2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

N-Storm: Efficient Thread-Level Task Migration in Apache Storm

Zhou Zhang[1, 2], Peiquan Jin[1, 2], Xiaoliang Wang[1, 2], Ruicheng Liu[1, 2], Shouhong Wan[1, 2]
[1]*School of Computer Science and Technology, University of Science and Technology of China, Hefei, 230026, China*
[2] *Key Lab. of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei, 230026, China*
Email: jpq @ustc.edu.cn

***Abstract*—Apache Storm is a widely used stream processing system, but it only supports offline task scheduling. Recently, some online task scheduling approaches were proposed to offer task migrations in Storm at runtime. However, most of them have to cost more than 10 seconds during a task migration, because they employed a Worker-level scheme to stop or start Workers (implemented as processes in Storm). When an Executor within a Worker needs migrating, Storm has to kill the Worker, leading to the stop and restart of all Executors in the Worker. This process-level scheme introduces unnecessary stop and restart of Executors and Workers, resulting in poor performance of task migrations. Aiming to solve this problem, we propose N-Storm (Non-stop Storm). Instead of using the process-level migration scheme, N-Storm employs a thread-level scheme for task migrations. Particularly, we add a key/value store on each Worker node to make Workers be aware of the changes of the task schedule, so that the Workers on a Worker node can manage their Executors at runtime, i.e., killing existing Executors or starting new Executors. With this mechanism, we can avoid the unnecessary stop of Executors and Workers during a task migration, and thus improve the performance of task migrations. We further propose two optimizations to make N-Storm efficient for multiple tasks migrations. Our experimental results on a real Storm cluster show that N-Storm is able to reduce the stop time, increase the system throughput, and save significant migration time, compared with Storm and other approaches.**

Keywords—Apache Storm, task migration, online scheduling, thread management

I. INTRODUCTION

With the rapid development of cloud computing, social networks, and IoT (Internet of Things), human society has entered the era of big data [1]. Generally, big data has two kinds of processing modes, namely batch processing and stream processing [2]. Compared with batch processing, stream processing is more suitable for real-time applications, such as social networks, E-commerce, and video surveillance.

So far, a number of systems have been proposed for stream processing, among which Apache Storm (http://storm.apache.org/) is a widely used platform that has been adopted by many famous companies, e.g., Twitter [3]. However, the original Storm only supports offline task scheduling [4]. When Storm deploys a task schedule, it does not allow any changes to the task schedule until all running tasks are finished. In many real-world applications such as E-commerce applications, data are arriving at a fluctuant rate, and the old task schedule may not fit well with the new workload. In such situations, an online task scheduling solution is required, so that we can change a task schedule while the tasks are running. Here, the key point is that all running tasks should not be affected during an online task scheduling.

Online task scheduling requires that tasks have to be migrated among nodes while the system is kept running. The original Storm is not able to handle online task migrations. It has to first stop all running tasks (or wait until they are finished), and then start a new task schedule. Presently, some online schedulers, such as T-Storm [4] and TS-Storm [5], typically cost more than 10 seconds during a task migration. Such poor performance is not acceptable for real-time data processing. We noted that this is mainly because of the internal process-level task management scheme in Storm. A process-level task migration, which has been employed in existing works like T-Storm [4] and TS-Storm [5], will introduce unnecessary stop of Workers and Executors, yielding significant stop-and-restart time of Workers and Executors. As a result, a process-level mechanism is suitable for online task migrations in Storm.

In this paper, instead of adopting a process-level task migration scheme, we propose a thread-level non-stop scheme called N-Storm to implement online task migrations in Storm. N-Storm only restarts the Executors (threads) to be migrated but does not affect the Worker process and other Executors within the Worker node. With this mechanism, we can avoid the stop of unnecessary Workers and Executors during a task migration, and therefore improve the performance of online task migrations. Briefly, we make the following contributions in this paper:

(1) We find out the intrinsic cause of inefficient task migrations in Apache Storm, i.e., the process-level task management mechanism, and propose a thread-level non-stop scheme named N-Storm (Non-stop Storm) to improve the efficiency of online task migrations in Storm.

(2) We further propose two optimization strategies for N-Storm, namely delaying the killing of Executors and adjusting the synchronization cycles, to make N-Storm efficient for migrating multiple tasks at once.

(3) We verify the efficiency of N-Storm and the optimization strategies in a real Storm cluster. The experimental results show that N-Storm is able to reduce the stop time, increase the system throughput, and save significant migration time, compared with Storm and other approaches.

II. BACKGROUND AND MOTIVATION

A. Basic Concepts of Storm

Storm packages the logic of a real-time application as a Storm topology (hereinafter denoted as Topology). Topology is an abstraction of complex computing tasks in

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00219

Storm. A Topology is a DAG (Directed Acyclic Graph). Each vertex in a Topology represents a component. There are two types of components in Storm: Spout and Bolt. Spout is responsible for receiving data tuples from the data source and sending tuples to the Topology. Here, tuple is the unit where Storm processes data. Bolt is responsible for encapsulating the processing logic and processing specific data. A Bolt can only handle one tuple each time. In Storm, each component can have one or more tasks. Such a technology is called *component parallelism* [6-10].

