Stream Processing in Community Network Clouds

Ken Danniswara*, Hooman Peiro Sajjad*, Ahmad Al-Shishtawy†, and Vladimir Vlassov*
*KTH Royal Institute of Technokogy
Stockholm, Sweden
{kend,shps,vladv}@kth.se
†Swedish Institute of Computer Science (SICS)
Stockholm, Sweden
ahmad@sics.se

***Abstract*—Community Network Cloud is an emerging distributed cloud infrastructure that is built on top of a community network. The infrastructure consists of a number of geographically distributed compute and storage resources, contributed by community members, that are linked together through the community network. Stream processing is an important enabling technology that, if provided in a Community Network Cloud, would enable a new class of applications, such as social analysis, anomaly detection, and smart home power management. However, modern stream processing engines are designed to be used inside a data center, where servers communicate over a fast and reliable network. In this work, we evaluate the Apache Storm stream processing framework in an emulated Community Network Cloud in order to identify the challenges and bottlenecks that exist in the current implementation. The community network emulation was performed using data collected from the Guifi.net community network, Spain. Our evaluation results show that, with proper configuration of the heartbeats, it is possible to run Apache Storm in a Community Network Cloud. The performance is sensitive to the placement of the Storm components in the network. The deployment of management components on well-connected nodes improves the Storm topology scheduling time, fault tolerance, and recovery time. Our evaluation also indicates that the Storm scheduler and the stream groupings need to be aware of the network topology and location of stream sources in order to optimally place Storm spouts and bolts to improve performance.**

Keywords*—*Stream processing, Community network cloud, Community network, Apache Storm.

I. Introduction

Community Network Cloud is to maintain cloud services on top of community networks. The idea is that the cloud resources are provided by the community members [1]. The cloud services are hosted over multiple geographically distributed servers that forms the cloud's infrastructure. There is no control over the underlying network topology nor on its size which grows naturally as the community grows. Community Network Cloud enables community users to deploy and access a wider range of services, such as Video-on-Demand, Cloud storage and Internet of Things. Introducing new Cloud services and more user involvement will need processing more data, having more automation and fast decision makings. Stream processing is one of the paradigms that can enable us to satisfy these requirements. It enables us to process streams of data and extract information in real time [2] for different applications, e.g., churn prediction, social analysis, anomaly detection and sensor data interpretation.

Apache Storm is an open source distributed stream processing framework that has an active community [3]. Storm is designed to be used inside a data center, where all the data being accessed and processed through intra data center communications. Servers inside a data center are connected with a tree-like topology and top of the rack switches. However, topology of a community network can be variant and the servers of a community network are connected through heterogeneous links. Therefore, the performance of a Storm cluster inside a Community Network Cloud depends on the topology of the host servers and the links quality. To our knowledge, there has been no prior work to evaluate a distributed stream processing framework in a Community Network. Therefore, in this work, we evaluate Apache Storm as a popular distributed stream processing framework in a community network and identify the challenges and bottlenecks on leveraging it in such a network.

Our contributions include,

- Emulation of a community network with the CORE network emulator using real network traces from Guifi.net.
- Evaluation of Apache Storm stream processing system on top of the emulated community network.
- Identification of the main performance bottlenecks in leveraging Apache Storm on a community network and providing guidelines to increase the performance of the Storm in such an environment.

The structure of this paper is as follows. In Section II, we give an overview of Apache Storm and its components. Section III describes community network Clouds and the community network data set that we use in this paper. In Section IV, we explain the emulated environment. We discuss our evaluation results in Section V. In Section VI, we give our guidelines for the deployment of Storm in a community network. Finally, we give some conclusions and future work in Section VII.

II. Apache Storm

In this section we explain the main notions about Apache Storm and its components. Apache Storm is an open source distributed real time computation system. It is designed to process queries on unbounded streams of data. Its main difference with batch processing is that queries in Storm will be

978-1-4673-8103-1/15 $31.00 © 2015 IEEE
DOI 10.1109/FiCloud.2015.95

processing forever unless they are killed by its user. Storm is designed to be scalable and fault tolerant.

