batchScheduler();
  Unit batchManager(Batch batch);}
```

When a user launches SSP, method confSetup is invoked to create the requested set of num worker nodes with the requested resource specification rs. Each worker node is modeled by class Worker on an independent deployment component with the resource specification rs of type RSpec:

```
data RSpec = Res(Int cores, Rat speed, Int memory)
```

Here, cores, speed, and memory are accessor functions to the values of the respective resources in the resource specification. Class Worker uses method exe to execute one job of a batch processing with the corresponding cost expression. Exploiting the resource-constrained semantics of Real-Time ABS, the job execution time is determined by the cost expression and the CPU processing speed of the worker.

After all the worker nodes have been created, method confSetup triggers methods streamReceiver, batchGenerator, and batchScheduler to start running the streaming application. As illustrated in Figure 2, streams of data are sent to method streamReceiver based on the pre-configured data inter-arrival pattern. Upon receiving data, method streamReceiver keeps it in the memory buffer of the Spark driver. Method batchGenerator periodically generates batches based on the pre-configured batch interval and inserts these batches into a queue. Method batchScheduler schedules batch processing by invoking method batchManager for each batch so that the batchManager is able to manage the corresponding batch processing. In the following, we here focus on describing methods batchGenerator, batchScheduler, and batchManager.

Method batchGenerator

This method periodically generates a batch at the rate given by the batch interval bi. Using the await duration statement of Real-Time ABS, this method suspends its execution and resumes again after the time interval bi. Each time, method batchGenerator considers all received data in the memory buffer as a batch, inserts the batch into the queue, and empties the memory buffer to avoid reprocessing the same data. Each batch has a unique bID and has size bSize where bSize=dataSizeInBuffer, i.e., total size of the data in the memory buffer.

```
Unit batchGenerator (){
  Int bID=1; Int bSize=0;
  while (True){
    await duration(bi,bi);
    bSize=dataSizeInBuffer;
    queue=appendright(queue,Batch(bID,bSize));
    dataSizeInBuffer = 0; bID=bID+1;
  }}
```

Method batchScheduler

Similar to batchGenerator, method batchScheduler is also a non-terminating process. It uses the default first-in-first-out scheduling approach to schedule the execution of batches. As long as the total number of currently running jobs is less than conJobs (i.e., runningJobs < conJobs) and there is an unprocessed batch in the queue (i.e., length(queue)>0), method batchScheduler invokes method batchManager to create a separate object for managing the processing of the head-of-queue batch. Note that the await-statement suspends the active process in Real-Time ABS and therefore allows SparkDriver to interleave the different stream processing activities in a flexible way.

```
Unit batchScheduler (){
  while (True){
    await (runningJobs<conJobs);
    await (length(queue)>0);
    Batch batch = head(queue);
    this!batchManager(batch);
    queue=tail(queue);
  }}
```

Method batchManager

When batchManager is requested to manage a batch processing, it retrieves the corresponding batch processing workflow. As long as there are unfinished stages (i.e., length(fin)<totalJobs), batchManager checks if any of these jobs can be executed immediately based on the corresponding constraints. When the constraint of a job is resolved (i.e., check(constr(s),fin) evaluates to True), batchManager awaits until there is available worker in the workerList. When this condition is satisfied, method batchManager executes the job on the worker by invoking the corresponding exe method. At this point, the global parameter runningJobs increases by one. When the job is finished, runningJobs decreases by one, which enables batchScheduler to schedule another batch processing. By repeating the above process, batchManager is able to manage the entire workflow of the batch processing. When all the jobs are finished, the batch processing is complete.

```
Unit batchManager (Batch batch){
 List<JobID> fin=Nil;
 List<JobInfo> jobList=jobs(wf);
 Int totalJobs=length(jobList);
 while (length(fin)<totalJobs){
   JobInfo s=head(jobList);
   JobID jobID=jID(s);
   if (check(constr(s), fin)==True){
     await (length(workerList)>=1);
     Container wr=head(workerList);
     workerList=tail(workerList);
     Bool res=False;
     if (isEmptyBatch(batch)==True){
       wr!exe(jobCostEmptyBatch(jID(s)),bID(batch),jobID);
     }
     else{
       wr!exe(jobCostNonEmptyBatch(jID(s)),bID(batch),jobID);
     }
     runningJobs=runningJobs+1;
     //... await until the job is finished.
     fin=appendright(fin,jID(s));
     jobList=tail(jobList);
     runningJobs=runningJobs-1;
   }
 }
 if (length(fin)==totalJobs){//The batch processing is finished.