Storm is typically deployed on a cluster using the master-slave architecture. It runs a process on the master node called Nimbus, which is the administrator of the entire system. Other nodes are treated as slave nodes, which are also called Worker nodes. A Worker node consists of a Supervisor process and some Worker processes. The Supervisor communicates with Nimbus through Apache Zookeeper. In Storm, all modules do not maintain their running states themselves, and all states are stored on Zookeeper, meaning that all modules in Storm are stateless [1, 2]. When a process is killed, it only needs to restart and continue working without affecting the system. Here, we clarify that our work is based on the stateless scheme of Storm.

In addition to a Supervisor process, a Worker node also consists of some Worker processes (Workers in short). A Worker is actually a JVM (Java Virtual Machine) process that runs on a Worker node with configured resources. Each Worker can have one or more Executors. An Executor is a thread responsible for processing one or more specific tasks. A task in Storm refers to some computation work. When a new Topology is submitted, the Nimbus will divide each component in the Topology into one or more tasks and perform task scheduling to assign tasks to Worker nodes.

B. *Problem of Task Scheduling in Storm*

The task scheduling algorithm in Storm aims to (1) assign Workers to Worker nodes, and (2) assign Executors to Workers. Storm employs an offline task scheduling policy, meaning that when a task scheduling is deployed, it does not allow changing the schedule until all tasks are finished. Such an offline scheme is inefficient for many stream processing applications like Twitter [2], in which workloads may frequently change with time and adjustment of the task schedule is required to make the task schedule better suitable for the change of workloads. That means, the task assignment and configuration in Storm has to adapt to the change of workloads.

The offline task scheduling algorithm in Storm has poor performance. More specifically, Storm uses the *rebalance* command for task migrations. This command needs to first stop running tasks and kill all Workers. During this time, the Nimbus generates a new task schedule. When all Workers are killed, the Nimbus restarts the entire Topology according to the new task schedule. According to such a scheme, the system needs to wait until all tasks in the queue finish. However, as the Nimbus and Supervisors do not know whether each Executor will complete work, Storm sets a threshold of wait time, which is 30 seconds by default. This means that the *rebalance* command will make the system stop at least 30 seconds.

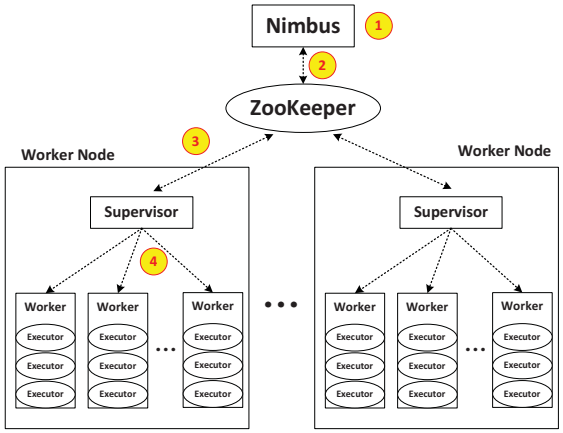

Figure. 1. Task management mechanism of Storm

Thus, an efficient online task scheduling scheme is needed for Storm. Instead of stopping the running of Topology, we can directly send the new task schedule to the Supervisor at each Worker node, and let the Supervisors perform task migrations [4, 5]. The Supervisor can close the Workers whose Executors need to be updated, and start new Workers if needed. Meanwhile, other Workers can keep running regardless of the shutdown and starting of Workers. This approach is more efficient than the *rebalance* command. However, it still has a time cost of more than 10 seconds when performing a task migration.

Through the investigation of the source code of Storm, we found that the intrinsic cause of the poor performance is the process-level task management mechanism in Storm. Note that the task scheduling in Storm is accomplished by a thread-level mechanism, that is, the Executors for task scheduling are implemented as threads. However, the current task management scheme in Storm does not support thread-level. It can only kill or start process-level components including Supervisors and Workers, but cannot update thread-level Executors at runtime.

Fig. 1 shows the task management mechanism in Storm, which consists of four steps:

(1) Nimbus generates a task schedule for the given Topology.
(2) Nimbus writes the task schedule into Zookeeper.
(3) The Supervisor on each Worker node gets its own tasks from Zookeeper.
(4) Each Supervisor checks the Workers whose tasks have changed, and kills or starts Workers.

Here, the key point is that Supervisors and Workers are both separate processes, and Storm does not provide methods for managing thread-level Executors within each Worker. Thus, if a Worker needs to kill only a few Executors but not all the Executors, Storm has to restart the entire Worker and all the Executors within the Worker as well, which will lead to unnecessary time overhead.

III. N-STORM

In this paper, we propose a novel thread-level task migration scheme called N-Storm (Non-stop Storm) to overcome the problems of the process-level scheme in Storm. The goal of our proposal is to kill/start necessary fine-grained Executors but not the coarse-grained Workers.

A. Architecture of N-Storm

In N-Storm, we refine Storm's task management granularity to thread-level, enabling the Supervisor to directly control the Executors within each Worker. N-storm can shut down an Executor or start a new Executor while the Worker is running, without affecting other Executors. Fig. 2 shows the task management mechanism of N-Storm. Specially, N-Storm adds a key/value (k/v) store on each Worker node to support communication between the Supervisor and the Workers within the same Worker node. The first three steps in Fig. 2 are the same as those in Fig. 1. In the fourth step, the Supervisor on a Worker node periodically writes messages to the k/v store. In the final step, each Worker periodically accesses the k/v store to get the latest message and update the Executors it manages.

