Efficient Distributed Query Processing

Roman Kolcun, David E. Boyle, and Julie A. McCann

Abstract—A variety of wireless networks, including applications of Wireless Sensor Networks, Internet of Things, Cyber-physical Systems, etc., increasingly pervade our homes, retail, transportation systems and manufacturing processes. Traditional approaches communicate data from all sensors to a central system, and users (humans or machines) query this central point for results, typically via the web. As the number of deployed sensors, thus generated data streams, is increasing exponentially, this traditional approach may no longer be sustainable, or desirable in some application contexts. Therefore, new approaches are required to allow users to directly interact with the network, for example requesting data directly from sensor nodes. This is difficult, as it requires every node to be capable of point-to-point routing, in addition to identifying a subset of nodes that can fulfil a user’s query. This paper presents DRAGON, a platform that allows any node in the network to identify all nodes that satisfy user queries, i.e. request data from nodes, and relay the result to the user. The DRAGON platform achieves this in a fully distributed way. No central orchestration is required, network overheads are low, and latency is improved over existing comparable methods. DRAGON is evaluated on networks of various topologies and different network densities. It is compared to the state-of-the-art algorithms based on summary trees, like Innet and SENS-Join. DRAGON is shown to outperform these approaches up to 88% in terms of network traffic required, also a proxy for energy efficiency, and 84% in terms of processing delay.

Note to Practitioners:

Abstract—This work is motivated by the continuing deluge of constrained, wirelessly connected sensing and control devices. Networks of communicable sensors and actuators are finding increased applicability across a range of industries and application scenarios. They are often thought of as a subset of the Internet of Things. However, due to the inherent difficulty in building these systems, technically and in terms of balancing the trade-offs between (economic) cost and performance (energy, latency, reliability, determinism), uptake has been slow. The community is relatively small, and therefore has not overcome all of the problems that present themselves considering required functionality of industrial applications. There is a need to find new ways to interact with these devices, particularly those with heterogeneous attributes. There is also clear motivation to progress from traditional system architectures, whereby all data sensed are transmitted to centralised storage and management platform, to decentralised means of interrogation and control. This work proposes a solution to this problem, describing and evaluating a novel framework to query constrained networked devices based on two key improvements over the current art. The first is construction and management of a dynamic routing mechanism that facilitates the second; a method to store static attributes in a distributed manner, which can be queried by a user - man or machine - in a decentralised way. In combination, they are presented as a platform, essentially a middleware, implementable in embedded firmware that sits between the lower (i.e. physical and medium access control) and upper (i.e. application) layers of the stack. The algorithms are presented in full, and extensively evaluated with respect to existing approaches. It is shown that the proposed platform significantly outperforms the existing art considering efficiency (latency and energy), maintaining modest implementation overheads in computation and memory. Ongoing and future work concerns extending this capability to continuous operation, considered essential for distributed control and autonomous systems applications.

Primary and Secondary Keywords Keywords—Distributed Query Processing, Wireless Sensor Networks, Cyber-Physical Systems, Autonomous and Automated Sensing and Control Systems, Routing Algorithm, Distributed Storage

I. INTRODUCTION

This work extends contributions presented in [1]. There are 10 billion wirelessly connected devices estimated to be in routine use today, and it is predicted that this figure will exceed 30 billion by 2020 [2]. These devices have become ubiquitous, and have found application in consumer electronics, transportation, environmental monitoring, and industrial automation. However, as the number of these devices exponentially grows, it becomes more challenging to manage interaction with and processing of the myriad of data streams produced by these devices. Traditional methods that tend to transmit all data to large centralised server systems are becoming outdated due to the physical upper bounds on communication capacities, systems resilience concerns and scalability issues. Additionally, a centralised approach raises significant security and privacy questions, where strict separation of users continues to be problematic, and contemporary systems are increasingly being proven vulnerable to attack.

Devices’ data, sensor readings for example, are typically forwarded to central processing systems through a single networked node (or reduced subset of nodes) with broader connectivity (i.e. from an ad hoc network to the Internet). Such a node is usually referred to as a base-station, sink or gateway; and is a potential bottleneck and/or single point of failure.

An alternative is where any node can communicate with a user, accept and evaluate a query, and return the result to the user. This happens without interaction or orchestration by a central node or federating system, allowing a user to interact with the network directly, therefore making the system fully decentralised and more reliable.

This is particularly important in automated systems, where an entity (human operator or machine) may request data from a set of relevant sensors, and depending on the outcome, a
message may be generated and sent to a specific actuator, for example. Consider the following illustrative scenario:

A remote oil field requires many miles of various pipes to transport liquids and gases. These pipes create a complex inter-connected system. Each segment of a pipe is monitored by a number of sensors. The field and pipe infrastructure is regularly monitored by an engineer who is responsible for the smooth operation of the process. Depending on the summary of information available, the engineer may be required to go into the field and investigate the cause of a problem.

Traditional approaches periodically collect data from every sensor to a central server. Users then query, often automatically, the server in order to obtain information. This approach usually requires long range communication. Users may also retrieve data from the main server using the long-range communication. However, it is possible that an Internet connection may be too expensive or otherwise unavailable. Similarly, the server may be overloaded and not capable to answer the queries [3]. If an engineer wants an answer to a query such as “what is the average gas pressure in a pipe in given segment”, any node in the network must be able to perform the following tasks: i) identify a set of sensor nodes capable of satisfying the requested criteria (i.e. pressure sensors on given segment of a pipe only) ii) request data from all participating nodes, and iii) report the result back to the user.

This work proposes and extends the DRAGON platform [1], which provides methods to tackle the first two of the aforementioned problems by allowing any node in the network to find a set of nodes fulfilling given static requirements and providing a reliable point-to-point communication. More precisely, the contributions of the paper are as follows:

- **Distributed Static Attribute Table (DSAT)** - a table used to store static information about each node in a scalable way, allowing any node to find a list of nodes fulfilling given static attributes by communicating efficiently in the local neighbourhood (this was originally referred to as Distributed Data Table (DDT) in [1])
- **Peer-to-peer routing protocol** for wireless sensor networks that routes messages via near-optimal routes without the need to search for a path in advance
- **DRAGON platform** built upon the DSAT and P2P routing protocol capable to answer one-time (snapshot) queries in timely manner and with low network overhead
- **Extensive evaluation** of DRAGON platform on networks of different topologies and various densities.