   ... //Record the time info of the batch processing.}}
```

4 Modeling Spark Streaming on YARN

In this section, we model the deployment of Spark Streaming on YARN to provision the stream processing framework with worker nodes. This is done by integrating SSP with ABS-YARN, which is an executable framework for modeling and simulating Apache YARN. ABS-YARN (Lin et al., 2016) focuses on modeling the following four important components of a YARN cluster:

- ResourceManager (RM): It is the master server in a YARN cluster to manage all resources and allocate resources to different completing applications, such as MapReduce, Spark Streaming, graph applications, etc.
- Slave nodes: Each slave node provides computation resource and storage capacity.
- Containers: Each container is a logical resource combination (e.g., 1 CPU core and 1 GB memory) of a particular slave node to execute a task.
- MapReduce Application Master (MapReduceAM): It is

the master of a MapReduce application to request containers from RM and manage the execution of the MapReduce application.

RM is modeled by a class RM with four methods: initialization, getContainer, free, and logger. Method initialization is used to initialize the YARN cluster, including RM and the slave nodes. Each slave node is modeled as a database record with a unique SlaveID and an initial resource specification. After initialization, the cluster is ready to serve client requests. Method getContainer allows a MapReduceAM (modeled by class MapReduceAM) to request containers from RM. When invoked, getContainer tries to allocate desired containers from the available slave nodes to the caller. This method may fail if no slave node has sufficient resources to meet the container resource requirements within a certain time. When a container finishes its task, method free is called to release the resources of the container and method logger is invoked to record the execution statistics.

In ABS-YARN, each container is modeled by class Container, which has a method exe to execute a task. Each container is deployed as a deployment component with the given resource specification. MapReduceAM is modeled by class MapReduceAM, which has a method req to request a container for each map/reduce task from RM. Once the container is obtained from RM, the MapReduceAM can execute a task on the container by calling the exe method of the container. Since each MapReduce application is a terminating application with only two stages (i.e., map and reduce), MapReduceAM is unable to model the Spark driver of a Spark Streaming application.

In order to deploy SSP on ABS-YARN, we further create a new class StreamingAM on ABS-YARN by instantiating the SparkDriver class of the SSP model as a class StreamingAM. The resulting model of SSP on ABS-YARN is illustrated in Figure 3. In this model, classes RM, Container, and MapReduceAM remain as in ABS-YARN except that the exe method of Container can be invoked to execute either a map/reduce task or a job of a batch processing. In this model, users can specify not only all the parameters discussed in Section 3 for Spark Streaming, but also the following parameters to configure the scale of the underlying YARN cluster:

- the number of slave nodes in the YARN cluster
- the CPU capacity for each slave node
- the memory capacity of each slave node

When a user launches this model to run a Spark streaming application on YARN, method confSetup will first be triggered to request a set of desired containers from RM by invoking method getContainer. When the required containers have been obtained, confSetup triggers methods streamReceiver, batchGenerator, and batchScheduler to start the streaming application. Note that when a container finishes a job of a batch processing, method free will not be invoked to release the container's resources. Instead, the container will be returned to a container pool and wait for another execution.

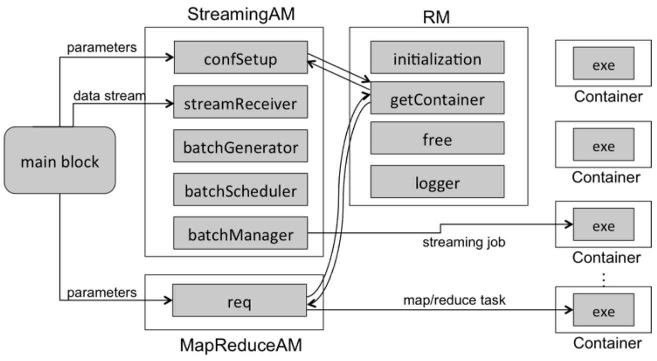

Figure 3. The modeling architecture of Spark Streaming on Apache YARN.

Our model exploits the concurrent objects of Real-Time ABS to capture concurrent batch processing. The deployment components of Real-Time ABS are used as an abstraction of containers with execution capacities for batch processing. The language's support for timed behavior is used to model how the resources of containers are consumed by stream processing as time passes. Furthermore, we exploit cost annotations to abstract from concrete job executions and focus on the resource and performance aspects of job computations. Hence, users can abstract from specific application details or black box components and still be able to experiment with the performance of the overall streaming application.