As Storm offers an internal k/v store in its implementation, we simply utilize this k/v store to maintain the task assignment information that is needed in N-Storm. There may be other alternative implementations such as message queue for maintaining the task assignment information, but it is more practical and requires less modification of codes to use the internal k/v store in Storm. More specifically, the task assignments in N-Storm are stored in the k/v store as a map containing three kinds of mappings, namely *node→ host*, *executor→ node and port*, and *executor→ start-time*. Based on such an implementation, each Worker in the Topology can obtain the task assignment information of its own Executors.

Our design follows the stateless principle of Storm. When the Supervisor or some Worker is down, Storm can restart it directly without any further work. Here, the key idea is that the task assignment information has been persistently stored on the k/v database. Next, we present the improvements to Supervisor and Worker in detail.

B. Improvements in Supervisor

The Supervisor on a Worker node periodically executes two routines, namely synchronizing with Zookeeper and managing Workers within the same Worker node. In the routine of managing Workers, the Supervisor periodically compares the new task assignment with the current task assignment. When it finds that the Executors assigned to a Worker have changed, it kills the Worker and restarts a new Worker.

N-Storm uses a new scheme for the Supervisor to manage Workers. Compared to the original Storm, it introduces two improvements to the Supervisor. First, differing from the original Storm in which Executors changes will result in restart of Workers, N-Storm does not restart Workers when the Executors within the Workers are changed. Thus, the performance of task migration can be improved because N-Storm reduces unnecessary Worker restarts. Second, after requesting data from Zookeeper, we write the task schedule to the k/v stores at each Worker node. Differing from the original way in Storm that only Supervisors are aware of task schedules, the k/v stores in N-Storm can be accessed by all Workers within the same Worker node. This makes Workers be aware of the updating information of task schedules. Further, Workers can update the Executors it manages to adapt to the change of the task schedule.

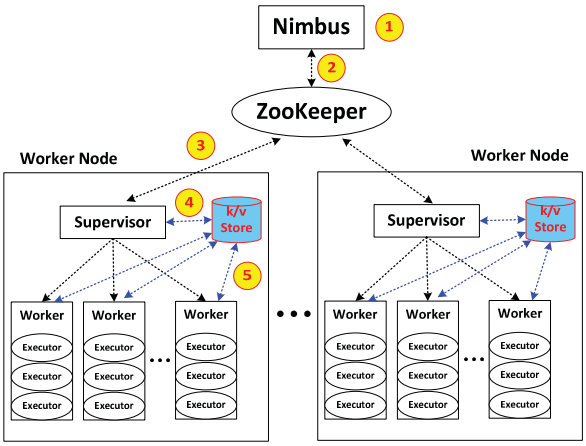

Figure. 2. Task management mechanism of N-Storm

C. Improvements in Worker

In Storm, each Worker manages its Executors which process the tasks assigned to the Worker. For each Executor, a corresponding message queue is established to store Tuples that need to be processed by the Executor. After a Worker starts up, the configurations about Executors, tasks, and the mapping from Executors to message queues are saved in a big map which is not updatable. This means that the task assignment on each Worker cannot be changed. The configurations are frequently used by some important functions in a Worker. For example, the message sending function will use them to determine whether the target Executor for a message is within the same Worker. The function which is responsible for updating the socket connection is called *refresh-connections*. It uses tasks and Topology information to find out the Workers that need to communicate and then establishes socket connections with them. This function is called periodically by each Worker.

While Workers in Storm are not aware of task schedule information, in N-Storm we add in each worker node a k/v store which saves necessary configurations about Executors, tasks, etc. Thus, Workers can access the k/v store to know the change of the task schedule. Once a Worker finds that the task schedule has changed, it can conduct a thread-level task migration to make the task assignment adaptive to the change of the task schedule.

We have developed the new algorithm for the thread-level task migration and implemented it in each Worker, called TLU (Thread-level update). In N-Storm, we use Clojure's ATOM modifier [11] to enable the atomic update of Executors, tasks, message queues, and other related

contents. Clojure provides two primitives, namely "*reset*!" and "*swap*!", to ensure the atomic update of variables. We also add the atomic update function in Workers, so that each Worker can access the k/v store on the worker node periodically and atomically modify the Executors assigned to the Worker.

TLU uses the above-mentioned *refresh-connections* function. The input to TLU is *worker*, *executors*, and *refresh-connections* function (Clojure is a functional language that allows functions to be arguments of another function). The *worker* is the big map storing the configuration information of the Worker. The *executors* is a map storing all Executor objects on the Worker. The *executors* map maintains the mapping between Executor ids and active Executors. TLU first accesses the k/v store on the worker node, gets the latest schedule, and extracts the Topology and Executors allocated to the Worker from the schedule. Then, it checks whether the Topology that the Worker belongs to has changed, and whether the Executor set is empty. If either is true, it waits for the Supervisor to turn off the Worker. Otherwise, it will make use of the difference between the currently running Executor set and the newly assigned Executor set. To get the *new-executors* using the newly assigned Executor set minus the currently running Executor set, and to get the *kill-executors* using the currently running Executor set minus the newly assigned Executor set. If both result sets are empty, it invokes the *refresh-connections* function once and ends, because the Executors it needs to communicate may have migrated. Otherwise, if both result sets are not empty at the same time, meaning that the Executors on the Worker have changed, it updates the Executor set, the task set, the message queue, and other information in the *worker* using the '*reset*!' primitive, and then invokes *refresh-connections* function to update the socket connection. Finally, it stops or starts the corresponding Executors and updates the *executors* using the '*swap*!' primitive.

