A managed distributed processing pipeline with Storm and Mesos

Dragos Dena, Mihai Bucicoiu, Mircea Bardac
Faculty of Automatic Control and Computers,
University "POLITEHNICA" of Bucharest,
dragos.dena@gmail.com, mihai.bucicoiu@cs.pub.ro, mircea.bardac@cs.pub.ro

***Abstract* — Running an *always-on* distributed system that has the task of continuous message processing can raise a series of problems like resource sharing with the other services running in the cluster, controlling the locality of the worker processes and the ability to move workers from one machine to another without impacting the functionality of the system or inducing data loss. In this paper are making an in-depth analysis of these problems and design a system that addresses them. Our computation pipeline takes the form of a Storm topology and with dynamic resource management and isolation for it being achieved through Apache Mesos. The focus is on isolating and distributing resources between the nodes of one topology instance and other services.**

Keywords — Storm; Mesos; distributed processing pipeline; resource management; resource isolation; virtualization

I. INTRODUCTION

The management of an *always-on* real-time system can prove a difficult task. This system is expected to run as a distributed system in a cluster, sharing the resources with other services, like Hadoop. It has the additional constraint that it shouldn't be turned off. We will enumerate the problems that occur and, later on, we will propose a series of solutions for these problems.

A. Motivation

The first problem is scalability. How do you implement a system like our pipeline that needs to dynamically scale when adding machines? Besides the design that should bring a speedup as close to linear as possible, we are also faced with the problem of how the implementation will deal when a new node is added to our system. This situation can arise in multiple scenarios. Some machines are added or removed from the cluster and this must be done in such a matter that it will not bring data loss. Another scenario is that the service you are serving is faced with a data spike and you must dynamically supply your cluster with new machines.

Another problem would be resource allocation between multiple frameworks running in your cluster. Regardless of how well the planning is done when deciding the appropriate cluster configuration based on your expected workload, you will most likely be faced with data spikes that will bring some resources to or above their limits, or you can get in the opposite scenario which is arguably worse – underutilization of your cluster resources. This is quite a problem because if 90% of the time you are only using 5% of your CPU or your memory, it means that you acquired other machines for your other services which are also underutilized. In the long run this can prove to be an expensive issue.

What would we want to do in this case is to be able to take those free resources and give them to some other services, thus, reducing the number of physical machines needed. This is also the case for our processing pipeline, which is designed in such a way that it should share the same cluster with other frameworks like Hadoop or Spark. This type of resource sharing can prove quite useful when you can balance jobs that have different workloads depending on the time of the day. For example, if we would use our processing pipeline for real-time computation for a website service we will probably have more traffic during the day, which leaves room to schedule more resources during the night for asynchronous jobs like Hadoop Map-Reduce.

B. Used technologies

For our pipeline we are using Storm [1], which is an open-source project that provides the means for doing distributed real-time computation. Unlike Hadoop Map-Reduce jobs, which are designed for batch processing and last on average minutes (though indeed they can span for hours depending on the dataset against which these jobs are running), a Storm unit of execution, which is called a *topology*, does continuous computation. A Storm topology is defined as a graph of logical entities called *spouts* (from which the messages originate) and *bolts* (which do the message computation/transformation). These nodes, spouts and bolts, are connected through edges called *streams*. Each node in this graph, which is a logical entity, has more instances spread across the cluster. These instances are actual threads called in the Storm terminology *executors*. The processes in which the executors are running is called a *worker*. There can be multiple workers on one machine and they are managed by a daemon called the *supervisor*, which, in turn, are managed by the Storm master process – *nimbus*. The nimbus process has only one instance per cluster.

While Storm provides the means for doing this continuous computation and manages its workers, it still requires some additional supervision for it to run in a fully managed mode. For example, Storm has a scheduling system that automatically reallocates workers on new machines if one goes down. It will also display the status of the topology and allow the user to restart/rebalance it. Still, the default Storm scheduling system

has its limitations. It has a fair strategy, trying to allocate the same number of workers on all the machines in the cluster.