The rest of the paper is structured as follows: in Section II the problem is stated in more detail and in the Section III a solution to the problem is demonstrated. The solution is experimentally evaluated in Section IV. Findings are concluded in Section V.
as key-value storage engines. Pastry uses a hash function to map the key to an id, and the node with the closest nodeId is used to store the value. The value is also replicated on the k nodes with the closest nodeIds. During the lookup of the value, Pastry forwards the message towards the replica closest to the user. However, it is not Pastry’s aim to spread different replicas uniformly throughout the network such that every node can access them with minimum overhead. A similar approach to Pastry is used by Chord [12], which also stores the given value on the k closest nodes to the nodeId retrieved by hashing. Chord uses caching to speed up the search process, however, its aim is not to store the replicas close to all nodes in the network.

B. In-network Processing

Performing computation inside the network has several advantages: i) network traffic can be reduced, ii) single points of failure are reduced, and iii) computation latency is reduced. Minimising network traffic is especially important for WSN where as much as 80% of the overall energy consumption is attributed to the radio. Thus, by decreasing the network traffic it is possible to significantly increase the lifetime of the network. Decreasing latency is especially important in networks with actuation capabilities where an action may have temporal constraints.

The simplest variant of in-network processing is at the base-station, where only the relevant nodes send data to the base-station that processes them. Ciciriello et al. (2006) propose an abstraction of virtual nodes where a node collects and processes data from its neighbourhood [13]. However, discovery of relevant nodes is based on flooding the network x hops from the node. Additionally, the platform assumes a grid network topology. Stern et al. (2009) propose a two-phase approach, where summaries are firstly collected from the whole network, then, at the base-station, candidates fitting the query are chosen. In the second phase, only data from chosen candidates are retrieved [7]. The last approach uses pairwise joins, which splits the processing into pairs, and for each pair of sources it finds a node on the path between them that processes data [14]. This approach can significantly lower the number of messages, but only where the selectivity of the join, i.e. the percentage of tuples fulfilling the join predicate, is very low and the processing can be split into pairwise joins (i.e. a pairwise join operates only on exactly two streams of data, and produces only a partial result). The final join is carried out at the base-station.

Stone et al. (2014) propose a Hill Climb algorithm for GaianDB [15] to evaluate snapshot queries. Similarly to the proposed approach, any node can receive a query from a user. After receiving the query, the node uses the Hill Climb algorithm to find a node with the highest degree. GaianDB assumes that the data are uniformly distributed throughout the network, and the query performs an aggregation on data where the size of the result is much smaller than the sum of retrieved data. Once the node with the highest degree is found it propagates the query by flooding the network, similarly to the AODV [16] routing protocol. The nodes which contain data relevant for the query reply to the node with the highest degree. After processing data, the result is sent back to the user via the node which received the query.

Ma et al. (2013) [17] proposed a declarative approach to distributed computing where nodes are represented as input/output automata which could be analysed using Answer Set Programming. They introduce language $\mathcal{D}$ which can be used to specify distributed systems. However, their implemen-ration of the language is done in Java, and insufficiently considers the unique constraints of WSN-type IoT applications.

A primary disadvantage of the aforementioned approaches is that their set-up phases are very expensive, often heavily relying on the base-station (i.e. traditional base-station processing, the two-phase approach, and the second half of the pairwise join). To the contrary, DRAGON allows any node to find all relevant nodes and request data from them in a demonstrably more efficient way.

C. Routing

The most common routing mechanisms used in WSN are CTP [18] and RPL [19]. Both exploit a tree structure rooted at the base-station. Of these, only RPL supports point-to-point communication. Here, the message is routed up the tree until either the destination, an ancestor that has a known route to the destination, or the root is reached. An alternative designed for in-network processing is Innet [14], which exploits three summary trees rooted in different parts of the network to establish a path between two nodes. The number of paths discovered is equal to the number of summary trees. The shortest path is chosen for further communication between the nodes. Innet is able to find close to optimal paths between any two nodes but at the expense of a costly search, in terms of time and numbers of messages and therefore it is not suitable for ad hoc communication.

AODV [16] is an established routing protocol allowing P2P communication. The protocol first floods the whole network with a request, and the destination node replies to the request. During the request, a distance vector is built that is then used for communication. As the request floods the whole network, the path set-up overhead is even larger than in case of Innet.

Other routing protocols rely on knowledge of the geographical location of nodes, which is not always possible. Routing protocols like GPSR [20] or GEAR [21] cannot cope with obstacles or voids in the network, resulting in a node being unable to find a path to another node. The last group of protocols are referred to as hierarchical routing protocols [22]. Here, those that support P2P routing cannot do so in an optimal way with respect to the length of the discovered paths.

III. COMPUTATIONAL PLATFORM

DRAGON (Figure 1) is a platform for WSN, which supports efficient and reliable peer-to-peer communication in a multi-hop environment. DRAGON efficiently distributes information about static attributes of each node throughout the network, allowing any node to easily find other nodes matching given criteria.
In the rest of this section each subsystem is described in more detail. Before the platform can be used for query processing a bootstrapping phase is required, during which each node learns the Routing Table (RT), splits the Distributed Data Table (DSAT), and fills the DSAT with Static Attributes (SA). On completion, every node in the network can receive queries or send control/actuation messages. For each query or command the node finds all participating nodes, then either requests data from, or sends control messages to them.

A. Routing Table Discovery

Many of DRAGON’s subsystems rely on a routing table stored at every node. The routing table (RT) stores for each node in the network three pieces of information: destination, next hop, and distance. For the distance the number of hops was chosen as the simplest, yet representative metric; but any other kind of additive metric could be used (e.g. ETX in RPL [19]).

During the bootstrapping phase each node runs an algorithm (Algorithm 1) inspired by Netchange [23]. Netchange was designed for wired distributed computer networks with no broadcast capability and which assumes reliable packet delivery. Alg. 1 is optimised for wireless networks which are characteristically unreliable, but have real broadcast capabilities, i.e. a packet can be received by all nodes within communication range.

Initially, each node creates a record in its local RT and broadcasts a RT discovery packet to all its neighbours. A RT discovery packet contains a list of \( \langle \text{destination}, \text{distance} \rangle \) pairs. Upon receiving a RT discovery packet the receiving node updates its records in the RT. If there is no record for the destination a new record is created (line 4). If there is a record and the received distance is shorter than the one already learned or the same node sends an updated record (possibly with longer distance), the routing record is updated. As the “next hop” is set as the node from whom the message was received. The record is marked as “updated”. After every iteration, all updated records are broadcast to all neighbours.

