Towards Resilient Data Management for the Internet of Moving Things

Elena Beatriz Ouro Paz[1], Eleni Tzirita Zacharatou[2], Volker Markl[3]

Abstract: Mobile devices have become ubiquitous; smartphones, tablets and wearables are essential commodities for many people. The ubiquity of mobile devices combined with their ever increasing capabilities, open new possibilities for Internet-of-Things (IoT) applications where mobile devices act as both data generators as well as processing nodes. However, deploying a stream processing system (SPS) over mobile devices is particularly challenging as mobile devices change their position within the network very frequently and are notoriously prone to transient disconnections. To deal with faults arising from disconnections and mobility, existing fault tolerance strategies in SPS are either checkpointing-based or replication-based. Checkpointing-based strategies are too heavyweight for mobile devices, as they save and broadcast state periodically, even when there are no failures. On the other hand, replication-based strategies cannot provide fault tolerance at the level of the data source, as the data source itself cannot be always replicated. Finally, existing systems exclude mobile devices from data processing upon a disconnection even when the duration of the disconnection is very short, thus failing to exploit the computing capabilities of the offline devices. This paper proposes a buffering-based reactive fault tolerance strategy to handle transient disconnections of mobile devices that both generate and process data, even in cases where the devices move through the network during the disconnection. The main components of our strategy are: (a) a circular buffer that stores the data which are generated and processed locally during a device disconnection, (b) a query-aware buffer replacement policy, and (c) a query restart process that ensures the correct forwarding of the buffered data upon re-connection, taking into account the new network topology. We integrate our fault tolerance strategy with NebulaStream, a novel stream processing system specifically designed for the IoT. We evaluate our strategy using a custom benchmark based on real data, exhibiting reduction in data loss and query runtime compared to the baseline NebulaStream.

Keywords: Mobile Stream Processing, Internet-of-Things, Fault Tolerance, Buffering

1 Introduction

Existing research and development for the Internet-of-Things (IoT) has primarily focused on stationary objects that are associated with a fixed location. However, some of the most important pieces in the IoT landscape are devices that can move (i.e. dynamically change their geo-spatial position), such as mobile phones and tablets. Mobile devices have become ubiquitous; smartphones, tablets and wearables have all become daily commodities for many people. A report published by GSMA in 2020 estimates that by 2025 there will be 5.8 billion

[1] Teradata (this work was done while the author was at TU Berlin, Germany), elenabeatriz.ouropaz@teradata.com
[2] TU Berlin, Germany, eleni.tziritazacharatou@tu-berlin.de
[3] TU Berlin, DFKI GmbH, Germany, volker.markl@tu-berlin.de

mobile subscribers and that nearly 80% of connections will be made from smartphones [GS]. The ubiquity of mobile devices combined with their ever increasing computing capabilities give rise to new systems that leverage mobile devices in the processing of streaming data. Furthermore, they enable new IoT applications where mobile devices are used to both generate and process data. While mobile devices open numerous possibilities for the IoT, using them to process data is challenging. Mobile devices are notoriously prone to transient disconnections due to network instabilities while their near-constant mobility within the network renders much of the existing research in the field of stream processing inapplicable.

State-of-the-art stream processing systems that use mobile devices base their fault tolerance strategies on replication [OSP18, CS20] or checkpointing [WP14, MRH14]. Neither of these techniques is optimal for using mobile devices as data sources that also perform processing. Checkpointing-based techniques require the devices to store and broadcast snapshots of their state periodically. This is a pessimistic approach that introduces a considerable overhead irrespective of whether failures occur or not, and can drain the scarce resources of mobile devices. Replication-based techniques, on the other hand, replicate query operators across multiple nodes in the network. However, replicating the source can be impractical or costly, as it requires to have redundant sensors capturing the same input. When redundant sensors are not available, replication-based techniques cannot provide fault tolerance for data sources. Finally, existing approaches assume that disconnections are permanent and exclude the disconnected device from data processing. As a result, any data generated during the disconnection are lost. Furthermore, when the device re-connects, it is treated as a new node and all the queries that use it as a source have to be redeployed. This imposes an undue performance overhead when the disconnection is only transient.

In this paper, we propose a fault tolerance strategy for overcoming transient disconnections of mobile devices that both generate and process data in a streaming system. We particularly focus on transient disconnections caused by mobility, where the device might move through the network during the disconnection period, and then re-connect at a different point in the network. In addition, we assume that we do not have control over the data source, and therefore we cannot pause and restart the source or change the data generation rate. The goal of our proposed strategy is twofold: (a) avoid losing data during the disconnection, and (b) resume query execution quickly and consistently when the device re-connects. To achieve this goal, we employ the following mechanisms. First, we propose a data logging mechanism that enables mobile devices to keep processing the data they generate while they are disconnected by temporarily storing the processed data in a circular buffer. That way, we leverage the idle disconnection period for computation, and reduce, or completely avoid, data loss. Furthermore, we propose a query restart process that achieves efficient and consistent resumption of query execution, even when mobile devices move during the period of disconnection and reappear at a new point in the network. Our query restart process examines whether the device can still reach its previously assigned downstream node after the re-connection, and only redeploys queries when the downstream node cannot be reached. When a query is redeployed, the device may be assigned different tasks than those it was originally performing. Our restart process handles this situation by generating new paths through which the buffered data can be forwarded directly to the pertaining node, when needed. This is achieved by taking advantage of the natural way in which stream processing systems model queries as directed acyclic graphs (DAGs). Our contributions can be summarized as follows:

- We mitigate the impact of transient disconnections of mobile devices with an end-to-end reactive fault tolerance strategy that leverages the offline processing capabilities of the devices. In our strategy, mobile devices continue processing the data they generate locally during a disconnection, and store the processed data in a circular buffer until connection is regained.
- We propose a query-aware replacement policy to make space in the buffer when needed.
- We develop a query restart process that addresses the mobility of devices during transient disconnections.
- We integrate our solution with the NebulaStream (NES) platform [Ze20a], a novel streaming system for the IoT. We call our solution NebulaStream-MSS, where *MSS* stands for *Mobile Source Support.*
- We evaluate our solution on real data collected from a sensor attached on a football player during a match. Our results show that buffering reduces data loss by 63% over a disconnection period of 30 seconds compared to the vanilla NES. Additionally, when several disconnections occur during query execution, our restart process reduces the query runtime by more than 2× by avoiding query redeployment whenever possible.

We envision our approach as a building block that can be combined with other fault tolerance strategies to handle other failure scenarios, such as devices that permanently exit the network, or failures occurring in nodes that are not data sources.

In the remainder of this paper, we first motivate our work in Section 1.1 and then present the background concepts that lay the foundation for NebulaStream-MSS in Section 2. Section 3 describes our approach, which we then experimentally evaluate in Section 4. Finally, we present an overview of related work in Section 5 before concluding and discussing future work in Section 6.

1.1 Motivational IoT Application Scenario

To illustrate the importance of a fault tolerance strategy tailored for mobile data sources, let us look at an IoT disaster management application that monitors the vital signals of firefighters responding to a fire incident and the temperature around them. The goal of the monitoring is twofold: (a) pull a firefighter out of the scene when a life-threatening condition is detected, and (b) make better decisions through fast and accurate assessment of the situation. Each firefighter is monitored by a single wearable sensor, and therefore previously proposed fault tolerance strategies that rely on the availability of a back-up device capturing the same input are not applicable. The wearables are interconnected using a wireless network.