A query in Storm is defined by creating a graph of computation called a *topology* [4]. Each node of the topology contains a processing logic, and connections between the nodes indicate the way data should be propagated between them. A stream in Storm is an unbounded sequence of tuples. A topology is made of two types of nodes for doing stream transformations, *spouts* and *bolts*. A spout is a source of streams, e.g., a spout may read tuples from a message queuing system or connect to Twitter API to emit a stream of tweets. A bolt receives any number of input streams, process them and may emit new streams either to another bolt or another application. Bolts can do any kind of processing, e.g., running functions, aggregating streams, doing streaming joins and talking to databases or filesystems.

Each node in a Storm topology consists of one or multiple *tasks*, which do the actual data processing. One or multiple tasks are executed in parallel through one or multiple threads inside a number of predefined *workers*. Each thread is called an *executor*. Each worker is a Java Virtual Machine process. Workers can execute only the executors of the same topology. The amount of parallelism for each node can be defined by the user and Storm will execute the node across the cluster using that number of threads. *Stream grouping* is an important notion in Storm. A stream grouping specifies how tuples should be sent to bolts. There are seven predefined stream grouping in Storm:

- **Shuffle:**Tuples are uniformly randomly distributed across the bolt's tasks.
- **Fields:**Tuples are partitioned by a specified field. Tuples with the same value for that field will always go to the same task.
- **Partial key:**Tuples are partitioned as in fields grouping, but load balancing between two bolts when the incoming data is skewed.
- **All:**Tuples are replicated across all the tasks.
- **Global:**All tuples will be sent to only one task.
- **Direct:**The producer decides which task of its consumers will receive this tuple.
- **Local or shuffle:**Colocated tasks in the same worker will do in-process shuffling. Otherwise, this acts as a normal shuffling group.

Storm has a master-worker architecture. There are three types of nodes in the cluster. It has a master node, which runs a daemon called *Nimbus*. Nimbus uploads the user defined topologies for execution and distributes the code across the clusters. It launches workers and monitors and reallocate workers as needed. *Supervisor*s are supervisor daemons running on each worker node that communicate with Nimbus and manage workers. *ZooKeeper* nodes coordinate the storm cluster. All the communications between Nimbus and Supervisors are through the ZooKeeper nodes.

A logical topology, at the execution time, will be compiled and transformed into a physical execution plan. On the physical level, each worker node has a predefined number of slots available to execute worker processes. Nimbus will use a scheduler to plan the execution of tasks in the cluster. The default scheduler of the Nimbus, using a round-robin strategy, evenly distributes the execution plan on the available nodes.

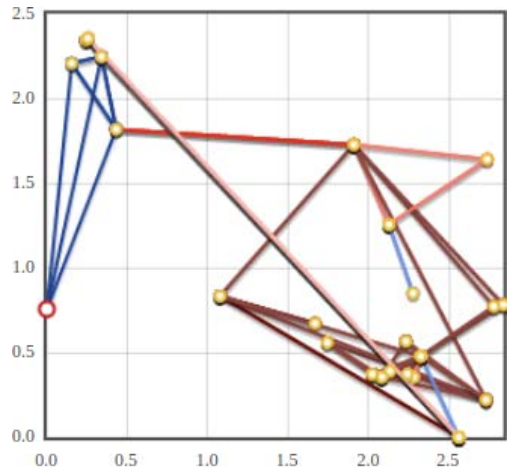

Fig. 1: A snapshot of the QMP Sants-UPC community network taken from [6]. Axes are in km.

III. Community Network Cloud

A Community Network Cloud provides cloud services on top of community networks. Cloud resources are provided by the community members [1] and due to the distributed nature of the community networks, cloud services are hosted over multiple geographically distributed servers. There is no control over the underlying network topology and it grows naturally as the community grows.

In this work, we evaluate our work on the data collected from a small part of Guifi.net [5]. We collected a data set of the QMP Sants-UPC (Figure 1) [6] network for doing our experiments. This network is a wireless multi-hop network. A monitoring system collects information of nodes and links on an hourly basis. We use the data collected in 24 hours. To make an estimation about the quality of links, we calculate the average bandwidth and latency of the links. We filter the links that their monitoring information is missing. This can be due to either a very bad quality link or they're nodes are off. We also rule out all the disconnected nodes. The final data set contains 52 nodes and 112 edges. In this work, we assume that all the nodes have enough resources to host at least one Storm component.