IV. Optimizing N-Storm for Multiple Tasks Migration

In this section, we further optimize N-Storm to make it efficient for migrating multiple Executors simultaneously. Particularly, we propose two optimization strategies, which are delayed killing of Executors and shortening the execution cycle of the update function.

A. Delayed the Killing of Executors

In TLU, after we get the new schedule, we kill the Executors in *kill-executors* immediately, and then start the new Executors in *new-executors*. However, the Executors to be killed may be performing calculations, and there may be tuples in the message queue that are going to be processed. Thus, when Executors are killed, the associated tuples in the message queue will be lost, which will eventually raise timeout to Storm. In this case, Storm will re-process these tuples according to some fault-tolerant mechanism. As a consequence, killing the Executors immediately will probably cause additional time overhead.

Aiming to address the above issue that impacts the performance of task migration in N-Storm, we propose to delay the killing of Executors. Specifically, when we need to kill an Executor, we do not kill it immediately but let a timer thread in the Worker monitor and perform the killing operation. The timer thread will wait a few seconds before it finally kills the Executor. As modules such as Nimbus, Supervisor, and Worker all run one or more timer threads to support deferred and periodic execution functions in Storm, our timer-thread-based delayed killing of Executors can use one of the existing timer threads in the Worker. Fig. 3 illustrates the idea of delayed killing of Executors. Here, we want to move Executor 2 in Worker 1 to Worker 2. Worker 1 first gets the latest schedule from the k/v store and decides to kill Executor 2. Instead of killing it immediately, we invoke the timer thread in Worker 1 to manage the killing of Executor 2. The timer thread will wait a few seconds (2 seconds in our experiments) before it kills Executor 2. Meanwhile, Worker 2 gets the latest schedule from the k/v store and decides to start Executor 2. During the delayed time interval, two identical Executors may exist in two Workers, but this will not affect the property of the processing. After the *refresh-connections* function has been executed, the Executors that send messages to the old Executor 2 in Worker 1 will start sending messages to the new Executor 2 in Worker 2. In this way, no tuples in the wait queue belonging to the old Executor 2 are lost during the migration. Thus, we need not trigger any fault tolerance process. In addition, the Topology does not have to wait for the creation of the new Executor 2, meaning that N-Storm implements non-stop migrations.

B. Adjusting Synchronization Cycles

The performance of N-Storm is highly impacted by two kinds of synchronization cycles. The first is the cycle for a Supervisor to communicate with Zookeeper (Supervisor-Zookeeper Cycle), and the second is the cycle for a Worker to visit the k/v store (Worker-Store Cycle). We noted that Supervisors and Workers work asynchronously in Storm, i.e., a Supervisor communicates with Zookeeper every 10 seconds in Storm, and a Supervisor communicates with its Workers every 3

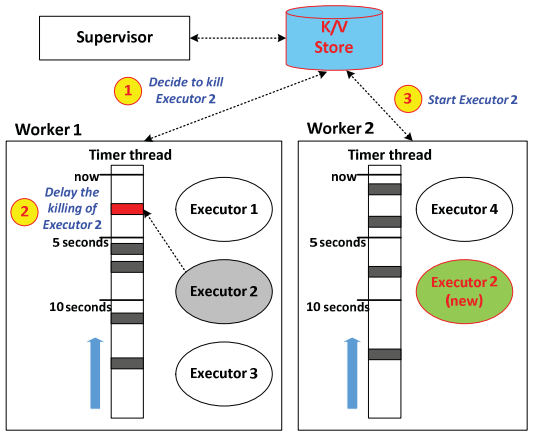

Figure. 3. Delayed the killing of Executors

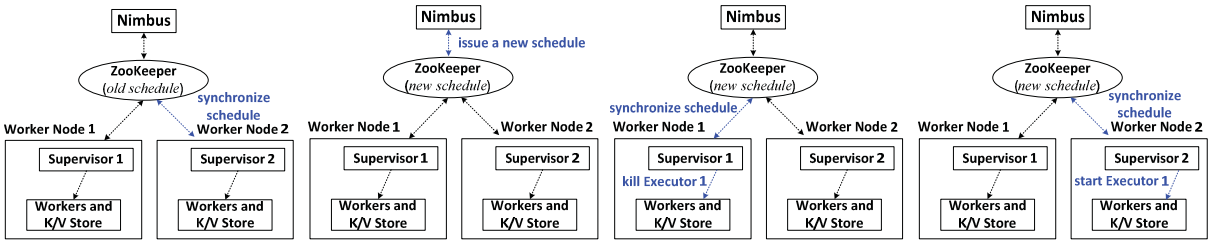

Figure. 4. The worst case of asynchronous working mechanism

seconds by default. This asynchronism will affect the duration of performance fluctuation.

