Experimenting with Recovery of Stream Processing Systems

Editor: Stefanos Kalogerakis
FORTH-ICS & Computer Science Department, University of Crete, Greece

I. INTRODUCTION

In this project, the main focus was to experiment with fault-tolerance and recovery mechanisms of Stream-Processing Systems(SPSs) and evaluate whether aligning recovery tasks with externally stored state can speed up the recovery process.

For the purposes of this project, the Apache Flink SPS is utilized along with its state-of-the art incremental distributed checkpoint capabilities. In a nutshell, incremental checkpointing is currently performed initially locally (on the local RocksDB instance of each node whose partial state is being checkpointed) and then followed by a remote copy to a distributed HDFS instance for reliability. After a crash of a Flink Task Manager and during its recovery, when the failed Task Manager cannot recover its local state(e.g., when the machine hosting it is not booting up again) it fetches state from a remote replica of HDFS. This whole mechanism is put to the test.

The notion of alignment between processing task and external state is expected to significantly improve the efficiency of recovery compared to transferring state from a remote replica as the original version of Flink. To this end, the key challenge is to control task recovery decisions and task placement in the Flink SPS to ensure that recovery tasks are placed at the same nodes with the HDFS replicas required throughout the recovery process. This requires cross-layer coordination between two different middleware systems(Flink and HDFS) with the minimal intrusion in the underlying software possible. While HDFS is currently used as a backend, the principles are general and can be applied to other distributed storage backends

II. RELATED WORK

In the recent past, topics such as fault-tolerance and state migration in stream processing systems are getting increasingly important and gain a lot of traction. While significant progress has been made in understanding how to checkpoint state remotely and using it to recover to a consistent state of a streaming job, handling efficiently very large state (of sizes that cannot possibly fit in the memories of backup nodes via active replication) in such systems is still a challenging and multi-faceted problem.

An interesting work that was the inspiration for this project is Rhino [1]. Rhino is described as a library for efficient management of very large distributed state that is compatible with stream processing systems based on the streaming dataflow paradigm. In a nutshell, it introduces a library that is built upon two protocols, a handover, and a replication protocol. The handover protocol is used for processing and state migration of running operators among workers, while the replication protocol provides proactive and asynchronous incremental state checkpointing on a set of workers. The protocols are considered to be tailored for resource elasticity, fault-tolerance and runtime query optimizations.

While Rhino's ideas and results may seem impressive at a first glance certain observations lead us to suggest alternative routes. Rhino is a library that implements the protocols and their functionality from scratch, without taking into consideration the existing functionality of SPSs. More specifically, both Rhino protocols essentially reproduce functionality that is already implemented in systems such as Flink and HDFS. The existing implementation examines whether the mechanisms that Flink provides in combination with HDFS, could essentially replace mainly the handover protocol by exploiting locality during state recovery.

III. APACHE FLINK

A. Background information

Apache Flink [2] is an open-source framework and a distributed processing engine built for data streams analysis in real-time. Flink supports stateful computations, at in-memory speed, over both unbounded and bounded data streams at any scale. Executing applications in Flink is handled by either one of the integrated cluster managers (such as Hadoop Yarn, Apache Mesos, Kubernetes) or a standalone cluster. Furthermore, Flink provides a powerful checkpointing mechanism that guarantees exactly once state consistency in case of failures.

Flink follows a master-slave architecture model with three main components: the JobManager, the TaskManager(s) and the client. **JobManager** is the master node which mainly contributes to coordinating each application's distributed execution. Handling checkpoint and failures, scheduling new tasks, keeping track of the state of each stream and operator are some of JobManager's

