Open Access

Open Journal of Internet of Things (OJIOT)
Volume X, Issue X, 20XX

http://www.ronpub.com/ojiot
ISSN 2364-7108

Streaming Data through the IoT via Actor-Based Semantic Routing Trees

Dimitrios Giouroukis[1] Johannes Jestram[1,†] Steffen Zeuch[1,2] Volker Markl[1,2]

[1]Technische Universität Berlin, [2]DFKI GmbH
[1]{firstname.lastname}@tu-berlin.de [†]{firstname.lastname}@alumni.tu-berlin.de [2]{firstname.lastname}@dfki.de

Abstract

The Internet of Things (IoT) enables the usage of resources at the edge of the network for various data management tasks that are traditionally executed in the cloud. However, the heterogeneity of devices and communication methods in a multi-tiered IoT environment (cloud/fog/edge) exacerbates the problem of deciding which nodes to use for processing and how to route data. In addition, both decisions cannot be made only statically for the entire lifetime of an application, as an IoT environment is highly dynamic and nodes in the same topology can be both stationary and mobile as well as reliable and volatile. As a result of these different characteristics, an IoT data management system that spans across all tiers of an IoT network cannot meet the same availability assumptions for all its nodes. To address the problem of choosing ad-hoc which nodes to use and include in a processing workload, we propose a networking component that uses a-priori as well as ad-hoc routing information from the network. Our approach, called Rime, relies on keeping track of nodes at the gateway level and exchanging routing information with other nodes in the network. By tracking nodes while the topology evolves in a geo-distributed manner, we enable efficient communication even in the case of frequent node failures. Our evaluation shows that Rime keeps in check communication costs and message transmissions by reducing unnecessary message exchange by up to 82.65%.

Type of Paper and Keywords

Regular research paper: *Edge Networking, Topology Management*

1 Introduction

The Internet of Things (IoT) with billions of sensors brought forward the need to redesign existing data management systems. The main game changer is the massive data production at the edge, in particular from mobile devices that are located *outside* the cloud [23]. Gartner predicts that cloud providers will manage 20% of all edge and mobile computing platforms by 2023, from the current 1% [13]. Recent extensions to the cloud computing paradigm, such as Fog-Cloud [4], Edge Computing [19], Sensor-Clouds [22], and Cloudlets [18] consider mobile nodes as an integral part of data management systems. Fundamentally, these extensions propose the unification of centralized and immobile nodes in the cloud together with distributed and mobile wireless sensor networks (WSNs). At the same time, intermediate gateway nodes that connect the edge to the cloud are also capable of processing information [9, 26]. However, current approaches assume uninterrupted connectivity to mobile devices, which is seldom the case in today's dynamic and volatile environments such as smart cities [4]. Thus, systems need to incorporate the notion of mobility for data sources and infrastructures into their core design to really enable future IoT applications.

IoT data management concerns the management and processing of data streams, potentially in combination with data at rest, in a heterogeneous distributed environment of cloud and mobile edge devices. One of the main challenges

for IoT data management systems is to timely answer user queries that span across cloud and mobile WSN nodes. To this end, a system requires real-time knowledge of the underlying topology to keep track of volatile nodes in a scalable manner. In particular, maintaining a topology that unifies mobile as well as stationary nodes is challenging as nodes connect or disconnect from the system *anytime* and *anywhere* in the network. At the same time, mobile nodes induce *fluctuations in bandwidth*, *unstable latency*, and *increased communication overhead* [11]. The inherent volatility in the topology complicates the deployment and routing decisions within an IoT data management system. As a result, one major challenge for an IoT data management system is to determine which nodes and paths provide answers to queries and which of these options induce minimal communication overhead.

Motivating Example Figure 1 shows a simplified view of an IoT installation in a smart city, containing a sink ▼, gateways ◆, and edge devices with sensors ●. The edge devices are attached to mobile phones to keep track of their environment. Multiple gateways, installed at the sides of streets, offer connectivity to the edge devices with one gateway per street. The sink is located at the offices of the data analysis department of the municipality, using their own municipal cloud. The dotted ring indicates the boundary between the edge level (before the gateways) and the municipal infrastructure (gateways and cloud). The ring also indicates the difference in communication methods between nodes. Gateways request information from an edge device, hence they utilize *pull-based* approaches. After gateways acquire values from the edge, they forward it to the municipal cloud by utilizing a *push-based* communication scheme.

Suppose a data analyst wants to send a push notification to edge devices on three different streets. If the device user accepts the notification, the edge device sends sensor data in real time to the cloud. At the same time, the analyst wants to avoid notifying edge devices with low battery. The data analyst thus would issue the following query: `SELECT * FROM edge_nodes WHERE battery_level >= battery_threshold AND street_name IN (s_1,s_2,s_3).` To answer the query, an IoT data management system has to ask all edge devices in `s_1`, `s_2`, and `s_3` about their current battery level, using the closest gateway to the edge device. In Figure 1, *red* routes indicate edge devices that do not have enough battery. Even though these edge devices should not report back, some systems would still ask them for reporting, even in subsequent queries [1, 21]. In contrast, the *green* route contains a edge device with enough battery and thus information has to move from the yellow edge device to the sink in ▼. In this scenario, a system needs to communicate and keep track of the yellow edge device as it moves potentially to streets outside of the original query. At the same time, a system has to reduce communication costs with