IV. CORE Network Emulator

CORE (Common Open Research Emulator) is a network emulation tool made by NCS group in United States Naval Research Laboratory (NRL) [7]. CORE is an open source application based on IMUNES [8] with more features of creating virtual wireless network, distributed emulation between physical machines, python scripting, remote API, and easy-to-use GUI.

In CORE, users can create virtual nodes with different roles, e.g., PCs, Servers, Routers, Hubs, and Switches. Nodes

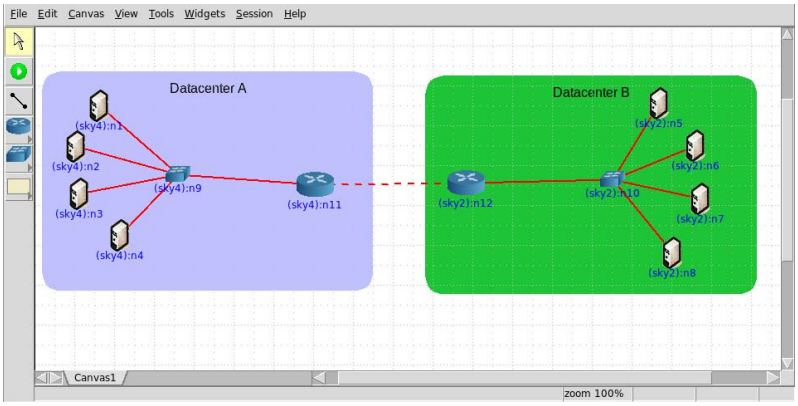

Fig. 2: CORE network emulator

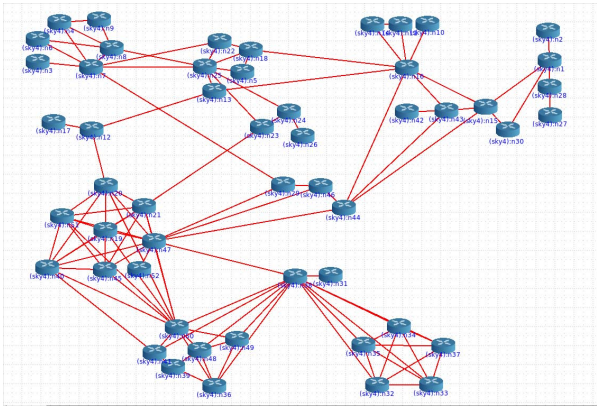

Fig. 3: Representation of Guifi.net community network on CORE

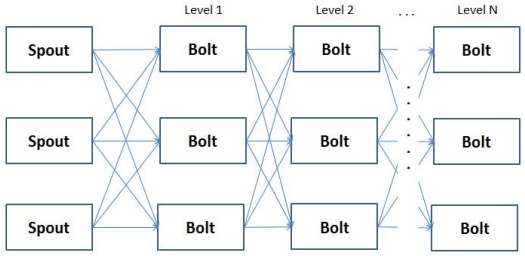

Fig. 4: Storm-perf-test topology

can be connected through the links, which can be configured to set maximum bandwidth, delay, and jitter for each way or both. CORE's main application runs as a daemon that can be connected from any core-GUI located in the same or a different machine. The daemon will create the virtual nodes and network links inside its own machine. This feature enables the management of multiple core-daemons located on multiple machines. This is useful to run the emulation with high number of nodes and links that requires load distribution among multiple machines. Fig. 2 shows a sample emulation of two connected data centers.In this figure, Datacenter A runs on server named "sky2" and Datacenter B runs on "sky4". The dashed link, that connects the two data centers, is a GRE tunnel between the two machines.

Emulation of Guifi.net community network, that is explained in section III, can be seen in Fig. 3. We use physical connection between the virtual nodes rather than wireless network because Guifi.net use directed radio wave communication between the devices [5]. Every link is assigned a maximum bandwidth and latency according to the data set.

V. Evaluation

We evaluate Apache Storm in a community network in two series of experiments. First, we evaluate how the different placement of management components, namely Nimbus and Zookeeper, on nodes of different types (with various degrees of connectivity) effects the Storm topology scheduling time. In the second series, we evaluate the run-time behaviour of Storm in terms of network traffic for different tasks assignments to different types of nodes and different types of stream grouping.

