Building Python-Based Topologies for Massive Processing of Social Media Data in Real Time

Rodrigo Martínez-Castaño
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
rodrigo.martinez@usc.es

Juan C. Pichel
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
juancarlos.pichel@usc.es

David E. Losada
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
david.losada@usc.es

ABSTRACT

In this paper we propose a streaming approach for real-time processing of huge amounts of data. Catenae is a library for easy building and execution of Python topologies (e.g., web crawler, classifier). Topologies are designed for their deployment inside Docker containers and, thus, horizontal scaling, granular resource assignment and isolation can be achieved easily. Furthermore, micromodules can have its own dependencies (including the Python version), allowing the user to limit resources such as CPU or memory by instance. We describe an implementation of a use case composed of two topologies: (1) a crawler for tracking users in social media and (2) an early risk detector of depression. We also explain how Catenae topologies can be connected to non-Python systems.

CCS CONCEPTS

• **Information systems** → *Data extraction and integration*; *Content analysis and feature selection*; • **Computer systems organization** → *Cloud computing*; *Real-time system architecture*;

KEYWORDS

Social Media, Text Mining, Depression, Stream Processing, Real-Time Processing, Docker, Python

1 INTRODUCTION

In recent years, numerous technologies have been developed for massive data processing. Many of them are open source and available for free. These platforms focus on horizontal scalability, so they can manage a larger amount of tasks by adding a proportional number of (not necessarily more powerful) nodes to an existing cluster. In order to scale horizontally, it must be possible to partition the data so they can be processed independently by different processes (distributed by the nodes of a cluster), even if at some point partial results have to be merged. Due to the democratization of these technologies that are capable of scaling on commodity hardware, the number of applications that make use of these platforms has increased a lot.

There are two main types of processing technologies of this family: batch and stream processing. In batch processing, results are obtained together at the end of an execution. To perform the computation, a processing topology is defined. A topology is composed by several stages where data are filtered or transformed following a path. Each node of a cluster can execute many instances of the topology, which will receive a fraction of the total input data. Instances are isolated among them and, thus, unbalanced. Therefore, if the input data are not shuffled, bottlenecks may occur. In stream processing, the different stages of a topology are not dependant of a topology instance and they can be individually instantiated multiple times. These processing technologies are suitable for building real-time applications since individual results are reflected instantly once they are obtained in a final stage instance. In addition, the different stage instances are permanently listening for new input data, so the topology does not need to be relaunched as new data are collected. With regards to resource allocation, resources and number of instances can be set at stage level. Since not all the stages of a topology take the same average time to process an input, more resources can be assigned to heavier modules, balancing the execution time per stage.

In Information Retrieval, many tasks can take advantage of stream processing. Many applications must run in real time and have several easily identifiable stages. Data follows a unique or multiple paths through stages forming a topology (a module could serve data to and/or receive from multiple modules). A data extraction stage is always present and usually a final stage to store results. Some examples are Real-Time Filtering (e.g., filtering stage), Real-Time Summarization (e.g., filtering and summarization stages), Real-Time Clustering (e.g., topic extraction), Real-Time User Classification (e.g., user type classifier), Real-Time Trend Detection (e.g., filtering and counting stages), etc.

In this paper, we propose Catenae[1], a new library that facilitates the process of building real-time applications at scale (stream processing). Data are processed through topologies in the shape of directed graphs (see Figure 1). Graph nodes represent points of data transformation and/or filtering and edges symbolize data flowing between nodes. Nodes can be connected to multiple nodes both to send and receive data. Cycles may exist since data can be redirected to previous stages (e.g., if a temporal condition is not met in a certain moment).

The paper is structured as follows: Section 2 describes our Python library to build processing topologies. Section 3 describes a use case where the library is used to build an early risk detection of depression system in real time. Section 4 explains how non-Python systems can be connected to a Catenae topology. In Section 5, related technologies are compared with our library. Finally, Section 6 contains the main conclusions of this study and future work.