Due to the unreliability of wireless communication, some nodes may not receive the message, hence they may learn a sub-optimal route to some nodes. This is mitigated by proactively broadcasting better paths, should a node identify one, and by exploiting overhearing of neighbours updating their tables. Assume nodes \( n_1, n_2, n_3 \) are neighbours. Node \( n_1 \) broadcasts a path to node \( n_x \) with a distance \( d \) which is received by node \( n_2 \) but not by the node \( n_3 \). In the next iteration node \( n_2 \) broadcasts the path to node \( n_x \) with distance \( d+1 \) which is now received also by node \( n_3 \), so node \( n_3 \) learns a path to \( n_x \) with distance \( d+2 \) via node \( n_2 \). When node \( n_3 \) broadcasts this path further, node \( n_1 \) receives this message and compares it with its RT (line 10). Because its distance to \( n_x \) is \( d \) it means that the distance to \( n_x \) of any of its neighbours should be at most \( d+1 \). Node \( n_3 \) can learn a better path via \( n_1 \).

Once the node has not updated its RT for some predefined time \( \Delta t \) it assumes that the RT is complete and it switches to the stable phase. In our case we set \( \Delta t = 10 \times t_b \), where \( t_b \) is the basic interval at which the algorithm checks the routing table and broadcasts updated records. During the stable phase it broadcasts small parts of its RT in a round robin fashion as a heartbeat beacon but only if no other message is scheduled.

During the bootstrapping phase every node broadcasts every routing record it receives, plus a small number of records which are re-broadcast if a neighbour’s suboptimal path is discovered. Therefore, the complexity of the Routing Table algorithm is \( O(N^2) \) where \( N \) is the number of nodes in the network. Because every routing record consists of only \( \langle \text{destination}, \text{distance} \rangle \) pair, one packet may contain up to \( r \) routing records. The message complexity then decreases to \( O(N \times N/r) \).

In case a node detects a failure of a neighbour, the node executes a failure recovery procedure. A disadvantage of

```
Algorithm 1 Routing Table Discovery
1: procedure RECEIVE_RECORD(record, senderId)
2: localRecord ← findRecordInLocalTable(record)
3: if localRecord = null then
4:    addRecord(record)
5:    markUpdated(record)
6:   else if localRecord.hops ≥ record.hops + 1 OR
7:        localRecord.nextHop ← senderId
8:    localRecord.nextHop ← record.hops
9:    markUpdated(localRecord)
10:   else if localRecord.hops ≤ record.hops − 2 then
11:    markUpdated(localRecord)
12: end if
13: end procedure

14: procedure SEND_RT(rt, packet)
15: for all updated record in rt do
16:    addToPacket(packet, record)
17: end for
18: if notEmpty(packet) then
19:    broadcast(packet)
20: end if
21: end procedure
```
Every node broadcasts at predefined interval a short message, referred to as a heartbeat message, informing its neighbours it is alive. If a node has not received five consecutive heartbeat messages from a neighbour it assumes the neighbour has failed and executes FAILEDNEIGHBOURDETECTED procedure. The procedure marks the failed node as “failed” and all destinations, where the failed node is set as the “next hop”, as unreachable. A request which contains the failed node, list of unreachable destinations, and the node’s distance to the failed node (line 2–4) is broadcast. This process can be seen in Figure 2 a), where nodes 2, 4, and 5 marked node 1 as the failed one. Each of these nodes broadcast a message notifying their neighbours about the failed node and informing them about destinations to which they have lost route.

Upon receiving the message, the receiving node compares its distance to the failed node with the one received from the sending node. If the receiving node is closer to the failed node than the sender, the message is ignored (line 3). The message is ignored so the information is propagated further into the network only. If the receiving node is further or the same distance to the failed node, the message is processed. The message contains a list of all destinations the sender cannot reach. The receiving node adds the sender to a list associated with every unreachable destination sent in the message. This list stores the nodes through which the receiving node will not be able to reach the destination (line 12). As there might be more shortest path passing through the same failed node the node has to wait for all the messages from the nodes closer to the failed node to be received. For example, in Figure 2 a) node 7 must wait for messages from nodes 4 and 5, as the node 7 can be reached via paths 7, 4, 1, 0 and 7, 5, 1, 0. Therefore, node 7 ought not to broadcast the path to 0 via 5 immediately after it receives the unreachable list from node 4. Node 7 must wait for the list of unreachable nodes from node 4, too.

Figure 2 a) shows the list created at node 7. It shows that destination node 0 is currently not reachable via nodes 4 and 5, destination nodes 2 and 3 are not reachable via node 4, and the destination node 4 cannot be reached via node 5. This information is used when the node checks its routing table and invalidates outdated records.

If a node is informed about a failed node for the first time, it sets a timer for a random timeout after which it will update its routing table (described below) and propagates information about the failed node further into the network (line 15). The timeout depends on the distance from the failed node - the further away the node is the larger the timeout is. The increasing timeout is important so a node waits long enough to receive messages from neighbours closer to the failed node.

Once the timer expires, the node checks its routing table. For every unreachable destination it has received from neighbours which are closer to the failed node, the node checks the next hop for given unreachable destination. If the RT lists as the “next hop” a node which is associated with an unreachable destination, the destination is marked as unreachable (line 22). Otherwise, it means the node has an alternative path to the destination, and marks the record as updated. Updated record