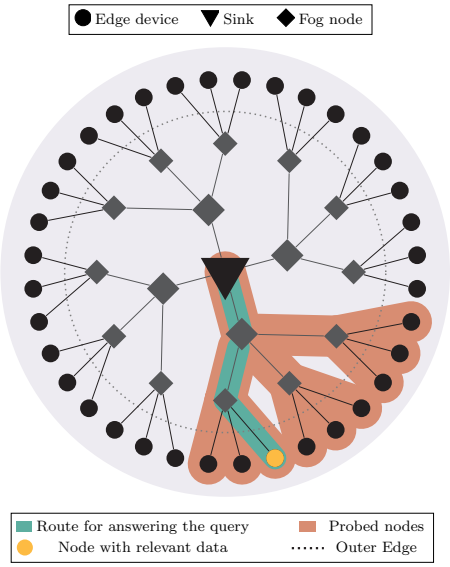


Figure 1: An IoT topology showing edge devices participating in a query. *Red* indicates edge devices that are not relevant but are still probed for values. *Green* indicates vehicles that contain data relevant to the query and have to report their values back.

edge devices that do not have enough battery. As a result, the system has to unify a pull-based approach (edge level) with a push-based approach (gateways and upstream) in order to answer queries in different physical partitions of the network.

Contributions To enable query processing within a mobile environment, we address the challenges of heterogeneous communication schemes with Rime. Our solution is a geo-distributed approach for propagating routing information efficiently in an IoT environment. Rime organizes a topology into parent nodes with multiple children nodes, thus forming a hierarchical tree structure over the topology. Rime bridges the gap between centralized and mobile approaches by extending the concept of Semantic Routing Trees (SRTs) [12] from WSNs to the IoT. In this paper, we make the following contributions:

- We propose Rime, our approach for efficiently building and maintaining an overview of an IoT topology.
- We introduce a novel parent node selection scheme that introduces new nodes to the network with minimal overhead.
- We redesign SRTs to use instant communication between parent nodes within an IoT topology.
- We evaluate Rime in an emulated IoT network and show that it reduces communication overhead for queries with high index selectivity significantly.

We organize the remainder of the paper as follows. In Section 2, we provide the necessary background for our work. Then, we propose the core design decisions of Rime in Section 3 and evaluate them in Section 4. After that, we discuss related work in Section 5 and provide an outlook into future work together with our conclusion in Section 6.

2 BACKGROUND

In this section, we detail the foundational concepts behind our approach. We introduce the basic networking models in Section 2.1. Then, we discuss the concept of SRTs in detail in Section 2.2 as well as the common parent selection scheme in Section 2.3. Finally, we introduce the actor networking model that Rime employs in Section 2.4.

2.1 Networking Model

Edge deployments combine two main networking models: *1) flood-like* [10] and *2) index-based* [17] networks. Flood-like networks are highly distributed and peer-to-peer (P2P) or P2P-like and assume that nodes are capable of moving. In contrast, index-based networks use meta-information about the physical topology for making routing decisions on the application level.

Nodes in a flood-like network broadcast (flood) messages across the network in order to propagate information and processing. To this end, flood-like networks usually employ a multi-hop design for messages [1]. As a result, nodes can not rely on the topology to be the same during the entire broadcast duration. One example of a flood-like network is a WSN, where processing and communication are fully decentralized. Sensor nodes gather and process data in-situ to reduce transmission costs and later forward results to a sink through multi-hop communication.

Index-based approaches utilize centralized indexes for book-keeping of metadata, which supports decision making. Applications can access the index and make routing decisions timely, as the index is in main memory. Depending on the application, the index can be centralized or decentralized [14]. The nodes can utilize the index for making any decision and can assume that the topology is reliable. An example of such an application is an *overlay* network of data-centers, where a data structure keeps track of link-qualities between nodes and later categorizes them based on that attribute. An external application (e.g.: a stream processor) can later utilize that meta-information for performing operator placement.

2.2 Semantic Routing Trees

TinyDB [12] introduces the concept of *SRTs* to bridge the gap between a flood-like and index-based approach. SRTs are data structures that stem from work on WSNs and they store information for each node, e.g., the value range that the node produces. In an SRT, nodes fall into the categories of parents, children, or both. A parent keeps track of its children and their attribute values. The attribute values determine what data is produced by the node, e.g.: CO2 emission readings or temperature data between 2 and 8 degree centigrade. In contrast, a child node produces values from sensed phenomena but might also acts as an intermediate parent, for multi-hop communication. The main task of a parent is to decide if its children need to participate in a query and if required, it will route any necessary information to them.

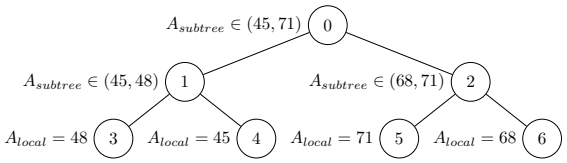


Figure 2: Exemplary topology with an SRT over Attribute A. Parents only know the attribute range within their subtree.

In Figure 2, an SRT acts as a distributed index over an attribute *A*. N_0 is the root of the tree with an index over *A* that contains values in range $(45{,}71)$. N_0 is the parent of N_1 and N_2, with ranges of $(45{,}48)$ and $(68{,}71)$. The range of N_1 stems from the values of N_3 (48) and N_4 (45). If a query requires values in the range $(45{,}47)$, then only the subtree of N_1 replies to the query and the other whole subtree of N_2 is ignored. This process allows SRTs to avoid unnecessary communication because a parent node knows the range of all values of its children. The parent node is also responsible for disseminating only relevant queries to its children. The main implication is that the choice of a parent strongly affects future maintenance costs and query latency [12]; thus, parent selection is an important performance factor.

2.3 Parent Selection

When a new node enters the network, choosing a parent from a set of candidates or choosing to become a parent impacts query performance as well as future maintenance overhead. Ideally, a parent has children that produce values with small variation, such that the range of attribute *A* is small, e.g., temperature values only differ by several degree. A small attribute range affects the number of children that a parent will adopt as well as the number of topology updates. In the presence of multiple candidate children nodes, parents prefer children that produce *similar* values to their attribute range.