2 PYTHON TOPOLOGIES WITH CATENAE

In Catenae, the communication between nodes is managed in the form of message queues by Apache Kafka [4]. Each node (stage) can be instantiated multiple times in such a way that if one type

[1] Publicly available at: https://github.com/catenae

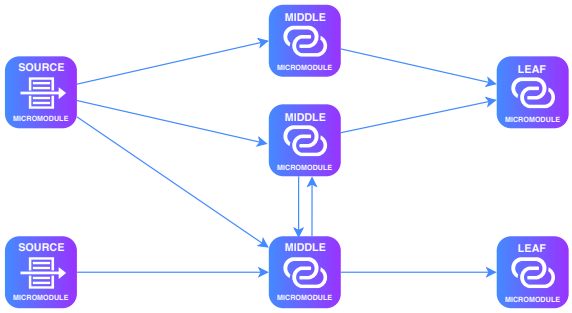

Figure 1. Catenae topology diagram example.

of node becomes a bottleneck, it is only necessary to replicate it. In addition, unlike popular batch processing frameworks where resources are assigned to the whole topology, Catenae assigns the hardware resources individually to each node instance.

In our library, graph nodes are called micromodules, which are tiny, loosely coupled Python scripts packaged inside Docker containers. These modules can scale up and down automatically launching or destroying instances respectively.

There are three types of Catenae micromodules:

- Source links. These modules do not receive data from Kafka but collect them by custom ways such as external APIs connections, custom crawlers or database reads.
- Middle links. All the modules that transform the input data and/or filter them. They consume and emit data within the topology (e.g., filters, classifiers). In Figure 2, a simple filter is implemented with Catenae.
- Leaf links. These nodes consume data from the topology but do not emit back to it. They store the results in a custom way such as database writes or output files (e.g., store alerts based on the output of a classifier).

Running the micromodules inside Docker containers allows the user to scale up a topology by launching more containers. This approach also avoids the Python GIL problem[2] (multiple threads cannot execute Python bytecodes at once) since each instance (container) will have its own CPython interpreter. Furthermore, since modules are isolated, they can have their own dependencies, avoiding execution problems caused by dependency lacks or conflicts in the cluster nodes. Even the Python version used in each module could differ while the transmitted data between nodes is compatible among them.

At the time of writing these lines, parameters such as input and output modules or the Kafka bootstrap address and port have to be indicated individually for executing each module (See Figure 3). Although it is not supported yet, this process will be managed automatically in future versions following a topology definition file (`topology.yaml`) as described in Figure 4. Also, the required directory structure for future automated building and deployment can be observed in Figure 5. For each module of the topology (i.e., `token_filter`), a directory with its name has to be created containing the following items:

[2]https://wiki.python.org/moin/GlobalInterpreterLock

```python
from catenae import Link, Electron


class TokenFilter(Link):

    def setup(self):
        self.allowed_tokens = \
            self.load_object('allowed_tokens')

    def transform(self, electron):
        tokens = electron.value.split()
        if self.allowed_tokens.intersection(tokens):
            return electron

if __name__ == "__main__":
    TokenFilter().start()
```

Figure 2. Micromodule implementation of a token filter (Middle link).

```
# Host
python token_filter.py \
-i <PREVIOUS_MODULE> \
-o <NEXT_MODULE> \
-b <KAFKA_BOOTSTRAP_ADDRESS>:<KAFKA_BOOTSTRAP_PORT>

# Docker container
docker run -d --net=host <DOCKER_IMAGE> \
-i <PREVIOUS_MODULE> \
-o <NEXT_MODULE> \
-b <KAFKA_BOOTSTRAP_ADDRESS>:<KAFKA_BOOTSTRAP_PORT>
```

Figure 3. Example execution of a topology module.

```
modules:
  <PREVIOUS_MODULE>:
    output: token_filter
    instances: 1
  token_filter:
    input: <PREVIOUS_MODULE>
    output: <NEXT_MODULE>
    instances: 4
  <NEXT_MODULE>:
    input: token_filter
    instances: 1
conf:
  kafka:
    address: <KAFKA_BOOTSTRAP_ADDRESS>
    port: <KAFKA_BOOTSTRAP_PORT>
```

Figure 4. Topology definition file example.

```
|-- topology.yaml
|-- <PREVIOUS_MODULE>
|   |-- <PREVIOUS_MODULE>.py
|   |-- requirements.txt
|
|-- token_filter
|   |-- token_filter.py
|   |-- requirements.txt
|
|-- <NEXT_MODULE>
    |-- <NEXT_MODULE>.py
    |-- requirements.txt
```

Figure 5. Topology directory structure example.

```
FROM catenae/link
COPY token_filter.py /usr/local/bin
ENTRYPOINT ["token_filter.py"]
```

Figure 6. Example of dockerfile script for creating a Docker image of a module.