<table>
<thead>
<tr>
<th>Algorithm 2 Routing Table Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure FAILEDNEIGHBOURDETECTED(failedNode)</td>
</tr>
<tr>
<td>2: mark failedNode as failed</td>
</tr>
<tr>
<td>3: mark all records, for which failedNode is the “next hop”, as unreachable</td>
</tr>
<tr>
<td>4: broadcast failedNode along with the list of all unreachable destinations</td>
</tr>
<tr>
<td>5: end procedure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>procedure RECEIVEFAILEDNODEMSG(sender, failedNode, distance, unreachableDestinations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7: mark failedNode as failed</td>
</tr>
<tr>
<td>8: if my distance to failedNode &lt; distance then</td>
</tr>
<tr>
<td>9: return</td>
</tr>
<tr>
<td>10: end if</td>
</tr>
<tr>
<td>11: for all destination in unreachableDestinations do</td>
</tr>
<tr>
<td>12: add sender to list associated with destination in global variable unreachableList</td>
</tr>
<tr>
<td>13: end for</td>
</tr>
<tr>
<td>14: if first message for the failedNode then</td>
</tr>
<tr>
<td>15: set timer for time in the future depending on my distance to failedNode</td>
</tr>
<tr>
<td>16: end if</td>
</tr>
<tr>
<td>17: end procedure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>procedure TIMEREXPIRES(failedNode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18: for all destination in unreachableList do</td>
</tr>
<tr>
<td>19: record ← find record for destination in local routing table</td>
</tr>
<tr>
<td>20: if record.nextHop is in the list associated with destination then</td>
</tr>
<tr>
<td>21: mark record as unreachable</td>
</tr>
<tr>
<td>22: else</td>
</tr>
<tr>
<td>23: mark record as updated</td>
</tr>
<tr>
<td>24: end if</td>
</tr>
<tr>
<td>25: end if</td>
</tr>
<tr>
<td>26: end for</td>
</tr>
<tr>
<td>27: if there is a record marked as updated then</td>
</tr>
<tr>
<td>28: broadcast failedNode along with the list of unreachable destinations</td>
</tr>
<tr>
<td>29: else</td>
</tr>
<tr>
<td>30: broadcast failedNode only</td>
</tr>
<tr>
<td>31: end if</td>
</tr>
<tr>
<td>32: end procedure</td>
</tr>
</tbody>
</table>

Figure 2 shows the full process can be found.
is broadcast in the next iteration and the neighbour can learn this alternative path (line 24). In networks where nodes have several neighbours, the unreachable list is rarely propagated more than one or two hops away.

Finally, the node checks whether there are any unreachable destinations in its routing table. If there are, the list is propagated to the neighbours (line 28). Otherwise, only information about failed node is broadcast (line 30). This final state is shown in Figure 2b).

Because the algorithm requires that the number of hops to the failed node is increasing, we achieve that the information is propagated from the failed node further away. If a node misses a message, it may temporarily assume it has a path to an unreachable destination. However, as the nodes repeatedly broadcast their unreachable nodes, their neighbours eventually learn this information and propagate it further into the network. This behaviour was observed in numerous experiments carried out in networks of various topologies and densities. A formal proof showing convergence of the algorithm is left for future work.

In the case whereby a node failure does not lead to any “unreachable nodes” in the neighbours RT, then only the information about the failed node needs to be propagated throughout the network. Therefore, in this case, the message complexity of the algorithm is \( O(N) \), where \( N \) is the number of nodes in the network. However, when a node failure leads to appearance of “unreachable nodes” in the RT of the failed nodes neighbours, new routes must be discovered. This information is propagated further away from the failed node until new paths are discovered. The information about the new paths is propagated back towards the neighbours of the failed node. Therefore, the complexity of the algorithm is \( O(N + an \times 2) \), where \( an \) is the number of affected nodes, whose routing table(s) must be updated. This number is multiplied by 2 because a message with new paths is returned to the affected nodes.

The value of \( an \) depends on the network topology in addition to network density. It is unique for every node, as the failed node may be a part of a cluster of highly-connected nodes, where a node failure does not represent any increase in path distances. In this case \( an \) is equal to the number of immediate neighbours of the failed node. On the other hand, a node may be on a critical path between two node clusters. In this case, both clusters need to find new paths to each other, possibly taking much longer. Here, the node failure affects many nodes, as the information spreads into the network until new paths are found.

### B. Distributed Static Attribute Table

In [1] the Distributed Static Attribute Table is referred to as Distributed Data Table (DDT). The name is revised to better reflect its functionality.

Most contemporary WSN systems do not readily allow a node to search the network based on a given criteria [4]. An example of such search could be a node looking for all nodes with the same type of sensors, or monitoring the same phenomena in a heterogeneous sensing environment [24].

Each node in a network may have a set of assigned static attributes (e.g., id, type, pipe id). All static attributes in a network can be represented as a table, wherein each column represents an attribute and each row represents a node. In current systems, strategies that a node may follow in order to find another node with a given static attribute are: i) flood the whole network with a search query and wait for a response from all nodes [5], [6], ii) use a summary of all static attributes which can probabilistically say whether a given node, or part of the network, has or does not have the specific attribute [7], [8], iii) store information on a set of predefined node [9], [10].
DRAGON implements a form of the latter, i.e. it stores all static attributes locally. Storing real values is preferable to storing summaries, as with the real data a node can instantly find out which node fulfils given static criteria and which does not. It is assumed that the static attributes will not change throughout the lifetime of the network, and that the higher cost of bootstrapping will be easily overcome by savings made during the lifetime of the network. This is based on the assumption that searching for other nodes with given static attributes will be a common type operation required to answer a query submitted by users, nodes, or other autonomous systems.

WSN nodes are typically limited in terms of memory, and fitting an entire table of static attributes on a single node may be impossible. Due to this limited memory, inspiration was taken from traditional databases where large tables are horizontally partitioned, i.e. the table is split amongst several computers where each computer holds one part of the table. To execute a query on a partitioned table, the query is sent to every computer holding a part of the table. Each node replies with a partial result, and the final result is composed on the node computer holding a part of the table. To answer a query submitted by users, nodes, or other autonomous systems.

How many parts the DSAT is split into depends on the number of nodes in the network, static attributes, and the size of available memory on a node. Each node will store one part of the DSAT, provided the network is homogeneous. If the network is heterogeneous, i.e. some nodes are more powerful and have a larger memory, one node may store several parts of the DSAT.

How the parts are assigned to nodes affects how many messages a node has to send in order to search in the DSAT. Here the objective is to assign parts of the table in such a way that if any node in the network wants to search the whole table it ought to send the minimum number of messages. It is assumed that the DSAT will not change during the lifetime of the network, therefore more than one node is allowed to store the same part of the DSAT, i.e. the parts are replicated throughout the network. In terms of communication, the lower bound of number of messages a node has to send in order to search in the whole DSAT can be defined. Let the DSAT be split into \( p \) parts, \( N \) be the number of nodes in the network, and \( D_n = \{d_{0}, d_{1}, \ldots, d_{N-1}\} \) be the vector of distances to all nodes from node \( n_i \) in ascending order. The minimum cost \( c \) in terms of the number of messages a node \( n_i \) must send is defined as:

\[
c_{i}^{\text{min}} = \sum_{j=0}^{p-1} d_{ij}
\]

In other words: the sum of distances to \( p - 1 \) closest nodes. However, this can be achieved only if each of the \( p - 1 \) closest nodes store a different part of the DSAT.