In order to stay updated with minimal performance impact, the SRT uses a *neighbor tracking policy*. This policy is responsible for parent assignment and determines how to propagate updates in the network. The original multi-hop, broadcast-based SRT design does not consider information exchange across nodes that are multiple hops away from each other. This communication limitation stems

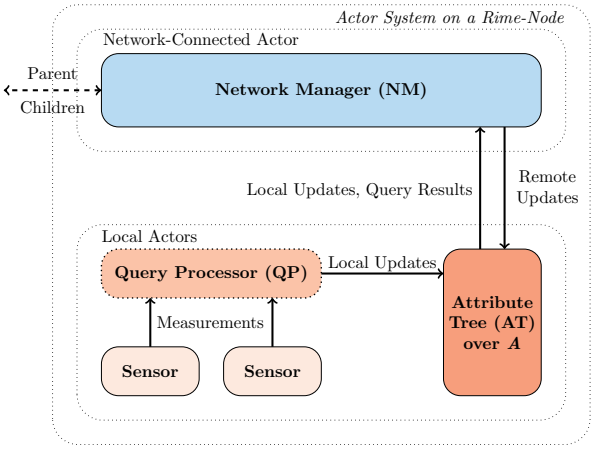


Figure 3: Overview of the architecture of Rime and the most important message exchanges between components

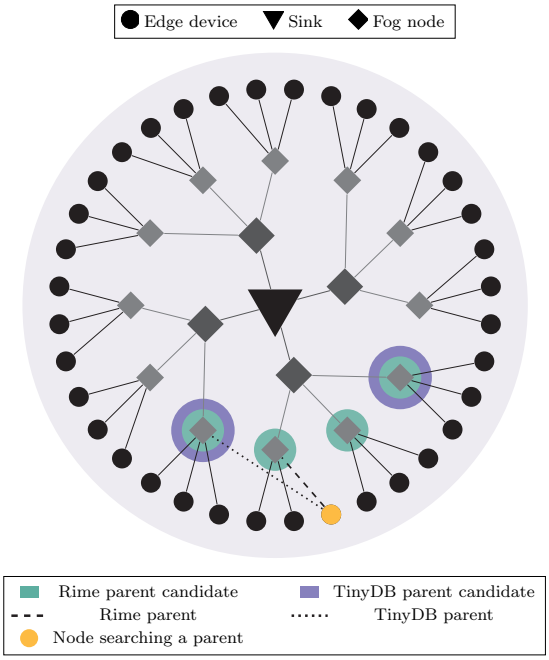


Figure 4: Distance between nodes corresponds to their physical distance. In TinyDB, only nodes in spatial proximity are parent candidates. Rime offers a broader parent pool than TinyDB with its new parent selection.

from the energy preserving design of WSNs, as antenna communication is the main culprit of energy expenditure. Thus, parents in WSNs only have partial knowledge of the network, which exacerbates the effects of a bad assignment and increases communication overhead for SRT updates. In the IoT, we can augment the tracking policy with different networking schemes in order to propagate SRT updates further than just the antenna range of a node.

2.4 Actor Networking

Rime employs the actor model for its networking stack. Here, the smallest unit of computation is an *actor*. Actors receive messages from other actors and respond to them by making local decisions, spawning other actors, or sending more messages. Basically, actors are software entities that tie together data- and control-flow as messages. Actors encapsulate state, behavior, and communication, translate state to messages and send them to other actor inboxes [7]. Actors enable sophisticated fault tolerance strategies through their messaging system [6]. Recent approaches introduce *virtual* actors, which are location transparent. As a result, the system does not know *if* and *where* a specific actor is currently materialized [3]. We do not use the virtual actor model in this paper because location-awareness is an important characteristic for an IoT data management system.

3 RIME

In this section we introduce Rime, a networking component that maintains highly volatile topologies efficiently. Our solution reduces communication costs by exchanging topology information over different gateway nodes and heterogeneous communication schemes. Rime extends the concept of SRTs by introducing an improved parent selection algorithm and a new node tracking policy. At the same time, Rime allows the exchange of topology metadata over different physical areas. The advantages of Rime are three-fold. First, by introducing a more efficient parent selection process that reduces the communication and integration overhead, Rime integrates new nodes faster into query processing. Second, to further reduce the amount of exchanged messages, Rime employs a novel tracking policy between nodes. The main goal of the tracking policy is to allow nodes to build an index from their "known" nodes. This information is the basis for building and maintaining the extended SRT locally. Finally, Rime propagates local topology updates within the network, e.g., in the case of node movement. Thus, nodes exchange topology updates with other nodes but do not broadcast them, which reduces the number of messages sent for operations like relocations. We will present the overall architecture of Rime in Section 3.1, our parent selection algorithm and its details in Section 3.2, and the tracking policy in Section 3.3.

3.1 Architecture

Figure 3 outlines the architecture of Rime. The main goal of Rime is to keep up with a volatile topology and to minimize communication overhead. Our implementation utilizes the C++ Actor Framework (CAF) [6]. Rime encapsulates state,

e.g., location updates, as part of the actor and models updates as actor messages. Each Rime node within the network is a distinct actor system, which consists of several local actors and a network-connected actor for communication. Each node contains a *Query Processor (QP)*, multiple sensors, and an *Attribute Tree (AT)*. A QP is responsible for gathering readings from sensors, perform processing on the readings, and finally update its local AT. The AT holds necessary state, i.e., the current and backup parent, handles to all its children, the local attribute value, and the range of values from the children. In sum, the AT is a superset to the SRT that holds more than just an attribute range. This is to keep the actor implementation simple and spawn as little actors as possible. The *Network Manager (NM)* serves as the networking gateway that listens for updates and propagates them to other nodes. During runtime, each AT propagates local updates through the NM to other ATs in other nodes. The NM is a stateful actor that is responsible for remote communication and maintaining a key-value store of attributes and handles to the AT. Furthermore, the NM can react to failures by monitoring its children and parents.