- The main Python script with the same name as the module (i.e. "token_filter.py").
- A file called "requirements.txt" which will contain all the required packages by the module scripts available at the Python Package Index[3].
- Extra custom Python files that the main module will import.

In order to create a Catenae Docker image with our module, the easiest way is to extended our base image as in Figure 6, which contains pre-installed the latest stable release of Python 3, the Kafka client and our library.

Catenae supports multiple inputs for the same node of a topology. There are two implemented modes to manage this situation:

- Parity. With this mode, the modules receive data indistinctly from their inputs.
- Exponential. Modules consume data from their inputs in time windows, assigning a fraction of the window that grows exponentially with the input priority.

The modules of a topology can emit any Python object, which will be serialized and compressed automatically. On the destiny module, the object is also deserialized automatically. Our library uses Pickle[4] to serialize Python objects, which guarantees backwards compatibility among Python interpreters.

The modules are deployed inside containers and thus, resources like CPU or memory can be restricted for each instance of a module.

Finally, Catenae allows the user to load external resources during the initialization of the module: a Python dumped object, for instance. Aerospike [2] is a key-value distributed store that is supported by our library and can be used to store and load data in a fast way during the initialization phase of the topology. It uses RAM memory or SSD disks to store data and supports disk persistence when using memory.

3 USE CASE: EARLY RISK DETECTION OF DEPRESSION

We have developed a system for real-time detection of signs of depression with Catenae [13]. The system uses the social network Reddit as data source. Following the lessons learned in [10], we implemented a dynamic strategy that works with a *depression classifier* (built from the training split detailed in [10]) and incrementally analyses the stream of texts written by each user.

Our system is oriented to early risk detection [11] and, thus, it is more reasonable to fire the alerts as soon as there is evidence of a potential risk (rather than accumulating evidences and making batch processing). Scaling up modules that constitute a bottleneck is easy with this architecture (stream processing). For instance, if a preprocessing module is slower than the predictor, incrementing the number of the preprocessing stage instances would dissolve the bottleneck. In contrast, when launching topologies with batch processing frameworks, resources have to be assigned to the full topology, although only one stage is active at a time. This is a problem in batch processing because any of the stages could not require as many resources as the heavier stage to perform properly.

3.1 The Reddit Crawler

Reddit is a website where users submit content such as text, images or links (submissions) and other users can comment and vote for or against. The platform is subdivided into communities (subreddits) focused on specific topics. It is currently ranked in Alexa [14] as the sixth website with more traffic in the world. The number of average monthly active users is higher than 330 millions and there are more than 138,000 active communities [1].

The first goal of our platform was to maximize the number of tracked users (collecting their new posts periodically). To meet this aim, we have built a web crawler for Reddit following the rules expressed in the `robots.txt` file. The crawler uses the Catenae library, as it is composed of multiple horizontal-scalable micromodules in a pipeline (see Figure 7):

- **Submission and comment crawlers.** They retrieve all new submissions and comments and extract author nicknames.
- **New user filter.** Nicknames are filtered to avoid repeated users. Aerospike is used to deal with this task efficiently as it is a memory-based store. Those users who pass the filter will be sent to a queue of new users.
- **User content crawlers.** In this stage, all texts (submissions/comments) written on Reddit by the users that passed the filter are extracted. Collecting all submissions of a user requires n calls, where n is the number of posts to retrieve (with a maximum of 100). On the other hand, retrieving the newest comments made by a user requires a single call. On every iteration, the system only retrieves the new texts available (it stores the identifiers of the last submission and comment for each user). The user content crawlers obtain user identifiers from different queues, ordered by priority. Based on the confidence score (as produced by the classifier) and the activity since the previous iteration, users can be allocated in different priority queues. A single output queue receives the extracted texts.
- **Post storer.** It is in charge of storing texts in a document-oriented database. In addition, these texts will also feed the early prediction pipeline.

There is a clear bottleneck in this crawling topology. Users can be extracted fast since in every iteration multiple users can be reached but the User Content Crawler has to scrape not only the comments and submissions main pages of every user, but a page per submission. This is the only module that requires a high number of instances.