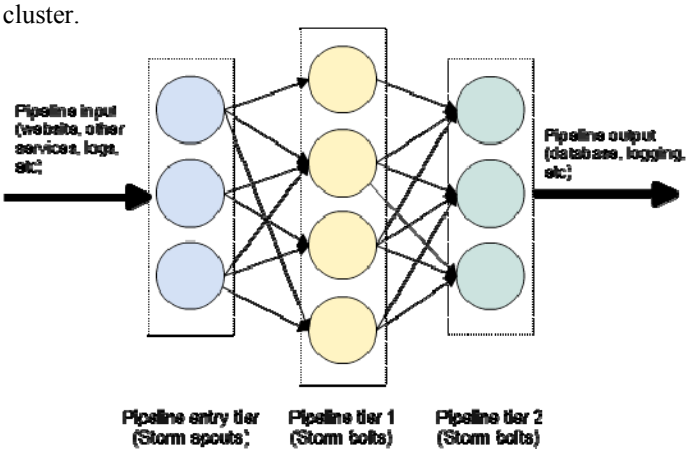

Fig. 1. *The processing pipeline based on Storm*

This does not solve our problem, as this does not imply that the resources are actually spread evenly between the workers, e.g., on one machine you could have only spouts that are I/O intensive, effectively wasting the CPU time on that node. It also does not factor in other frameworks running on those machines when doing the scheduling. One other important limitation is that the topologies are statically defined. There is no way to extend dynamically the topology with new executors without restarting it. It is also not possible to pause a set of nodes, temporary removing them from the topology while letting the data from the paused nodes drain.

We will look into what parts of Storm need to be changed/configured to solve the mentioned problems. Also, so we can provide resource sharing and isolation between Storm and other systems running in that cluster, we are integration torm with Apache Mesos in our design [2]. Mesos achieves these goals through a *master* process that decides which frameworks receive what resources. Each framework must implement a *scheduler* that will request new resources and it will get notified when new resources are available. Mesos *slaves*, running on each machine, will execute the tasks in cgroups containers.

C. Comparasion with related work

Unlike the Storm integration with Mesos that is currently used in production within Twitter, we will be more focused on isolating and distributing resources between the nodes of one topology instance, instead of between multiple topologies, thus introducing a more fine-grained control.

Also, compared with the Mesos integration for Hadoop, our scenario involves resource management for processes that cannot be restarted. This additional restriction brings additional problems like the need of pausing and moving workers.

D. Paper structure

We will start by presenting general aspects of the Storm processing pipeline, describing its main components in our design. We will also outline why resource sharing and isolation is our main goal and how we will achieve it trough integration with Apache Mesos.

In the "Customizing Storm to our needs" section we will describe step-by-step what Storm components require customization and configurations should be deployed so our resource sharing and isolation goals can be enabled.

Finally, in the "Mesos components" section we will enumerate the Apache Mesos components that require customization and configuration, also explaining how it needs to be done.

II. THE STORM PROCESSING PIPELINE

A. General aspects

As it can be seen in Figure (1), the Storm processing pipeline consists of more tiers. We only used three tiers in our diagram, but it can be easily generalized to as many as needed.

The input layer can be customized depending on the needs: it can either receive *hit events* directly from a website, it can stream data from other services, or it can read it from log files as they are written.

The output layer is again customizable. In most cases we will probably write to a persistent storage.

Any other layer on the way serves as a computation layer. The computation can be an aggregation or any stream processing as needed in the application.

In the diagram, we did a logical representation of the various components, neglecting the physical location. The first tier is essentially a storm *spout*, while the last two tiers are each a storm *bolt.* Each component of a tier is a storm *executor*. The edges connecting all these executors represent how data flows. The actual communication is done through 0mq sockets [3] between the workers, while the messages are buffered in inbound/outbound disruptor queues [4] in each worker process.

To which executor in the next tier a given message is passed is decided by a Storm concept called a *grouping*. With the grouping concept, you can shuffle messages randomly between all executors, partition them based on a key in the message or have a one-to-one communication system. As we will see in a later section, we are interested in modifying the default groupings given by Storm so we can avoid certain executors if configured as such.