Algorithm 1: Parent selection at the grandparent of the requesting node

```
   Input: find-better-parent request
   Data: curr_dist = distance to current parent
         att = requesting node tree attribute
   Output: new parent for requester
1  begin
2      best_dist = INTEGER_MAX;
3      best_idx = -1;
4      for child : self.allChildrenNodes do
5          dist = calcEuclideanDist(self.att, child.att);
6          if dist < best_dist then
7              best_dist = dist;
8              best_idx = children.current_idx;
9      request random_sibling from self.parent;
10     receive best_rand, best_rand_dist;
11     if best_rand_dist < best_dist then
12         return best_rand, best_rand_dist;
13     else
14         return children[best_idx], best_dist;
```

3.2 Parent Selection in Rime

Parent selection is a crucial step in SRTs as it impacts the attribute clustering in a specific sub-tree [12]. By clustering, we refer to the value range variance of the attributes stored in an AT. An AT helps pinpoint nodes that are relevant for a query and avoid unnecessary communication with nodes that do not contain relevant values. Thus, a small value range in an AT is the main goal when choosing a parent, as it reduces the amount of children when reporting *which* children are relevant to a query.

Rime uses an *enhanced closest-parent selection policy*, where the best parent candidate has the closest attribute value, e.g. location, to the requesting node. This is an extension to the *closest-parent selection policy* of TinyDB, where updates do not propagate between nodes. Essentially Rime exploits the fast network backhaul between gateways to share SRT updates between nodes. TinyDB does not offer this possibility due to the cost of constant broadcast communications, which decreases the life cycle of a battery-powered sensor node. Figure 4 shows the effects of the different parent selection schemes between Rime and the original SRT. The golden node is searching for a new parent as it just enters the network. Rime provides a larger pool of parents (green) compared to TinyDB (light blue) as it does not rely only on nodes that are in physical proximity. Rime suggests two more potential parents due to the exchange of routing information between nodes that are not in physical proximity. This allows nodes to continuously update their SRTs without having to perform costly multi-hop broadcast transmissions to all neighboring nodes.

Candidate Parents An *ideal* parent candidate has a narrow range of tree-attribute values and is closer to the sink than its children. For example, in Figure 2, node N_3 reports temperature value around (48) degrees. There are two candidate parents N_1 with value range $(45, 48)$ and N_2 with $(68, 71)$. In this case, N_1 is a better candidate as it has a narrower value range and its values are closer to N_3. Rime utilizes the parent selection algorithm in Algorithm 1 to determine the best parent. Algorithm 1 starts on a per-request basis from new nodes. The input data are: the distance to an existing parent and the *tree attribute* (*att*) of the new node. In Lines `2-8`, the existing node checks all its children and calculates the *euclidean* distance between the tree attributes of the new node and the current child. To this end, we assign the child with the smallest distance as a starting point. Then, in Line `9`, the existing node requests a *random* sibling from its own parent. The random sibling reply in Line `10` contains the best candidate child and its distance from the new node to the original random sibling of the existing node. The existing node returns the best child, after comparing its own best child and the best child from the sibling node, in Lines `11-14`. This way, Rime returns a larger and better pool of parent candidates for the new node.

3.3 Tracking Policy

Because nodes keep track of several other nodes, the question of *which* remote nodes to track arises. Tracking all nodes in the network is infeasible due to scalability reasons, e.g., high message overhead. The main goal of the tracking policy is to decrease the maintenance overhead of ATs by

reducing the number of tracked nodes. In order to build an AT within Rime, a node tracks the following: First, all nodes require information about their children, i.e., subtree-ranges, and thus track all children. Second, a node must know the tree-attribute range of a parent in order to determine how similar it is to its own. Third, there must be at least one backup node that is required in case of a parent failure. Rime picks the second best parent candidate from the selection process as a backup parent. With the above information, a node is able to perform routing on its own, without the immediate need to share knowledge to/from others.

Even though Rime allows for routing decisions using local information, it also enables specific nodes to gain knowledge about other nodes in two ways. First, a child that selects a new parent sends additional data to the parent, e.g., its IP-address, port, and information about its sensors. The new parent node tracks the child node by adding its information into its own state. Second, specific nodes exchange information about other nodes within the network. Exchange only involves specific children, parents, and backup parents and does not work in a broadcast fashion. Nodes maintain local state and data about several other nodes as they are able to share it with others on request. This is possible due to the actor-based message exchange of Rime through heterogeneous networks. Given that gateways and intermediate nodes are able to immediately communicate, they do not need to broadcast information to everyone. This is in contrast to TinyDB, where nodes broadcast all messages and only nodes within antenna range can receive messages. TinyDB is designed for antenna transmissions, where messages always assume a multi-hop scheme and it is not certain if they will reach their target. This design decision makes sense for a WSN but is not enough for a system that is expected to run on top of volatile, heterogeneous networks. Many core operations in TinyDB, e.g.: message stealing over antennas, are not feasible in different network configurations. This makes the benefits of the original broadcast-only system unclear in a modern IoT deployment.

