# Fault Tolerance in Heterogeneous Environments

Jessica Woods, Nikhil Singh, Fabian Hofmann
Complex and Distributed IT Systems
Technische Universitat Berlin, Germany
Email: jessica.woods@campus.tu-berlin.de, nikhil.singh@campus.tu-berlin.de, f.hofmann@campus.tu-berlin.de

***Abstract*—Fault tolerant and low-latency processing of data streams from distributed edge nodes has increasingly become important in a number of IoT applications. Such applications typically transmit data from edge nodes to intermediate nodes to clusters of larger resources in the cloud. By moving some of the computational workload towards the edge of the network, these applications can harness the computational capabilities of these untapped edge nodes. Typically, such streaming applications, which are time critical in nature, require guarantees in terms of latency, network congestion, and fault tolerance. In this paper, we investigate and evaluate different fault-tolerant recovery strategies that take the heterogeneity of resource capabilities, network topology, and recovery time into consideration and can tolerate a permanent failure in any one processor. Our approaches are based on a passive replication scheme capable of supporting one fail-silent (fail-stop) processor failure at any time. We evaluated our results in a virtual geo-distributed and heterogeneous environment and employed a Primary/Backup model in which a pool of heterogeneous resourced backups are executed only if a primary copy fails. Our results demonstrate the effectiveness of different fault tolerance strategies in the face of simulated real-world failures.**

***Index Terms*—Fault Tolerance, Quality of Service, Cloud computing**

## I. Introduction

The Internet of Things (IoT) has gained a lot of traction over recent years and can be seen in vast arrays of daily life. As these applications continue to grow in popularity so will the amount of data that will need to be transmitted and analyzed [1]. Typically for these types of applications, data is procured close to the source, i.e. at the edge of the network and executed in a geo-distributed, heterogeneous environment. This means that source data is most often times obtained far away from cluster resources located in the cloud, which makes real-time decisions a lot harder when analytics are performed on a distant cloud [2]. To address these issues, concepts key to Edge and Fog computing have become vital to these applications to reduce latencies and network congestion [1]. As data travels from the edge to the cloud, available resources on the path are increasingly used to execute portions of analytics pipelines [1]. Traditional approaches for static datasets cannot provide the low latency execution needed for continuous, real-time stream processing when new data is constantly arriving. Contrary to distributed batch processing, resource allocation and scheduling in distributed stream processing is much harder due to the dynamic nature of input data streams [3].

Therefore, efficient execution of applications in such environments requires effective scheduling strategies. Strategies that take into account both the heterogeneity of shared resources, the network topology, and the actual intensity of input data flow [3]. However, most current frameworks still typically expect to run in a homogeneous environment and do not take these factors into account when scheduling jobs onto resources [1]. These scheduling algorithms play an important role in obtaining high performance in heterogeneous, geo-distributed systems with the objective to map tasks onto resources to increase overall system performance [4]. Additionally, these scheduling algorithms need to be reliable and able to tolerate resource failures that may occur within the system. Since occurrences of faults are often times unpredictable, fault tolerance, i.e. to tolerate an arbitrary number of failures during execution, must be considered during the design of scheduling algorithms to make systems more reliable [4]. Therefore, techniques to achieve fault tolerance are important. The issues of heterogeneity and fault tolerance are difficult problems on their own and when combined create an even harder problem to solve [5]. In this paper, we examine strategies to reliably and proactively deal with faults on nodes within the Apache Flink framework. Our strategy employs a passive replication scheme (Primary/Backup model) to mask failures, and relies heavily on the knowledge and understanding of relationships among the primary and backup copies and the predecessors and successors of tasks.

### A. Contribution

For our evaluation, we implemented the extended distributed dataflow system Apache Flink for Edge that gives users the ability to express the resource requirements of tasks as well as the capabilities of worker nodes. We also performed an evaluation of fault tolerance methods in a simulated geo-distributed heterogeneous compute environment using Docker, the extended Apache Flink system, and an IoT benchmark application. Along with this environment, we also implemented tools such as Hadoop and Zookeeper for fault tolerant master nodes, checkpointing and overall system robustness. Pumba was utilized to inject failures, latencies, and bandwidth limitations, and Weavescope was implemented for visualization purposes.

### B. Outline

The remainder of the paper is structured as follows. Section II discusses related work for fault tolerant task placement of distributed dataflows in heterogeneous environments. Section III presents basic definitions and assumptions. We outline our approach in Section IV and present our evaluation in Section V. Section VI concludes this paper and also presents ideas for future work.