Let \( partId \in \{0, \ldots, p-1\} \) be the ID of a DSAT part a node is storing. Then the objective of the distributed algorithm is to find a mapping \( f \) between \( nodeId \) and \( partId \), i.e. \( f(nodeId) \rightarrow partId \). The simplest solution is to use a hash function that assigns \( partId \) to a node randomly. However, this approach does not take the locality of the nodes into consideration. In order to evaluate proximity of the parts to a node it is necessary to define a metric, as follows. The normalised cost \( c \) of a node \( n_i \) to perform a search in the DSAT is defined as:

\[
c_i = \frac{1}{p} \sum_{j=0}^{p-1} d_{ij}
\]

where \( d_{ij} \) is the number of hops from node \( n_i \) to a node which holds part \( j \) of the DSAT. For example, if the DSAT is split into 4 parts and a node’s neighbours hold different parts of the DSAT, the node must send 3 messages to retrieve the whole DSAT (it does not need to send a message to retrieve the part which is stored on the node), therefore the normalised cost will be \( c = 3/4 \). On the other hand, if a node has only 2 neighbours the minimum number of messages the node must send is 4 (one part is at least 2 hops away), therefore the normalised cost will be \( c = 4/4 \). To minimise number of messages in the network, the normalised cost \( C \) of the whole network is:

\[
C = \frac{1}{N} \sum_{i=0}^{N-1} c_i
\]

where \( N \) is the number of all nodes in the network.

To minimise the overall cost, the following distributed algorithm described in Algorithm 3 is proposed. The objective of this algorithm is not only to choose the \( partId \) but also to discover closest nodes that store the rest of the DSAT. This information is stored in the \( dsat \) variable which contains association between \( partId \) and \( nodeId \), i.e. the node which stores given part of the DSAT. Variable \( dsat \) is an array of size \( p \) where the index is \( partId \) and the value is a list of IDs of \( k \) closest nodes storing given \( partId \).

The algorithm is initiated by a single node, which is chosen randomly. The initiating node calls DSATTIMEREXPRES function in which the node chooses its \( partId \) randomly and broadcasts this information to all neighbours. Upon receiving a broadcast message a node updates its \( dsat \) variable which stores \( k \) closest nodes for each part of the DSAT. The distance to a node is retrieved from the routing table. If a node receives the DSAT association message for the first time it chooses the \( partId \) the node chooses depends on the current state.
of the \( dsat \) variable. First, the node chooses the part which has not been chosen by any other node. Otherwise, it decides using one of two techniques (line 11): \( i \) the node chooses the \( partId \) which has been chosen by a node furthest away, referred to as Maximum Distance strategy or \( ii \) the node chooses the \( partId \) which has been chosen by least number of neighbours, referred to as Minimum Neighbour strategy. Ties are resolved by choosing the \( partId \) randomly.

The first and the last steps of the process are demonstrated in Figure 3. The whole process is shown in Appendix B. In this scenario the node uses the first technique, i.e. node \( n \) in Figure 3. The whole process is shown in Appendix B.

The parameters \( k \), \( N \), and \( r \) are resolved by choosing the \( partId \). In the table summarising all \( dsat \) variables, the node in bold green is the node that has already chosen its \( partId \), while the value shown in bold red represents an updated value, i.e. part that will be re-broadcast in the next epoch.

The process is initiated by node \( n_1 \) which, in this case, randomly chooses part 3, and broadcasts this information. The message from node \( n_1 \) is received by its neighbour nodes \( n_2 \) and \( n_3 \), both of which start a timer with a random delay (Fig. 3(a)). Once a timer expires the node decides which part of \( dsat \) it will store and this information is broadcast to all the neighbours. Meanwhile every node receives updates from all its neighbours. There are two cases in which a node updates its \( dsat \) variable: \( i \) the node chooses which part of \( dsat \) it will store or \( ii \) the node receives information that a part of \( dsat \) is stored on a closer node than the currently discovered one. If the \( dsat \) variable is updated, the node re-broadcasts it. The DSAT algorithm converges very quickly, and within a couple of epochs all nodes have their \( dsat \) variables filled in (Fig. 3(b)).

Each node broadcasts a message whenever its \( dsat \) variable is updated either due to the receiving a message from a neighbour or due to choosing its own \( partId \). Because nodes first receive several messages from neighbours, they must only broadcast one message when they choose their own \( partId \).

---

### Algorithm 3 Distributed Static Attribute Table

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure RECEIVEDSATASSOCIATION(dsat)</td>
</tr>
<tr>
<td>2</td>
<td>update dsat</td>
</tr>
<tr>
<td>3</td>
<td>if DSAT association message was received for the first time then</td>
</tr>
<tr>
<td>4</td>
<td>set timer with a random timeout</td>
</tr>
<tr>
<td>5</td>
<td>end if</td>
</tr>
<tr>
<td>6</td>
<td>if dsat was updated then</td>
</tr>
<tr>
<td>7</td>
<td>broadcast dsat</td>
</tr>
<tr>
<td>8</td>
<td>end if</td>
</tr>
<tr>
<td>9</td>
<td>end procedure</td>
</tr>
<tr>
<td>10</td>
<td>procedure DSATTIMEREXPIRES</td>
</tr>
<tr>
<td>11</td>
<td>partId ← choose part which was chosen least times by neighbours or is furthest away</td>
</tr>
<tr>
<td>12</td>
<td>insert chosen partId into dsat</td>
</tr>
<tr>
<td>13</td>
<td>broadcast dsat</td>
</tr>
<tr>
<td>14</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