Rime extends the principles behind SRTs in order to remove the inherent limitations that come with broadcast transmissions. To this end, nodes in Rime are able to gain topology information to help them to form a better AT and reduce the number of exchanged messages. At the same time, the parent selection and the tracking policy reduce communication while propagating updates of the topology to other locations. These locations are not reachable using only antenna transmission, contrary to the original SRT design. The design of Rime is made from the ground-up to exploit the communication capabilities of IoT infrastructure and bridge the gap between WSNs and the cloud.

4 EVALUATION

In this section, we evaluate the performance of Rime. First, we describe our test environment in Section 4.1. Then, we introduce our emulation setup in Section 4.2. After that, we evaluate different aspects of Rime, i.e., failure recovery in Section 4.3, efficient query dissemination in Section 4.4, scalability and maintenance overhead in Section 4.5, and the support of geo-spatial movement of nodes in Section 4.6. For the sake of reproducibility, we release Rime as well as experiment configurations in a public repository. [1]

4.1 Test Environment

Our test environment consists of emulated nodes deployed on a single server. All benchmarks were performed on a server with Ubuntu 18.04 on an Intel Xeon E5620 (2.4GHz) CPU with 47GiB memory. The hardware resources of the server allows for emulating topologies with sizes up to 73 nodes. Table 1 lists all the topologies used throughout our experiments. We found that topologies larger than T_{73} degrade performance on the host due to hardware limitations. We use a tree-like topology with configurable `height` and `degree` across all experiments. For node emulation, we use Containernet [16], a network emulator that is an extension of Mininet [8] which is capable of running Docker [2] containers. This setup allows for flexible deployment of nodes or experiments and supports emulating node failures in a reproducible manner. We utilize emulation rather than network simulators since simulators use models to represent the current environment, hence they abstract details for the sake of accuracy. A network emulator on the other hand leads to easier testing on a diverse set of hardware and allows for faster and prototyping [20].

Assumptions In order to evaluate our approach, we make some simplifications about our running environment. In all our tests we assume that the root node does not fail. One possible solution to root node failures would be a consensus mechanism that can be utilized in order to elect a new root between candidates. However, we argue that a root could be hosted within a cloud and thus exclude this aspect from the following considerations.

Baseline To the best of our knowledge, there exists no open-sourced implementation of SRTs that is suitable as a baseline for our evaluation. The closest is TinyDB [12] that was part of TinyOS in a single release that is not available anymore.[3] TinyOS and its `nesc` compiler are not further developed and have transitioned to getting volunteer updates.[4] Therefore, we compare Rime to a baseline-system that connects nodes using an immutable

[1] https://github.com/jo-jstrm/rime-data-streaming-iot
[2] https://docker.com
[3] http://telegraph.cs.berkeley.edu/tinydb/software.html
[4] https://github.com/tinyos/tinyos-main

Table 1: Experiment configurations used for evaluation.

ID	# of Nodes	Degree	Height
T_7	7	2	2
T_{13}	13	3	2
T_{15}	15	2	3
T_{21}	21	4	2
T_{40}	40	3	3
T_{73}	73	8	2

routing tree. We consider this to be a suitable baseline with low maintenance-overhead at the cost of no failure recovery and update propagation. To implement the baseline, we compile Rime without any SRT-features, i.e., no parent-selection algorithm, backup parent, and state-tracking of remote nodes. This ensures that Rime and the baseline are as similar as possible. We elaborate further on our baseline assumptions and practicalities in Section 4.2.

Running Example Figure 2 shows the running example topology that we use for our experiments. The tree resulting from this topology has a $height$ of three, which is the highest number of hops on the path from root to leaf. Furthermore, it has a $degree$ of two, which is the number of children of each node, including the root. The node IDs are allocated using breadth-first search. For our experiments in Section 4.5, we extend this basic topology by increasing height and degree, as well as the number of nodes. We show the resulting deployment configurations in Table 1. The geo-spatial location for all nodes is randomly assigned within the interval $\{latitude, longitude\} \in (0.0, 100.0)$ meters of a uniform distribution created with an MT19937 random number generator.[5] In Section 4.3 and Section 4.4 experiments start after location assignment and parent selection are complete. This is necessary to let Rime propagate updates and create an SRT. For all experiments, we set the sampling rate of each sensor at every node to two tuples per second, which is common in WSNs [12, 2].

4.2 Exploration of Emulation Boundaries

As a first step for our evaluation, we test the maximum throughput and average latency of our setup with the baseline system over a static topology. This provides a practical line rate when the system only performs routing of tuples from the edge to the cloud nodes, without applying processing. Edge nodes do not move or change location and the keep producing two tuples per second (t/s) uninterrupted. We observe the average combined incoming throughput at the sink of the query is 146 t/s. The average latency of a single packet traveling from an edge node to the base station is 96.5 milliseconds (ms). This is in line with the randomized latency that we insert in the topology through the emulator, where we use a random latency range of [60-150]ms for the edge nodes and [10-30]ms for parents to base-station.

[5] https://www.cplusplus.com/reference/random/mt19937/

Table 2: Baseline messaging-overhead compared Rime for different selectivities. Rime significantly reduces the number of messages needed to get a query to all applying nodes.

Selectivity	Baseline Overhead	Rime Overhead	Rime Savings
10%	87.67%	5.02%	82.65%
25%	70.32%	1.83%	68.49%
80%	24.20%	2.74%	21.46%

In a second experiment, we evaluate the cost of node failure due to movement with the baseline. Recall that the baseline has Rime features turned off, thus there is no way for a disconnected node to recover. The impact of node movement is directly proportional to the size of the sub-tree of the failing nodes. As an example, on a tree with $degree=8$ and $height = 2$, if a node at $height = 1$ fails, then the whole sub-tree is disconnected from the network; thus the network loses $\approx \frac{1}{8}$ of the nodes. We provide details about the cost of recovering from failure using Rime in Section 4.3.