## II. Related Work

This section presents an overview of work related to fault tolerant scheduling algorithms in geo-distributed dataflow systems and heterogeneous environments.

The Fault-tolerant scheduling algorithm proposed by [4] is a scheduling algorithm for precedence-constrained tasks in real-time, heterogeneous systems. Tasks are assumed to be non-preemptive and each task is assigned two copies, which are scheduled onto different processors and mutually excluded in time [4]. This scheduling algorithm achieves performance gains by allowing a backup copy to overlap with other backup copies on the same processor. The algorithm is able to do this because the technique assumes that only one processor can fail at any time and that a second processor cannot fail before the system recovers from the first failure. In addition, this algorithm also takes reliability measures of the system into account. However, the two objectives- reliability and performance- are not considered simultaneously. First, the algorithm tries to guarantee the timing constraints of the tasks. Then, among the processors, on which the time constraint of a task is guaranteed, the task is mapped to a processor that will minimize the failure probability of the system. Time constraints in this algorithm have priority over reliability.

Fault-tolerant Scheduling Algorithm (FTSA) [5] is an efficient fault-tolerant scheduling algorithm for heterogeneous systems based on an active replication scheme. This approach is capable of supporting an arbitrary amount of fail-silent (fail-stop) processor failures. The technique focuses on two main criteria, where the algorithm either aims to minimize latency given a fixed number of failures in the system, or minimize the number of failures given a fixed latency. Additionally, the authors also propose in [5] a Minimum Communications-FTSA (MC-FTSA), a variant of the algorithm, which aims to reduce the number of additional communications induced by the replication mechanism. Unlike our strategy, we do not implement an active replication backup model but we do focus on minimizing latency when recovering from processor failures.

Apache Flink for Edge proposed in [1] is a scheduling strategy for stream processing tasks on geo-distributed, heterogeneous resources. This scheduling approach focuses on finding the optimal placement for dataflow tasks with regard to minimizing latencies and avoiding bandwidth congestion [1]. The heterogeneity of resources derives from the dependencies of tasks, the input/output rates, and the operator requirements defined by users. These properties of the network and dataflow jobs make up the components of their offline-scheduler, which executes before the dataflow job takes place. Thus, the authors do not consider changes to the network topology or data rates. Our approach to the heterogeneity of compute resources and bandwidths between links is closely aligned to this strategy. However, the main difference between this study and ours is that the authors do not cover failure in processor resources and thus does not support fault tolerance. Our proposed strategy, however, is able to tolerate any one processor's failure, thus making the system more dependable.

## III. Background

In homogeneous environments, all nodes within a cluster have identical resources and a scheduler will evenly distribute the workload between the nodes of the cluster. Effective task scheduling in a homogeneous environment is determined by the quantity of nodes rather than their individual qualities [3]. This differs greatly to the resource allocations in a heterogeneous environment. Typically, nodes in a heterogeneous cluster vary in both computing performance and capacity. To balance the workload optimally, a scheduler has to know two key things: the performance characteristics of individual nodes within the cluster and the computation characteristics of incoming tasks and input data [3]. The unpredictable variability of input data in stream processing makes guaranteeing knowledge of the second requirement more difficult. To address this issue, user-defined tagging of input data can be implemented to help the scheduler with optimal performance [6]. With the knowledge of both the node performance characteristics and the input computation characteristics, a scheduler will have enough information to optimize both task assignments and application throughput [3].

While a large number of algorithms have been proposed for the scheduling of distributed stream processing in heterogeneous environments, fewer have dealt with the issue of fault tolerance. Fault tolerance is the ability of a system to continue operating even in the midst of a failure. Applications typically implement some sort of fault tolerance methodology in order to improve system reliability and achieve system robustness. Unfortunately, fault tolerance becomes harder the more heterogeneous the resources become. Errors in heterogeneous environments are harder to localize and have a higher probability of occurring the more geographically separated the nodes are [7]. Therefore, understanding and finding optimal fault tolerance solutions is incredibly important.

In dynamic environments such as the ones found in the cloud, many different types of system errors can occur- some transient, others unrecoverable (fail-stop) [8]. These two major types of failures affect a system differently. Transient failures only invalidate the current task but allow nodes experiencing the failure to recover and execute any subsequent tasks. Fail-stop failures, however, are unrecoverable. This means that once the fault occurs, the corresponding processor is down until the end of the entire execution [8]. In our paper, we consider only fail-stop failures, i.e. unrecoverable failures that interrupt the execution of the application.