3.2 The Early Risk Detection Pipeline

The early prediction pipeline is a Logistic Regression classifier with L1 regularization, implemented in Python with *scikit-learn*. The classifier is built with a training set of 486 users (83 positive, 403 negative) [10]. Users are represented with a single document,

[3] https://pypi.python.org/
[4] https://docs.python.org/3/library/pickle.html

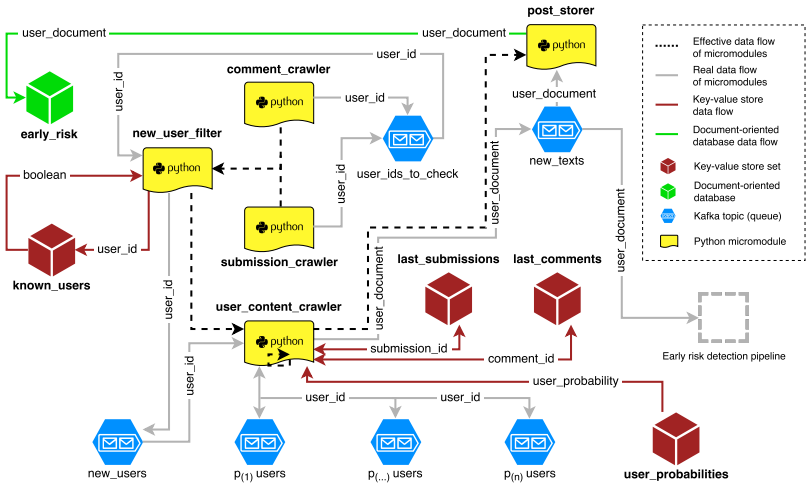

Figure 7. Architecture diagram of the Reddit crawler.

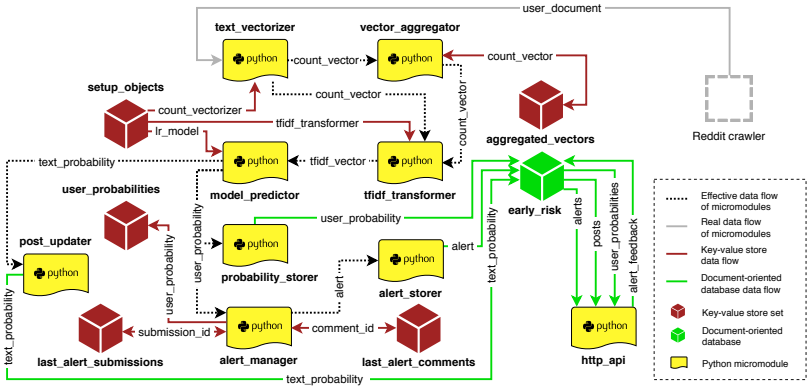

Figure 8. Architecture diagram of the early risk detection pipeline. Main queues omitted for simplicity.

consisting of the concatenation of all their writings. The prediction process has four micromodules (see Figure 8):

- **Text Vectorizer**. It transforms an input text into a vector of token counts.
- **Aggregator**. It accumulates a vector of token counts that represents all submissions and comments of each user. The aggregator merges the current vector of counts with the vector obtained from any new submission or comment.
- **Tf–idf Transformer**. It transforms the aggregated vector of counts to a normalized tf-idf representation.
- **Model Predictor**. It produces the probability of risk of depression for users given their tf–idf representation. In addition, it produces an individual probability for each document which will be stored by the Post Updater micromodule.

The Probability Storer micromodule is fed with user probabilities by the Model Predictor and stores user probabilities. The Alert Manager receives the same input data but stores alerts if the probability is over a certain threshold.

The aggregated vectors and the Python objects (count vectorizer, tf–idf transformer, and the classification model) are stored in the Aerospike store. In this way, the vector updates and the micromodule initializations are fast.

The web interface is composed of three main views that are connected to our HTTP API. In "Alerts" (Figure 9), the fired alerts by the system can be inspected in real time, sorting them by recency or priority (higher confidence first). Alerts correspond to a given user and all the related submissions and comments can be read (ordered by decreasing probability) so the reviewer can tag the alert as a true positive (risk) or false positive (risk free). For each alert, it is also represented the evolution of the user in a line chart (See Figure 11). Each point corresponds to the aggregated probability once a new text (submission or comment) is extracted by the crawler for that user. This view also contains three tabs. The default tab is intended for untagged alerts. The other two tabs contain the tagged alerts: true positives and false positives. The "Datasets" view contains three download options: true positives, false positives and *everything* (See Figure 10). For risk and risk free, the dataset is generated with the users which where tagged at some point with a given tag. In this way, the platform facilitates the creation of labelled collections and benchmarks. Observe also that any classifier can

Figure 9. Real-time alert view of web interface (early risk detection of depression).