In this application, the most time sensitive operation is the detection of abnormalities in the firefighter's vital signals. To reduce the latency and increase the survival chances of the firefighter, we need to perform this operation as close as possible to the source, i. e. the wearable sensor. Even if the wearable suffers from a transient disconnection, it is critical to keep processing the captured data offline, as otherwise we might miss a sign that the firefighter needs help. That way, when the wearable regains connectivity to another network node, it can transmit an alert without delay.

To assess the overall situation, the fire area is divided in zones and every wearable generates a periodic report of the maximum temperature that was recorded in each zone. These reports are then aggregated, to obtain the global maximum temperature for each zone. This task uses the combined processing capabilities of multiple nodes in the network. To get an accurate assessment of the situation, we need to minimize the amount of data loss when wearables suffer from a transient disconnection. Furthermore, as the wearable sensors that generate data are worn by firefighters, they will move through the network with them. We therefore need to ensure that query execution resumes once connection is regained, even when a sensor moves to a different point in the network topology during the disconnection.

2 Background: NebulaStream

In the following, we give an overview of data stream processing, focusing on NebulaStream, the IoT data streaming platform into which we integrate our fault tolerance strategy.

IoT infrastructures are highly distributed, with sensors spanning entire cities or even countries. In addition, many IoT applications require prompt responses. One natural solution to support such applications are Stream Processing Systems (SPS), i. e. systems capable of performing computational tasks over a potentially infinite sequence of data items, called *tuples*. NebulaStream (NES) is a novel end-to-end SPS specifically designed for IoT applications [Ze20a, Ze20b], born from the need for a SPS that can better deal with the main challenges in IoT settings. NES aims to provide similar functionalities as widely used SPS such as Flink [CEH15] or Spark [Za16] while better supporting IoT applications. To that end, NES introduces mechanisms to handle, among others, the hardware heterogeneity in IoT environments, the high distribution of data and compute, and the unreliable communication in fog and sensor networks.

Prior systems employ either the cloud or the fog paradigm. Cloud-centric systems (e. g. Flink, Spark or Kafka [NSP17]) collect all data in a data center before processing them. Given that upcoming IoT applications will require processing data from millions of distributed sensors,

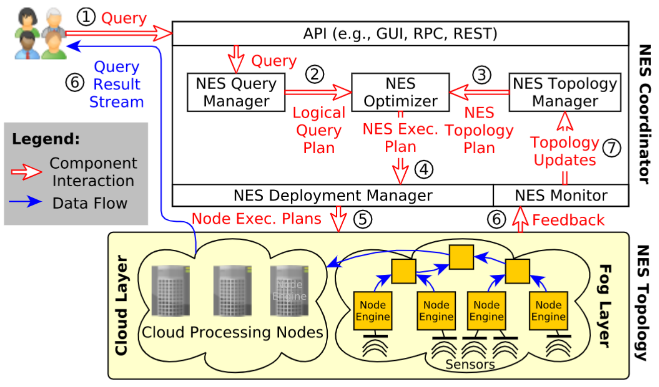


Fig. 1: NebulaStream's Architecture. Figure from [Ze20a].

this centralized approach results in a bottleneck and lacks scalability. Existing systems based on fog computing are also not optimal for the IoT. Systems like Frontier [OSP18] or CSA [Sh16] bring computations closer to the sources, thus reducing the bottleneck of cloud systems. However, they advocate a completely decentralized approach and only exploit the computing capabilities of nodes at the edge of the network. In the field of Wireless Sensor Networks (WSN) we can find Database Management Systems (DBMS) such as TinyDB [Ma05] that exploit the combined capabilities of sensors and actuators. These systems provide strategies to optimize query execution for devices with limited battery life but offer neither strong fault tolerance nor correctness guarantees. NES takes the state-of-the-art one step further and unifies the cloud, fog and sensor layers, leveraging the individual advantages of each layer and enabling optimizations across them. Specifically, data are typically generated within the sensor layer and routed through intermediate nodes up to the cloud. Any of the nodes in the path between the sensors and the cloud can access any data being routed through them and perform data processing tasks [Ga20]. The cloud performs any remaining processing as a fall-back.

NES Topology. There are three types of nodes in NES: *Workers*, *Sensors*, and a *Coordinator*. *Workers* process data by employing *operators* that consume tuples, apply a computational task (e.g. filter, map, sum, count), and emit new tuples. *Sensors* generate data and can also perform computational tasks. Mobile devices in NES fall in that category. Finally, the *NES Coordinator* is in charge of administering the network: it is aware of the current topology, it registers/deregisters nodes and streams, and deploys/undeploys queries. A single physical device can potentially host multiple NES nodes.

The network topology maintained in the coordinator is modeled as a graph consisting of sensor, worker and coordinator nodes and network links among them. Initially, the network is formed solely by the coordinator and one worker contained in it. Further workers and sensors can join the network by sending a request to the coordinator. They are then added to the topology graph. In the case of sensors, the streams that the sensors generate are also registered in the topology graph. Once registration is complete, the new node is ready to start participating in query processing. When a node needs to exit the network, it does so by sending a deregistration request to the coordinator. The coordinator then removes the node and any streams it might generate from the topology graph, and undeploys the affected queries that were using the removed streams.

NES Query Deployment. Figure 1 displays an overview of NebulaStream's architecture. A query submitted through the NES API ① contains information regarding the streams from which data should be obtained and the computational tasks that should be applied to them. Currently, NES uses a centralized deployment process. The coordinator is comprised of several components that handle different aspects of the deployment. First, the *NES Query Manager* transforms the user query into a directed acyclic graph (DAG) that contains source, processing, and sink nodes and directed links among them. A *source* is a node that generates data streams, i. e. provides the input, and a *sink* is a node that consumes data streams without generating new ones, i. e. provides the output. The links represent the communication channels for the data exchange among operators. The DAG is essentially a logical query plan that describes conceptually the operations that need to be performed over a data stream. To execute queries in a SPS, we need to generate a physical query execution plan (QEP) which maps the logical query plan to physical nodes. The *NES Optimizer* takes the query DAG from the Query Manager ② and the topology graph from the *NES Topology Manager* ③ and creates a physical query execution plan (QEP) which dictates the assignment of the different operators in the logical query plan to physical nodes in the topology ④. The process of assigning operators to physical nodes is known as *operator placement*. The next paragraph discusses operator placement in further detail. The *NES Deployment Manager* takes the execution plan and deploys the operators to their assigned nodes in the topology ⑤. Finally, the *NES Monitor* collects information about the changes occurring in the network topology ⑥ and sends the updates to the Topology Manager ⑦.

NES Operator Placement. The choice of the placement strategy depends on the optimization goal, e. g. low latency, high throughput, or minimal use of resources. NES supports several operator placement strategies. In addition to them, we also implemented a shortest-path based placement strategy. In contrast to the already existing strategies that always place the sink operator on the coordinator, the shortest-path based placement strategy allows to place the sink operator on any arbitrary node in the network. That way, we enable greater proximity to end-users and can avoid streaming data to the cloud-based coordinator. Clearly, the source operator is assigned to the sensor node that generates the data for the queried stream. The shortest-path based strategy places the remaining operators as follows. It first finds the shortest path between the source and the sink. Then, it applies a "push-down as far as possible" strategy: It starts placing operators on the shortest path from the source all the way up to the sink according to the available resources on each node, i. e. while there are remaining resources on a node, it keeps assigning operators to it. This strategy is particularly suited for queries that consist of filter operators as it reduces downstream data traffic. Once the assignment is complete, the coordinator sends to each of the nodes the part of the query execution plan that they need to execute (i. e. a pipeline of operators), and indicates from which node they will receive their input, and to which node they should send their output. Some nodes in the path might be assigned a forward operator, which means that their only mission is to transmit data to the next node in the execution plan without processing them. Our fault tolerance solution is independent of the choice of the placement strategy and can thus be used in conjunction with any strategy. The NES Optimizer uses a placement strategy to generate a QEP given a query DAG and a network topology. As we will describe in more details in Section 3, our approach does not modify the QEP. It simply determines *when* a new QEP needs to be generated upon a device re-connection, and analyzes the newly generated QEP to determine *how to forward* the data that have been buffered during a data source disconnection.