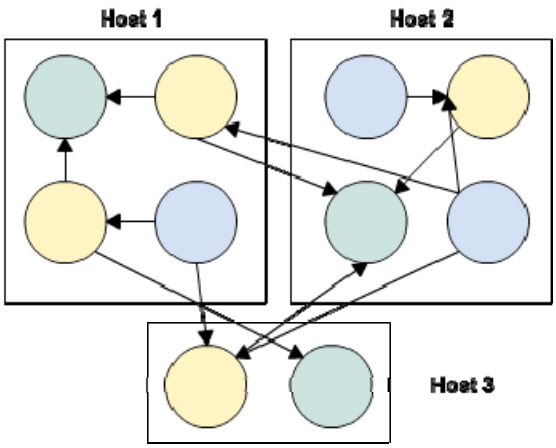

Fig. 2. *A physical representation of the same pipeline*

In our design, since we are not having dedicated machines for specific services, there will not be a need to have each tier run isolated on a set of machines. Thus, the actual physical representation of our topology will look like in Figure (2). Regardless of the tier an executor belongs to, it can get scheduled on any host and be collocated with any other executor, regardless of its type. This sort of shuffling is actually the default built-in mechanism in the Storm scheduler, although with enough customization, locality restrictions can be implemented.

B. *Resource sharing considerations*

As we mentioned earlier, some companies already use in production a Storm version that is integrated with Mesos. Still, that version does not fully satisfy our needs, as we are not interested in resource isolation between topologies, but between different tiers in our processing pipeline, thus we need resource isolation for internal components of a given topology. Translated into Storm terminology, we would need to allocate resources for a specific spout or bolt, not for all the topology.

We are interested in this type of resource allocation because different tiers have different resources needs. We can make this assumption because otherwise two consecutive tiers would have been merged into a single one if they had the same profile. For example, we could identify the following common profiles by which we will want to allocate resources with a different strategy:

1) The *entry tier* will read the data and push it downstream in the topology. Thus, we can expect it to be I/O intensive. If it reads the data from the disk, we will need to allocate it more disk time. If we need to receive the data through the network, we will need to allocate more network bandwidth.

2) *Processing tiers* are expected to be heavy computation intensive, so we will need to allocate them more CPU and memory.

3) *Outbound tiers,* in a similar fashion to the entry tier, will either need disk or network bandwidth, as this layer will write to the local storage or send the data to other services.

C. *Integration with Apache Mesos*

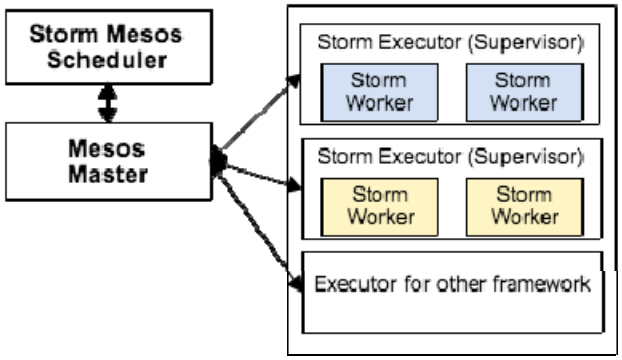

Fig. 3. *The integration between Storm and Mesos*

The integration between Storm and Apache Mesos implies a few things that need to be implemented:

1) A *Storm Mesos Scheduler*. This component is responsible with deciding how to handle resource offers or to make actual resource requests. We also require an external component (a GUI or a command line shell) that will offer the user the possibility to tune the resources used by Storm. That external component will communicate with this scheduler, that will make a resource request to the Mesos Master or it will try to reduce the resources used by an existing executor.

2) The default *Mesos allocator* needs changes so it can fit our needs regarding the policy in our cluster: we need to give priority to our pipeline to take up at least a minimum configured value of resources. That way, if we tune our pipeline to occupy a fixed amount of resources and we start a great number of tasks belonging to other frameworks, we would like to be able to raise the resources occupation level of our pipeline without having a race between all the frameworks.