5 Model Validation

To validate the proposed model, we compared our model (i.e., SSP on ABS-YARN) with Spark Streaming deployed on a YARN cluster running Hadoop 2.2.0 (Apache Hadoop) and Spark 1.5.1 (Spark 1.5.1 released). The cluster has one virtual machine acting as RM and 30 virtual machines acting as slave nodes. Each virtual machine runs Ubuntu 12.04 with 2 virtual cores of Intel Xeon E5-2620 2GHz CPU and 2 GB of memory. To achieve a fair comparison, we also configured our model to have one RM and 30 slave nodes; each with 2 CPU cores and 2 GB of memory. For both our model and the real Spark Streaming environment, the resource requirements for each container are 1 CPU core and 1 GB of memory.

We chose three benchmarks called JavaNetworkWordCount (NWC for short), JavaSQLNetworkWordCount (SQLNWC for short), and JavaNetworkFunctionCost (NFC for short) from (Spark Streaming Example; Spark Streaming Programming Guide) as example streaming applications to conduct our validation. Table 1 summarizes the characteristic of these benchmarks. For some performance considerations, we noticed that, in Spark Streaming, the number of jobs in a batch processing for a same streaming application might change. For instance, when we run SQLNWC and NFC, some of their batches require 2 sequential jobs to finish, but some others require 3 sequential jobs to finish. However, this phenomenon does not happen when NWC was executed. Therefore, our model considers the worst-case batch processing workflow, i.e., NWC has one job in its batch processing workflow, but both SQLNWC and NFC have 3 sequential jobs in their batch processing workflows.

Table 1. Three Spark Streaming benchmarks.

Benchmark	batch processing workflow
NWC	1 job
SQLNWC	3 sequential jobs
NFC	3 sequential jobs

We employed Netcat (The GNU Netcat) to continuously send data from the Wikipedia website of Apache Spark (Wikipedia: Apache Spark) to these benchmarks. The size of each data item is around 1 KB and the data is sent in a dynamic inter-arrival pattern following an exponential distribution (Koralov and Sinai, 2007) with the average time of 1.96 sec and the standard deviation of 1.768 sec. In order to assign appropriate execution costs for each job of these benchmarks, we conducted an experiment on one of the slave node to study if there is a linear relationship between batch size and job execution time. However, the results of NWC illustrated in Figure 4 show that the job execution time does not always increase when the size of a batch increases. The other two benchmarks also have a similar phenomenon, implying that it is inappropriate to use batch size to assign job execution costs. As an alternative, we separately executed each benchmark on the same slave node without any workload to measure the average job execution time and standard deviation. Table 2 lists the corresponding results, which we used to assign job execution costs in our model.

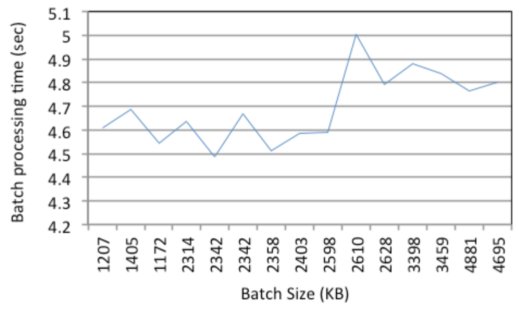

Figure. 4. The batch processing time of NWC under different batch sizes.

Table 2. The average job execution time and standard deviation of each benchmark.

Benchmark	Average Job Execution Time (sec) & Standard deviation (sec)	
	For non-empty batches	For empty batches
NWC	3.88 & 0.24	0.1 & 0
SQLNWC	3.85 & 0.35 for job1 0.58 & 0.28 for job2 0.26 & 0.05 for job3	0.06 & 0.02 for job1 0.53 & 0.12 for job2 0 & 0 for job3
NFC	4.08 & 0.30 for job1 4.22 & 0.41 for job2 3.99& 0.13 for job3	0.09 & 0.04 for job1 0.7 & 0.02 for job2 0.1 & 0.04 for job3

In Spark Streaming, a streaming application is stable if each of its batches can be scheduled immediately. For stability, the parameters conJobs and bi are highly influential. Hence, for each benchmark, we designed two scenarios, one is unstable and the other is stable, to see if our model can correspond to Spark Streaming on YARN in the following two performance metrics:

- Batch scheduling delay: The time a batch waits for scheduling.
- Batch processing time: The total time to process a batch.

5.1 Benchmark 1: NWC

First, we simulated the execution of NWC on our model and compared the corresponding results with those of NWC running on Spark Streaming by separately designing an unstable scenario and a stable scenario. In the unstable scenario, conJobs=1 and bi=2 sec, implying that at most one job is allowed to run at any time, and a batch is periodically generated every two seconds. On the other hand, in the stable scenario, conJobs=15 and bi=4 sec. Our goal is to see how well our model can simulate Spark Streaming when a streaming application is executed with both unstable and stable parameter settings.