We show the influence of this asynchronism on performance in Fig. 4. Assume that a Topology runs on Worker Node 1 with Supervisor 1, as well as on Worker Node 2 with Supervisor 2. Assume that we need to migrate Executor 1 on Worker Node 1 to Worker Node 2. In the worst case, Supervisor 2 visits Zookeeper at clock T=0 second, and finds that the task schedule has not changed. Then, after half a second, Nimbus issues the new schedule to Zookeeper. After another half a second, Supervisor 1 accesses Zookeeper and updates the tasks for Worker Node 1, and kills Executor 1. However, Supervisor 2 has to wait until the clock T=10 seconds to get the new schedule and start the new Executor 1, which causes Executor 1 to stop working for about 9 seconds. As a consequence, in the worst case the stopping time of the migrated Executor is nearly the size of the Supervisor-Zookeeper Cycle (10 seconds by default in Storm). The influence of the Worker-Store Cycle is similar.

Thus, we propose to reduce the impact of Executor migration on performance by shorten the Supervisor-Zookeeper Cycle as well as the Worker-Store cycle. Generally, when the cycles are set to a small value, the expected stopping time of migrated Executors can be reduced, so that the overall performance of task migration is expected to improve. Meanwhile, a small cycle may also lead to the frequent invoking of the synchronization operation, which will introduce more CPU costs and increase the workloads of Supervisors and Workers. In Section V, we test the influence of the setting of the cycles on the performance of N-Storm and find that setting a middle value for the cycles can get the best performance.

V. Evaluation

A. Experimental Setup

Storm only supports data processing, thus we need additional applications to provide input data. Our experiments use DRPC (Distributed Remote Procedure Call) to provide input data to Storm. DRPC, whose structure is shown in Fig. 5, has been integrated into Storm. We implement the Word Count Topology as the Topology for testing. The input of the Topology is English sentences. The responsibility of the first Bolt is to divide sentences into words. And the second Bolt is responsible for counting the number of word occurrences and outputting statistical results.

We conducted all experiments on Storm 0.9.7. N-Storm has also been implemented on this version of Storm.

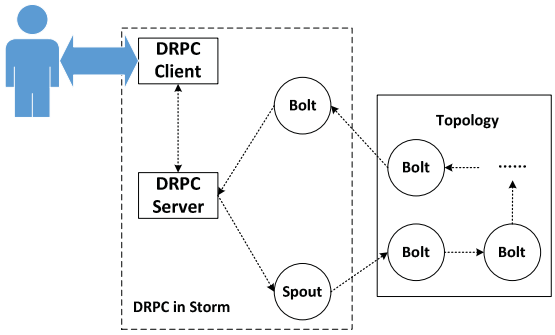

Figure. 5. Structure of DRPC

Later versions of Storm add some additional functionalities and interfaces, but use the same task management mechanism as before. We build a Storm cluster consisting of three servers, each of which is equipped with two Intel Xeon e5-2620 v4 CPUs and 128 GB memory. The Nimbus and DRPC servers run on one server, and the other two servers are used as Worker nodes. Each Worker node opens 4 slots, and we limit the heap size of each Worker to 1 GB. In addition, we deploy Zookeeper 3.4.10 on all three servers. The DRPC client runs on a personal computer, which is equipped with an Intel Core i5-4590 CPU and 8 GB memory.

Note that there are three parameters in our experiments, namely M, T_S, and T_W. M represents the number of Executors to be migrated in each task re-assignment. T_S represents the cycle that Supervisor visits Zookeeper to get the task schedule, and T_W represents the cycle that Worker visits the k/v store to get the task schedule. Table I summarizes all parameters and settings in the experiments. All experimental results are the averaged

TABLE I. Parameters for the Experiments

Parameter	Value
Slots per Worker node	4
Heap size of Worker	1 GB
Topology parallelism	8
Bolt 1 parallelism	12
Bolt 2 parallelism	24
Task re-assignment cycle	60 seconds
Number of Executors to be migrated per reassignment *(M)*	1
Cycle of Supervisor for update task assignment *(T_S)*	10 seconds
Cycle of Worker for update task assignment *(T_W)*	3 seconds
Cycle of Supervisor for Worker management *($=T_W$)*	3 seconds
System running time	600 seconds

value over 10 repeated runs.

All the task migration methods involved in the experiments are summarized as follows:

(1) Storm [8, 9, 12, 13]: This strategy uses the *rebalance* command before each task migration, which is recommended by Storm.

(2) Storm* [4, 5]: This strategy directly dispatches the new task assignment to the Supervisor, and the Supervisor kills or starts Workers. T-Storm and TS-Storm both use this strategy.

(3) N-Storm: The task migration method we proposed in Section III, which is toward single task migration.

(4) N-Storm+: The optimized N-Storm we proposed in Section IV, which is toward multiple task migration.

B. Performance of N-Storm for One Task Migration

In this section, we compare the performance of N-Storm with Storm and Storm*.The parameters are set by default, where M=1, T_S=10 seconds, and T_W=3 seconds.

We capture the throughput of the first 150 seconds of the three methods, as shown in Fig. 6. In the stable running stage, the throughputs of the three methods are very close. This indicates that N-Storm performs as well as Storm when task migrations are not invoked. We make the first task re-assignment at the 50th second. Storm needs to suspend the Topology running before the task re-assignment, and the entire stop time is around 30 seconds. Storm* reduces the degradation time of the throughput by more than a half, and the system throughput only fluctuates by about 10 seconds. N-Storm reduces the stop time of performance degradation to around 1 second, which is far better than the other two strategies. In addition, we find that the first throughput drop of N-Storm occurs at the 62nd second, with a delay of about 10 seconds from task re-assignment. This is because that T_S is set to 10 seconds in our experiment, and in the worst case there will be a delay of more than 10 seconds for task migrations.