3 Fault Tolerance for Mobile Data Sources

Our approach, NebulaStream-MSS, is an extension of NebulaStream that provides fault tolerance capabilities for mobile data sources such as mobile phones and wearables. Our approach is holistic: it handles the disconnection process, the data buffering during the disconnection, as well as the process of resuming query execution after the re-connection. Specifically, our approach employs a circular buffer that temporarily stores the data that are processed locally during a transient disconnection. We couple the buffer with a query-aware replacement policy that evicts data when the buffer becomes full before connectivity is regained without penalizing any query unequally. To handle the mobility of devices during a disconnection, we combine buffering with a restart process that determines which of the queries that involve the device require redeployment. When a query is redeployed, the device might be assigned different tasks than before the disconnection. Our restart process handles this situation by generating new paths through which the buffered data can be forwarded directly to the appropriate node, ensuring that the buffered data are processed correctly. We describe NebulaStream-MSS in more details in the following sections.

3.1 Disconnection Process

Our solution aims to handle transient disconnections of mobile devices. This raises the question of how we can distinguish a transient disconnection from a more permanent failure. From the point of view of the coordinator both look the same: a node has unexpectedly disappeared from the network. The short answer is that we cannot know with certainty when a disconnection is transient. We, however, follow an optimistic approach where at first we assume that all the disconnections that happen to mobile devices are transient, i.e. once a mobile device disappears from the network, the coordinator assumes that it will return. This is in contrast to the current approach in NES, where the coordinator completely removes the node and any associated streams or queries from the system upon a disconnection. If the device does not return after a timeout period, the coordinator assumes that the disconnection is permanent and acts accordingly. Choosing the timeout duration is challenging and depends on both the network characteristics and the application requirements, as different applications can tolerate different delays. The problem of finding the timeout duration is orthogonal to the problem addressed in this work and we leave it open for future research.

3.2 Data Buffering

Mobile devices acting as data sources can still both generate data and perform computations while transiently disconnected. Furthermore, in many real-world scenarios (e. g. the one described in Section 1.1), we have no control over the data source, and thus pausing data generation or modifying the data generation rate is infeasible. Therefore, to avoid data loss during a disconnection, we need to temporarily store the generated data on the device. To reduce the amount of data that needs to be stored and to exploit the offline processing capabilities of the device, we also propose to keep processing the data locally during the idle disconnection period. To that end, we base our solution on a buffering component that stores the data that are generated and processed on a data source node during a disconnection period. In the following, we discuss our buffering component in more details.

Buffer Structure. To ensure efficient space utilization and minimize access latency, we use a circular buffer. Circular buffers are particularly suited when the data are accessed in a First In First Out (FIFO) order. Our buffer stores batches of tuples after they are processed. As our device may have several streams or queries attached to it, every batch may have a different size and contain tuples of different formats. To retrieve batches from the buffer, we need to be able to infer their size. For this reason, we always precede batches with a control block, i.e. a small memory region in the buffer that contains metadata about the batch of tuples that follows it. Specifically, a control block stores a query identifier that indicates the query to which the batch belongs, the tuple size, and the number of tuples in the batch. Figure 2 presents the conceptual view of our circular buffer. As with any circular buffer, we always maintain two pointers: one pointing to the address of the first control block stored in the buffer (*Read Pointer*) and one pointing to the first free address in the buffer where new batches can be added (*Write Pointer*).

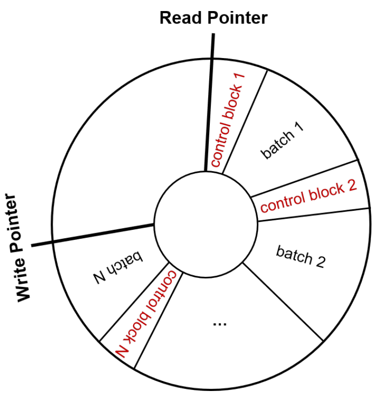


Fig. 2: Conceptual view of the circular buffer in NebulaStream-MSS.

Data Insertion. Once the data source has executed all operators assigned to it over the generated data, it sends the processed batch to the buffer. Inserting the batch into the buffer is fairly trivial as long as there is enough available space to store the incoming batch and the corresponding control block. The control block is simply written into the first free address in the buffer indicated by the Write Pointer, followed by the batch of tuples.

Query-aware Replacement Policy. As stated before, our buffer might store results corresponding to different queries. To achieve fairness among different queries running on the device, we propose a query-aware FIFO replacement policy that aims to always keep some results of each query in the buffer. Specifically, whenever we need to make space for a new batch of tuples, our policy replaces the oldest batch of the same query if possible.

If there are no other batches of the same query, or if removing the batch is insufficient, it removes the oldest batch of the query with the highest number of batches in the buffer. The process is repeated until there is enough space in the buffer for the new batch. One drawback of this policy is that it might cause segmentation in the buffer, i.e. fragments of empty space. To address this, whenever we remove a batch from the buffer, we shift all subsequent batches up, filling up the generated gap. Algorithm 1 presents the data insertion process using our query-aware FIFO replacement policy.

Algorithm 1: Data insertion with query-aware FIFO

```
Incoming Data: a batch of tuples batch for query with queryID
while (available space <(controlBlock.size + batch.size)) do
    delID = queryID
    if (there are no results of query with queryID in the buffer) then
        delID = query with highest batchCounter
    end
    delete oldest batch of query with delID
    move up subsequent batches
    batchCounter[delID]- -
end
copy batch at writePointer
writePointer += (controlBlock.size + batch.size)
batchCounter[queryID]++
```

Data Extraction. Once the mobile device regains connection to the network, we can proceed to send the buffered data. Extracting data from our buffer is fairly simple. Starting at the Read Pointer, we calculate up to what address in the buffer each result is stored, as we have information about the size of the batch in the control block. Since we use a circular buffer, no data shuffling is required on data extraction. Once we have retrieved a batch, we advance the Read Pointer up to the next control block.

3.3 Query Restart Process

Once the connection of a mobile device is reestablished, we have to restore the system to its normal working state. First, the device notifies the coordinator of its return, indicates the queries it was involved in, and transmits its current position by announcing which devices are in its neighborhood. Based on that information, there are two possible scenarios that determine the actions that should be taken by our system: re-connection without mobility and re-connection with mobility. In the first scenario, the device can still reach its sink nodes. This typically happens when the device did not move, or moved very little, during the disconnection period. In the second scenario, the device can no longer reach its original sink nodes, which happens in the case of high mobility.