To combat these failures, systems can implement a number of techniques. These fault tolerance techniques can either be reactive or proactive [7] as shown in Figure 1. Reactive fault tolerance techniques aim to reduce the effects of failures after one occurs. Techniques include checkpointing/restart, replication, and task re-submission. Proactive fault tolerance, on the

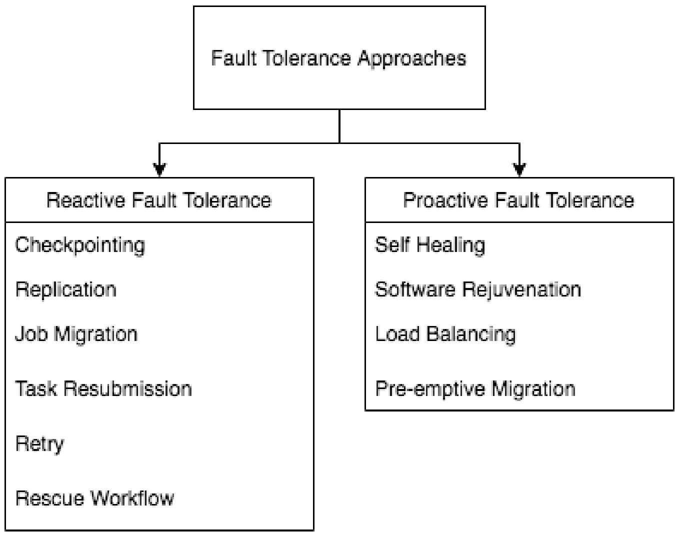

Fig. 1. Fault tolerance techniques.

other hand, tries to avoid recovery from faults by predicting them and proactively replacing faulty components with working components [6]. For our investigation, we focus on only reactive fault tolerance techniques. More specifically, we study checkpointing/restart mechanisms found in Apache Flink and passive replication techniques using a Primary/Backup model for resource nodes and checkpointing data. By studying these techniques, our paper aims to provide a better understanding of fault tolerance challenges in geo-distributed, heterogeneous environments and examine the various tools and techniques used for fault tolerance.

## IV. Approach

As already outlined in Section I, the focus of our work is the Apache Flink framework and making it fault tolerant for a heterogeneous environment. In this section we go into detail on how we approached this issue and summarize the different existing implementations and tools we utilized to guarantee fault tolerance.

### *A. Environment and Scheduling*

Because a heterogeneous environment consists of multiple layers of the network, such as cloud, fog, and edge, and data is processed on every layer, it is important to schedule a job properly so that it is optimally executed. As mentioned and explained in Section II, Janßen et al. provide a reliable scheduling strategy for stream processing tasks in an extended Apache Flink framework in a geo-distributed, heterogeneous environment [1]. To accomplish this, every node in the network topology contains information about its location within the environment and its own resource capabilities. The scheduler, on the other hand, knows the entire topology. To distinguish resource capabilities, such as `Edge`, `Fog`, and `Cloud` processing power, each node's capability must be identified before execution. Through a second variable, users can define the neighbors of a node. This helps the scheduler determine the entire network topology. Next, users must identify the requirements for a task into three distinct categories – high, medium, and low. This allows the scheduler proposed by Janßen et al. to not only resource fit tasks to nodes more efficiently, but also helps to better resource fit backup nodes if primary nodes face failures.

For our approach, we chose to require expert knowledge to define the locations and capabilities of nodes and the requirements of each individual task in the environment. By utilizing the external scheduler presented by Janßen et al., we also assume a static environment in the sense that bandwidths between nodes do not change and the general topology does not change. The external scheduler, a server accessed by the Flink job manager when scheduling tasks, should be placed in a location close to the job managers. For our study that means on the cloud. This helps to guarantee fast scheduling and low latency communication between the job manager and external scheduler.

### *B. Detection*

As mentioned in Section III, our approach solely assumes a fail-stop model, i.e. unrecoverable failures that interrupt parts or all of the execution. Because our approach is reactive to faults, the first step of our approach is to detect failures for both task and job managers. To detect a failed task manager in Flink, the job manager, the master, will request periodic heartbeat signals from all active task managers [9]. If the job manager does not receive an expected heartbeat from a node, the node is seen as failed and the recovery process is initiated. Failed job managers, on the other hand, are detected by their peers,if these exist, as described in the following section.

### *C. Recovery*

Due to the assumption of the fail-stop model, any failed task or job manager is guaranteed to never return, i.e. once a node disappears from the network, it is gone for good. This implies that there is no way to recover a faulty state without the use of backup nodes. However, the different types of nodes in a Flink application vary in responsibilities and are handled differently.