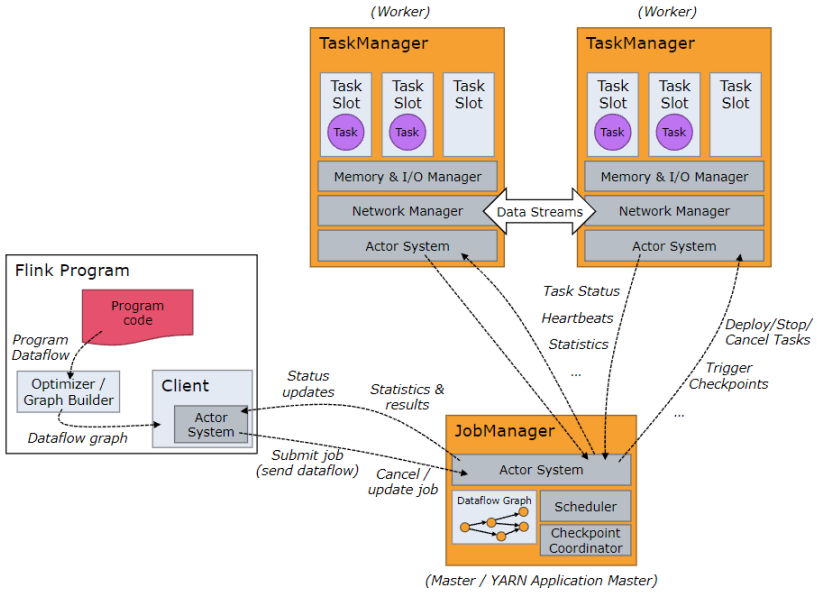

Figure 1: Apache Flink: Architecture Overview [3]

responsibilities. **TaskManagers**, the worker nodes, are responsible for executing tasks and processing dataflows. At least one TaskManager is required for the successful execution of a Flink application. Finally, the **client** is Flink's first interaction with the user. Despite not being part of the runtime environment, the client transforms program code to dataflow graph and submit it to the JobManager.[4][3]

Before presenting any more Flink features, it is important to introduce the notion of **state**.[5] This term is used to describe operations that remember and preserve their information across time. In Flink, state describes snapshots of operators that will know everything that happened in the application up until a particular point in time. Having knowledge of the state enables both rescaling Flink applications, by redistributing state across parallel instances, and fault-tolerance.

In order to provide its fault-tolerance mechanism, Flink combines **stream replay** and **checkpointing**. Checkpoints draw global, asynchronous snapshots in each of the input streams in conjunction with the corresponding state for each of the operators. The mechanism for drawing snapshots is inspired by the Chandy-Lamport algorithm for distributed snapshots. Maintaining consistency in streaming dataflows is also rather challenging, but with its checkpointing mechanism that enables restoring state of the operators and replaying the records from the latest checkpoint, Flink ensures exactly-once semantics without data loss.

The recovery under the aforementioned mechanism is relatively simple. When a failure occurs, Flink as a first step halts the distributed streaming dataflow. Then, the system restarts the operators and resets them to the last fully successful checkpoint, while the input stream is reset to the point of the state snapshot. In production however, it is common for streaming applications to reach GBs and TBs levels, which would result in both very time and resource consuming checkpoints. Therefore, a special type of checkpointing was introduced in Flink 1.3, the incremental checkpointing. In this special case of checkpoints, instead of preserving in each checkpoint the full state, simply maintain the differences between each checkpoint and store only the "delta" between the last checkpoint and the current state. They were built after the observation that the changes between checkpoints is not so great. Incremental checkpoints can have significant performance impact especially when working with large states, which is usually the case.

Currently, incremental checkpointing is only supported with a RocksDB which brings us to the different state backends[7] that Flink provides. Essentially, the state backend specifies how state is represented internally, and how and where it is persisted upon checkpoints. Currently Flink supports two state backends out-of-the-box, the **HashMapStateBackend** and the **EmbeddedRocksDBStateBackend**. The HashMapStateBackend, which is the default state backend, preserves data internally as objects on the JVM, while the EmbeddedRocksDBStateBackend on the other hand, holds in-flight data in a RocksDB that is stored in the TaskManager local data directories.

Choosing between those two state backends, de-

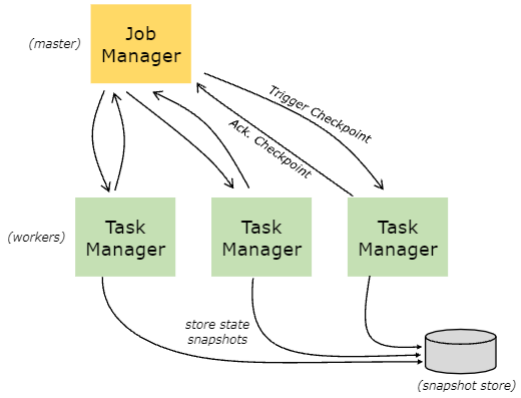

Figure 2: Apache Flink: Checkpoint Snapshot [5]