Next, we calculate the average throughput per second when the system was running smoothly. Based on this, we calculate the duration of system performance degradation in each set of experiments. The performance degradation percent is set to 20%, 40%, 60%, 80%, and 100%, as shown in Fig. 7. Then, we measure the duration of performance degradation at each level. We specially focus on the 60% and 100% levels. If the performance degradation level is over 60%, the system performance is seriously worsen. In addition, a drop of 100% means that the system is stopped. Storm has 279 seconds in case of 60% performance degradation, and 267 seconds in case of 100% degradation. Storm* reduces the duration of 60% performance degradation to 155 seconds, which is 44% less than Storm. However, Storm* still accounts for more than 1/4 of the total system running time. In contrast, N-Storm only costs 20 seconds at the level of 60% performance degradation, which is 93% less than that of Storm, and 87% less than that of Storm*. Fig. 8 shows the total throughput of the system. We can see that Storm has the lowest throughput, and the throughput of Storm* is 41% higher than that of Storm. N-Storm has the highest

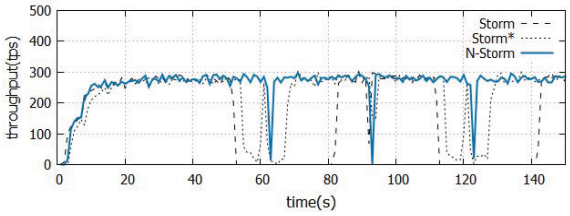

Figure. 6. Throughput fluctuations

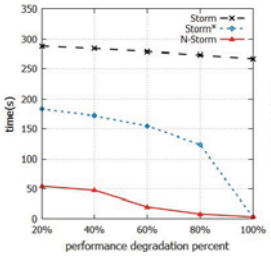

Figure. 7. Duration of performance degradation

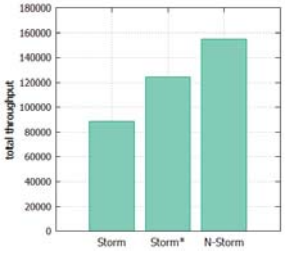

Figure. 8. Total throughput of different methods

throughput, which is 25% higher than that of Storm*, and 75% higher than that of Storm.

In this subsection, N-Storm only migrate one Executor at a time, meaning that N-Storm only needs to stop and restart one Executor. In the next subsection, we increase the number of Executors to be migrated to evaluate the performance of N-Storm for multiple tasks migrations.

C. Performance of N-Storm for Multiple Task Migrations

In this subsection, we measure the performance of N-Storm for multiple task migrations. We still set T_S=10 seconds and T_W=3 seconds. Fig. 9 shows the different performance degradation duration under different values of M. When M is 1, 2, and 4, there are few differences in the duration of performance degradation in the five levels. And there is a small difference in the duration time at the 60% and 80% levels. However, when the value of M is 8 and 16, the duration of performance degradation significantly increases, and the system throughput drops to 0 in many cases. Especially when M=16, the duration of 60% performance degradation is 118 seconds. This is because that nearly half of Executors need to be migrated for each task re-assignment.

Fig. 10 shows the total throughput of N-Storm under different values of M. Only when M=16, the total throughput decreases significantly, and the total throughputs of M=1, 2, 4 and 8 are not significantly different. Surprisingly, the total throughput at M=4 is greater than that at M=2. By looking at throughput per second over the entire experiment, we find that when M=4, after two or three task re-assignments, the throughput of the system after the migration increases by about 5% compared to the initial state of the system. This is because that the migrated Executors are randomly selected, which means that the initial task assignment is not as effective as the random task assignment.

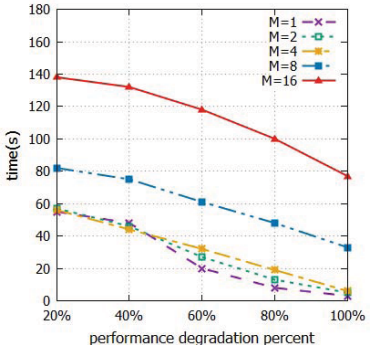

Figure. 9. Duration of performance degradation of multiple tasks migration of N-Storm

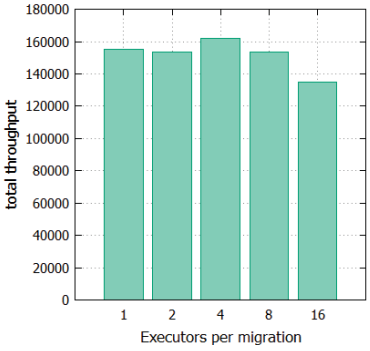

Figure. 10. Throughput of multiple tasks migration of N-Storm

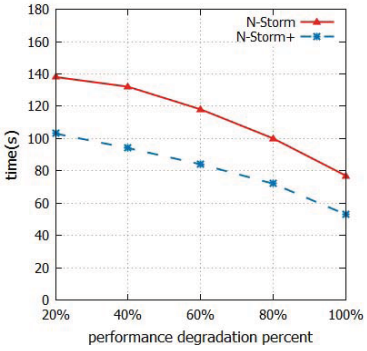

Figure. 11. Comparison of duration of performance degradation between N-Storm and N-Storm+