III. CUSTOMIZING STORM TO OUR NEEDS

In this section we will try to cover all the Storm components we need to modify so we can get more control regarding:

1) *Workers locality*. Depending on their type, we might want certain type of workers be positioned on certain machines. While the usage of Mesos, which takes care of the resource management, reduces the need of making in Storm the scheduling effort that would place workers on certain machines depending on their resource consumption profile, we would still want a degree of control here. One example where that might be

useful is the scenario in which we would want to place a portion of our workers in an external IaaS provider like Amazon Web Services to handle traffic spikes.

2) The ability to *pause a certain portion of our pipeline*. This need occurs mainly due to a Storm limitation – you cannot dynamically add workers to an existing topology while it is running without issuing a rebalance/restart operation. A solution for this would be to have allocated more workers than needed to begin with and have the unneeded ones in a pause/pending mode. When the extra computing power is needed, those workers can be woke up.
3) *A controlled way to migrate workers while avoiding data loss*. This situation can rise in multiple scenarios:
    a. A restart is needed for one machine in the cluster. Before issuing the restart, it is ideal to move the workers to another host without loosing the data.
    b. New machines are added to the cluster and some workers should be moved to those machines to spread the load.
    c. You are faced with a traffic spike and you want to migrate some workers to dynamically allocated machines like virtual machines in Amazon Web Services.
4) The need to have *exclusively executors of the same type in a worker*. By having executors (threads) from different tiers in the same worker, it would make it impossible to achieve resource isolation between multiple tiers in our pipeline. Thus, from now on, we will consider that workers can be of a given *type* - type that is associated with the type of the node/tier in the pipeline. This can be easily achieved through the Storm scheduler.

We require a way to control all the needs we mentioned earlier. The worker locality will be controlled through configurations files that will specify a mapping from workers of a given type to a specific supervisor/host. If no locality restriction is specified, then the worker should be able to get allocated anywhere.

For other commands like pausing a worker or moving a worker from one host to another need a way to interact with the user (the system administrator). This interface can be a command line shell or a graphical user interface. We are not focusing on describing the user interface, as it is not within the scope of this paper. Instead, we are focusing on what Storm components should be modified to achieve our goals and how we should communicate with these components so they can process our requests.

A. Controlling worker locality

Storm provides a pluggable mechanism through which the assignment, a mapping executor to worker slot, can be customized. It is sufficient to write a class that implements Storm's IScheduler interface. This method will be called by the Storm master, *nimbus*, each 10 seconds. Here, we need to read a configuration file and see if there are any locality restrictions. If yes, we will need to make sure the workers are started on the correct host.

We also need to make sure and check if the configuration changed and the worker is already started on another machine. In this case, we will need to move that worker to a new machine, a use case that we will cover later on.

B. Pausing the pipeline

Let us first consider the case in which we want to pause the smallest possible entity: a storm task. By pausing it, we mean it will effectively get removed from the topology, no data going inside/outside that node.

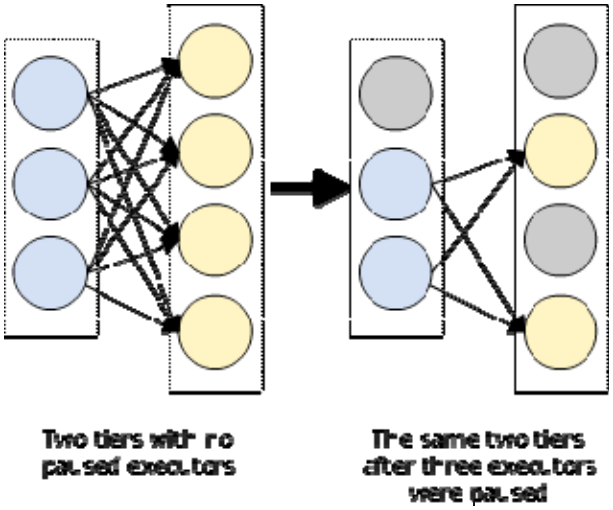

Fig. 4. *Pausing a set of executors*

To achieve this, we need to extend the Storm class CustomGrouping and in its implementation to always avoid the task that we want to ban. By doing this and configuring Storm to use our grouping, which will be called to determine the destination tasks on each message, always avoid the task, isolating it. Obviously, we will need to communicate the tasks we want to avoid to the groupings in all the workers in our cluster. For this, we have different approaches: we could use sockets to connect to each worker and send them the commands or we could use a distributed coordination system like ZooKeeper [5].