We create 52 network nodes emulated in one physical machine (HP ProLiant DL380 server with 2 x Intel Xeon X5660, 24 threads in total, 44GB of RAM, and 2TB of storage, RHEL 6 OS). In evaluation experiments, the Storm topology is based on the storm-perf-test benchmark by Yahoo [9]. The design of the storm-perf-test topology is shown in Fig. 4. The topology starts by a Spout (level 0) that creates dummy tuples with the size of 500 bytes and the rate of 20 tuples per second and sends the messages to the processing tasks (bolts) at level 1. Each tuple is passed to the higher level bolts until it reaches the last level specified by the user. There is a 33% chance that a tuple is discarded on every bolt. The discard probability is defined based on the idea that aggregation or filtering processes are common in stream processing, where the number of input tuples on each task is higher than the number of output tuples. Every run in this evaluation experiment uses the same topology.

We categorize community network nodes based on their degree of connectivity. A node with 5 or more direct connections to other nodes is called a SuperNode; whereas a node with less than 5 connections is called an EdgeNode. According to our categorization, in the network considered in this study, 22 nodes are SuperNodes and 30 nodes are EdgeNodes. As

TABLE I: Storm.yaml configuration for the experiment

Parameters	Storm Default value	Modified value
Worker Heartbeat frequency (s)	1	10
Worker timeout (s)	30	80
Supervisor heartbeat frequency (s)	5	20
Supervisor timeout (s)	60	150
Nimbus task timeout (s)	30	80
Nimbus monitor frequency (s)	10	40
Zookeeper session timeout (ms)	20000	50000
Zookeeper connection timeout (ms)	15000	40000

one can observe in Fig. 3, a group of multiple SuperNodes can appear as a cluster, where nodes have high availability and good connectivity.

A. Placement of Management components

Storm supervisors (worker nodes) and nimbus (a master node) are tightly connected with Zookeeper that serves as coordinator. Zookeeper continuously receives heartbeats from worker processes, supervisors and nimbus. These heartbeats are used to detect failures in the system and are configurable. The default value for the parameters related to the heartbeats are shown in Table I. We have observed that applying default configuration, when running Storm on the QMP Sants-UPC community network, leads to false-positive detection of failures. In other words, some worker nodes are detected as failed nodes even though they are still healthy. Therefore, there is a need for reconfiguration of these parameters in order to adapt the Storm cluster to work on such heterogeneous network. Table I shows the configuration of heartbeat parameters used in our experiments. As it can be seen, we have increased the timeouts for around 2.5 times of their default values. The new configuration avoids the wrong failure detection of the nodes. We have also decreased the heartbeat frequencies in order to reduce the network traffic. By reducing the heartbeat rate and increasing timeouts, it is expected to prolong the recovery time of the system. Study on the recovery time of Storm cluster is beyond the scope of this paper.

When a topology is submitted to Nimbus, the scheduler inside Nimbus assigns the tasks to worker nodes. Distribution of the tasks among the worker nodes can be different depending on the scheduler considerations. The default scheduler of Storm is round-robin that distributes tasks evenly among the worker nodes to achieve load balancing. In [10], the authors propose a more sophisticated scheduler that takes into account the Storm topology, nodes capabilities and the network traffic when making scheduling decisions. Every scheduling process takes an amount of time, which we call *scheduling time*, to make scheduling decisions and to assign the tasks to worker nodes. In the case of latency-critical stream processing, the scheduling time becomes even more important as the tasks rescheduling is considered as the system downtime (overhead). We evaluate how the placement of the Nimbus and Zookeeper components on the community network nodes affects the scheduling time. The result is presented in Fig. 5.

We select two nodes out of the 52 nodes: one for Nimbus and one for Zookeeper, and 30 nodes to be worker nodes. In total, in each run there are 32 nodes hosting Storm components. We give permanent locations to the worker nodes, which are spread in the whole network, because we want to focus only on the placement of Nimbus and Zookeeper. We run

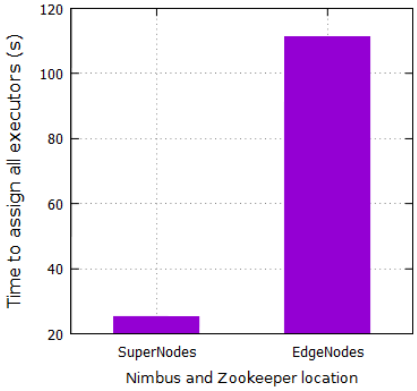

Fig. 5: Average time until all tasks assigned to workers and acknowledged by the Zookeeper