### Algorithm 4 Static Attribute Propagation

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure RECEIVESTATICATTRIBUTE(staticAttribute, senderId)</td>
</tr>
<tr>
<td>2</td>
<td>if staticAttribute is received for the first time then</td>
</tr>
<tr>
<td>3</td>
<td>insert staticAttribute into buffer</td>
</tr>
<tr>
<td>4</td>
<td>associate timer with a random timeout with staticAttribute</td>
</tr>
<tr>
<td>5</td>
<td>associate list of neighbours with staticAttribute</td>
</tr>
<tr>
<td>6</td>
<td>if this node stores the part of DSAT to which staticAttribute then</td>
</tr>
<tr>
<td>7</td>
<td>insert staticAttribute into DSAT</td>
</tr>
<tr>
<td>8</td>
<td>end if</td>
</tr>
<tr>
<td>9</td>
<td>else</td>
</tr>
<tr>
<td>10</td>
<td>fetch staticAttribute from the buffer</td>
</tr>
<tr>
<td>11</td>
<td>end if</td>
</tr>
<tr>
<td>12</td>
<td>remove senderId and all common neighbours with the senderId from the list of neighbours associated with staticAttribute</td>
</tr>
<tr>
<td>13</td>
<td>end procedure</td>
</tr>
<tr>
<td>14</td>
<td>procedure STATICATTRIBUTETIMEREXPIRES(staticAttribute)</td>
</tr>
<tr>
<td>15</td>
<td>if list of neighbours for staticAttribute is not empty then</td>
</tr>
<tr>
<td>16</td>
<td>broadcast staticAttribute</td>
</tr>
<tr>
<td>17</td>
<td>end if</td>
</tr>
<tr>
<td>18</td>
<td>remove staticAttribute from buffer</td>
</tr>
<tr>
<td>19</td>
<td>end procedure</td>
</tr>
</tbody>
</table>

Therefore, the complexity of the algorithm is \( O(N) \), where \( N \) corresponds to the number of nodes in the network.

### C. Static Attribute Propagation

Static Attributes (SA) that describe each node in the network are stored in the DSAT. The DSAT must be completed with SA of each node. Theoretically, in order to propagate nodes’ lists of SA to every other node, each node must broadcast the list of every other node, leading to exchange of \( N^2 \) messages; where \( N \) is the size of the network.

However, this number is significantly reduced using Algorithm 4. Prior to starting the algorithm, each node learns a list of common neighbours with every other neighbour. When a receiving node \( n_r \) receives a list of static attributes \( sa \) from a sending node \( n_s \) for the first time (line 2), \( n_r \) stores \( sa \) in a buffer. Along with \( sa \) two additional pieces of information are stored: \( ln \) - a list of neighbours (of \( n_r \)) and a timer with a random timeout after which the \( n_r \) will broadcast the received \( sa \). It can be assumed that all \( n_s \)'s neighbours have also received the \( sa \), so \( n_s \) and all common neighbours with \( n_s \) can be removed from the \( ln \) (line 12). If \( n_r \) has already received the \( sa \), \( n_r \) removes \( n_s \) and all \( n_s \)'s common neighbours from the \( ln \).

Once a random timeout expires, \( n_r \) may broadcast the \( sa \) using the function STATICATTRIBUTETIMEREXPIRES. Prior to broadcasting, the node checks the \( ln \) (line 15). If \( ln \) is empty, i.e. all node’s neighbours have received the \( sa \) from other nodes, the node removes the \( sa \) from the buffer without broadcasting the \( sa \). If the \( ln \) is not empty the node broadcasts the \( sa \) (line 16). The timeout is chosen randomly in order to avoid all nodes broadcasting at the same time.

The algorithm finishes after a predefined \( \Delta t \) from the time the node has received the last static attribute update. Next, the node checks whether it has received static attributes about every node it is supposed to store information about. Each node knows the list of nodes whose static attributes it should store by
combining the information from the routing table and knowing which part of DSAT the node stores. In the implementation presented here, a simple modulo function is used, i.e. the node $n_i$ stores SA of a node $n_j$ if and only if

$$j \mod \text{partId} = 0 \quad (4)$$

Due to unreliability in wireless communication, it may happen that a node has not received the list of static attributes for some node $n_x$. In this case static attributes about node $n_x$ are first requested from the nodes storing the same part of DSAT. If these nodes also miss this information, static attributes are requested directly from node $n_x$.

If every node broadcasts every other node's static attribute the message complexity of the algorithm is $O(N \times N)$, where $N$ is the number of nodes in the network. However, using Algorithm 3 the complexity could be reduced to $O(N \times N \times p)$, where $p$ is the average probability that a node sends a message after expiration of the timer. The average probability is defined as

$$p = \frac{1}{N} \sum_{i=0}^{N-1} p_i \quad (5)$$

where $p_i$ is a probability that the node $n_i$ has to broadcast a message after the timer expires. The upper bound of the node's probability is defined as

$$p_i = \frac{nn_i}{mn_i} \quad (6)$$

where $nn_i$ is the number of neighbours of node $n_i$ and $mn_i$ is the minimum number of neighbours that have to broadcast the static attribute message before the expiration of the timer at node $n_i$ in order for the node $n_i$ to discard the message.

The average probability $p$ depends on the topology and the density of the network. While in dense topologies the $p$ is lower because nodes have more neighbours, in a sparse network the $p$ is higher. In the extreme case of a line topology $p = 1$, while in the opposite case of a clique topology, i.e. a fully-connected mesh network, $p = 1/N$. In uniform topologies the variance of $p_i$ is very low, while in the case of a random topology $p_i$ varies from node to node.

The message complexity can be further reduced by merging several static attributes into a single message, should the size of the message be large enough. The message complexity is then reduced to $O(N \times ((N \times p)/r))$, where $r$ is the number of records in a message.

### IV. Evaluation

The various parts of the proposed platform were evaluated in the TinyOS simulator, TOSSIM [25]. TOSSIM was chosen because of its reasonable accuracy in the simulation of real WSN, and its widespread use and popularity in the research community. The built-in radio and noise models were used. The packet size was set to 21 bytes.

DRAGON was evaluated on two different topologies: uniform and random. For each topology networks of four different densities were generated: $i)$ dense (D for uniform and RD for random topology, with 12 neighbours on average), $ii)$ medium dense (MD/RMD, 10 neighbours), $iii)$ medium sparse (MS/RMS, 7 neighbours), and $iv)$ sparse (S/RS, 5 neighbours). For each density three 250-node networks were generated. Each experiment was executed 10 times and the results presented take the average values.

#### A. Routing Table Discovery

The Routing Table Discovery algorithm is evaluated using the routing stretch metric. Routing stretch is defined as:

$$s = \frac{d_{\text{found}}}{d_{\text{optimal}}} \quad (7)$$

where $d_{\text{found}}$ is the number of hops between the nodes found by the routing algorithm, and $d_{\text{optimal}}$ is the number of hops of the shortest path between two nodes. The lower the routing stretch is, the better paths the routing algorithm is able to
find. In an ideal scenario, when the algorithm always finds an optimal path, the routing stretch \( s = 1 \).

Finding the shortest path between two nodes is important as it reduces network traffic. This becomes more influential when a path is used for long term communication, i.e. if two nodes communicate using this path for a longer period.