*1) Job Manager:* In the case of a failed job manager, the node that is responsible for both scheduling and resource management, is gone [9]. This is a single point of failure and brings the whole system to a halt. Therefore, Flink provides an optional high availability mode, that allows a single leading job manager to have multiple active standby replicas. These standbys are elected to take over leadership in case the current leading job manager fails. Leader election is organized using a ZooKeeper quorum which provides the necessary recovery mechanism in case the current leader – the active job manager, fails [10]. It also provides the newly elected job manager with the necessary state information of the failed master, so the job manager can just resume where the other one stopped.

*2) Task Manager:* Meanwhile, when the failure of a task manager is detected by the leading job manager, things are handled differently. In data streaming applications, it is important that every record from the data stream is handled exactly once [11]. As a result, when a single node fails, one can not just recover the task to another node in active standby, because records could be skipped or be processed multiple times. Therefore, a global state of the entire streaming process has to be recovered. In Flink, this is done via snapshots [9]. Periodically at certain points in the data stream, snapshots from each task manager are taken and stored in some state backend storage. Now, when a failure occurs, Flink can simply restart the operators and reset the whole system to its last saved global state through the use of the distributed snapshots.

This, combined with the external scheduler explained in Section IV-A, results in the following process when a task manager fails: As soon as the failure of a task manager is detected by the lack of a heartbeat signal, the entire job is stopped by the job manager. The job manager then requests a new schedule for the job from the external scheduler, passing along topology and capability information about the active task managers to the scheduler. As a result, failed tasks could be rescheduled to other task managers in active standby which could change the desirability of nodes in that neighborhood. For example, because of a failure in one task manager, its neighbors could become less attractive for the new schedule, thus resulting in those nodes becoming active standby after recovery.

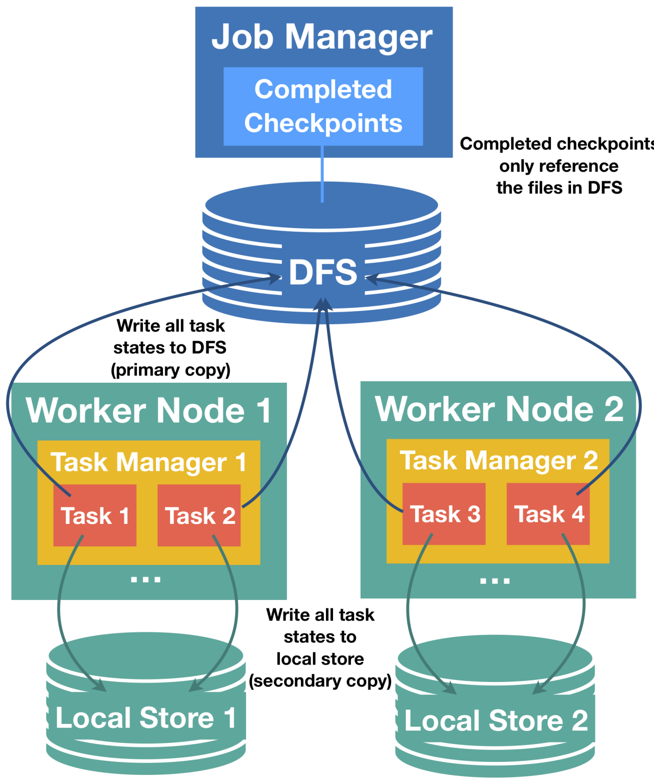
Job Manager
Completed Checkpoints
Completed checkpoints only reference the files in DFS
DFS
Write all task states to DFS (primary copy)
Worker Node 1
Task Manager 1
Task 1
Task 2
...
Worker Node 2
Task Manager 2
Task 3
Task 4
...
Write all task states to local store (secondary copy)
Local Store 1
Local Store 2

Fig. 2. Apache Flink primary and secondary state snapshots [9].

### D. Checkpoint Storage

As explained in the previous section, checkpointing is crucial for Flink to uphold the exactly-once semantic in case of a failure. However, there are several ways to store these checkpoints in a state backend. Each of which come with their own challenges and advantages [9].

*1) Heap-Based:* The first one is the easiest one to configure, checkpoints are simply sent from the individual task manager to the job manager where they are stored within the job manager's heap memory. The main advantage of this storage strategy is that management is centralized. Communication is organized solely between task managers and a single job manager. However, it becomes increasingly inefficient for applications with larger checkpoint states and applications that need a fault tolerant job manager. As described in Section IV-C1, because state information of a job manager has to be managed via Zookeeper, the heap based storage becomes a single point of failure, that has to be managed via Zookeeper [9].