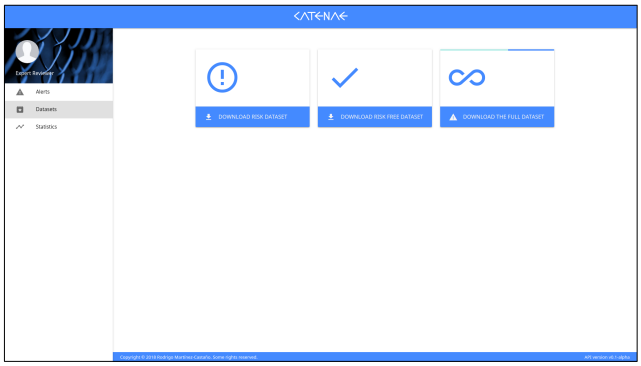

Figure 10. Dataset download view of web interface (early risk detection of depression).

be plugged into the system and, therefore, Catenae supports real-time analysis of users in a number of domains or tasks. Finally, the "Statistics" view shows some information about the running system in real time such as total number of extracted submissions, comments, users and processing speed (See Figure 12).

3.3 Performance Evaluation

In order to test our Catenae use case, we have deployed it in AWS EC2 virtual machines running Amazon Linux 2. AWS gives their users the possibility of running a wide variety of virtual machines in their EC2 infrastructure. In our case, we have used `c5.4xlarge` instances with the following characteristics:

- CPU: Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz with 16 assigned virtual cores.
- Memory: 32 GiB.
- Storage: 100 GiB SSD General Purpose volumes (EBS).

According to the specifications provided by AWS, `c5.4xlarge` instances have a dedicated bandwidth up to 2,250Mbps for EBS storage. The general network performance is around 5Gbps.

In these experiments, we consider that the processing is performed in real time when all the extracted texts are classified within seconds, without a growing queue of unprocessed items.

For both topologies, we have defined experimentally the proportion of micromodule instances of each kind in order to remove bottlenecks. In the case of the crawler, the User Content Crawler is the only module that we have considered necessary to scale. For the classifier, the critical micromodules are: Text Vectorizer, Tf-idf Transformer, Vector Aggregator and Model Predictor. We assigned the proportion 2-4-4-1 respectively in order to balance the topology.

The parallelism refers to the number of times that the selected micromodules with the previous proportions are replicated. For instance, a parallelism 2 for the classifier topology would mean 4x Text Vectorizer, 8x Tf-idf Transformer, 8x Vector Aggregator and 2x Model Predictor.

In Figure 13 it can be observed the extracted texts per second for two crawling tests executed in different days. Only one virtual machine was used and the crawler parallelism was configured from 1 to 640. Each configuration was tested for 5 minutes. Both tests behave slightly different due to external factors (Reddit response time varies). Due to this fact, the optimal number of instances of the crawler varies: for higher response times, higher number of instances are needed. It can be observed that for parallelisms higher than 320 units the performance begins to degrade on both tests. However, in Test B, the number of extracted texts recovers due to a reduction in the response time.

In Figure 14, two tests were performed in different days with both the classifier and the crawler. The classifier parallelism was determined depending on fixed crawler parallelisms from 1 to 50 so real-time processing of the retrieved texts was achieved. With higher crawler parallelisms, real-time processing was not possible. Again, each configuration was tested for 5 minutes. The experiment was replicated in Test B with the same parallelism configurations, obtaining a better performance of the crawler while maintaining the processing in real time.

Finally, a 3-node cluster was set up with Apache Kafka. During a three-hour experiment (Figure 15) it can be observed huge fluctuations on the crawling capacity between 500 and 1,000 extracted texts per second. Due to these fluctuations, the queue of unclassified texts grows and decreases accordingly. At least with this scenario, a fixed parallelism for both topologies or even a fixed number of nodes is not an optimal configuration. Topologies should be able to scale up and down automatically for optimal performance and to avoid the waste of resources. Due to this fact, it is necessary to develop a topology controller that automatically balances the parallelism configurations of the running topologies according to their load.