Section II-C describes and categorises routing algorithms. In this evaluation, the proposed algorithm is compared only with algorithms capable of peer-to-peer communication. Collection algorithms like CTP [18] or Backpressure [26] are omitted, as they support routing towards one node only. From the peer-to-peer group of routing algorithms, algorithms based on geographic routing are excluded, e.g. GPSR [20] or GEAR [21], as they rely on the exact geographic location of network nodes (which either requires specialised hardware or localisation algorithms). The specialised hardware increases the cost of the nodes as well as their energy consumption. Localisation algorithms require additional communication amongst the nodes. Additionally, geographic routing algorithms are not capable of dealing with routing voids.

Ad hoc routing algorithms are also excluded from evaluation, e.g DSR [27] or AODV [16]. These algorithms are based on flooding the whole network in order to find the destination node. This type of searching is extremely expensive in terms of network traffic and not scalable.

Therefore, DRAGON’s routing algorithm is compared only to those which are capable of restricting the search space. Specifically, comparison is drawn against three groups of algorithms: i) algorithms based on one routing tree, ii) algorithms based on several routing trees, and iii) hierarchical routing.

Representative of the first group is RPL [19]. Although RPL was designed to route data towards a base-station, it also supports peer-to-peer communication. By default data are routed towards a base-station via a tree. The tree is rooted at the base-station. Any node in the network can act as a routing node. This node stores a routing table for all nodes in a sub-tree rooted in given node. A packet sent to a specific destination is routed up the tree until it reaches a node that has a record in the routing table for the destination. Subsequently, the packet is routed down the tree towards the destination node. The base-station has records of every node in the network. In this implementation, it is assumed that every node may act as a routing node. Therefore, the route discovered is the shortest path in a tree. However, a recent study by Istomin et al. (2015) showed that RPL is not ready to handle P2P communication required in actuation networks and its strength lies in its collection capabilities [28].

The second group of algorithms is based on several routing trees, e.g. Innet [8]. Each routing tree is rooted in a different part of the network. The principle of routing is the same as with just one routing tree, however, because a packet is routed in several directions, there is a higher possibility that a shorter path will be discovered.

The last group of algorithms is hierarchical routing. This is not implemented for experimental evaluation, relying on extensive evaluation of this type of routing presented by Iwanicki and van Steen (2012) [22], where it is shown that the average routing stretch is 25%.

The comparison of routing stretch of various algorithms is depicted in Figure 4. As it can be seen (or in this case, as it cannot be seen) DRAGON’s routing stretch is constantly lower than 0.1% regardless the network topology or the network density. Therefore, it can be claimed that DRAGON can route messages via optimal (or near-optimal) paths.

The routing algorithm which exploits three different routing trees in order to find the shortest path performs rather well in all but the random sparse topology. The routing stretch ranges from only 3% to 28%, with an average of 11%. In the case of the algorithm based on one routing tree, where every node acts as a router, the routing stretch ranges from 16% to 50%, with an average of 33%.

It is important to note that after bootstrapping, DRAGON and Hierarchical Routing algorithm can start routing packets immediately, whereas algorithms based on routing trees must first discover the paths between the nodes. This discovery phase requires additional messages, hence these platforms are not suitable for ad hoc communication.

\[ C_{\text{min}} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{1}{p_i^C} \]  \hspace{1cm} (8)
$C_{min}$ is the normalised cost computed by summing of distances to $p - 1$ closest nodes. It is assumed that in an optimal assignment each node can find the rest of the DSAT on the nearest $p - 1$ nodes. However, in reality, such mapping may not be possible.

Figure 5 shows the results of a comparison of two versions of the parts assigning algorithms with a random assignment and a theoretical minimum in uniform networks of dense and sparse density. Due to limited space only two densities are shown as they represent extremes. As the results for random and uniform topologies are very similar, only results for uniform topologies are presented. The x-axis shows how many parts the DSAT is split into, while the y-axis shows the normalised overall cost as defined by Equation 8 and the “theoretical optimal” cost defined by Equation 8. Although on average both algorithms perform similarly (with an exception of sparse networks), taking a closer look at the results shows that the Maximum Distance version of the algorithm performs better in more sparse networks or if the DSAT is split into more parts. On the other hand, Minimum Neighbour version of the algorithm performs slightly better in dense networks or if the DSAT is split into fewer parts. This behaviour is to be expected as, in dense networks, each node has more neighbours. In this case, the Maximum Distance algorithm chooses the parts randomly, as many nodes holding other parts of the DSAT are equally distant. However, the Minimum Neighbour algorithm in this case takes into account how many neighbours have chosen a given part and chooses the part which was chosen least times.

Table I summarises the results for uniform topologies. It shows minimum, maximum, and average gain for both versions of the algorithm when compared to random assignment. Results are grouped by network density. It can be seen that the average gain is approximately 10%, while the maximum gain can reach 17%.

### C. Static Attribute Propagation

Static Attribute Propagation (SAP) problem can be seen as a dissemination problem, where each node must disseminate its static attributes to other nodes in the network. However, there are several differences between the traditional dissemination problem and the static attribute propagation problem. First, in the case of dissemination, one node typically needs to disseminate a certain value to all other nodes, whereas in the case of the static attribute propagation problem, all nodes must disseminate their values to other nodes. Secondly, the disseminated value must reach all nodes, where in case of static attribute propagation the value has to reach only a subset of nodes, i.e., those nodes which store a certain part of the DSAT. Third, a node cannot establish by itself whether it has the latest disseminated value, while in case of static attribute propagation, a node can find all missing values locally, and request these data from other nodes.

Nevertheless, value dissemination is the closest problem solved in WSN, therefore the SA propagation algorithm is comparable to dissemination protocols. Trickle [29] is the most established dissemination protocol where each node, upon receiving a disseminated value, broadcasts the value several times, with each subsequent broadcast occurring after a longer delay from the previous broadcast. In order to reduce the number of messages, in the evaluation each node broadcasts the message only once.

Other dissemination protocols reduce the number of messages by overhearing dissemination messages from neighbours. A node, on receiving a message, waits for a random delay. During this delay, the node listens to neighbours and counts how many of them broadcast the message. Let $x$ be the percentage of neighbours that have broadcast the message at the time when the random delay expires. If $x > t$, where $t$ is a threshold, the node discards the message without broadcasting it. This approach differs from SA propagation algorithm in that the node is not aware of which neighbours may or may not have received the message.