*2) Distributed:* An alternative to a heap-based checkpoint storage is the organization of checkpoints via a distributed file system (DFS), such has the one provided by hadoop (HDFS) [12]. In HDFS, the storage is distributed among multiple nodes that each individually only contain parts of a single checkpoint. When storing a new checkpoint, the file is split into equally sized blocks, before it is saved to a single node of the file system. Each block is then replicated at least two times, depending on the configuration, and distributed among all other nodes. This means that even in case of a node failure, the file is not lost. It is important to note however, that not only the task managers need to know the location of the individual blocks, but also the job manager [9]. During failure of a task manager, the job manager has the responsibility of finding the location of the latest checkpoint which will be used in recovery. Finally, it is important to note that checkpoint retrieval does not happen only from a single node – e.g. HDFS allows for retrieval of any blocks of a file concurrently from multiple nodes to optimize their data retrieval process.

Compared to a heap-based storage, DFS provides us with the necessary fault tolerance we need. Furthermore, this storage is accessible to all nodes and checkpoints can be easily redistributed. However, it also carries higher recovery overhead costs in terms of required storage, configuration, and data retrieval latencies. Additionally, DFS nodes need to be placed in areas of the network with higher resource capabilities such as the cloud. There, the storage nodes will have access to the necessary resources to guarantee their performance.

*3) Task-Local:* Lastly, there is a task-local storage, i.e. the checkpoint is directly stored on the task manager that created the snapshot [9]. This of course must be seen as a secondary option, for in case of the node's failure, the checkpoint stored locally would also be lost. However, there are scenarios where tasks could be recovered locally. In the scenario of a single task manager failure, each task manager would have to be recovered to its latest checkpoint before the job can continue. Normally when using a distributed storage this requires the

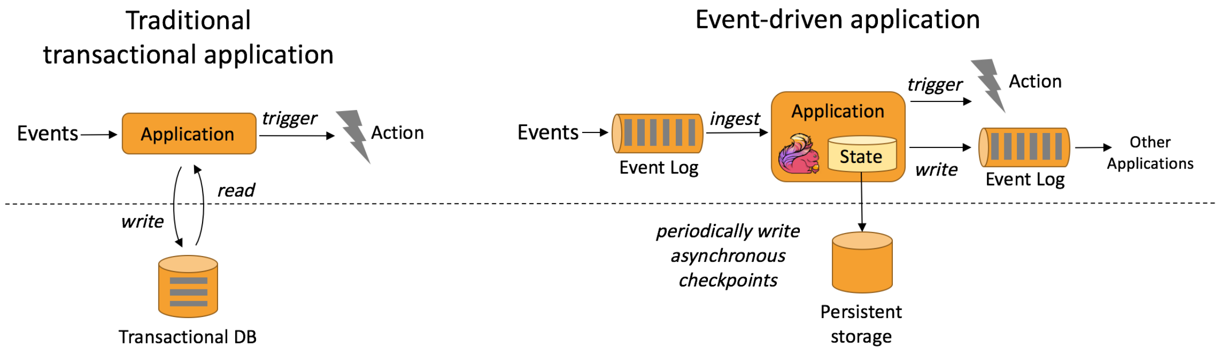

Fig. 3. Traditional vs. event driven application [9].

job manager to send out the latest checkpoint handles so that the individual task managers can then restore their state from the distributed storage. However, using this strategy, all tasks would need to retrieve their state from multiple nodes of the distributed file system, over the network and at the same time. The would result in high latencies, especially in a heterogeneous environment, where latencies between locations could have high variability. Therefore, a task-local recovery, where checkpoints are saved in the local memory, serves as a speedup for the entire recovery process [9]. As Figure 2 illustrates, for every checkpoint, each task does not only save its checkpoint remotely, but also keeps the checkpoint locally. This means that those nodes, which were unaffected by a failure, can reduce network congestion by bypassing a call to the distributed storage and instead recovering from a locally stored checkpoint. However, because the local storage does not ensure reliability in case of a node failure, this makes the local storage merely a secondary copy. Thus, for checkpointing to be guaranteed as successful, the primary is the only copy that must be saved successfully. Also of importance, because task managers manage their own local checkpoints, the secondary copy is always used first, before the recovery is attempted via the remote store. By using this optional feature provided by Flink, we are able to better cope with the latencies between the task managers and the distributed file system in case the recovery process is initiated.

## V. Evaluation

There are several mechanisms to enable fault tolerance in Flink, as illustrated in the previous section. We evaluated our investigated methods in Apache Flink using a virtual test bed to simulate geo-distributed, heterogeneous nodes. This section provides a short overview over our implementation and our test bed. Afterwards we present and discuss the results.