Figure 5 illustrates the batch scheduling delay of NWC in the unstable scenario. We can see that the batch scheduling delay on Spark Streaming keeps increasing as more batches are generated, implying that the generated batches were unable to be processed immediately. The reasons are twofold: First, the value of conJobs was only one, so other jobs needed to wait in the queue. Second, setting bi to two seconds was too short. A shorter bi implies that batches are generated more frequently, so more containers are required to process these batches. Despite the poor performance, it is clear that the batch scheduling delay in our model follows the same trend as in Spark Streaming.

Figure 6 shows the batch processing time of NWC in the unstable scenario. Because data arrived every 1.96 sec in average and the standard deviation was 1.768 sec, data might sometimes arrive frequently (i.e., when the data inter-arrival time is shorter than 2 sec) and sometimes more seldom (i.e., when the data inter-arrival time is longer than 2 sec). Due to the fact that bi is 2 sec, a lot of generated batches were empty. This explains why the batch processing time in Spark Streaming fluctuated between 4 sec and 0 sec. This phenomenon also occurs in our model since our model considers both empty batch processing and non-empty batch processing.

Table 3 lists the average batch scheduling delay and standard deviation results of NWC in the stable scenario. It is clear that NWC in the Spark Streaming environment became more stable compared with that in the unstable scenario because the average batch scheduling delay were close to zero second. This means that each batch could be scheduled almost immediately. This phenomenon is also captured and reflected by our model. Figure 7 shows the batch processing time of NWC in the stable scenario. We can see that a lot of batch processing time in Spark Streaming are slightly lower than the batch processing time simulated in our model. This is because that our model follows the worst-case batch process workflow. In addition, we can observe that the number of empty batches in this scenario is far lower than that in the unstable scenario (compare Figure 7 with Figure 6). The main reason is that the value of bi in this scenario was double of the value in the unstable scenario, meaning that the probability of generating empty batches is reduced. From the results, we can see that our model also captures this change.

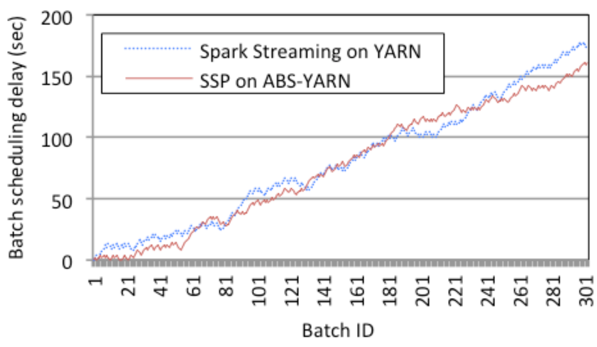

Figure 5. The batch scheduling delay of NWC in the unstable scenario.

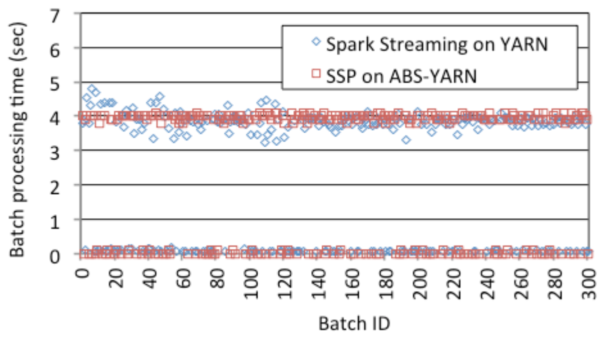

Figure 6. The batch processing time of NWC in the unstable scenario.

Table 3. The batch scheduling delay of NWC in the stable scenario.

Environment	Average (sec)	Standard deviation (sec)
Spark Streaming on YARN	0.024	0.004
SSP on ABS-YARN	0.00	0.00

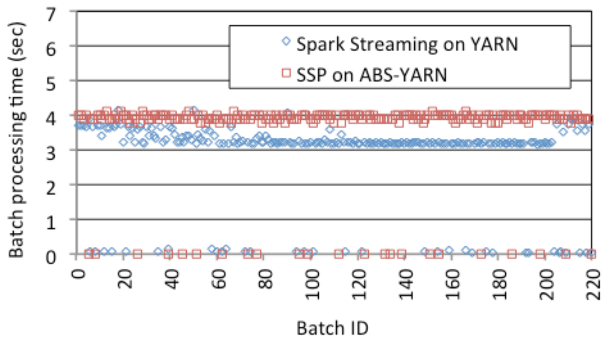

Figure 7. The batch processing time of NWC in the stable scenario.