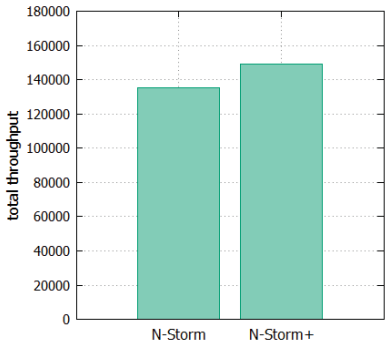

Figure. 12. Comparison of the throughputs of N-Storm and N-Storm+

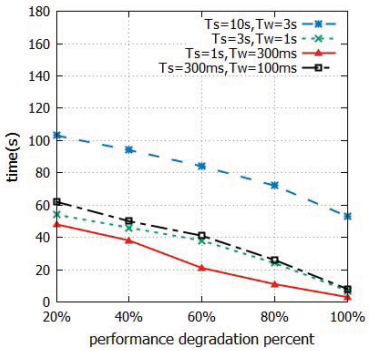

Figure. 13. Duration of performance degradation under various cycles of N-Storm+

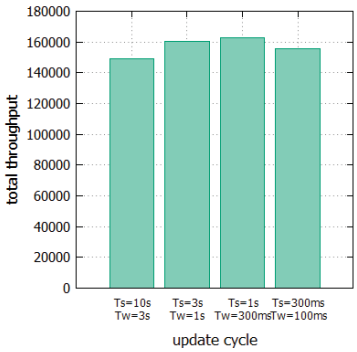

Figure. 14. Throughputs under various cycles of N-Storm+

D. Performance of the Delayed Killing of Executors

In this subsection, we still set T_S=10 seconds, T_W=3 seconds, and M=16. Previous experiments show that N-Storm does not perform well in this setting. We add the delaying killing Executors strategy to N-Storm and set the delay time to 2 seconds. This delay is sufficient to ensure that tuples in the task queue are completed and slightly less than the initial T_W. We test the performance of N-Storm+ with N-Storm under the same settings.

Fig. 11 shows the duration of the performance degradation. We can see that the degradation duration time of N-Storm+ is less than that of N-Storm in all five degradation levels. Among them, the duration of 60% performance degradation of N-Storm+ is 84 seconds, which is 29% less than that of N-Storm. The duration of 100% performance degradation of N-Storm+ is 53 seconds, which is 31% less than that of N-Storm. Fig. 12 shows that the total throughput of N-Storm+ is 11% higher than that of N-Storm. In conclusion, the delayed killing of Executors in N-Storm+ can improve performance of task migrations.

E. Impact of T_S and T_W

In this subsection, we study the impact of T_S and T_W on the performance of N-Storm+. All experiments in this subsection are based on N-Storm+ and M is set to16. Since the Supervisor first gets the message from Zookeeper and sends it to Workers through the k/v store, T_W should be less than T_S. We still let T_S be about three times of T_W, and gradually reduce the value of T_S and T_W.

Fig. 13 shows the duration of performance degradation under different T_S and T_W values. We observe that the performance degradation time of the system continues to decline with the decrease of T_S and T_W in the first three groups of experiments. However, in the last group of experiments, when T_S=300 ms and T_W=100 ms, the performance degradation time of the system is longer than that of the middle two groups. The best performing set is T_S=1 second and T_W=300 ms. The duration of 60% performance degradation is only 21 seconds, and the duration of 100% performance degradation is almost 0 second. Fig. 14 shows the total throughput under different update cycles, and the results obtained are consistent with Fig. 13, *i.e.*, the shorter the performance degradation time, the higher the total throughput of the system.

In conclusion, the experimental results confirm our previous analysis. In general, the shorter the synchronization cycle is, the better the performance of the system will reach. However, this cycle cannot be too short, otherwise the Supervisor and Workers will be overloaded, which will introduce side-effects. Compared to the experimental results of N-Storm in subsection V.*C*, we can conclude that the duration of 60% performance degradation of N-Storm+ is reduced by 82%, demonstrating the efficiency of the optimization strategies.

VI. Related Work

The default scheduling strategy in Storm adopts a simple round-robin method. Many algorithms have been proposed to optimize the allocation of resources and improve the fault tolerance of the system [14-17], but they

are offline algorithms and cannot cope with fluctuating data flows. If the data flow rate changes, online scheduling methods [4-5, 18-20] can re-calculate the new schedule and update the task assignment, which triggers task migrations. However, due to the process-level task management mechanism in Storm, each task migration will result in severe performance fluctuation.

The method proposed in [6] is similar to that in [7]. They dynamically adjust the degree of component parallelism, *i.e.* the number of Executors for a component, by monitoring Executors' data arrival rate and data outflow rate. Li *et al.* [9] proposed a dynamic algorithm for Topology optimization. Some researchers used machine learning methods to obtain the optimal Storm parameter configuration, such as AdaStorm [12] and OrientStream [22]. These methods implemented an elastic mechanism on Storm that can dynamically adjust parallelism and other parameters, which will trigger task splitting or merging.

T-storm [4] proposed an optimization scheme to delay the stop of Workers. However, delaying the shutdown of Workers has to use more resources. Moreover, it needs to stop for 10 seconds. Therefore, this method cannot effectively solve the performance fluctuation problem. Yang *et al.* [21] proposed a smoothing task migration idea for Storm in a two-page way. They analyzed the performance cost of the task migration in Storm and proposed to change the task-migration granularity from Worker to Executor. Our study is inspired by this visionary work but we present a systematic framework for the Executor-level task migration. In addition, we proposed new optimization strategies for the original Executor-level task migration to make it efficient for multiple tasks migration, which were discussed in Section IV.