### *A. Prototype*

To begin, we first create our geo-distributed, heterogeneous nodes. Our test bed includes 3 Edge nodes, 2 Fog nodes, and 3 Cloud nodes each with low, medium, and high resource capabilities, respectively. We were able to specify memory and cpu limitations on specific nodes by utilizing docker-compose commands and we introduced latency and bandwidth limitations by using Pumba. Our implementation uses the external scheduler proposed by Janßen et al. [1] in order to test different fault tolerant strategies. Our study focuses on 3 fault tolerance strategies in particular. The first strategy evaluates the recovery of a system configured to store checkpoints in the heap memory of the job manager. In the next strategy, we configure our system to enable task-local recovery, i.e. storage of checkpoints in the task manager's memory and evaluate its recovery process. Finally, we implement a replicated HDFS storage to test recovery with a remote storage system. Since the external scheduler requires extra information about the topology of the network and the capabilities of the individual task managers, we tag task managers with the necessary information upon startup [1].

### *B. Experiment Setup*

For the evaluation, we created an event driven application much like the one shown in Figure 3, which records and process the temperature readings in the remote forests designated on the edge of the network to prevent ”Fire”. It triggers a ”Danger” alert if the temperature exceeds 50 degree Celsius. Along with the alert, the application also calculates the average temperature from the last ’n’ readings. This was an added feature we used to ensure that our application was stateful which is key to our checkpointing evaluation. Then, we designed a heterogeneous cluster from Docker containers, which were designated for specific types of resource capabilities. With different CPU and memory limitations in place, we then assigned bandwidth limitations between resource capabilities to imitate real world latency and network congestion. Figure 4 illustrates visually our virtual evaluation test bed.

We found node placement to also be a key factor in our evaluation setup. To minimize the effects of bandwidth limitations and latencies, we purposefully placed each node into a specific area of the network. Because job managers frequently communicate with other nodes of the network and play a vital role in application execution, we placed those nodes into the cloud and assigned them high resource capabilities. Apache Zookeeper, which is used to enable High availability and to prevent Job manager failure, was also placed into the cloud along with the external scheduler. Checkpoints stored in HDFS were placed in the Fog. This was done to both

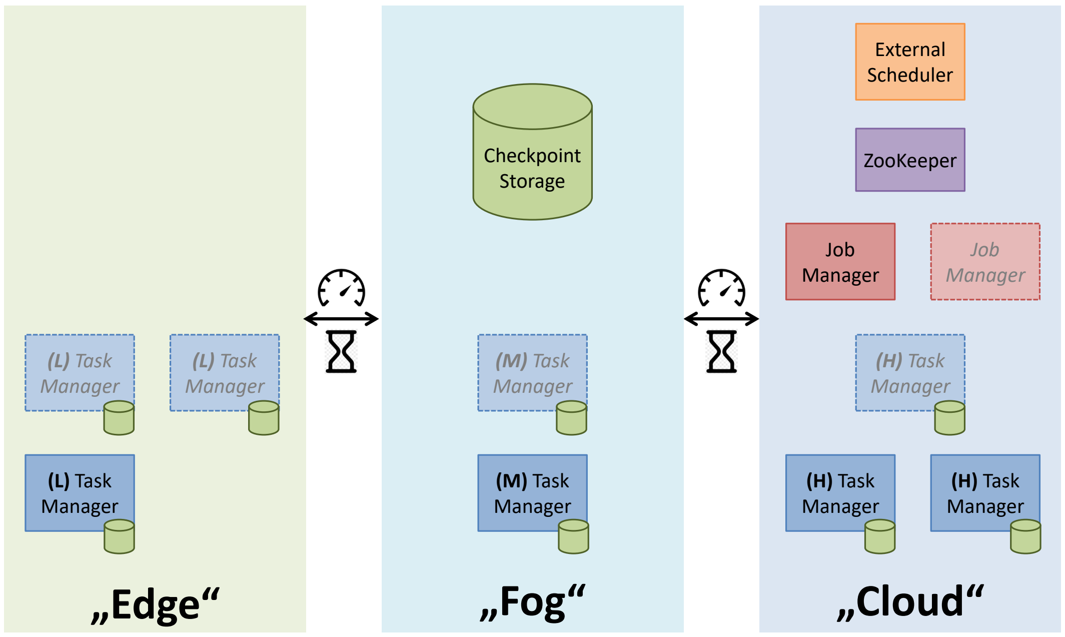

Fig. 4. Evaluation setup of test bed.

minimize checkpoint retrieval latencies between the 3 resource capabilities but also to ensure the HDFS has enough resources to guarantee performance.