4 COMPATIBILITY WITH OTHER SYSTEMS

Catenae facilitates the creation and deployment of Python topologies. Our library uses Kafka behind the scenes, and, thus, topologies can be connected with other systems using the Kafka client. This connection can be done both for emitting data to the topology (producer) and consuming the output data from it (consumer). It is only necessary to integrate a Kafka producer or consumer in the external system.

Catenae can be connected to other frameworks such as Polypus [12], which is a modular framework that provides the following

Figure 11. Risk alert view of the web interface (early risk detection of depression).

Figure 12. System statistics view of the web interface (early risk detection of depression).

functionalities: (1) massive text extraction from Twitter, (2) distributed non-relational storage optimized for time range queries, (3) memory-based intermodule buffering, (4) real-time sentiment classification, (5) near real-time keyword sentiment aggregation in time series, (6) a HTTP API to interact with the Polypus cluster and (7) a web interface to analyse results visually. Polypus' main modules are not coded in Python and Polypus does not use Catenae. However, it could be useful to reuse the Twitter crawler as data source for a Catenae topology. The original crawler was connected to two different databases to store the content of the extracted tweets. One of them, Aerospike, was also used to store tweet identifiers in memory. Thereby, checking the existence in the system of a new extracted tweet could be fast while the state is preserved among nodes and between executions. The original outputs for extracted contents were substituted with a Kafka producer (see Figure 16). The Kafka Java API was used to adapt the crawler so it can emit tweets to the input topic of a Catenae topology. Since we are connecting a module written in Java and a Python topology, data should be emitted with basic types. As a matter of fact, we have already taken the first step to connect Catenae topologies with the Polypus's Twitter crawler with a simple Python sentiment analyser.

5 RELATED WORK

MapReduce [7] is a programming model introduced by Google for processing and generating large data sets on a huge number of computing nodes. A MapReduce program execution is divided into two main phases: *map* and *reduce*. The input and output of a MapReduce computation is a list of key-value pairs. Users only need to focus on implementing map and reduce functions. In the map phase, map workers take as input a list of key-value pairs and generate a set of intermediate output key-value pairs, which are stored in the intermediate storage (i.e., files or in-memory buffers). The reduce function processes each intermediate key and its associated list of

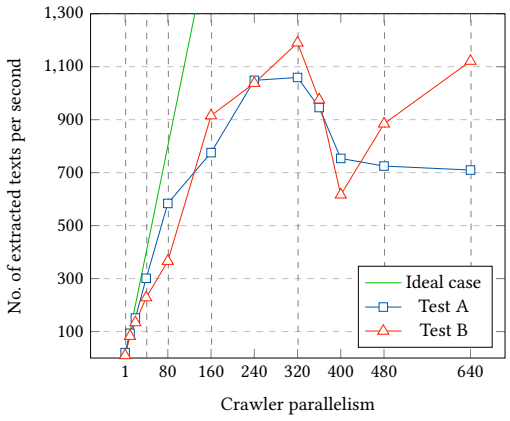

Figure 13. Crawler performance with different parallelism configurations in 1 node.

values to produce a final dataset of key-value pairs. In this way, map tasks achieve data parallelism, while reduce tasks perform parallel reduction. Currently, several processing frameworks support this programming model.

With Apache Spark [16], arbitrary workflows with several processing stages can be defined, so it is a more flexible model than Hadoop MapReduce [3]. It supports several functional programming operations beyond *map* and *reduce*. On the one hand, Spark, written in Scala (JVM language) supports Python, however, non-JVM languages are less efficient since they are not natively executed. Building Catenae modules requires fewer modifications on the existing code since custom data structures are not needed and the transition is more natural if the modules were already distributed in multiple scripts forming a pipeline. In addition, Python dependency management is not trivial in Spark since libraries must be available in all the nodes of the cluster. With Catenae, by contrast, each module has its own encapsulated dependencies in a Docker image (including the Python interpreter). The performance and context of a Python module with Catenae will be the same as when executed

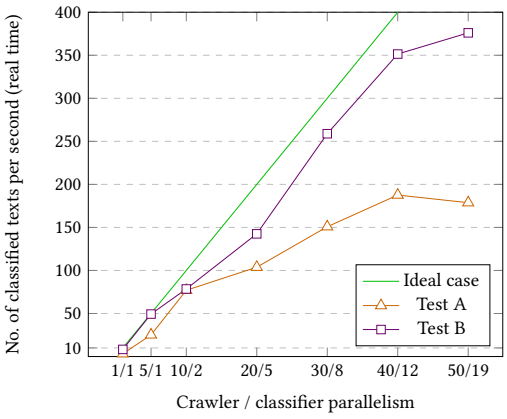

Figure 14. Parallelism configurations to achieve real-time processing in 1 node.