4.3 Failure Recovery

In this experiment, we explore how Rime performs recovery from failure, even if the node is a parent. Failure of a parent node disconnects all its children from the rest of the tree. Therefore, no sensor readings from the affected subtree can reach the sink. Rime offers a coping mechanism for parent-failure through backup parents. The backup parent is the second best candidate from the parent selection process.

To quantify the ability of Rime to recover from failures, we measure the *recovery latency*. We define the recovery latency as the time between the failure of one or several nodes and the point in time when regular operation resumes, i.e., the successful registration of all children of the failed node to their backup parents. We use the same base topology T_7 as the running example in Section 4.1 and we measure the recovery latencies of 10 runs when either node 1 or node 2 fails. In each run, we let the parent with most children fail, such that a failure has the highest possible impact on Rime. Throughout the experiment, the recovery latency varies between $46ms$ and $366ms$, with a median of $201.5ms$. We account the variations of the recovery latency to *1)* the varying number of queued messages at the receiver and *2)* scheduling on the emulating host. Recovery from node failure is not possible in the baseline-system, whereas Rime timely reconnects the nodes. Thereby, Rime resumes routing with insignificant tuple loss after a failure.

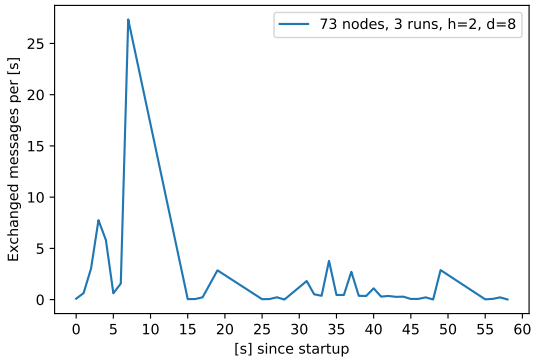


Figure 5: Message transmissions per node of T_{73}. The spikes starting after 31s shows the number of transmissions during a stepwise tree-attribute change.

4.4 Efficient Query Dissemination

In this experiment, we quantify how efficiently Rime filters out unnecessary nodes for a query. A key benefit of using SRTs in Rime is its ability to discard whole subtrees of a query if the nodes in a subtree do not contain relevant values. To evaluate the benefit of this characteristic, we measure the overhead of Rime as well as the baseline in Table 2. The overhead is defined as the ratio of the number of nodes that received the query and the number of nodes that have relevant data for the query. Our query selects values from all location sensors installed at the nodes, inside a 2D area. We use topology T_{40} and vary query-selectivitiy, i.e., number of nodes participating in the query, between 10%, 25%, and 80%. Table 2 presents the average overhead for each selectivity. As shown, Rime reduces the average overhead by up to 82.65%. This means that Rime performs better for queries with a lower selectivity. The main reason is that Rime is able to choose only relevant nodes for a query, which is emphasized in queries with low selectivities. The results indicate that the overhead of Rime is minimal compared to the baseline.

4.5 Scalability and Maintenance Overhead

Building and maintenance of SRTs require message exchange between nodes, which is the main cost of building an SRT. To determine the required message transmissions, we measure the number of exchanged messages per second in Rime during the first minute after start-up. To determine the scalability of Rime, we use data from all topologies detailed in Table 1. We choose T_{73} as an example for showing the behavior of Rime *during* and *after* startup. We use the largest topology to fully show the effects of scalability in Rime. Figure 5 shows the average number of exchanged messages per second over three runs using

Table 3: Message statistics for all topologies, taken from the first 20s after startup.

ID	Total Messages	Messages per node	Messages per second per node
T_7	175	25.12	1.26
T_{13}	454	34.90	1.74
T_{15}	563	37.54	1.88
T_{21}	841	40.04	2.00
T_{40}	2292	57.29	2.86
T_{73}	3721	50.97	2.55

T_{73}. The communication peak during the first $10s$ stems from the initial data exchange and the requests from children for parent selection. The reoccurring local maximum of message transmissions at $20s$, $35s$, and $50s$ originate from the regular collection of the current tree topology. Furthermore, we discuss the peaks at $31s$, $34s$, and $37s$ in Section 4.6, as they are related to node movement.

Across all topologies, we identify spikes of message transmissions until $15s$ after startup. Figure 6 shows the sum of exchanged messages *per node* for all topologies during the first $20s$, such that we account for all startup spikes. This way we account for the overhead of extra messages needed to build and update the SRT in the same time-frame. We observe that the messaging overhead increases with topology size. As shown in Table 3, topology T_{73} induces the overall maximum number of messages. However, topology T_{73} induces less messages per node than T_{40}, even though T_{73} contains more nodes. This is due to the larger height of T_{40}, which leads to a larger pool of candidate parents. Thus, T_{40} induces more parent selection messages. We observe that the height of the tree is another important factor to message overhead, which can dominate in topologies with complex structure. Table 3 shows that the average increase in transmissions per node is near to linear, under the same height. Furthermore, the number of transmissions stays low, with periodical spikes during topology collection. Comparing the messaging overhead of 2.86 messages per node per second with T_{40} with the throughput of modern SPEs, which is magnitudes higher [24], we find the overhead negligible.