5.2 Benchmark 2: SQLNWC

To see how well our model can simulate Spark Streaming when SQLNWC is executed, we also designed an unstable scenario and a stable scenario. In the former, conJobs=1 and bi=4 sec. In the latter, conJobs=2 and bi=4. Note that here we randomly chose these two settings to separately achieve unstable and stable execution and to verify if our model can correspond accordingly.

Figure 8 illustrates the batch scheduling delay of SQLNWC in the unstable scenario. The batch scheduling delay in both our model and Spark Streaming increased when SQLNWC generated more batches. However, the trend in our model is steeper than that in the real Spark Streaming environment. The key reason is that our model adopts the worst-case batch processing workflow, so the total number of jobs for all batch processing in our model is more than that in Spark Streaming, which causes this phenomenon. Nevertheless, our model is still able to reflect this unstable scheduling delay. In addition, Figure 9 shows that our model captures the batch processing time of SQLNWC running in Spark Streaming. Due to the worst-case batch processing workflow, we can see that most batch processing time in our model are slightly longer than those in Spark Streaming.

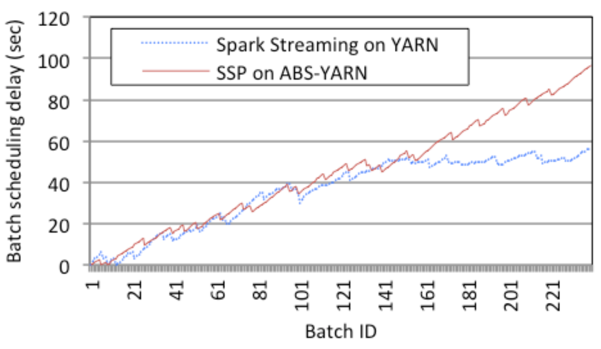

Figure 8. The batch scheduling delay of SQLNWC in the unstable scenario.

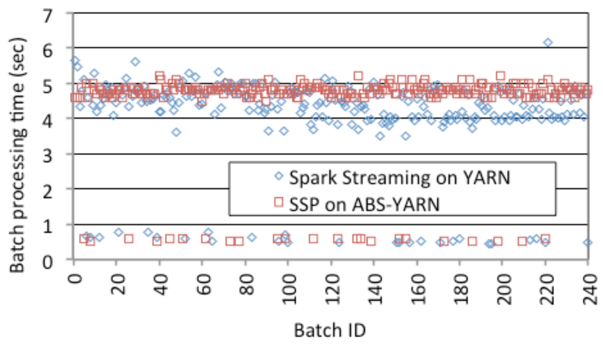

Figure 9. The batch processing time of SQLNWC in the unstable scenario.

On the other hand, in the stable scenario, the batch scheduling delay of SQLNWC in Spark Streaming dramatically decreased (see Table 4) and our model is still able to capture this performance change. In addition, Figure 10 suggests that our model is also able to reflect the batch processing time of SQLNWC running in Spark Streaming.

Table 4. The batch scheduling delay of SQLNWC in the stable scenario.

Environment	Average (sec)	Standard deviation (sec)
Spark Streaming on YARN	0.16	0.10
SSP on ABS-YARN	0.00	0.00

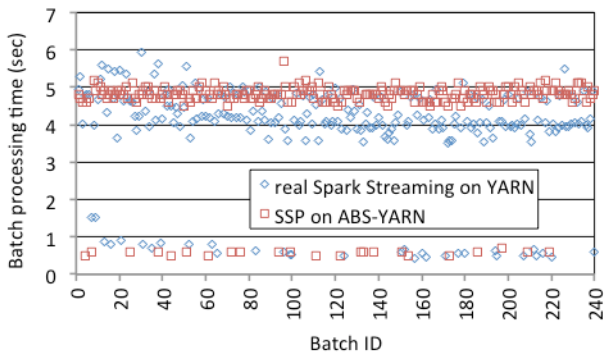

Figure 10. The batch processing time of SQLNWC in the stable scenario.

5.3 *Benchmark 3: NFC*

Similar to the previous two benchmarks, we randomly designed two scenarios to achieve an unstable execution and a stable execution for NFC. In the unstable scenario, conJobs=1 and bi=8 sec. In the stable scenario, conJobs=2 and bi=8 sec. Figure 11 and Figure 12, respectively, illustrate the batch scheduling delay and batch processing time of NFC in the unstable scenario. It is clear that our model is able to simulate the unstable performance behavior of NFC. Similarly, due to the adoption of the worst-case batch processing workflow, the batch scheduling delay and batch processing time in our model were longer than those in Spark Streaming.