We would also want to extend the ISpout/IBolt interface and add a new method – *pause*, which must be implemented by the components that require special behavior when paused. One example of such a behavior would be draining all your data to the next tier in the processing pipeline.

When pausing multiple tasks, we would need to do it in a synchronous matter – pause one task, wait for it to finish, pause the next one. One obvious way to minimize the time waiting for the tasks to pause themselves is to sort the nodes we want to pause depending on their position in the pipeline (if closer to the end of the pipeline, then ranked higher).

An example of pausing a set of tasks can be seen in figure (4). Here, one task from the first tier and two tasks from the second tier were paused.

Of course, it should be possible to un-pause the tasks by notifying the grouping object, which is responsible for deciding the recipient of all the messages, that it should stop ignoring a given task.

C. Migrating workers to other machines

The worker migration is triggered when the Storm scheduler changed the topology assignment. We require hooking into the default scheduler policy here and, when wanting to move a worker, we should send a message to that worker notifying it that it will get relocated to another machine. The user should be able through an interface to notify the scheduler that a worker should be moved.

Migrating a worker involves migrating all the tasks that are contained in that worker. To achieve this, we would want to first pause all the tasks, making sure that until the worker was successfully moved to another host, no messages will be queued to it.

Further, we are adding two more methods to the ISpout/IBolt interface:

```
byte[] getInternalState();
void resumeExecution(byte[] internalState);
```

The first method, *getInternalState,* should return the serialized state of the task, if any. This will be called after pausing the worker, before it is moved. The second method, *resumeExecution,* is to be called after the worker moved, passing it the internal state as given by the worker prior to moving it. Basically, we ensure through this mechanism that the worker will be started on the new machine with the same state.

IV. MESOS COMPONENTS

In this section, we will analyze the Mesos components we need to implement and how they should be implemented. Beforehand, we will mention the Mesos setup we will use.

There will be one Mesos Scheduler. This will make resource requests to the Mesos master and will handle resource offers when resources are available.

Another important component is the Mesos allocator that sets the policy regarding how resources will be allocated. We want fine-grained control to the resources used by our pipeline through this.

We also require a user interface that will allow the user to communicate with the Mesos allocator and make the requests to increase or decrease the resources used by our pipeline.

We need Mesos tasks to start up a Storm supervisor. Each cgroups container will run exactly one supervisor. That supervisor will only start workers of a given type (from only one tier in the pipeline). Our customized Storm scheduler will guarantee this. This is needed to achieve resource isolation between different tiers in the pipeline.

Another very important aspect that is required by our Storm scheduler so resource allocation will work correctly is to only allocate one active worker per supervisor (a worker is considered active in our case if it is not paused). That way know exactly how many resources a worker uses. Obviously, we also require a fixed number of Storm executors per worker for our allocation algorithm to make sense.

A. Setting the quantity of allocated resources

It should be noted that it will not be possible to allocate resources at a finer granularity that the number of workers. Thus, if we have *W* workers of a given type *T*, and workers of type *T* that consume the quantities *[Q1, ... Qn]* for the resources *[R1, ... Rn],* then we have the following restrictions regarding resource allocation:

1) We can set for a given resource *Ri* the quantity that is used by the workers of type *T* anywhere in the set *{k * Qi for 0 < k <= n}*.
2) When setting the quantity used by workers of type *T* for the resource *Ri* to *k * Qi*, then for each resource *Rj* we also need to set it at *k * Qj*.

B. The Mesos Scheduler for Storm

To implement a Mesos scheduler we just need to implement an interface. We will not analyze all the methods that must be implemented in the Mesos scheduler interface, but we will talk about the important ones that define how the system works.