We configured Flink with exactly-once level of fault tolerance guarantee and setup checkpointing intervals to prevent resource exhaustion. Flink offers users the flexibility to customize checkpoint configurations and restart strategies. By default, when checkpointing is not enabled Flink follows a 'No Restart' strategy. The job will simply fail when it encounters an error. When checkpointing is enabled and a restart strategy is not configured, Flink will provide the system with a default 'Fixed delay' restart strategy. Flink also provides the option of a 'Failure rate' strategy, which fails a job when it exceeds some specified failure rate value. In our experiment we used a 'Fixed Delay' restart strategy with 3 restart attempts each after 5 seconds. We used Pumba for resilience testing to pause, stop, and kill containers and emulate network conditions such as latency and network congestion. Results will be discussed in the next section but here we briefly specify the 3 fault tolerance strategies for state backends that our study evaluates:

- `MemoryStateBackend` (Heap-based)
- `FsStateBackend` (Task-Local)
- `HDFS` (Distributed Storage)

### *C. Results*

*1) MemoryStateBackend:* We observed that, when we submit the job, Flink's job manager's runtime receives it and its execution graph switches the state from CREATED to RUNNING. Afterwards, the operator's state switches from CREATED to SCHEDULED and the runtime CheckpointCoordinator triggers checkpointing. If operators are in a SCHEDULED state, the CheckpointCoordinator aborts checkpointing. Flink's runtime registers the job manager at the resource manager and allocates a leader ID. Then runtime requests a new slot and resource profile, such as cpu cores and memory, from the resource manager. After allocation, the stream operator's state switches from SCHEDULED to DEPLOYING and DEPLOYING to RUNNING. After this process, the runtime CheckpointCoordinator triggers and completes the checkpoints. Flink's runtime then sends heartbeats to the task managers. If the task manager fails or if a network connection problem occurs, the runtime will try to reconnect several times. However, when the heartbeat times out, it closes the task executor connection which will then switch the operator's state from RUNNING to FAILED. The CheckpointCoordinator then discards the latest checkpoint which hasn't been completed and the runtime will transition the job from RUNNING to FAILING and the operator's state from RUNNING to CANCELING to CANCELED. Flink's runtime now tries to restart based on the configured restart strategy and switches the job's state from FAILING to RESTARTING to CREATED. It will then try to restore the job from the latest valid checkpoint and switch the state from CREATED to RUNNING. The runtime now switches the Socket Stream operator's state from CREATED to SCHEDULED on a second available backup task manager and the whole process repeats. Once the Socket Stream operator's state switches from DEPLOYING to RUNNING, the CheckpointCoordinator begins triggering and completing checkpoints again. What we found is that although this approach is fast and fault tolerant, it is limited to the size of the stored state.

*2) FsStateBackend:* In this case, many of the steps are the same as case 1. However, checkpoints are stored in a file state backend within the individual task manager under '/tmp/flink/checkpoints'. Once the Socket stream operator switches from DEPLOYING to RUNNING, Flink's runtime CheckpointCoordinator triggers and completes the first checkpoint and continues on. Whenever the heartbeat of task manager times out, the recovery mechanism is put into motion. The Socket Stream switches from RUNNING to FAILED and the CheckpointCoordinator discards the last triggered incomplete checkpoint. Flink's runtime now tries to restart the job but cannot establish the connection due to the host being unavailable. In this case, the CheckpointCoordinator cannot find the appropriate state to restore the job to and therefore fails the job execution. The job manager stops job execution and the runtime disconnects the job from the resource manager. With this strategy, the application cannot recover from a node failure meaning this approach is not fault tolerant.

*3) HDFS:* In case 3, we modified our evaluation setup so that Flink would store the state backend in Apache Hadoop. Hadoop is an HDFS framework that provides persistent, reliable and distributed storage. Again, many of the steps during failure and recovery are identical to the two previous cases. Upon startup, the Flink runtime will start a standalone session cluster entrypoint and check for a Hadoop dependency. Once one is found, the runtime will load the configuration properties and the resource manager will register task managers with unique resource ids. The CheckpointCoordinator then triggers checkpointing and changes the state of the job from DEPLOYING to RUNNING. Checkpoints are completed successfully until an error (fail-stop) occurs, which will prompt the runtime to switch the RUNNING job to a FAILED state and transition the operators RUNNING state to a CANCELED one. By implementing fault tolerance mechanisms, Flinks runtime will then try to restart the failed job which will change the state from FAILING to RESTARTING to CREATED. Because Flink will automatically write and store state snapshots during execution, it is able to restart the failed jobs by locating the latest, valid checkpoint. With the assistance of the Job Manager, Flink is able to accurately locate and retrieve the latest checkpoints from the persistent HDFS located in the Fog. Once checkpoints have been successfully retrieved, the job will then switch states from CREATED to RUNNING and the Socket Streams operator state will switch from CREATED to SCHEDULED on a similarly resourced, available backup task manager. Once the operators state transitions from DEPLOYING to RUNNING then the job is will be back to a functioning state and is said to be recovered.