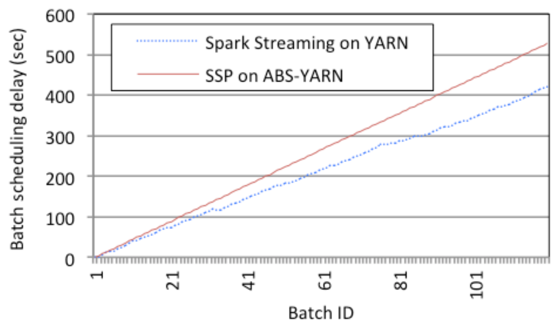

Figure 11. The batch scheduling delay of NFC in the unstable scenario.

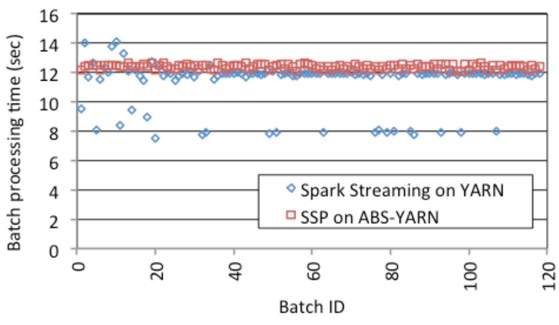

Figure 12. The batch processing time of NFC in the unstable scenario.

Furthermore, from the results shown in Table 5 and Figure 13, we can see that our model also reflects the batch scheduling delay and batch processing time of NFC when NFC running on Spark Streaming in the stable scenario.

Based on all the above results, we conclude that our model indeed captures the properties of Spark Streaming and it provides a good approximation of the performance of Spark Streaming on YARN under the dynamic data traffic pattern for both unstable and stable scenarios. Users can easily model their streaming applications in our model and compare by means of simulations and know how different parameter configurations affect the performance of their applications before these applications are deployed on the Spark Streaming framework.

Table 5. The batch scheduling delay of NFC in the stable scenario.

Environment	Average (sec)	Standard deviation (sec)
Spark Streaming on YARN	0.03	0.02
SSP on ABS-YARN	0.00	0.00

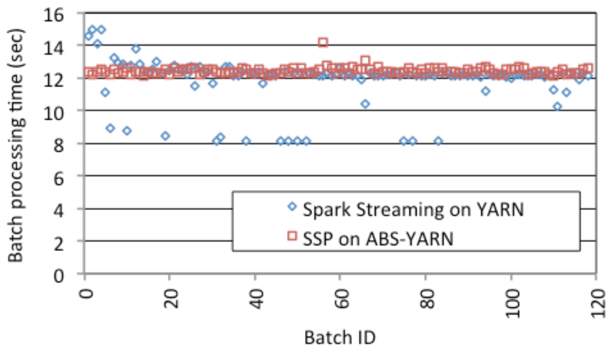

Figure 13. The batch processing time of NFC in the stable scenario.

6 Related Work

There have been some research efforts devoted to configurable and executable modeling. Lin et al. (2016) developed a generic framework called ABS-YARN for Hadoop YARN, which is a cluster platform for executing both batch processing and streaming processing. ABS-YARN enables users to configure a Hadoop YARN cluster (including cluster size and resource capacity) and determine job workload and job inter-arrival patterns to evaluate their deployment decisions. ABS-YARN was validated through a comprehensive comparison. The results show that ABS-YARN provides satisfactory modeling and offer users a dependable framework for making deployment decisions about YARN at design time. However, users are unable to directly model the detail of stream processing on ABS-YARN since ABS-YARN is mainly developed to model the underlying cluster platform.

Lin et al. (2018) introduced a model to simulate Spark Streaming and allow users to specify different parameters. The model focused on batch processing that consists of only one job, and it only provides the modeling from the stage level. To validate the model, the authors chose JavaNetworkWordCount to be their example application and designed two scenarios to separately enable and disable concurrent job processing. The simulation results show that the proposed model provides a good approximation of Spark Streaming in terms of the batch scheduling delay and batch processing time. Different from their model, the model we proposed in this paper emphasizes the modeling from the job level, which allows to model batch processing consisting of multiple jobs and reduces users' burden for assigning cost annotations for their batch processing workflows. Besides, our model not only models and simulates Spark Streaming, but also Apache YARN, offering users more flexibility to control the underlying cluster resources.