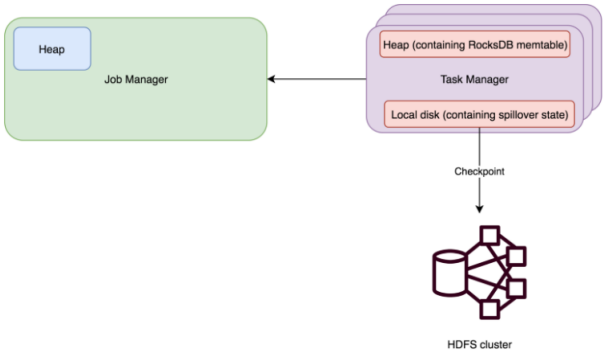

Figure 3: Apache Flink: RocksDB checkpoint [6]

pends on the application and the importance of performance compared to scalability. The HashMapStateBackend is very fast as operations take place in memory, but the size is limited by the memory size within the cluster. In contrast, RocksDB scales based on the available disk space and is the only state backend to support incremental checkpoints, however the performance can drop significantly because of the disk accesses in comparison to in-memory computations. Scaling is a major factor in our task, so working with RocksDB state-backend and incremental checkpoint is the best design choice to make.[8]

B. Environment Setup

In order to work with Apache Flink, it is necessary to set up the required environment for that purpose. Mastering and automating all the steps is very important since Flink must be installed in multiple machines for testing.

1. *Okeanos*

Cloud computing services are getting increasingly popular these days, with more and more organizations choosing to deploy their systems in the cloud to take advantage of features such as virtualization, improved security, and high availability. Under this premise, Okeanos *Infrastructure-as-a-Service (IaaS)* services were utilized to create our cluster. The two main services provided by Okeanos were **Cyclades** and **Pithos**.

The **Cyclades** [9] is the Virtual Compute and Network service of Okeanos. The Cyclades allows building, managing, and destroying virtual machines all of which can be accomplished through any web browser.

Pithos [10] on the other hand is the Virtual Storage service of Okeanos. Pithos enables online file storing, sharing, and accessing files from anywhere, anytime. Pithos combines harmonically with the Cyclades to allow users fast, simple, and secure data sharing across VMs.

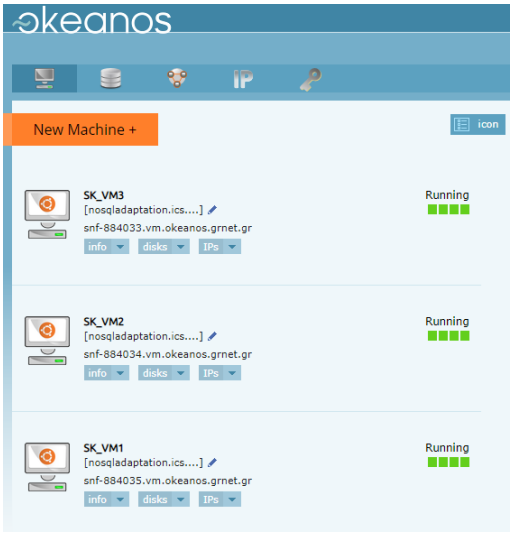

Figure 4: Okeanos VM Deployment

2. *Setup a Multi-Node Flink cluster*

The easiest way to get started with Apache Flink is by downloading the binaries from the official website(link). The standalone cluster version does not require any extra effort to work in its simplest form. Despite that, tweaking the configuration files is obligatory since even simple scenarios fail to execute under the default configurations.

A multi-node Flink cluster is essentially a topology of different machines connected with each other. In order to form a cluster the first step involves installing the standalone version in each machine as mentioned in the previous paragraph. Afterwards, after modifying the corresponding configurations regarding Job Manager and Task Manager IPs across all the machines the cluster should be ready.

Especially in distributed clusters and in production mode, it is highly advised to modify Flink configurations. Modifying the default memory configs, choosing an appropriate state backend or checkpointing options can lead to huge performance differences given the job in hand and the workload.

3. *Building from source*

In order to achieve the goal of the project and the desired functionality, it is also mandatory to change the source code of Flink. The different versions of source code can be found in a public repository in Github.

A crucial detail is paying attention to the version of the project. It is common in open source projects to find multiple variations even of the same version which can lead to confusion. It is always advised to work with the latest stable edition otherwise errors and unexpected behavior is anticipated.

Building the project is proven to be very time-consuming as it takes more than 30 minutes, without including the tests. Despite the fact that the project is split into multiple modules, for some reason changes in code take effect only after building the whole project from scratch. Of course, this makes testing completely ineffective, so determining a better way to build Flink after applying small changes will be necessary for the future if extra functionality is demanded.