Re-connection without Mobility. In this scenario, query execution can continue as normal, i.e. there is no need to modify the query execution plan. Once the coordinator receives the re-connection notification, it checks the position of the device and determines

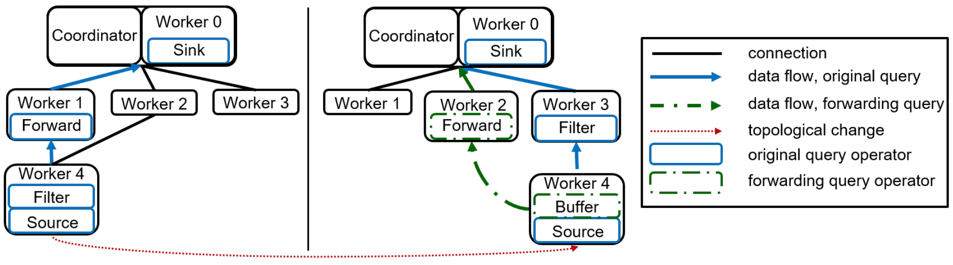


Fig. 3: Mobility during the execution of a simple filtering query. Before the disconnection (left), the device (Worker 4) acts as the source in the QEP, and also applies a filter operator. During the disconnection, Worker 4 moves out of the range of its donwstream node (Worker 1). Once it returns at a new point in the network (right), the new QEP (shown in blue) assigns the filter operator to Worker 3. As the buffered data have been already filtered, we want them to bypass Worker's 3 filter. This is achieved by deploying a forwarding query (shown in green) that sends the buffered tuples to Worker 2.

that the device remains connected to its previously assigned sink nodes. Then, query execution resumes and the device transmits the data stored in the buffer.

Re-connection with Mobility. In this scenario, the device moves through the network during the disconnection period, reappearing at a different position in the topology than the one that it last registered for. When this occurs, the mobile device might be unable to reach the sink nodes that had been previously assigned to it. Figure 3 shows an example: the mobile source (Worker 4) was original connected to Workers 1 and 2 (left). After the disconnection (right), it remains connected to Worker 2, but moves out of Worker's 1 range. Instead, it is now connected to Worker 3. This mobility affects query execution and requires action from the system. Once the coordinator receives the notification that the device has returned, it can see from its current position that it has moved. As a result, it triggers the redeployment of every query for which the device can no longer connect to its sink nodes. The coordinator can employ any of the available operator placement algorithms to perform the redeployment. There are three possible scenarios that can arise in the newly generated query execution plan: the device is assigned (a) the **same** operators as before, (b) a **superset** of the original operators, or (c) a **subset** of the original operators.

If the data source is expected to apply the same set of operators over new data once the re-connection is complete, then all operators have already been applied to the data that were processed during the disconnection period. Therefore, we can simply retrieve the data from the buffer and send them to the nodes indicated by the new query execution plan.

If the data source is assigned a superset of the operators that it was previously executing, the data contained in the buffer have not been processed fully. In this case, every time a batch of data is retrieved from the buffer, we first need to apply the remaining operators in the pipeline to it, before sending it to the nodes specified in the query execution plan.

In the last scenario, the data source is assigned a subset of the original operators. In this case, we cannot simply send the buffered data to the next node in the new query execution plan, as that node would re-apply some operators that have already been applied to them. For that purpose, we propose a mechanism that forwards the buffered data to the pertinent node in the network and skips over the operators that have already been applied to them. As described before, NES supports the forwarding of data through the topology using forward operators. Therefore, a natural way to create a forwarding path is by issuing a new query that uses forward operators to bypass the operators which have already been applied to the buffered data. We call this query a *forwarding query*. The coordinator generates automatically the forwarding query as follows. First, it specifies that the source of the forwarding query is only the buffered data, and not the newly generated tuples. Then, it examines the QEP of the original query to determine the node that contains the first operator that has not been applied to the buffered data yet. This node is the sink of the forwarding query, named *fwd_sink*. Next, the coordinator executes the shortest-path based operator placement algorithm and places a forwarding operator in every node of the path between the source and the fwd_sink. Lastly, the coordinator has to determine the operators that will be placed on the fwd_sink node. To do so, it examines the operators that are assigned to the fwd_sink node in the original QEP. If none of the operators has been applied to the buffered data, then the coordinator does not need to place any additional operators on the fwd_sink node. Instead, the tuples of the forwarding query will join the QEP of the original query. This is achieved by notifying the fwd_sink node about the mapping between the original and the forwarding query, so that the sink node can match the information arriving from the forwarding query to the original QEP. However, if the fwd_sink node contains some operators that have already been applied to the buffered data, the forwarding and the original query cannot be merged yet. Instead, the coordinator generates a new pipeline of operators for the forwarding query that contains only the operators that have not been applied to the buffered data yet. In this case, the tuples of the forwarding query join the original QEP in the next downstream node. Figure 3 shows a simple example of mobility during disconnection that requires the deployment of a forwarding query. Forwarding queries are short-running: once the buffer is empty, they are removed from the system and processing continues normally.

Imagine now that for the scenario shown in Figure 3 (right), one of the nodes connecting the source and sink in the forwarding query suffers from a disconnection as well. To handle such cascading disconnections, we enable a certain level of replication in our forwarding queries. We do so by finding as many paths between the source and the sink node as possible, up to a maximum indicated by a user-specified replication rate. The buffered data are then redundantly transmitted through all those paths, to avoid data loss in case of an intermediate node suffering from a failure. Each path is modelled as a separate forwarding query, i.e. we issue multiple forwarding queries concurrently that all have as input the same buffered data. To avoid processing the same data received through different paths multiple times at the sink node, each batch is assigned a timestamp and the sink node keeps track of the timestamps that it has already seen for each query. These timestamps are flushed periodically at the sink node, as we no longer expect to see duplicate batches after a certain period of time.

4 Experimental Evaluation

In this section, we first describe the experimental setup and then present the evaluation of our approach to support mobile sources in NebulaStream. The aim of the performed experiments is to showcase the fault tolerance capabilities of our approach and its efficiency when compared with the vanilla NebulaStream. Section 4.2 presents a comparative analysis while section 4.3 analyzes the performance of our approach in more details.

4.1 Experimental Setup

Hardware. The experiments were performed on a machine with a 10th generation Intel® Core™ i7 processor and 16 GB of RAM running Ubuntu 20.04 LTS.

Implementation. NES (and therefore also NebulaStream-MSS) is implemented in C++. Control messages between the coordinator and other nodes are handled through gRPC [GRP], while data transfer is performed using ZeroMQ [Hi].

Data. As we explain in Section 1, in our work we assume that we cannot control the data source, i. e. we cannot pause and restart the source and we cannot modify the data generation rate. The data source simply generates data infinitely, at a certain frequency. However, in our experiments we want to show the benefit of processing data during the disconnection period, which is what our approach enables, versus pausing query execution completely during the disconnection, which is what NebulaStream does. For that purpose, we use a CSV file as a source, which allows to pause data generation and easily restart it by storing the last read position in the file. We use the DEBS 2013 Grand Challenge dataset (described in [MZJ13]) which contains data generated from sensors attached on football players during a match. Since we focus on a single data source, we extract the data corresponding to a single player, the one with ID 10. The resulting CSV file contains a total of 6,575,830 tuples, each of 62 bytes. In all the experiments, our CSV data source produces data at a rate of 3.33 MBps, i. e. we read 56,375 tuples every second. In addition, we use a binary generator that produces tuples containing a single 64-bit unsigned integer field at a rate of 488 KBps.

Queries. We use two queries, one for each of the aforementioned data sources. The first, referred to as "Query 1", is a filtering query that takes as input the CSV source. It applies two filters: filter 1 selects tuples that satisfy the condition `vx > 0` and filter 2 selects tuples satisfying the condition `az > 1`. The overall query selectivity is 24%. Once filtered, the results are written to a file. Our second query, "Query 2", simply streams the tuples of the binary generator source through the topology and prints them in standard output at the sink.