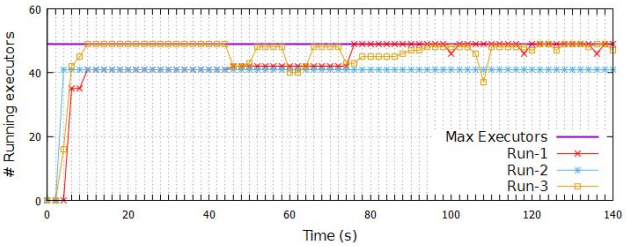

Fig. 6: Number of tasks running at run-time. Nimbus and Zookeeper located on EdgeNodes

the experiment three times (Run-1,2,3) for each category of nodes (SuperNodes and EdgeNodes). On each run, we place Nimbus and Zookeeper on different nodes but still close (at most two hops) to each other in order to ensure that the high network traffic at run-time does not affect communication between Nimbus and Zookeeper. Our evaluation shows that if the management components are deployed on the highly connected nodes (SuperNodes), the scheduling time significantly improves by around 4 times compare to scheduling time in the case of placing management components on the EdgeNodes. This is because better connectivity of the SuperNodes (higher bandwidth, shorter latency and larger number of links). If the Nimbus and Zookeeper are placed on the EdgeNodes, the heterogeneity of latency and bandwidth between nodes in the network may negatively affect the speed and time of the information distribution to the worker nodes. The effect may get worse for the nodes with farther distance from Nimbus.

B. Worker nodes placement

Fig. 6 and Fig. 7 show a detailed view on state of the tasks in each run for different placements of the management components, Nimbus and Zookeeper. By default, there exists one executor per task. The executors are considered "Running" after their process is created in the worker node and is registered to Zookeeper. "Max Executors" is the total number of executors that should be running. Executors running in the worker nodes with good connectivity to Zookeeper reach to the "Running" state much faster than those executors that their worker nodes have poor connectivity. As it can be seen in

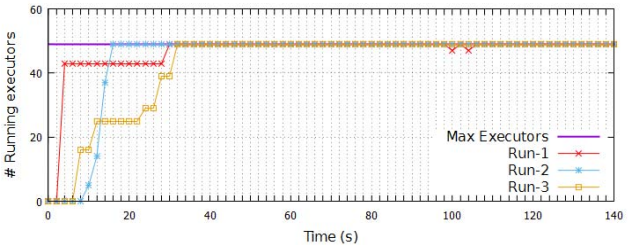

Fig. 7: Number of tasks running at run-time. Nimbus and Zookeeper located on SuperNodes

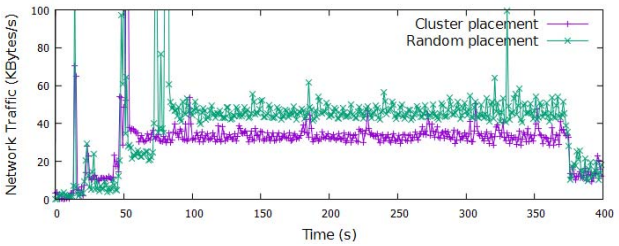

Fig. 8: Average nodes traffic for different Bolt placement scenario

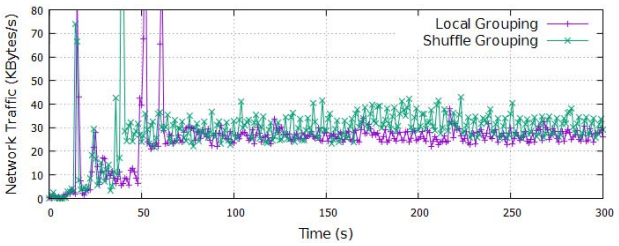

Fig. 9: Average nodes traffic for Shuffle and Local grouping. Bolt tasks only assigned on Cluster of SuperNodes