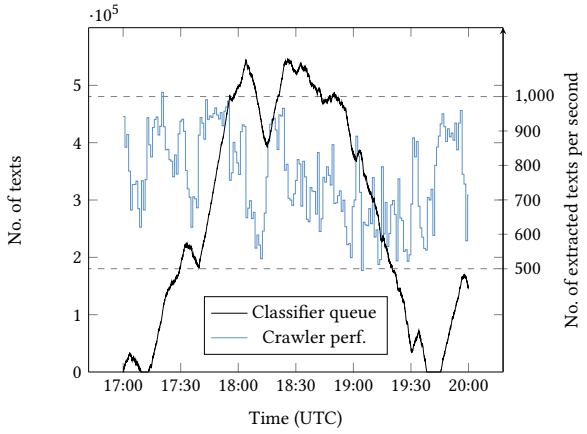

Figure 15. Three-hour experiment on April, 30th 2018 with 3 nodes and 170/0 - 0/50 - 0/50 parallelism (crawler/classifier).

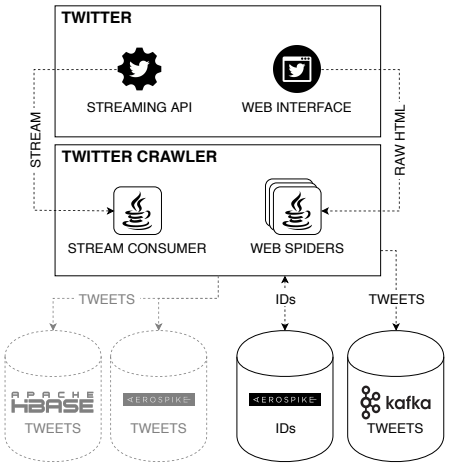

Figure 16. Polypus' Twitter crawler.

locally within a Docker container. A workflow in Spark must be a directed acyclic graph, so data cannot go back to early stages. With our library, as it is focused on real-time applications, cycles could happen (e.g., a module could check a time condition in order to let the data continuing its course).

All these processing technologies require a cluster manager to execute an application (e.g., Apache Hadoop YARN [15], Apache Mesos [9]), whereas Catenae only requires Docker, a *de facto standard* framework for containers since native clustering functionality is provided with Docker Swarm.

Apache Storm [5] is a framework with the aim of processing streaming data in real time. It requires the definition of *topologies*, which are computational graphs (workflows) where every node represents individual processing tasks. Edges correspond to the flowing data between nodes, which are the responsible of exchange data using *tuples*. Tuples are ordered lists of values, where each value has an assigned name. In particular, nodes exchange non delimited sequences of tuples called *streams*. Every node listens to one or more streams as input. In the Storm terminology, *spouts* are the sources of a stream within a topology, which usually read data from an external source. Finally, *bolts* are the consumers of the streams and they perform calculus and transformation tasks on the received data. Bolts can emit none, one or more tuples to the output streams. When a topology starts, it stays in execution waiting for new data to process. Storm clusters are composed by two types of nodes: master (Nimbus) and workers (supervisors). Storm uses Apache Thrift [6] and since it can be used in any language, topologies could be defined and submitted from any language. However, non-JVM languages interact with Storm through a JSON-based protocol over stdin/stdout[5] (not very efficient). Again, there is a problem with Python dependencies which will have to be installed in the cluster nodes and the management could be tedious since different topologies could need different versions of the same package. Despite there are libraries such as StreamParse[6] that automatize the installation of dependencies, the problem persists if different

[5]https://storm.apache.org/about/multi-language.html
[6]https://github.com/parsely/streamparse

modules in the same topology require incompatible dependencies among them.

Kafka is a distributed message broker for high-throughput, low-latency handling of real-time data feeds. Kafka is used as the mechanism to distribute messages between the modules of a Catenae topology. Using Kafka directly requires to manage Kafka producers and consumers for each node of a topology. There exists an official Kafka library for building these kind of real-time topologies: Kafka Streams. However, this library is only available for Java and Scala. During the development of Catenae, a unofficial Kafka Streams library was developed for Python. Our library makes much easier to develop Python topologies since Kafka Streams is a lower level library. Features such as inputs with configurable priority are already implemented and not available in Storm or Kafka. Resource assignment can be easily achieved encapsulating the modules inside Docker containers. Catenae abstracts the developer from Kafka offering a context of minimum intrusion with the existing code.