Network Topology. All the experiments presented in this evaluation, start with the same network configuration depicted in Figure 4 (left), consisting of the coordinator (that also contains the sink), and 8 other workers, one of which contains the source for both queries. Even though in a real-world scenario the full topology of the network would be significantly larger, we consider our topology to be representative of the relevant part of the network

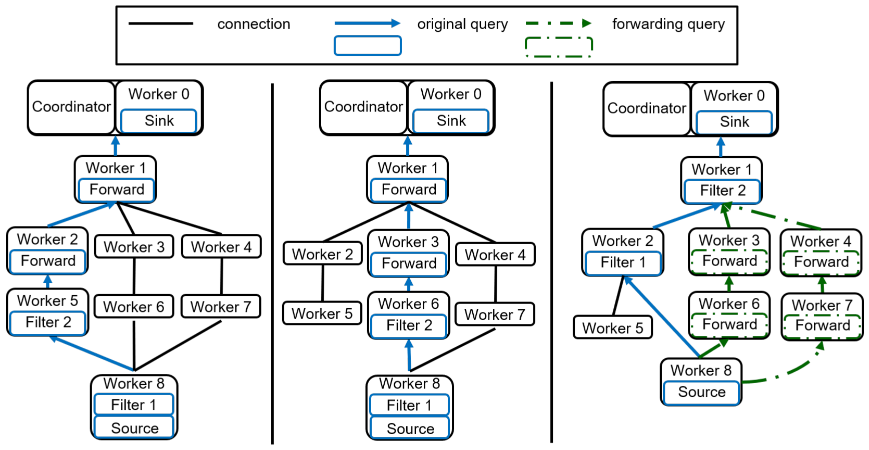


Fig. 4: Three topology snapshots along with the placement of Query 1 operators. (Left): Initial topology and placement. (Middle): After a transient disconnection, worker 8 gets disconnected from worker 5, which affects the placement of Query 1. There is no need for a forwarding query, as worker 8 applies the same operator as before (filter 1). (Right): After a transient disconnection, worker 8 gets disconnected from worker 5 and connected to worker 2. In the new placement, worker 8 no longer applies filter 1, and thus two forwarding queries are deployed.

that is involved in processing data captured by a given mobile data source. As we deal with mobility of the data source during disconnections, the topology is not static. Following a transient disconnection, the source might re-connect at a different location within the network. The middle and right parts of Figure 4 show the different topology snapshots that we use to simulate mobility. Given the different topology snapshots, we deploy Query 1 as described next. As you can see in Figure 4 (left), in the initial topology, one filter operator is placed at the source (worker 8), worker 5 applies the second filter, while workers 2 and 1 forward the data to the sink. The placement shown in Figure 4 (middle) is similar to the original one. The only difference is that, since now the source is disconnected from worker 5, the second filter is applied by worker 6, while it is worker 3 that forwards the results to worker 1. The last operator placement corresponds to the re-connection scenario that requires query redeployment, and the source is assigned a subset of the original operators after the redeployment (Figure 4 (right)). In this case, filter 1 is already applied to the buffered data, so we forward them directly to worker 1 which applies the second filter. For that, we employ the two forwarding queries shown with green color. “Query 2” (not shown in the figure), simply forwards the generated data through the shortest path.

Configuration Parameters. Unless otherwise stated, the buffer size is set to 50MB. We chose this size considering the characteristics of modern smartphones and wearables, but also the fact that we focus on short disconnections during which a limited amount of data are generated. When applying data forwarding, we use a replication factor of 2.

Evaluation Metrics. We use two metrics throughout our evaluation: query runtime and data loss ratio. Query runtime gives us insight into the performance of our approach, while the data loss ratio indicates the level of fault tolerance that we can achieve. We define *query runtime* as the time between the moment that a new query starts processing data and the moment that the query has processed a fixed amount of data. Specifically, in our runtime experiments we measure the total time that it takes for Query 1 to process 2M tuples of our CSV data source. *Data loss ratio* is the ratio of data that were evicted from the buffer due to lack of space over the total amount of data that were generated during the disconnection period after applying all local operators. Specifically, the data loss ratio (%) is calculated over a fixed disconnection period as: $\frac{\text{amount of data evicted from the buffer}}{\text{total amount of generated data after applying local operators}} \cdot 100$. We ran each experiment 5 times and report the average results.

4.2 Comparative Analysis

We first perform a comparative analysis of NebulaStream-MSS with the vanilla implementation of NebulaStream. The baseline NES considers all node disconnections to be final, i. e. upon a disconnection, the node is completely removed from the system. In the case of a data source, this results in halting the execution of the queries that the source is involved in. Furthermore, the data that are produced during a disconnection are lost.

Data Loss Ratio. This experiment explores the fault tolerance capabilities of NebulaStream-MSS, based on the amount of data that are lost during transient disconnections. We assume that the mobile data source continues to produce data while being transiently disconnected from the other nodes in the network, as it would happen with a sensor in a real-world scenario. We run concurrently Query 1 and Query 2 in the topology of Figure 4 (left), with worker 8 hosting the data sources for both queries. Worker 8 also applies Filter 1 of Query 1, even during the disconnection, and only buffers the filtered data. The filter selectivity is 67.6%. To show the benefits of our query-aware FIFO replacement strategy, we also implemented a simple FIFO strategy. As Figure 5 shows, the replacement strategy impacts the amount of data loss. This is because we are running two different queries that generate batches of different sizes, 3.33 MB and 488 KB respectively. Since the FIFO strategy always removes the oldest batch, even when we only need space for 488 KB in the buffer, we might end up removing a batch of 3.33 MB, if this batch is the oldest. This

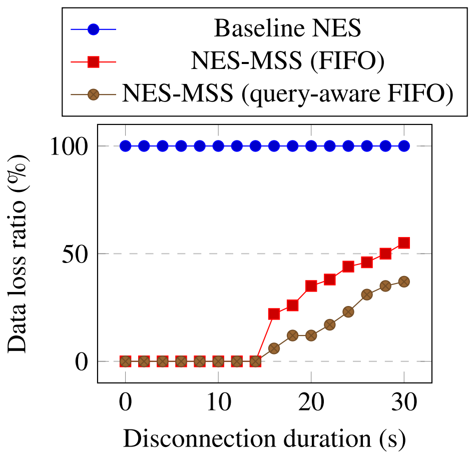
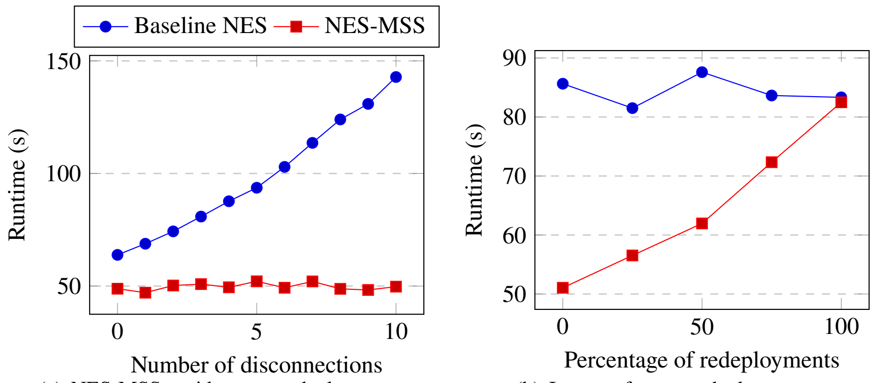


Fig. 5: Data loss ratio during a transient disconnection. After 15 seconds of disconnection, the buffer becomes full and tuples start to be dropped.