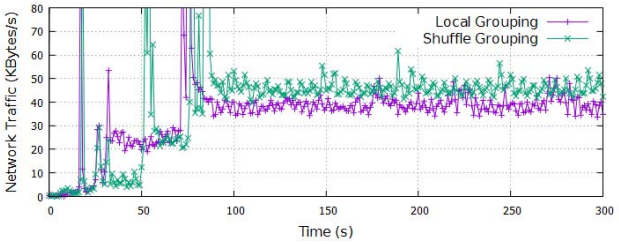

Fig. 10: Average nodes traffic for Shuffle and Local grouping. Bolt tasks assigned randomly between the available worker nodes

Fig. 6 RUN-2, it takes even more than 140 (214) seconds for some executors to establish connection with Zookeeper. It can also be seen that some of the executors fail and restart during the run-time. Therefore, the number of running executors fluctuates during the run-time. This is more frequent when Zookeeper and Nimbus are placed in the EdgeNodes than when they are placed in the SuperNodes. This is due to the fact that workers that run the executors keep losing their connections with Zookeeper, and therefore, Nimbus considers them as dead workers. However, as it can be seen in Fig. 7, the executors fail rarely when the management components are hosted on the SuperNodes.

Inside the community network, stream processing can be used to process the data closer to the source of the data, e.g., where sensors (temperature, humidity and etc) or logs from different processes are accessible through the community network nodes. Assuming that we know on which nodes sources of the data are located, our idea is to allocate the Storm topology tasks on those nodes. This concept is a bit different from the data center stream processing deployment where the data sources from different places are usually pooled into a message broker system such as Kafka [11] or non-SQL databases such as HBase [12], Cassandra [13].

From 52 nodes available in the emulation, we choose 29 nodes randomly to serve as worker nodes and 2 SuperNodes for Nimbus and Zookeeper. In contrast to the first experiment, in this experiment, the placement of Nimbus and Zookeeper is static. Each worker nodes have a single spout task that generates the tuples, to emulate the distributed sources. We create 10 bolts on each topology level and deploy them based on two types of placements: random placement and a cluster of SuperNodes. In each placement, there are 10 nodes with collocated spout and bolt tasks, and 19 nodes with only spout tasks. For the SuperNodes cluster, the group of 10 SuperNodes with high connectivity between each other have been chosen to form the cluster. In this experiment, we measure the amount of inout network traffic in each of 52 nodes using the ifstat utility. This information allows to quantify how the stream processing effects the whole community network, not only nodes that host the Storm cluster. Fig. 8 shows the average network traffic for different placements of bolts. Bolts placed in the SuperNodes cluster generate 30% less traffic compare to the amount of traffic generated by bolts placed randomly. This is because the network traffic generated by Storm in the SuperNodes cluster configuration is mostly circulated within the cluster; whereas in the configuration with randomly placed bolts, the tuples need to travel more number of hops between bolts.

Other interesting thing to explore is to see the effect of using different types of Storm stream grouping (described in Section II) on the amount of network traffic generated by Storm in the community network. In this experiment, we compare two types of Storm stream groupings, namely, the default shuffle grouping and the local-or-shuffle grouping. As one can see in Fig. 9 and Fig. 10, for each of two studied configurations (a SuperNodes cluster and a cluster of randomly selected nodes) apparently there is no significant difference in the amount of network traffic between shuffle and local-or-shuffle groupings. This is because the number of bolts is lower (around $\frac{1}{3}$) than the number of spouts. Therefore, there are few bolts that are collocated with the spouts and in most of the cases local-or-shuffle grouping behaves similar as shuffle grouping.

VI. Guidelines for the deployment of Apache Storm in a community network

In order to deploy the Apache Storm stream processing system in a community network, we need to consider the situation of running stream processing on distributed nodes with imperfect connectivity, rather than in a well-connected

data center. In this situation, as our evaluation has shown, placement of Storm components on distributed nodes is very important as it affect the amount of network traffic, and as a consequence, the stream processing performance. There are a number of issues to be considered when deploying Storm in a community network.