4.6 Geo-spatial Movement of Nodes

In this experiment, we measure the number of exchanged messages in a dynamic environment, where nodes change their positions. In this experiment we move a node step-wise, where each step is an increase of latitude and longitude by three meters within the space defined in the running example in Section 4.1. Rime needs to account for movement in order to maintain efficient query dissemination. Movement might change which parent is the closest one for a node. Therefore, each node initializes parent selection after its movement has ended. The overhead induced by parent

selection and metadata exchange for topology updates immediately affects the scalability of Rime. Figure 5 shows the number of message transmissions per node per second, during runtime. The spike at $31s$ marks the beginning of a continuous movement, where five random children nodes move three times during nine seconds, ending at $38s$. The last spike at $40s$ is the parent selection after movement stops. We start movement after $30s$ in order to give Rime enough time to create and populate the tree. The creation process induces a lot of concurrent updates and does not reflect the state of the network during regular runtime. If no movement occurs, Rime exchanges 233.3 messages (3.64 per node) on average with the same topology during the same period. During the interval $(31s, 41s)$, where nodes are moving, 1582.3 messages (24.7 per node) are transmitted on average.

In order to further quantify the impact of movement within Rime, we run the same experiment with two different configurations over the topology T_{73}. The configurations differ in the number of moving nodes. In the first configuration, we let one node move, where in the second one we increase this number to 10%, i.e., seven nodes. During the movement period, the nodes in the first configuration exchange on average 0.37 messages per node per second. For the second configuration, this number rises to 2.55. Our results show that the number of exchanged messages per node per second scales linearly with the number of moving nodes.

Overall, Rime drastically increases the efficiency of query dissemination compared to the baseline, as shown in Section 4.4. Rime can cope with failures in the topology, something that the baseline is not capable of, as well as recover timely. With the addition of the parent selection algorithm, Rime keeps in check the overhead of maintaining an SRT and allows it to scale up to the number of updates. Finally, Rime exploits edge resources and multi-scheme communication between physically distant nodes in order to overcome the original limitations in the radio-based transmission design of SRTs.

5 Related Work

Rime combines ideas from multiple subfields. This section covers the related work, as well as the differentiating factors, towards envisioning and implementing Rime.

Wireless Sensor Networks TinyDB [12] coined the term of Acquisitional Query Processing (ACQP) for WSNs. It is one of the earliest attempts to utilize the inherent ability of sensors of controlling *where*, *when*, and *how frequently* to physically acquire values as part of TinyOS. In that context, TinyDB uses SRTs to address the question of *where* to sample data and discard areas that are not relevant to the active query. In Rime, we extend SRTs by allowing the exchange and update of routing trees between nodes in different physical locations.

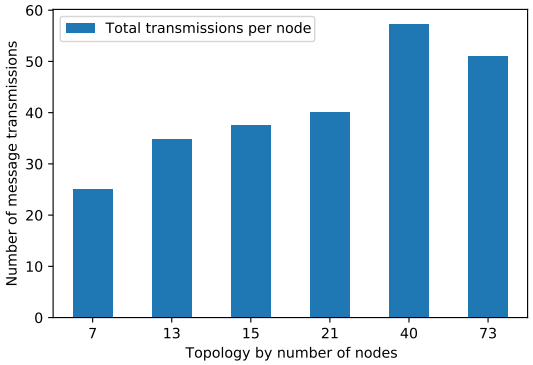


Figure 6: Average number of message transmissions per node during the first 20s of deployment.

In-Network Processing and Protocols The sensor-cloud, an amalgamation of heterogeneous resources, utilizes different communication models depending on the location. On one hand, edge nodes tend to utilize a more relaxed approach to connectivity, in order to handle movement and transitivity. On the other hand, edge gateways and cloud nodes assume robust interconnects. Existing work on WSNs lead to the creation of power-sensitive protocols. Trickle [10] aids synchronization of state in a distributed WSN. It performs a "polite gossip" between nodes. The protocol aims to quickly detect and resolve an inconsistent state within the network while performing as few message exchanges as possible. Trickle offers a trade-off between quick adaption to changed state and maintaining a low messaging- and processing-load. However, the protocol assumes only a multi-hop communication style, which is not the case in Rime. The protocol is not fit for an Edge deployment as it cannot exploit the ability to target different physical locations from a gateway node. Rime accounts for non-broadcast communication in its core design. This way a lot of multi-hop communication is reduced and nodes can have better routing information available anytime.

Mobility-Aware Stream Processing Mobility-Aware Stream Processing refers to Stream Processing Engines (SPEs) that process tasks on mobile nodes first. This is achieved by distributing workload among nodes or providing a unified view on the available resources. Our work is similar to systems that focus on maximizing routing and path diversity. Frontier [15] is a stream processing engine for the IoT. Frontier utilizes a modified back-pressure routing algorithm for load balancing between nodes. Frontier deploys query operators across multiple nodes, thus replicating the dataflow graph. This enables parallel processing of data by sending batches of data to replicated operators on distinct nodes. This approach also helps in case of node failures, because the replication of operators

enables re-routing. However, if a query path in Frontier becomes unavailable due to a node failure, the same path is not available for the system to use, even if it recovers. Therefore, Frontier addresses the challenge of failing nodes in the IoT only partially compared to the always-updating approach of Rime. In Rime, information propagate from the nodes themselves, which are also responsible for updating their list of connections. By propagating this list to other nodes, based on the tracking policy, Rime ensures the path will be available again after recovery.

Similarly to Frontier, R-MStorm [5] utilizes the notion of path diversity for data flowing from the sources in the edge to the sinks in the cloud. R-MStorm follows a weighted-link strategy, where it gives higher weights to links that have been known to be in "good" quality over time. R-MStorm does not provide any guarantees at the level of the source, while Rime allows for reconnection of nodes regardless of their type.

NebulaStream [23, 25] is an SPE whose goal is the integration of the heterogeneous hardware in an IoT deployment under the same runtime. NebulaStream exploits any hardware capabilities in order to further optimize queries and perform runtime re-optimizations. Rime can enhance the networking stack of NebulaStream, as NebulaStream does not contain the notion of SRTs. By doing so, Rime can make NebulaStream more robust under heterogeneous networking schemes and strengthen its mobile processing capabilities even under frequent failures.