Before going on, we should note that we would require keeping the state of the pipeline in this component. Thus, we need to know which Storm workers are paused and for which tier. We will also require knowing how many resources a worker of a given tier occupies.

When resources are available in our cluster, a method called *resourceOffers* will be called. We will want to look what workers are paused, check their resource consumption profile and try to fit as many as possible in the available given resources until we reached the global resource limit for our pipeline. Any other resources should be rejected.

When a task is lost, the *statusUpdate* method is called. We do not need to worry about rescheduling the workers on another supervisor, as the Storm scheduler automatically takes care of it, but we need to update our internal data structures, noting down what workers went down. The Storm scheduler, when faced with a supervisor going down, must make sure the worker is started again in pause mode. Thus, in the Mesos scheduler, we should consider that all workers that are in a lost Mesos task/Storm supervisor are in the paused state.

C. Changes required in the Mesos allocator

The main objective for the allocator is to guarantee that our processing pipeline is using exactly the number of resources we configured it to. For example, we would want through a user interface to tune the pipeline to use 80% of all the available CPU power in the cluster. The user interface will need to talk to the allocator to process its request.

When increasing the number of resources, we should just send resource offers to the Storm Mesos scheduler if available.

If no resources are available, we should wait for a given time for tasks belonging to other frameworks to finish. This will directly impact the time the resource allocation will take to converge to what the user configured.

When decreasing the number of resources used by the pipeline, we should see which workers are up and stop them until we are under the accepted threshold. We should determine which supervisor is associated with the workers we want to stop and kill the Mesos task that is executing that supervisor. Various policies can be implemented to decide which workers will be paused. By default, we would want to choose the minimum number of workers to be paused.

V. CONCLUSIONS

We analyzed and solved the problems regarding resource sharing and seamless worker migration in the system we designed based on Storm and Mesos.

Storm offers the foundation for the real-time computation pipeline we proposed and, with enough changes and configurations, it can be made to run in a managed mode where we can pause workers, move workers around and decide the locality based on the profile of that worker.

Using our Storm modifications, adding Mesos on top of it and with a few more other modifications in the Storm scheduler we can make our processing pipeline run in a managed mode in the cluster, sharing physical resources with other services, while having a guaranteed quantity of resources. This guaranteed quantity of resources can be tuned depending on the expected workload dynamically.

[1] Jonathan Leibiusky, "Getting started with Storm", O'Reilly Media, September 2012

[2] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", University of California, Berkley, September 2010.

[3] Martin Sustrik, "0MQ: The theoretical fondation", July 2011

[4] Martin Thomson, Dave Farley, Michael Barker, Patricia Gee, Andrew Stewart, "High performance alternative to bounded queues for exchanging data between concurrent threads", May 2011

[5] Patrick Hunt, Mahadev Konar, Flavio P. Junkiera, Benjamin Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems"

[6] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, Sean Quinlan, "Availability in Globally Distributed Storage Systems", 2010

[7] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", 2004

[8] Tom White, "Hadoop: The Definite Guide, Third Edition", 2010

[9] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, Majors, J., Manzanares, A., Xiao Qin, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters", 2010

[10] Kc, K., Anyanwu, K., "Scheduling Hadoop Jobs to Meet Deadlines", 2010

[11] Paul B. Menage, "Adding Generic Process Containers to the Linux Kernel", 2007

[12] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar and Andrew Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters"

[13] Rajesh Raman, Miron Livny, Marv Solomon, "Matchmaking: An extensible framework for distributed resource management", 1999

[14] R. Deaconescu, R. Rughini□, and N. □ăpu□, "A BitTorrent Performance
Evaluation Framework," in The Fifth International Conference on Networking and Services ICNS 2009, 2009, pp. 354–358.

[15] S. Costea, M. D. Barbu, and R. Rughini□, "Qualitative Analysis of Differential Privacy Applied Over Graph Structures," in The 11th International RoEduNet Conference, 2013, pp. 1–4.

.