After successfully building the source code, a new directory /build-target is produced under the build path. The new directory contains all the files found in a typical installation from binaries, so the installation steps follow the directions from the previous section

IV. HADOOP DISTRIBUTED FILE SYSTEM

A. Background information

The Hadoop Distributed File System (HDFS) [11] is the open-source implementation of a distributed file system that derives from Google File System (GFS) [12], a proprietary highly-available, fault-tolerant data storage file system that operates on commodity hardware. It is optimized when handling big data, as the design purpose was to overcome issues like reliability and speed that traditional databases could not.

Following the GFS architecture, the HDFS incorpo-

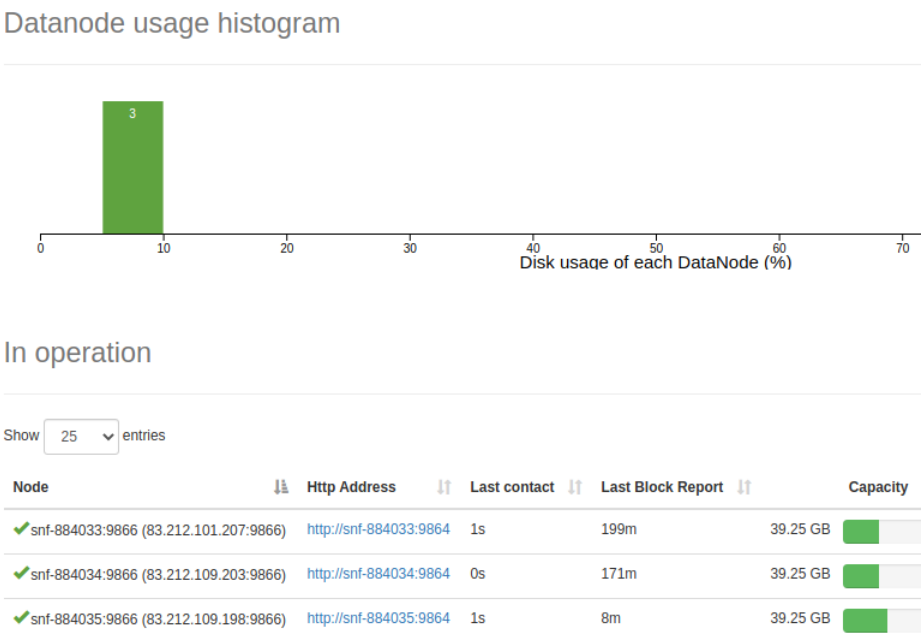

Figure 5: HDFS Multi-Node Cluster status

rates a master-slave topology in which the master node is called **NameNode** and multiple slave nodes that are called **DataNodes**.

While HDFS applies many interesting policies to achieve all its features, this section examines some fundamental block and replica placement policies to provide a better understanding of the upcoming sections.

In HDFS, each system file is stored as a sequence of blocks on DataNodes. The default block size is *128MB* in Apache Hadoop 2.x and 3.x versions (*64MB* in Apache Hadoop 1.x and GFS). The block size is modifiable through the configuration. Before the NameNode can store and manipulate data, files are divided into smaller block-sized chunks stored as independent units.

The number of blocks depends on the size of the file to be stored. All but the last block are the same size as the configured block size (128MB by default), while the last one is what remains of the file. Finally, each block is replicated into several copies based on the replication factor.

It is important to note that the selection for a default block size of 128MB, transpired to perfectly balance the tradeoff between overhead and processing time. In case the block size is smaller, too many data blocks along with metadata can cause increased overhead, whereas in case the block size is very large, the processing times for each block increases.

Data replication methods are commonly utilized in distributed systems to handle unexpected failures and data loss. NameNode creates several copies of every data block (3 by default, can be modified through configuration), distributed across different DataNodes. Replicas are not distributed randomly, as HDFS implements rack awareness policies to ensure high availability and fault tolerance while maximizing network bandwidth.

The HDFS Rack Awareness policies include:

1. One DataNode can store one replica of a data block.
2. The same Rack cannot assign over two replicas of a single block.
3. The number of racks used inside an HDFS cluster must be smaller than the number of replicas

B. Environment Setup

Before start interacting with HDFS, it is crucial to ensure that all systems are installed and operate properly. Since the ultimate goal is to work on a multi-node HDFS cluster, which includes installing HDFS in multiple environments, optimizing the installation process is a necessity, in order to save valuable time.