6 Conclusion and Future Work

In this paper, we proposed Rime, a geo-distributed approach to propagating routing information efficiently in an IoT environment with heterogeneous communication schemes. Rime copes with a volatile topology while also reducing communication costs. Our work extends concepts from distributed sensor networks to in-application routing while accounting for the complex needs of an IoT environment. We show that Rime reduces message overhead for queries by up to 82.65% by controlling the parent selection. Rime enables fast updates over a dynamic topology through its efficient node tracking process and bridges the gap between WSNs and post-cloud paradigms.

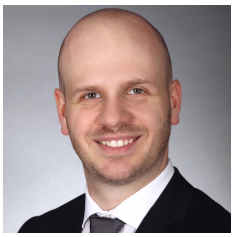
As future work, we plan to extend our node tracking policy and re-design Rime as a drop-in networking component for SPEs. This approach will allow for a tighter coupling of an optimizer of the SPE with Rime when making routing decisions. The optimizer will benefit from Rime by getting timely topology updates and offload reconfigurations.

Acknowledgements

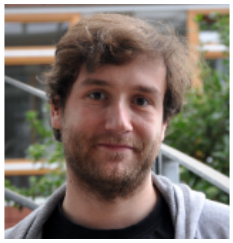
The authors would like to thank Moysis Symeonides and Daniele Miorandi for their comments, feedback, and constructive criticism during the writing of the paper. This work is part of a project that has received funding from the European Union's Horizon 2020 researchand innovation programme under the Marie Skłodowska-Curiegrant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein. This work is supported by the German Ministry for Education and Research as BIFOLD (01IS18025A, 01IS18037A) and the German Federal Ministry for Economic Affairs and Energy as ExDra (01MD19002B).

Author Biographies

Dimitrios Giouroukis is a Ph.D. candidate at the DIMA group (TU Berlin). His research interests include adaptive sensor data management, stream processing, and the IoT. He got his M.Sc. in Computer Science from Aristotle University of Thessaloniki.



Johannes Jestram studies Information Systems Management (ISM) in the Master programme at Karlsruhe Institute of Technology. His interests include distributed data processing and computer vision. He got his B.Sc. in ISM at TU Berlin and wrote his Bachelor thesis at the DIMA group.



Steffen Zeuch is a Senior Researcher at the DIMA group (TU Berlin) and IAM group (DFKI). His research interests are modern hardware and the IoT. He published research papers on query optimization and execution as well as novel system architectures. He did his Ph.D. in Computer Science at Humboldt University Berlin.

Volker Markl is a Full Professor and Chair of the Database Systems and Information Management (DIMA) Group at TU Berlin, Chief Scientist and Head of the Intelligent Analytics for Massive Data Research in DFKI, and Director of the Berlin Institute for the Foundations of Learning and Data (BIFOLD). He has published numerous research papers on indexing, query optimization, lightweight information integration, and scalable data processing. He is a Fellow of the ACM as of 2021.

REFERENCES

[1] C. C. Aggarwal, Managing and mining sensor data. Springer Science & Business Media, 2013.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," Computer Networks, 2002.

[3] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed virtual actors for programmability and scalability," MSR-TR-2014-41, 2014.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in SIGCOMM MCC, 2012.

[5] M. Chao and R. Stoleru, "R-mstorm: A resilient mobile stream processing system for dynamic edge networks," in ICFC, 2020.

[6] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native actors – a scalable software platform for distributed, heterogeneous environments," in SPLASH AGERE, 2013.

[7] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in IJCAI.

[8] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in SIGCOMM HotNets, 2010.

[9] A. Lerner, R. Hussein, and P. Cudré-Mauroux, "The case for network accelerated query processing," in CIDR, 2019.

[10] P. Levis, E. Brewer, D. Culler, D. Gay, S. Madden, N. Patel, J. Polastre, S. Shenker, R. Szewczyk, and A. Woo, "The emergence of a networking primitive in wireless sensor networks," Communications of the ACM, 2008.

[11] Y. Li, Q. Li, Z. Zhang, G. Baig, L. Qiu, and S. Lu, "Beyond 5g: Reliable extreme mobility management," in SIGCOMM, 2020.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," TODS, 2005.

[13] J. McArthur, A. Chandrasekaran, and T. Bittman, "Predicts 2021: Cloud and edge infrastructure," Accessed 2021-04-05. [Online]. Available: https://www.gartner.com/en/documents/3994091/predicts-2021-cloud-and-edge-infrastructure

[14] G. Mühl, L. Fiege, and P. Pietzuch, Distributed event-based systems. Springer Science & Business Media, 2006.

[15] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: resilient edge processing for the internet of things," VLDB, 2018.

[16] M. Peuster, H. Karl, and S. v. Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in IEEE NFV-SDN, 2016.

[17] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in ICDE, 2006.

[18] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," IEEE Pervasive Computing, 2009.

[19] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, 2016.

[20] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in Symposium on Edge Computing (SEC). IEEE/ACM, 2020.

[21] A. Woo and D. E. Culler, "A transmission control scheme for media access in sensor networks," in MobiCom, 2001.

[22] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing," in NBiS, 2010.

[23] S. Zeuch, A. Chaudhary, B. Monte, H. Gavriilidis, D. Giouroukis, P. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform: Data and application management for the internet of things," in CIDR, 2020.

[24] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," VLDB, 2019.

[25] S. Zeuch, E. T. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Del Monte, D. Giouroukis, P. M. Grulich, A. Ziehn, and V. Mark, "Nebulastream: Complex analytics beyond the cloud," VLIoT, 2020.

[26] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in SIGCOMM, 2020.