Catenae topologies can be deployed with Docker [8] containers, which allow us to obtain the benefits of virtualization (isolation, flexibility, portability, agility, etc.) without penalizing the I/O performance considerably. Docker makes use of resource isolation characteristics of the Linux kernel, so independent containers can be executed on the same host. Containers supply a virtual environment with their own space of processes and networks. The containers are built with stacked layers. When a container is in execution, a new writeable layer is created over a set of read-only layers which define a Docker image. The Docker images are always built from a base image, ultimately the *Scratch* image. These images can be easily distributed via the official Docker registry[7], with our own registry or with *tarballs*. Images can be built with a custom scripting language (*dockerfiles*) or by saving the state of a running container.

6 CONCLUSIONS AND FUTURE WORK

Catenae is a Python library which allows the user to build scalable real-time streaming applications easily. Data can flow through the topology in any direction and even backwards. Dependency problems such as library versions or lack on the target cluster are avoided through the use of Docker containers. Furthermore, the Python's GIL problem is bypassed since scalability is achieved by launching more containers for the same module (different processes). The isolation property also allows the user to run scripts written for different Python versions (e.g., Python 2.7, Python 3.6). Basic input data prioritization politics are implemented and they have not to be manually coded. Finally, serialization and deserialization of Python objects is performed automatically. In order to serialize Python objects, our library uses Pickle, which guarantees backwards compatibility across Python releases. There are base containers for building Catenae modules and deploying Kafka.

As future work, new queue handlers will be implemented among the possibility of use custom ones. With Docker as single dependency, deployment of topologies will be handled automatically. Moreover, it will exist the possibility to deploy a disposable Kafka cluster with the topology. The topology manager, in addition to deploying topologies, will let the user to manage a running topology (scaling it up and down). It is usual for real-time systems to have serious changes on their load during its execution, so we have also considered adding support for automated scaling.

[7] https://hub.docker.com/

ACKNOWLEDGMENTS

This work has been supported by MINECO (TIN2014-54565-JIN, TIN2015-64282-R), Xunta de Galicia (ED431G/08) and European Regional Development Fund.

REFERENCES

[1] About Reddit. 2018. https://www.redditinc.com/. [Online; accessed April, 2018].
[2] Aerospike. 2018. https://www.aerospike.com/. [Online; accessed April, 2018].
[3] Apache Hadoop. 2018. https://hadoop.apache.org/. [Online; accessed April, 2018].
[4] Apache Kafka. 2018. https://kafka.apache.org/. [Online; accessed April, 2018].
[5] Apache Storm. 2018. https://storm.apache.org/. [Online; accessed April, 2018].
[6] Apache Thrift. 2018. https://thrift.apache.org/. [Online; accessed April, 2018].
[7] J. Dean and S. Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation*. 10–10.
[8] Docker. 2018. http://www.docker.com/. [Online; accessed April, 2018].
[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 295–308.
[10] D. Losada and F. Crestani. 2016. A Test Collection for Research on Depression and Language Use. In *Proc. of CLEF*. 28–39.
[11] D. Losada, F. Crestani, and J. Parapar. 2017. eRISK 2017: CLEF Lab on Early Risk Prediction on the Internet: Experimental Foundations. In *Proc. of CLEF*. 346–360.
[12] R. Martínez-Castaño, J. C. Pichel, and P. Gamallo. 2018. Polypus: a Big Data Self-Deployable Architecture for Microblogging Text Extraction and Real-Time Sentiment Analysis. *CoRR* abs/1801.03710 (2018). arXiv:1801.03710
[13] R. Martínez-Castaño, J. C. Pichel, D. E. Losada, and F. Crestani. 2018. A Micromodule Approach for Building Real-Time Systems with Python-Based Models: Application to Early Risk Detection of Depression on Social Media. In *Advances in Information Retrieval*. Springer International Publishing, 801–805.
[14] Reddit on Alexa. 2018. https://www.alexa.com/siteinfo/reddit.com/. [Online; accessed April, 2018].
[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 4th Annual Symposium on Cloud Computing (SOCC)*. 5:1–5:16.
[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud)*. 10–10.