1. Multi-Node HDFS Setup

Deploying an HDFS Multi-Node Cluster was easier said than done; it required plenty of time and a lot of

trial and error to accomplish successfully. Even setting up the HDFS standalone version was particularly challenging, let alone mastering the installation process to perform in different machines. In addition, some scenarios lead to modifications in the HDFS source code that requires installing the HDFS from the source. All those consecutive steps followed are summarized below:

1. Installing HDFS Standalone version
2. Setup a Multi-Node HDFS cluster
3. Building from source
4. Automate building/testing

Installing HDFS Standalone version

The easiest and most common way to install HDFS is by downloading the binaries from the official Hadoop website. Before setting up a full Multi-Node cluster, it is suggested as a first step to install a local standalone version. However, deploying even a standalone version of HDFS is not a trivial procedure after all. Many different errors were encountered during the installation including permission errors, problems with environment variables and different configurations. For that reason, a 3-4 page manual was created highlighting in great detail all the different installation steps and troubleshooting methods for a variety of scenarios. It is also important to note, that there is an easier and more convenient way to use HDFS using existing Docker images. There are even existing images and solutions for Multi-Node cluster however, dockerized environments present constraints and don't allow high customizability which is a requirement in this project.

Setup a Multi-Node HDFS cluster

As mentioned in the previous section, VMs were utilized in order to create different machines and form a cluster topology. In order to create an HDFS cluster, the most important step is to successfully configure the standalone versions in each machine. The final stage includes modifying the corresponding configurations concerning master and slave IPs across those machines.

The key for a successful MultiNode cluster deployment is homogeneity amongst the different machines. Creating such a cluster is a complex multi-step process and maintaining things the same across machines is the best practice; otherwise monitoring and managing those machines would be impossible.

Building from source

Different project ideas may require working with the HDFS source code which leads to installing HDFS from source. Undoubtedly, this method is the most challenging and time consuming of the bunch, so it is not advised unless there is a specific reason for it.

Compared to building HDFS from binaries, the main difference lies in downloading and building the project from GitHub. Choosing a version can be tricky, as there are numerous versions and not all of them function properly. Also, different versions can have different requirements and can cause errors throughout the building process. In general, stable versions from the main branch are the most preferred ones.

Building the project for the first time is a very time-consuming process as it takes more than 30 minutes excluding tests and the pre-built requirements. The reason is that in the first execution buils the whole Hadoop project and produces binaries similar to those found on the official Hadoop website. From that point, steps 1 and 2 are utilized to deploy the HDFS cluster.

Luckily, modifying the code in HDFS does not require 30 minute build time in later stages. After the initial build, HDFS can be treated and built separately from Hadoop which drops execution time to less than a minute. The result of such build outputs a subset of the whole project and the new files should overwrite the old ones to produce the modified version of the code.

Automate building/testing

Replacing the newly produced files from the previous step can prove time consuming and frustrating at times. For that reason, a bash script was created to automate the building and testing process

```
#!/bin/sh
projectDir="/home/skalogerakis/Projects"

tar -xvf  ${projectDir}/hadoop/hadoop-hdfs-
    project/hadoop-hdfs/target/hadoop-hdfs
    -3.2.2.tar.gz -C $HOME

echo  "Extracting new version completed.\n\n"

rsync --update -raz --progress $HOME/hadoop-hdfs
    -3.2.2/. $HADOOP_HOME

echo  "Copy new version to previous config
    completed\n\n"

rsync --update -raz --progress etc/ $HADOOP_HOME

echo  "Completed"

# Remove temporary files
rm -r $HOME/hadoop-hdfs-3.2.2/
```

Listing 1: Shell Script to automate building/testing procedure from source

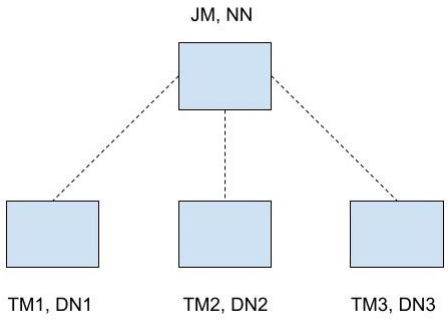

Figure 6: Cluster topology example: Both TaskManagers (TM) and DataNodes (DN) must be co-located on all machines. JobManager (JM) and NameNode (NN) be located on either the same or different machines.