(a) NES-MSS avoids query redeployment. (b) Impact of query redeployment.

Fig. 6: Runtime of Query 1 (processing 2M tuples). 6(a): By avoiding query redeployment, NES-MSS has a constant runtime irrespective of the number of disconnections. 6(b): NES-MSS's performance converges to the one of baseline NES when the query needs to be redeployed after every disconnection.

can rapidly escalate the amount of data loss. Our query-aware FIFO strategy is far less susceptible to the time at which batches arrive at the buffer since we remove batches of the same query to make space for new data.

Query Runtime. We evaluate the runtime of Query 1 (i. e. the total time for processing 2M tuples), with a varying number of disconnections occurring during its execution. In order to be able to fairly compare the performance of our approach (that buffers data) with baseline NES (that loses any data generated during a disconnection), in this experiment we use a source that can be paused and restarted so that both systems process the same data throughout the experiment. Figure 6(a) shows the case where the query does not need to be redeployed after the disconnections. As can be seen in the figure, the runtime of baseline NES increases with the number of restarts while the runtime of our approach is almost constant. This is because baseline NES views the failure as permanent and removes the disconnected data source and the queries running on it from the system. When the source regains connection, it is treated as a new node: first, the coordinator has to re-add the source to the topology, and to register the streams that it generates in the system. Then, the queries applied on the generated streams are redeployed and restarted (as described in Section 2). Our approach, on the other side, does not remove the source and the queries from the system, keeps processing data locally at the disconnected source, and resumes execution fast once the source re-connects. In a real-world scenario, we expect to see multiple transient disconnections of a given device over time, due to mobility and network instability. It is likely that in some cases query redeployment will be required. We therefore study the impact of query redeployment on execution and show the results in Figure 6(b). For a fixed number of disconnections occurring during the processing of 2M tuples with Query 1, we evaluate how the runtime varies based on the percentage of disconnections that require redeployment. As expected, we found that with fewer redeployments, data are processed faster. On the other side, when we need to redeploy queries after every single disconnection, the performance of NebulaStream-MSS converges to the one of baseline NES. That indicates the importance of avoiding query redeployment whenever possible and having a lightweight query restart mechanism, which is what our approach offers.

4.3 NebulaStream-MSS Analysis

In this section we take a closer look into the performance of our system to identify optimization opportunities.

Buffer Sensitivity Analysis. Figure 7 shows the impact of the buffer size on the data loss ratio for both buffer replacement strategies. We use the same setup as in the "Data Loss Ratio" experiment and we fix the disconnection duration to 30 seconds. As expected, there is an obvious correlation between the buffer size and the amount of the incurred data loss. Choosing an appropriate buffer size is vital to minimize the amount of data that are discarded during disconnections. The optimal size depends on the data generation rate, the disconnection duration, the query, and the available resources on the device. Furthermore, comparing the two strategies, we see that our query-aware FIFO strategy benefits more from a larger buffer size. This is because it makes better use of the available buffer space by dropping batches of the same size as the ones it needs to make space for. For 100 MB there is only a 6% difference between the two strategies, as the buffer can fit almost all the data that are generated during 30 seconds, while for 125 MB no data are discarded.

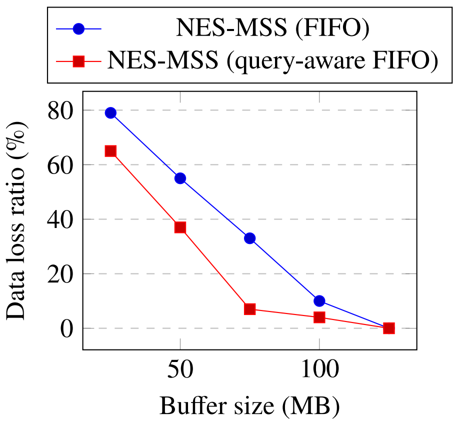
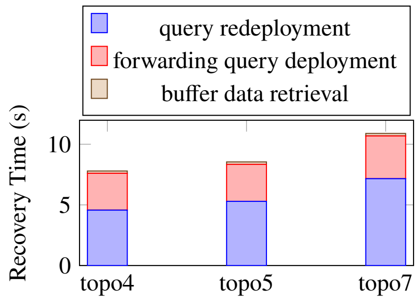
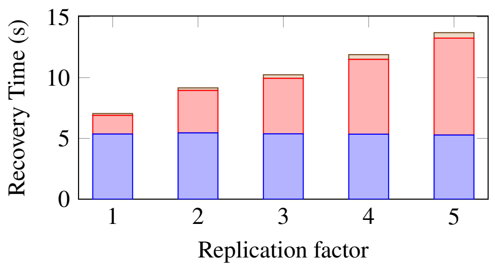


Fig. 7: Data loss ratio for a transient disconnection of 30 seconds. A buffer of 125 MB can fit all the generated data. When the available buffer space is more limited, the query-aware FIFO strategy can make better use of it.

Recovery Time Breakdown: Impact of Topology Size. This experiment investigates the time elapsed between the moment that a source executing a query regains its connection and the moment that the system resumes normal execution for different topology sizes. We run Query 1 on three topology configurations. *Topo5* corresponds to the topology of Figure 4 (left), where the query path between the source (worker 8) and the sink (worker 0) contains 5 nodes in total, while there are two more paths of size 5 between worker 8 and worker 0 (worker 8 -> worker 6 -> worker 3 -> worker 1 -> worker 0 and worker 8 -> worker 7 -> worker 4 -> worker 1 -> worker 0). *Topo4* is the same as topo5, but all paths are of size 4. Similarly, *topo7* is the same as topo5, but all paths are of size 7. The source disconnects after processing 2M tuples and re-connects after a short disconnection period. As Figure 8(a) shows, the largest portion of time is spent on redeploying the original query



(a) Impact of topology size.



(b) Impact of replication factor.

Fig. 8: Recovery time breakdown. 8(a): The redeployment time increases for larger topologies. 8(b): The time to deploy forwarding queries increases for higher replication factors.

and on creating and deploying forwarding queries, i. e. query deployment is clearly the main overhead. In addition, as expected, query redeployment takes longer for bigger topologies. We note, however, that we perform query redeployment only whenever necessary, i. e. when the source can no longer reach its original downstream node upon re-connection. In section 6 we discuss some possible directions to further mitigate the redeployment overhead.

Recovery Time Breakdown: Impact of Replication Factor. We repeat the previous experiment, but this time we vary the replication factor of the forwarding queries. The topology used in this experiment is similar to the one of Figure 4 (left) but wider: it contains 8 additional paths between worker 8 and worker 0. As expected, the results in Figure 8(b) show that the time to create and deploy forwarding queries increases for higher replication factors. The time to retrieve and send the buffered data also increases slightly, as the data are sent to more nodes. The query redeployment is independent of the forwarding queries, and remains, thus, constant. Overall, there is a trade-off between the level of fault tolerance and the recovery time, which we plan to further investigate in future work.