First issue is the heterogeneity of the latency and bandwidth between the nodes. A node with good computation power could perform very poorly if it has poor/imperfect connectivity with other nodes. There is a need to carefully position the Storm management components, Nimbus and Zookeeper, in the network on those nodes that have good connectivity with the nodes, which will serve as Storm worker nodes. The user also has to properly specify the Storm configuration parameters (shown in Table I) related to failure detection and fault tolerance, such as Worker heartbeat frequency and timeout, taking into account that Storm will operate in an inhospitable environment such as a community network with unreliable and heterogeneous nodes and links. Specifically, the values of all the parameters should be enlarged, as shown in Table I in order to prevent false-positive failure detection caused mostly by poor connectivity rather than node failures. The false-positive failure detection has negative impact on the system performance because it manifests task failures and requires restarting of tasks that actually did not fail. However, the values of the parameters should not be set too high in order to reduce the downtime in the case of actual failures.

Another important issue to be considered when deploying Storm in a community network is the topology of spouts (stream sources) and bolts (processing units). Spouts might be spread all over the community network, therefore it is important with respect to the system performance to achieve a good locality of bolts and spouts, i.e., to place bolts in close proximity to spouts in order to reduce network traffic, especially, if the links are heterogeneous and unreliable.

In this context, one important parameter is parallelism factor that defines the number of bolts and spouts assigned to the workers. The parallelism factor is mostly used to be able to cope up with the high data rates. When deploying Storm in the community network as explained in the Section V-B, the parallelism factor becomes important also with respect to location of spouts (data sources) because spouts are spread all over the network. If we try to distribute the bolts and spouts as close as possible to location of the data sources, we should consider the balance between the number of the bolt and spout tasks. If the parallelism factor for bolts is equal or close to the parallelism factor for spouts when using the local-or-shuffle grouping, more data can be processed locally to reduce the network traffic. On the other side, if the parallelism factor of bolts is less than the parallelism factor of spouts, then stream processing consumes less compute resources; however, the positioning of the bolts becomes important to reduce the inter-node communication. The Storm scheduler should find best nodes with good connectivity and betweenness centrality value based on the location of the spouts. Design of a new Storm scheduler, research on stream grouping as well as investigation on other Storm stream grouping such as field grouping, are subjects to our future work.

VII. Conclusion

In order to deploy a stream processing system, such as Apache Storm, in a community network, we need to consider the situation of stream processing on inhospitable environment of a community network with heterogeneous nodes and imperfect connectivity, rather than in a well-connected data center. Heterogeneity of the network bandwidth and latency in the community network raises an extra challenge in running the system smoothly. Default values of the heartbeat parameters need to be revised (enlarged) in order to prevent false-positive failure detection and, as a consequence, to decrease the downtime and to improve performance. The placement of the Storm components as well as tasks is important to reduce the traffic in the system. Our evaluation shows that the default stream grouping methods, namely, Shuffle and Local groupings, have similar results on network usage.

It is an open problem to make the Storm scheduler and stream groupings aware of the network topology, in order to provide optimal placement of spouts and bolts in the community network so that the traffic and inter-node communication are reduced. This is a subject for our future work.

Acknowledgment

This research is supported by the CLOMMUNITY project funded by the European Commission under FP7 Grant Agreement 317879; the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043; the ICT-TNG SRA initiative at KTH. We thank the anonymous reviewers for their valuable comments.

References

[1] J. Jimenez, R. Baig, P. Escrich, A. Khan, F. Freitag, L. Navarro, E. Pietrosemoli, M. Zennaro, A. Payberah, and V. Vlassov, "Supporting cloud deployment in the guifi.net community network," in *Global Information Infrastructure Symposium, 2013*, Oct 2013, pp. 1–3.

[2] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[3] (2015) Apache storm. [Online]. Available: https://storm.apache.org/

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[5] (2015) Guifi.net. [Online]. Available: http://guifi.net/

[6] (2015) Qmpsu. [Online]. Available: http://dsg.ac.upc.edu/qmpsu/index.php

[7] (2015) Common open research emulation core. [Online]. Available: http://www.nrl.navy.mil/itd/ncs/products/core

[8] M. Zec and M. Mikuc, "Operating system support for integrated network emulation in imunes," in *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004, pp. 3–12.

[9] (2015) Github: Storm-perf-test. [Online]. Available: https://github.com/yahoo/storm-perf-test

[10] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.

[11] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[12] A. HBase, "A distributed database for large datasets," *The Apache Software Foundation, Los Angeles, CA. URL http://hbase. apache. org*.

[13] A. Cassandra, "The apache software foundation," *URL: http://cassandra. apache. org/(visited on 01/05/2013)*.