V. CONTROL TASK RECOVERY WITH APACHE FLINK AND HDFS

Big Data frameworks such as Apache Flink are mainly designed to provide on-the-fly analytics on bounded or unbounded data streams. These frameworks usually do not provide built-in data storage capabilities, so combining them with external persistent file systems such as HDFS or S3 is a common use case. In the effort to build a recoverable stream processing system, persisting state in a reliable manner is crucial so that the system can recover from unexpected failures. In addition, HDFS is a distributed file system designed to handle large sets of data in high-throughput while achieving fault tolerance. Therefore, the combination of Flink and HDFS is ideal to accomplish our target.

The original motivation and goal behind the project is to experiment with the recovery process; controlling the task recovery and achieving locality is expected to result in improved end-to-end recovery time. Ideally, the implementation would be fully portable and would require minimal intrusion in the existing technologies, besides the mining and use of management information (ideally through standard APIs) from both systems.

The focus of this work is checkpointing as a common mechanism to maintain state in a fault-tolerant manner. Checkpointing mechanisms typically preserve state checkpoints in some durable medium (failing independently from the checkpointed system), such as a persistent file system. The HDFS distributed file system that we have used as a canonical such medium, can operate on multiple machines in different locations. To this end, in order to consider locality, it is a prerequisite that Flink and HDFS are co-located in the same machines (at least the Flink TaskManager and HDFS DataNodes, to create opportunities for local recovery, as shown in Figure 6)

In order to tackle this problem, an approach would be working on the Flink side and handling where recovered tasks are placed and executed after a failure, given apriori knowledge about the locations of the data. To ensure that, it is required to modify the Flink source code and add extra functionality. The reason is that by default Flink is Task Manager agnostic, i.e. it does not care which Task Manager executes the task at hand, as long as the resource requirements are met. Thus the decision is to identify the IPs of each TaskManager and to restart after failure a TaskManager on the machine that contains the checkpoint state data to achieve data locality (always after meeting the resource requirements).

Identifying where the checkpointed state resides, is a crucial task and not trivial. This is due to the nature of incremental checkpointing and the replica placement policies that HDFS provides. As stated in the background section, incremental checkpointing creates multiple "delta" files to store its state instead of full state snapshot at each checkpoint. These multiple files are spread across different DataNodes in the HDFS after applying its policies. For that reason, there is not a specific DataNode containing all the checkpointed state even in small cluster setups. Luckily, HDFS provides powerful reporting capabilities that allow insights into the replica placement status of a given directory(the checkpointed directory). This information can be used to derive the DataNode with the highest probability of achieving data locality(the more state stored in a machine the more likely it is that a piece of required data may be found in the same machine). However, this probabilistic approach is not good enough to get a clear perspective of the comparison between local and remote recovery, so the experiments are conducted in a manner that clearly separates the two methods(using a small cluster and appropriate replication factor in each case;see section VI)

VI. EVALUATION

This section simulates a simple execution scenario to examine the impact of locality performance-wise during state recovery in streaming applications. For the purposes of this demonstration Apache Flink is used as the processing engine and HDFS as state backend in order to preserve persistent state. The application that was designed may seem simplistic, but it is ideal to focus just on the recovery aspect of the problem and detach it from the application complexity. However, this does not mean that it cannot be generalized to much more complex stateful applications.

Initially, the idea was to work with stateful streaming workloads of the Nexmark benchmark. Nexmark ("Niagara Extension to XMark") is a benchmark suite for continuous queries modeling an online-auction system over continuous data streams (bids). The benchmark is built over a three entities model(person, auction, bid) representing on online auction system. A Person represents a person submitting an item for auction and/or making a bid on an auction while an Auction represents an item under auction. Finally, a Bid represents a bid for an item under auction. While the Nexmark benchmark illustrates a realistic application containing complex workload, the increased complexity that comes with its design and with validating our results lead quickly to adopt a different direction with much less complexity.

A. Experiment description and setup

The experiment is based on a WordCount application; an application that, as its name states, counts the number of words found on a given input. As mentioned earlier the goal of this project was to examine the performance impact of streaming applications during recovery, which can be particularly challenging when it comes to evaluation since streaming applications are naturally unbounded. While the implemented WordCount operates as a streaming application, in order to evaluate the results fairly the input data comes as files with specific file sizes.