Kroß and Krcmar (2017) presented an approach to model and simulate the performance of batch processing and stream processing by using and extending the Palladio component model (PCM), which enables engineers to describe performance relevant factors of software architecture. The authors use PCM to represent resource clusters, simulate parallel operations, and distribute them on a cluster of hardware resources. However, this approach does not allow users to configure the stream processing framework and their applications. SECRET (Botan et al., 2010) is a descriptive model for describing and predicting the behavior of diverse stream processing engines. This model focuses on time-based windows and single-input query plans and gives an end-to-end view of the effects of different execution semantics. However, SECRET is not a configurable and executable model. State Refinement (Dosch and Stümpel, 2004) is a formal method for transforming a stream processing function into a state transition machine with input and output. In this method, states are the abstraction of input history and state transition function are derived using history abstractions. Persistent Turing Machines (Goldin et al., 2004) endows classical Turing machines with dynamic stream semantics by formalizing the intuitive notion of sequential interactive computation. Event Count Automata (Chakraborty et al., 2005) is a state-based model for Stream Processing Systems by capturing the timing properties of data stream in terms of arrival and service pattern.

Another line of work focuses on modeling stream queries. Babcock et al. (2002) describe fundamental models and issues in developing a general-purpose data stream management system, especially related to stream query languages, requirements and challenges in query processing, and algorithmic issues. The authors extend standard SQL to allow the specification of sliding windows. Later, Kapitanova et al. (2011) have proposed a formal specification language called MEDAL to model data stream queries and data admission control. This language is based on Petri nets and focuses on modeling different stream-processing features such as collaborative decision-making and temporal and spatial data dependencies.

In contrast to the work discussed above, the model introduced in this paper targets the formalization of a state-of-the-art stream processing framework: Spark Streaming. We capture the main features of Spark Streaming on an elastic YARN cluster and address how processing capacities, data arrival patterns, and framework parameters affect the nonfunctional aspects of streaming applications, i.e., batch scheduling delay and batch processing time. Our model is executable and highly configurable and allows users to observe and compare the performance consequences of their streaming applications at the modeling phase.

7 Conclusion and Future Work

In this paper, we have presented SSP for modeling Spark Streaming. The proposed model enables users to configure the processing framework of Spark Streaming and adapt it to their streaming application settings, including streaming job workflow and execution cost. To model the deployment of Spark Streaming on Apache YARN, we have extended SSP by integrating it with ABS-YARN. The resulting model allows users to easily evaluate and compare how different parameter configurations and deployment decisions affect their streaming applications before these applications are actually deployed in the real world.

To increase the applicability of formal methods in the design of virtualized stream processing backends, we believe that it is crucial to show that the proposed model can faithfully reflect Spark Streaming once the model has been configured. To validate the proposed model, we have compared it with Spark Streaming running on a YARN cluster. The validation shows that 1) the model captures the key properties of Spark Streaming, including batch generation, empty batch processing, non-empty batch processing, and batch scheduling; 2) the model provides a good approximation of Spark Streaming on YARN in terms of the batch scheduling delay and batch processing time; and 3) the model enables users to predict the performance of their streaming applications on Spark stream processing framework with different configuration settings during the modeling phase and thereby to determine an appropriate deployment decision.

In future work, we plan to extend the model and further formalize virtualized stream processing by considering the modeling of multiple stream receivers and the failure of slaves, containers, and the network. Furthermore, we plan to investigate application-aware scheduling algorithms to optimize batch processing performance and reduce resource consumption using a formal approach.

Reference

1. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla G., and Román-Díez, G. (2014) 'SACO: Static Analyzer for Concurrent Objects' in *TACAS 2014*: the *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 8413 of Lecture Notes in Computer Science, Springer, pp. 562-567.
2. Amazon EC2. [online] https://aws.amazon.com/ec2/?nc1=h_ls

(Accessed 3 December 2018).

3. Apache Hadoop. [online] http://hadoop.apache.org/ (Accessed 3 December 2018).
4. Amazon Mesos. [online] http://mesos.apache.org/ (Accessed 3 December 2018).
5. Apache Spark JobScheduler. [online] https://github.com/apache/spark/blob/master/streaming/src/main/scala/org/apache/spark/streaming/scheduler/JobScheduler.scala (Accessed 3 December 2018).
6. Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002) 'Models and issues in data stream systems' in *Proceedings of the 21st ACM SIGMOD-SIGACTSIGART Symposium on Principles of Database Systems*, ACM, pp. 1-16.