Though the recovery process works much the same as the other cases, using an HDFS state storage comes with its own specific challenges and advantages. The biggest advantage is of course the persistence of data in the face of fault (due to replication) and the larger storage size in comparison to other storage options. However, even though HDFS systems are great for larger states, they incur higher latencies on data retrieval than the other cases [13]. We found that a FsStateBackend is faster in terms of recovery time but required a large amount of heap memory and was not fault tolerant in the face of fail-stop errors on resource nodes. The MemoryStateBackend we discovered to be more suited for testing purposes due to state storage size limitations in the job manager JVM heap but fast and fault tolerant upon job recovery. In the end, implementing a HDFS for state storage was determined to be the best-suited storage strategy in a geo-distributed, heterogeneous environment despite the high recovery overhead.

## VI. Conclusion

In our paper, we have discussed the challenges of fault tolerance in heterogeneous environments and examined various fault tolerance techniques to overcome those challenges. We mainly focused our study on fail-stop errors using reactive fault tolerance techniques such as checkpointing and passive replication in Apache Flink. Results demonstrate that the performance of fault tolerance strategies increases when combined with multiple fault tolerant techniques and this was most evident in the HDFS evaluation setup. However, a major flaw of an HDFS in the current implementation is that its placement does not really matter to an extent – nodes retrieve blocks of files from all the HDFS data nodes in the entire environment. This results in a high latency when a task manager on the cloud requests blocks of a checkpoint from a data node on the edge, though it could also be requested from a data node on the cloud layer as well. In our current implementation it is therefore not wise to distribute the data nodes, but to focus them on one location – the fog.

Therefore, in our future work, we intend to extend the block placement and retrieval policy by the DFS. Only when task managers are allowed to retrieve blocks from their closest neighbors, a low latency in the recovery process can be guaranteed. However, this does bring more challenges for the DFS data nodes: Currently, there already is a high requirement of expert knowledge that evaluates the resource requirement of tasks, the capabilities of task managers and knows the topology of the entire environment. Thus we suggest an automated process that explores the topology and learns capabilities and locations of each individual task manager and data node to reduce external assistance.

## References

[1] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen, "Scheduling stream processing tasks on geo-distributed heterogeneous resources," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 5159–5164.

[2] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, 2016, pp. 20–26.

[3] M. Rychly *et al.*, "Scheduling decisions in stream processing on heterogeneous clusters," in *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 2014, pp. 614–619.

[4] X. Qin, H. Jiang, and D. R. Swanson, "A fault-tolerant real-time scheduling algorithm for precedence-constrained tasks in distributed heterogeneous systems," *Technical Report TR-UNL-CSE 2001–1003*, 2001.

[5] A. Benoit, M. Hakem, and Y. Robert, “Fault tolerant scheduling of precedence task graphs on heterogeneous platforms,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[6] A. Bala and I. Chana, “Fault tolerance-challenges, techniques and implementation in cloud computing,” *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 288, 2012.

[7] Z. Amin, H. Singh, and N. Sethi, “Review on fault tolerance techniques in cloud computing,” *International Journal of Computer Applications*, vol. 116, no. 18, 2015.

[8] A. Benoit, L.-C. Canon, E. Jeannot, and Y. Robert, “Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms,” *Journal of Scheduling*, vol. 15, no. 5, pp. 615–627, 2012.

[9] A. S. Foundation, “Apache Flink 1.9 Documentation,” accessed 2019-09-27. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.9/

[10] ——, “ZooKeeper: Because Coordinating Distributed Systems is a Zoo,” accessed 2019-09-27. [Online]. Available: https://zookeeper.apache.org/doc/r3.5.5/

[11] P. C. A. Katsifodimos, S. E. V. Markl, and S. H. K. Tzoumas, “Apache flinktm: Stream and batch processing in a single engine,” *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng*, vol. 36, no. 4, 2015.

[12] A. S. Foundation, “Apache Hadoop 3.2.1  HDFS Architecture,” accessed 2019-09-27. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[13] D. Matar, “Benchmarking fault-tolerance in stream processing systems,” *Master’s thesis. TU-Berlin*, 2016.