5 Related Work

Mobile Stream Processing. Mobile Stream Processing (MSP) [Ni15] refers to performing stream processing tasks purely at the edge by combining resources of multiple mobile devices. Given that our work focuses on the use of mobile devices in stream processing, fault tolerance strategies that have been proposed in the context of MSP are the closest to our work. Existing fault tolerance strategies in MSP fall broadly into two categories: checkpointing-based and replication-based. MobiStreams [WP14] combines two checkpointing protocols, token-triggered and broadcast-based checkpointing, aiming to reduce the network overhead that checkpointing strategies typically induce. In token-triggered checkpointing, a controller node periodically prompts data sources to checkpoint their state. To coordinate the checkpointing process, the data sources generate tokens that are sent to the downstream node upon completion of the checkpoint. Once a node receives tokens of all its upstream neighbors, it checkpoints its own state and sends its token downstream. To avoid losing the checkpointed state when nodes exit the network, the broadcast-based checkpointing protocol broadcasts nodes' states in the network. The main shortcoming of MobiStreams is that it does not consider the limited resources of mobile devices. Checkpointing operations and state broadcasting have a considerable impact on battery life. In addition, MobiStreams consumes memory resources on the devices to store the checkpointed state even when the system is functioning correctly, introducing an unnecessary overhead. In contrast, NebulaStream-MSS follows a reactive approach and only buffers data upon a disconnection. Symbiosis [MRH14] attempts to optimize checkpoints not only for network overhead, but also for energy efficiency. It proposes to trigger checkpointing when a node in the network either moves outside the range of connectivity, or reaches a critical battery threshold. However, Symbiosis only considers failures caused by mobility and energy levels and does not account for sudden ad-hoc disconnections caused by network instabilities.

On the other end of the spectrum we find systems that base their fault tolerance strategy on replication. Frontier [OSP18] models queries as replicated data flow graphs, where each operator is replicated in multiple nodes resulting in multiple paths between a source and a sink node. It then adjusts the data flow dynamically based on a backpressure stream routing algorithm. To recover from the disconnections of data sources, Frontier applies replication also at the source level. In IoT scenarios, however, it is not always possible to have multiple sources (i. e. sensors) for the same data. MobileStorm [Ni15], is a stream processing platform for clouds of mobile devices, but does not provide support for transient disconnections. To address that, R-MStorm [CS20], an extension of MobileStorm, follows a similar approach as Frontier. It introduces path diversity by replicating operators, so that there is always a path between the source and the sink, even in case of failures, and performs dynamic path selection. In addition, R-MStorm attempts to improve the overall system availability by using an operator placement strategy that prioritizes devices with higher availability during operator assignment. R-MStorm does not provide fault tolerance at the data source level and therefore cannot recover from transient disconnections of data sources. Finally, Swing [FSL18] is a MSP that considers the dynamism of mobile devices. Each upstream node in Swing maintains routing information about the reachable downstream nodes. When a network link is broken, the affected upstream nodes update their routing records and re-route data to other nodes. However, any data that are generated from the moment of the disconnection until the completion of the reconfiguration are lost. In contrast, we reduce or completely eliminate data loss by buffering data on the disconnected device.

Distributed Stream Processing. Existing distributed stream processing systems provide fault tolerance through one of the two following approaches [Hw05]: upstream backup, where nodes buffer sent data while the downstream nodes process them and replay them to a recovery node upon a failure [Qi13, STO], or replication, where each node is assigned a second node as a backup [Ba08, SHB04]. As we already explained before, replication-based approaches are not practical in IoT environments, given the limited amount of resources, and can only provide fault tolerance at the data source level when there are multiple devices capturing the same input data. Similarly to our work, the upstream backup approach also relies on buffering. However, unlike our approach, upstream backups aim to handle failures at the receiver by buffering data at the sender. In our approach, we deal with failures at the data source which *does not have an upstream node*, i. e. is at the bottom of the QEP. We therefore need to buffer data at the source itself. Furthermore, while the upstream backup approach buffers data throughout query execution, our approach only buffers data reactively upon a disconnection. Finally, our approach also considers topological changes when forwarding the buffered data.

Mobile Computing and Networking. In the past decades, there has been a lot of research in the area of data management in mobile computing [Ba99]. In [BI94], the authors propose different caching strategies for mobile devices. They categorize devices into sleepers and workaholics based on the duration of their disconnection and show the impact of the disconnection duration on the effectiveness of the caching strategy. Wu et al. [WYC96], address the problem of selectively discarding caches in mobile devices that have been disconnected for a period of time. In contrast to the above work, we assume that the contents of the buffer do not become obsolete during the disconnection period, as we are focusing on transient disconnections. Another line of work, aims to provide network connection mobility. Persistent Connections [YS95] interpose a library between the application and the sockets API that provides the illusion of a single unbroken connection over successive physical connection instances. When a physical connection is lost, the sender stores data in a buffer. Once a new physical connection is established, any data buffered during disconnection are sent through it. Similarly, our approach uses buffering at the data source during a disconnection. However, instead of using an external library, we tightly integrate buffering inside the streaming engine. Furthermore, unlike Persistent Connections, our approach is holistic: in addition to buffering, we provide a query restart process that resumes query execution upon a re-connection.

Buffering for Fault Tolerance. Clearly, there are plenty of fault tolerance strategies in different domains that rely on buffering. Message logging and checkpointing are often used for fault tolerance in distributed systems [El02]. In [ZJ87], the sender stores messages in its local memory, which allows recovery from single failures, while [SBY88] builds upon that approach by selectively logging only data that cannot be otherwise reconstructed. Unlike our work, the above techniques do not support mobility of the sender. In addition, they follow a *pessimistic* approach that logs messages irrespective of failures, while we only buffer data upon a disconnection. DiscoTech [RGG12] is a toolkit for handling disconnections in groupware networks. It includes fault tolerance strategies that use event queues to store data in a centralized server during the disconnection of a receiver node. In contrast, we deal with failures at the source that sends the data, and store the data locally during the disconnection.

6 Conclusions & Future Work

This paper presents NebulaStream-MSS, an extension of NebulaStream that provides fault tolerance capabilities for mobile sources in IoT environments. NebulaStream-MSS focuses specifically on overcoming transient disconnections. At the core, the proposed solution is a data logging mechanism that allows mobile devices to continue processing data while they are in a disconnected state by temporarily storing the processed data in a circular buffer. That way, we exploit the processing capabilities of the mobile devices and transform the idle disconnection time into a productive period. Besides buffering data during transient disconnections, our fault tolerance strategy also includes a query restart process that ensures the consistent resumption of query execution, even when mobile devices move during the period of disconnection. Our restart process determines whether queries need to be redeployed, based on whether the device can still reach its downstream neighbours after the re-connection. In addition, we introduce forwarding queries, to address the case where the mobile device is assigned a subset of the operators previously assigned to it after a new deployment. These forwarding queries create new paths in the DAG of a query through which the data stored in the buffer can be forwarded further down the pipeline to avoid redundant processing. Using a custom benchmark based on real data [MZJ13], we show that NebulaStream-MSS reduces data loss by 63% over a disconnection period of 30 seconds and provides nearly constant query runtime with an increasing number of disconnections when no query redeployment is required.

In future work, we plan to further improve our buffering strategy. Since mobile devices have limited resources, minimizing the memory requirements is critical. For that purpose, we would like to investigate the use of more tailored replacement strategies that exploit application knowledge to determine the most relevant information. In addition, we plan to look into data compression and result sharing. Result sharing would allow us to only once store duplicate results produced by different queries. Moreover, we plan to further investigate the selection of an optimal buffer size based on the available resources of the device and the workload. Our evaluation also showed that the largest performance overhead stems from the redeployment of queries. To mitigate this overhead, we aim to perform dynamic and partial redeployment. That way, we can reconfigure the data flow without involving the coordinator thereby eliminating our system's bottleneck. Finally, we have proposed the use of a level of replication in our forwarding queries. To reduce network traffic, instead of broadcasting the data through all redundant paths, we could use an approach similar to Frontier's [OSP18] where data are sent through a single path chosen dynamically at runtime.