Cardellini *et al.* [13] proposed a strategy of automatically changing parallelism and designed a Storm based stateful task migration method. Li *et al.* [8] designed an elastic mechanism that needed to monitor the state of the system. They considered stateful operators and used an additional global state manager to persist the states of the operators to achieve stateful operator migrations. Shukla *et al.* [22] proposed several data-flow checkpoint and migration approaches. They also focused on stateful migrations. The studies [8, 13, 22] were all dedicated to migrate operator states, which were conflicted with the basic assumption of Storm. This is because Storm does not maintain the consistency of operator states [1, 2], as Storm modules are stateless and all states are kept in Zookeeper.

VII. Conclusions

In this paper, we propose N-Storm, a thread-level task migration scheme for Storm. Instead of using the process-level migration scheme, N-Storm employs a thread-level scheme for task migration. Particularly, we add a key/value store on each Worker node to make Workers be aware of the changes of the task schedule, so that the Worker process on a Worker node can manage its Executors at run time, i.e., killing existing Executors or starting new Executors. With this mechanism, we can avoid the stop of unnecessary Executors and Workers during a task migration, and thus improve the performance of task migration. We further propose two optimizations to make N-Storm efficient for multiple tasks migrations. The experimental results on a real Storm cluster show that our proposal can significantly reduce the time duration of performance degradation, and eliminate the stop time during task migrations.

Acknowledgment

This work is partially supported by the National Science Foundation of China (No. 61672479).

References

[1] M. Lyu, P. Jin, Z. Zhang, S. Wan, L. Yue. STEM: A simulation-based testbed for electromagnetic big data management. In *SEKE*, 2018, 230-229

[2] P. Jin, X. Hao, X. Wang, L. Yue. Energy-efficient task scheduling for CPU-intensive streaming jobs on Hadoop. *IEEE Transactions on Parallel and Distributed Systems*. 30(6), 2019, 1298-1311.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, Storm @Twitter, in *SIGMOD*, 2014, 147-156.

[4] J. Xu, Z. Chen, J. Tang, and S. Su, T-storm: Traffic-aware online scheduling in storm, in *ICDCS*, 2014, 535-544.

[5] J. Zhang, C. Li, L. Zhu, Y. Liu, The real-time scheduling strategy based on traffic and load balancing in Storm, in *Proc.s of HPCC/SmartCity/DSS*, 2016, 372-379.

[6] T. Fu, J. Ding, R. Ma, M. Winslett, Y. Yang, DRS: Auto-scaling for real-time stream analytics, *IEEE/ACM Transactions on Networking*, 25(6), 2017, 3338-3352.

[7] M. Sax, M. Castellanos, Q. Chen, M. Hsu, Aeolus: An optimizer for distributed intra-node-parallel streaming systems, in *ICDE*, 2013, 1280-1283.

[8] J. Li, C. Pu, et al. Enabling elastic stream processing in shared clusters, in *CLOUD*, 2016, 108-115.

[9] C. Li, J. Zhang, and Y. Luo, Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm, *Journal of Network and Computer Applications*, 87, 2017, 100-115.

[10] X. Hao, P. Jin, L. Yue. Efficient storage of multi-sensor object-tracking data. *IEEE Transactions on Parallel and Distributed Systems*. 27(10), 2016, 2881-2894

[11] Atoms in Clojure, *https://www.clojure.org/reference/atoms*.

[12] Z. Weng, Q. Guo, et al. AdaStorm: Resource efficient storm with adaptive configuration, in *ICDE*, 2017, 1363-1364.

[13] V. Cardellini, F. Presti, M. Nardelli, G. Russo, Optimal operator deployment and replication for elastic distributed data stream processing, *Concurrency and Computation: Practice and Experience*, 30, 2018, e4334.

[14] B. Peng, M. Hosseini, et al. R-storm: Resource-aware scheduling in storm, in *Middleware*, 2015, 149-161.

[15] M. Farahabady, H. Samani, et al. A QoS-aware controller for apache storm, in *NCA*, 2016, 334-342.

[16] J. Jiang, Z. Zhang, B. Cui, Y. Tong, N. Xu, StroMAX: Partitioning-based scheduler for real-time stream processing system, in *DASFAA*, 2017, 269-288.

[17] L. Su, Y. Zhou, Tolerating correlated failures in massively parallel stream processing engines, in *ICDE*, 2016, 517-528.

[18] W. Qian, Q. Shen, J. Qin, D. Yang, and Y. Yang, S-Storm: A slot-aware scheduling strategy for even scheduler in storm, in *HPCC/SmartCity/DSS*, 2016, 623-630.

[19] D. Sun, G. Zhang, et al. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments, *Information Sciences*, 319, 2015, 92-112.

[20] D. Sun, G. Zhang, et al. Building a fault tolerant framework with deadline guarantee in big data stream computing environments, *Journal of Computer and System Sciences*, 89, 2017, 4-23.

[21] M. Yang, R. Ma, Smooth task migration in apache storm, in *SIGMOD*, 2015, 2067-2068.

[22] A. Shukla, Y. Simmhan, Toward reliable and rapid elasticity for streaming dataflows on clouds, in *ICDCS*, 2018, 1096-1106.