All files are generated using the command shown below in the CLI(Command Line Interface), that generates random human readable characters (UTF-8) in a file of predefined size. This command is relatively time consuming since it generates UTF-8 characters, however this decision is crucial, so that it is possible to confirm from the application output that the checkpointing mechanism operates as expected.

$$head\ -c\ 600MB\ (strings < /dev/urandom) > file.txt$$

The experiment timeline follows this pattern: First start by executing the application normally in Apache Flink. Wait for the application to process the given input by observing the Flink Metrics in the WebUI. Since the files are of a predefined size, it is rather easy to understand when the processing is complete. After the first execution of each experiment, it is also possible to track the number of records for every different file size, information that was unknown in the first place. When all the records are processed, then the machine that executes the job is killed; the moment that Flink detects a failed Task Manager it activates its recovery mechanism and restarts the job in an available Task Manager.

To evaluate the performance of local in comparison to non-local recovery, two different experiments are used. In both experiments, the JobManager of Flink and NameNode of HDFS co-locate on the same machine. In the local case, 2 TaskManagers and 2 DataNodes co-exist in the same machine, which is different from the machine where JobManager and NameNode are found, and the replication factor is set to 2. In that way, we are certain that a replica of the data exists on each machine and will be found locally. The non-local case includes once again 2 TaskManagers and this time 1 DataNode found in a different machine than the TaskManagers. The replication factor is set to 1, and since the existing DataNode is detached with the TaskManagers no local replicas can exist.

B. Results

Table I and figure 7 showcase the collective elapsed time results for recovery for files of different sizes. The first column of Table I shows the file size of the processed files in MBs while the next two columns depict the elapsed time for state recovery in ms. It is important to highlight that the existing setup posed restrictions in the conducted experiments. More specifically, the initial idea was to execute experiments for a much larger input size, in the GBs scale. However, the larger the input and respectively the processing required, the larger the resource demands are. When the input file size is larger than 600MB, the backpressure is maximum which implies that machines are overloaded and essentially the processing gets stuck. That lead us to conduct experiments with a max input file size 600MB, as shown in table I

Since the number of records is also a widely used metric, Table II shows the correspondence between file size and the number of records.

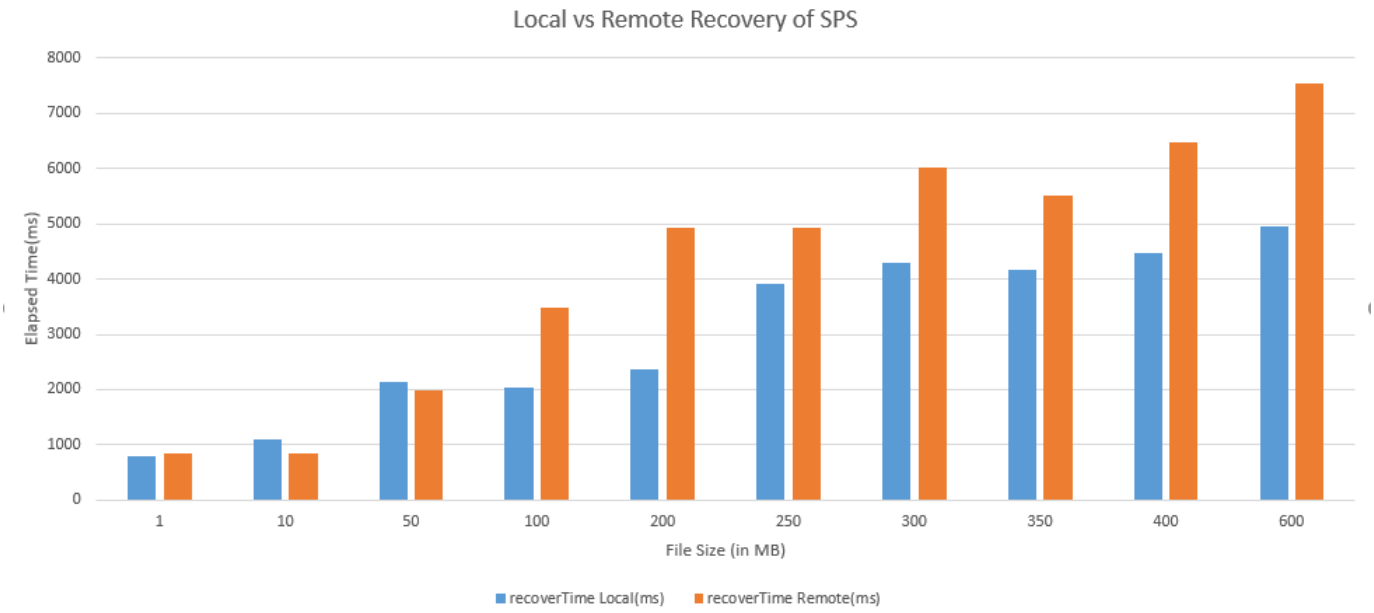

Figure 7: Result representation of elapsed time during local and remote state recovery