Acknowledgements
This work was supported by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref 01IS18037A).

Bibliography

[Ba99] Barbara, D.: Mobile computing and databases-a survey. IEEE Transactions on Knowledge and Data Engineering, 11(1):108–117, 1999.

[Ba08] Balazinska, Magdalena; Balakrishnan, Hari; Madden, Samuel R.; Stonebraker, Michael: Fault-Tolerance in the Borealis Distributed Stream Processing System. ACM Trans. Database Syst., 33(1), March 2008.

[BI94] Barbará, Daniel; Imieliński, Tomasz: Sleepers and Workaholics: Caching Strategies in Mobile Environments. In: International Conference on Management of Data SIGMOD. p. 1–12, 1994.

[CEH15] Carbone, Paris; Ewen, Stephan; Haridi, Seif: Apache Flink: Stream and Batch Processing in a Single Engine. In: IEEE Data Engineering Bulletin. volume 36, 2015.

[CS20] Chao, Mengyuan; Stoleru, Radu: A Resilient Mobile Stream Processing System for Dynamic Edge Networks. In: IEEE International Conference on Fog Computing (ICFC). 2020.

[El02] Elnozahy, E. N.; Alvisi, Lorenzo; Wang, Yi-Min; Johnson, David B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv., 34(3):375–408, 2002.

[FSL18] Fan, S.; Salonidis, T.; Lee, B.: Swing: Swarm Computing for Mobile Sensing. In: International Conference on Distributed Computing Systems (ICDCS). pp. 1107–1117, 2018.

[Ga20] Gavriilidis, Haralampos; Michalke, Adrian; Mons, Laura; Zeuch, Steffen; Markl, Volker: Scaling a Public Transport Monitoring System to Internet of Things Infrastructures. In: International Conference on Extending Database Technology (EDBT). pp. 627–630, 2020.

[GRP] Introduction to gRPC, `https://grpc.io/docs/what-is-grpc/introduction/` Last accessed 12/12/2020.

[GS] The Mobile Economy 2020, `https://www.gsma.com/mobileeconomy/wp-content/uploads/2020/03/GSMA_MobileEconomy2020_Global.pdf` Last accessed at 23/09/2020.

[Hi] ZeroMQ guide: Preface, `http://zguide.zeromq.org/page:preface` Last accessed 12/12/2020.

[Hw05] Hwang, J. .; Balazinska, M.; Rasin, A.; Cetintemel, U.; Stonebraker, M.; Zdonik, S.: High-availability algorithms for distributed stream processing. In: 2IEEE International Conference on Data Engineering (ICDE). pp. 779–790, 2005.

[Ma05] Madden, Samuel R.; Franklin, Michael J.; Hellerstein, Joseph M.; Hong, Wei: Tinydb: An acquisitional query processingsystem for sensor network. In: ACM Transactions on Database Systems (TODS). 2005.

[MRH14] Morales, Jefferson; Rosas, Erika; Hidalgo, Nicolas: Symbiosis: Sharing mobile resources for stream processing. In: IEEE Symposium on Computers and communications (ISCC). pp. 1–6, 2014.

[MZJ13] Mutschler, Christopher; Ziekow, Holger; Jerzak, Zbigniew: The DEBS 2013 Grand Challenge. In: International Conference on Distributed Event-Based Systems (DEBS). p. 289–294, 2013.

[Ni15] Ning, Qian; Chien-An; Stoleru, Radu; Chen, Congcong: Mobile Storm: Distributed Real-time Stream Processing for Mobile Clouds. In: IEEE International Conference on Cloud Networking (CloudNet). pp. 139–145, 2015.

[NSP17] Narkhede, Neha; Shapira, Gwen; Palino, Todd: Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly, 2017.

[OSP18] O'Keeffe, Dan; Salonidis, Theodoros; Pietzuch, Peter: Frontier: Resilient Edge Processing for the Internet of Things. In: PVLDB. volume 11, pp. 1178–1191, 2018.

[Qi13] Qian, Zhengping; He, Yong; Su, Chunzhi; Wu, Zhuojie; Zhu, Hongyu; Zhang, Taizhi; Zhou, Lidong; Yu, Yuan; Zhang, Zheng: TimeStream: Reliable Stream Computation in the Cloud. In: European Conference on Computer Systems (EuroSys). p. 1–14, 2013.

[RGG12] Roy, Banani; Graham, T.C. Nicholas; Gutwin, Carl: DiscoTech: a plug-in toolkit to improve handling of disconnection and reconnection in real-time groupware. In: ACM Conference on Computer Supported Cooperative Work. 2012.

[SBY88] Storm, Robert E.; Bacon, David F.; Yemini, Shaula A.: Volatile logging in n-fault-tolerant distributed systems. In: International Symposium on Fault-Tolerant Computing. 1988.

[Sh16] Shen, Zhitao; Kumaran, Vikram; Franklin, Michael J.; Krishnamurthy, Sailesh; Bhat, Amit; Kumar, Madhu; Lerche, Robert; Macpherson, Kim: CSA: Streaming Engine for Internet of Things. In: IEEE Data Engineering Bulletin. volume 38, 2016.

[SHB04] Shah, Mehul A.; Hellerstein, Joseph M.; Brewer, Eric: Highly Available, Fault-Tolerant, Parallel Dataflows. In: International Conference on Management of Data SIGMOD. p. 827–838, 2004.

[STO] Storm, `https://storm.apache.org/`.

[WP14] Wang, Huayong; Peh, Li-Shiuan: Mobistreams: A reliable distributed stream processing system for mobile devices. In: IEEE International Parallel and Distributed Processing Symposium. pp. 51–60, 2014.

[WYC96] Wu, Kun-Lung; Yu, Philip S.; Chen, Ming-Syan: Energy-Efficient Caching for Wireless Mobile Computing. In: International Conference on Data Engineering (ICDE). p. 336–343, 1996.

[YS95] Yongguang Zhang; Son Dao: A "persistent connection" model for mobile and distributed systems. In: International Conference on Computer Communications and Networks (ICCCN). pp. 300–307, 1995.

[Za16] Zaharia, Matei; Xin, Reynold S.; Patrick Wendell, Tathagata Das; Armbrust, Michael; Dave, Ankur; Meng, Xiangrui; Rosen, Josh; Venkataraman, Shivaram; Franklin, Michael J.; Ghodsi, Ali; Gonzalez, Joseph; Shenker, Scott; Stoica, Ion: Apache spark: a unified engine for big data processing. In: Communications of the ACM. 2016.

[Ze20a] Zeuch, Steffen; Chaudhary, Ankit; Monte, Bonaventura Del; Gavriilidis, Haralampos; Giouroukis, Dimitrios; Grulich, Philipp M.; Breß, Sebastian; Traub, Jonas; Markl, Volker: The NebulaStream Platform: Data and Application Management for the Internet of Things. In: Conference on Innovative Data Systems Research (CIDR). 2020.

[Ze20b] Zeuch, Steffen; Tzirita Zacharatou, Eleni; Zhang, Shuhao; Chatziliadis, Xenofon; Chaudhary, Ankit; Monte, Bonaventura Del; Giouroukis, Dimitrios; Grulich, Philipp M.; Ziehn, Ariane; Markl, Volker: NebulaStream: Complex Analytics Beyond the Cloud. Open J. Internet Things, 6(1):66–81, 2020.

[ZJ87] Zwaenepoel, Willy; Johnson, D.B.: Sender-based message logging. In: International Symposium on Fault-Tolerant Computing. 1987.