7. Bjørk, J., de Boer, F.S., Johnsen, E. B., Schlatte, R., and Tarifa, S.L.T. (2013) 'Userdefined schedulers for real-time concurrent objects', *Innovations in Systems and Software Engineering*, Vol. 9, No. 1, pp. 29-43.
8. Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R.J., and Tatbul, N. (2010) 'Secret: a model for analysis of the execution semantics of stream processing systems' *Proceedings of the VLDB Endowment*, Vol. 3, Issue 1-2, pp. 232-243.
9. Chakraborty, S., Phan, L.T., and Thiagarajan. P. (2005) 'Event count automata: A statebased model for stream processing systems' in *RTSS 2005*: *26th IEEE International Real-Time Systems Symposium*, IEEE, pp. 12-pp.
10. Dosch, W. and Stümpel, A. (2004) 'Transforming stream processing functions into state transition machines' in *International Conference on Software Engineering Research and Applications*, Springer, Berlin, Heideberg, pp. 1-18.
11. Goldin, D.Q., Smolka, S.A., Attie, P.C., and Sonderegger, E.L. (2004) 'Turing machines, transition systems, and interaction', *Information and Computation*, Vol. 194, No. 2, pp. 101-128.
12. Hähnle, R., and Johnsen, E.B. (2015) 'Designing resource-aware cloud applications', *IEEE Computer*, Vol. 48, No. 6, pp. 72-75.
13. Johnsen, E.B., Hähnle, R., Schäfer. J., Schlatte, R., and Steffen, M. (2011) 'ABS: A core language for abstract behavioral specification' in *FMCO 2010*: *International Symposium on Formal Methods for Components and Objects*, Vol. 6957 of Lecture Notes in Computer Science, Springer, pp. 142-164.
14. Johnsen, E.B., Schlatte, R., and Tarifa, S.L.T. (2015) 'Integrating deployment architectures and resource consumption in timed object-oriented models', *Journal of Logical and Algebraic Methods in Programming*, Vol. 84, No. 1, pp. 67-91.
15. Kapitanova, K., Wei, Y., Kang, W., and Son, S.H. (2011) 'Applying formal methods to modeling and analysis of real-time data streams', *Journal of Computing Science and Engineering*, Vol. 5, No. 1, pp. 85-110.
16. Koralov, L. and Sinai, Y.G. (2007) 'Theory of Probability and Random Processes', *Springer Science & Business Media*.
17. Kroß, J. and Krcmar, H. (2017) 'Model-based performance evaluation of batch and stream applications for big data' in *MASCOTS 2017*: *25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, IEEE, pp. 80-86.
18. Lin, J.-C., Lee, M.-C., Yu, I.C., and Johnsen, E.B. (2018) 'Modeling and Simulation of Spark Streaming' in *AINA 2018*: *32nd IEEE International Conference on Advanced Information Networking and Applications*, IEEE, pp. 407-413.
19. Lin, J.-C., Yu, I.C., Johnsen, E.B., and Lee, M.-C. (2016) 'ABS-YARN: A formal framework for modeling Hadoop YARN clusters' in *FASE 2016*: *19th International Conference on Fundamental Approaches to Software Engineering*, Vol. 9633 of Lecture Notes in Computer Science, Springer, pp. 49-65.
20. Spark 1.5.1 released. [online] https://spark.apache.org/news/spark-1-5-1-released.html (Accessed 3 December 2018).
21. Spark: Lightning-fast cluster computing. [online] http://spark.apache.org/ (Accessed 3 December 2018).
22. Spark Streaming. [online] http://spark.apache.org/streaming/ (Accessed 3 December 2018).
23. Spark Streaming Example. [online] https://github.com/apache/spark/tree/v2.4.0/examples/src/main/java/org/apache/spark/examples/streaming (Accessed 3 December 2018).
24. Spark Streaming Programming Guide. [online] https://spark.apache.org/docs/latest/streaming-programming-guide.html (Accessed 3 December 2018).
25. The GNU Netcat. [online] http://netcat.sourceforge.net/ (Accessed 3 December 2018).
26. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013) 'Apache Hadoop YARN: yet another resource negotiator' in *SOCC 2013*: *ACM Symposium on Cloud Computing*, pp. 5.
27. Wikipedia: Apache Spark. [online] https://en.wikipedia.org/wiki/Apache_Spark (Accessed 3 December 2018).
28. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., and Stoica, I. (2012) 'Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing' in *NSDI 2012*: *9th USENIX conference on Networked Systems Design and Implementation*, USENIX, pp. 2-2.
29. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013) 'Discretized streams: Fault-tolerant streaming computation at scale' in *SOSP 2013*: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ACM, pp. 423-428.