File size(in MBs)	recoveryTime Local(in ms)	recoveryTime Remote(in ms)	Speedup
1	801	843	1,05
10	1095	853	-
50	2143	1980	-
100	2027	3490	1,72
200	2359	4917	2,08
250	3909	4936	1,26
300	4297	6018	1,40
350	4166	5509	1,32
400	4465	6489	1,45
600	4967	7544	1,52

Table I: Results of elapsed time during local and remote state recovery

C. Discussion

The results shown in the last section are mostly expected. In the general case, when there is data locality it is expected to get faster execution time since the processing and the actual data co-locate on the same machine and there are no extra network IOs required.

In the first three cases, we observe little to no speedup in-between local and remote cases. This is explained for two reasons. First of all, as mentioned earlier Flink maintains a minimal checkpoint state for recovery, so the final file size for recovery can be much less given its processed state. The second is the fast network bandwidth in-between the VMs on Okeanos, which makes transferring data very fast. Those two reasons combined can make the difference between remote and local execution seem negligible and are considered noise to our results.

In the rest of the cases, that state for recovery is larger and we can observe the expected speedup in the local cases. The speedup results do not improve linearly or steadily with file size and this is due to the randomness of the generated files and the fact that every experiment provides different files as input. The state size is not always analogous to the input size and it varies given the processing it requires. For that reason, in order to produce fair results, we made sure that the input files for the local and remote cases were always the same for every given test. By using the recovery control mechanism described in the section V, it was possible to control the execution and always start on a specific machine with the same files and recover to a different machine.

File size(in MBs)	Number of Records Sent
1	185,032
10	1,248,834
50	6,971,555
100	12,273,865
200	21,734,179
250	27,383,167
300	32,574,495
350	37,768,358
400	43,033,466
600	66,452,131

Table II: Mapping between file size and Number of Records

While the results prove the point that locality improves the recovery time as a rule of thumb, the comparison is expected to be much clearer when examining much larger state. Therefore, the specific setup was not suitable to get the best results possible. The nature of incremental checkpointing and the requirement for a fair comparison between the distinct cases would demand a lot more complex mechanisms to allow different deployments, which is out of the scope of this course. Nevertheless, even this simple scenario showcases the performance gain when taking advantage of data locality during recovery.

VII. REFERENCES

[1] B. Monte, S. Zeuch, T. Rabl, and V. Markl, Rhino: Efficient management of very large distributed state for stream processing engines (2020) pp. 2471–2486.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. **38**, 28 (2015).

[3] Flink, *Concepts* (2021), `https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/flink-architecture/`.

[4] Flink, *Internals* (2021), `https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/general_arch.html`.

[5] Flink, *Stateful stream processing* (2021), `https://ci.apache.org/projects/flink/flink-docs-master/docs/concepts/stateful-stream-processing/`.

[6] TowardsDataScience, *"Here is how Flink stores your state"* (2021), `https://towardsdatascience.com/heres-how-flink-stores-your-state-7b37fbb60e1a`.

[7] Flink, *State backends* (2021), `https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/state_backends/`.

[8] Flink, *Incremental checkpointing* (2021), `https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html`.

[9] *Cyclades* (2021), `https://okeanos.grnet.gr/services/cyclades/`.

[10] Pithos, *Pithos* (2021), `https://okeanos.grnet.gr/services/pithos/`.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, The hadoop distributed file system, in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10 (IEEE Computer Society, USA, 2010) p. 1–10.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, The google file system, in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03 (Association for Computing Machinery, New York, NY, USA, 2003) p. 29–43.

[13] Okeanos, *Okeanos* (2021), `https://okeanos.grnet.gr/about/what/`.

[14] Flink, *Architecture* (2021), `https://flink.apache.org/flink-architecture.html`.

[15] S. Kalogerakis, *Migrating state between jobs in Apache Spark*, Diploma work, School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece (2020).

[16] K. Tucker, P. an dTufte, V. Papadimos, and D. Maier, NEXMark – A Benchmark for Queries over Data Streams (June 2010).

[17] Flink, *Checkpoints-State Backend* (2021), `https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/checkpoints/`.