In the evaluation, the threshold $t$ is varied. The higher the threshold is, the more neighbours have to broadcast the message before the node decides to discard it. The following thresholds are used: $t \in \{50\%, 75\%, 100\%\}$. If $t = 100\%$ every node broadcasts the dissemination message upon receiving it and therefore it mimics the Trickle algorithm. Lower thresholds $t \in \{10\%, 25\%\}$ were also evaluated, but even though they performed rather well in dense networks, in all other network densities they were unable to converge. Algorithms with such a small thresholds were able to propagate static...
TABLE II: Number of messages comparison of DRAGON with Trickle algorithm for static attributes propagation.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dense</td>
</tr>
<tr>
<td>Trickle t = 50%</td>
<td>48.7%</td>
</tr>
<tr>
<td>Trickle t = 75%</td>
<td>57.6%</td>
</tr>
<tr>
<td>Trickle t = 100%</td>
<td>59.0%</td>
</tr>
<tr>
<td>Topology</td>
<td>Random</td>
</tr>
<tr>
<td>Trickle t = 50%</td>
<td>45.3%</td>
</tr>
<tr>
<td>Trickle t = 75%</td>
<td>55.1%</td>
</tr>
<tr>
<td>Trickle t = 100%</td>
<td>57.6%</td>
</tr>
</tbody>
</table>

TABLE III: Time of static attributes propagation comparison of DRAGON with Trickle algorithm.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dense</td>
</tr>
<tr>
<td>Trickle t = 50%</td>
<td>23.9%</td>
</tr>
<tr>
<td>Trickle t = 75%</td>
<td>33.0%</td>
</tr>
<tr>
<td>Trickle t = 100%</td>
<td>34.8%</td>
</tr>
<tr>
<td>Topology</td>
<td>Random</td>
</tr>
<tr>
<td>Trickle t = 50%</td>
<td>8.3%</td>
</tr>
<tr>
<td>Trickle t = 75%</td>
<td>17.6%</td>
</tr>
<tr>
<td>Trickle t = 100%</td>
<td>21.4%</td>
</tr>
</tbody>
</table>

attributes within a small part of the network only. The nodes that did not receive the data started to request the missing data directly from the source nodes. This led to extremely high traffic and buffer overflows, causing the whole network to crash.

The evaluation focuses on two metrics: i) number of messages sent, and ii) time it takes to propagate all static attributes. The first metric shows the amount of energy spent to propagate static attributes throughout the network. The second shows how long it takes for the network to converge to the point when every node can start searching in DSAT.

The results comparing number of sent messages are provided in Figure 6a. These are grouped by network topology and network density. As can be seen, DRAGON is more energy efficient than other dissemination algorithms in all but random sparse networks, where it is slightly (by 4%) outperformed by the Trickle algorithm with \( t = 50\% \). It can be seen that the gain in performance of DRAGON algorithm is lower as the network becomes more sparse. This is understandable, as in sparse networks there are fewer common neighbours and, therefore, more nodes are required to broadcast a message.

The comparison of DRAGON and Trickle algorithms with several different thresholds is summarised in Table III. It can be seen that DRAGON sends \( 26 - 59\% \) less messages in the case of uniform topologies (with an average of \( 45\% \)), and \( 11 - 58\% \) less messages in the case of random topologies (with an average of \( 37\% \)). DRAGON is outperformed by Trickle algorithm with \( t = 50\% \) only in the case of random sparse topology, and then only by 4%.

Inspecting the graph, an obvious outlier can be seen in the case of random medium sparse networks. The reason lies in the topology of these networks. These networks consist of loosely connected large clusters of nodes. In this case, with a small threshold, it is more likely to happen that static attributes about a node are not propagated from one cluster to another. This leads to increased traffic once the nodes start to request static attributes directly from other nodes.

The next metric considers the time it takes for all static attributes of each node to propagate throughout the network. As it can be seen in the Figure 6b and the summary Table III DRAGON algorithm is faster in most of the studied cases by up to 35%. DRAGON is slightly slower in the case of random medium sparse network when compared to Trickle with threshold \( t = 75\% \) or \( t = 100\% \). Additionally, DRAGON is also slower in the case of a random sparse topology when compared to Trickle with threshold \( t = 50\% \). The trend is the same as with the number of messages, i.e. the more dense the network is, the faster DRAGON algorithm propagates static attributes. This is understandable, as the fewer messages the network has to send, the faster it reaches the final state. The duration may only be influenced by the final stage of the algorithm when nodes are requesting missing data directly...
from other nodes. This effect can be seen in the case of Trickle algorithm with threshold \( t = 50\% \) in random medium sparse network when many nodes were requested data from other nodes. This is consistent with the results of the number of sent messages.

**D. Sources Discovery**

In this section, **DRAGON**’s ability to find a list of nodes with certain static attributes, and request data from them with low overhead, is evaluated. **DRAGON** is compared to approaches based on summaries. How the number of summary trees, in addition to the number of summaries held by each node, influence the network traffic is studied. Particularly, the network traffic in platforms based on one and three summary trees (in the figures and tables marked as “1T” and “3T” respectively) is evaluated. Each node in a tree holds either one summary for the whole sub-tree rooted in given node (referred to as “tree summary” (TS)) or the node holds one summary for the whole sub-tree rooted in given node (referred to as “tree summary” (CHS)).

The number of trees and the number of summaries have a large impact on memory requirements. Here, six static attributes were assigned to every node in the network: \( id \) - a unique identifier, \( x \) - a random uniformly distributed variable, \( y \) - an exponential variable with \( \lambda = 0.05 \), \( z \) - an exponential variable with \( \lambda = 0.1 \), and \( coord_x, coord_y \) - virtual coordinates of the node. In the case of **DRAGON**, static attributes are stored in the DSAT split into 10 parts, i.e. every node stores information about 25 nodes. In the case of summaries, attributes \( id, x, y, z \) are stored using both, Bloom filters and count histograms, while \( coord_x, coord_y \) are stored using an R-Tree. Using both the Bloom filter and count histogram summary allows nodes to answer both equality queries and range queries. The cost of storing summaries in a memory is calculated as:

\[
c = \text{trees} \times \text{children} \times \text{summarizes} \times \text{summary length} \quad (9)
\]

For each summary, 16 bytes of memory is allocated. If it is assumed that a node has six children on average, the cost to store all summaries ranges from \( c_{1T \ TS} = 144 \text{ B} \) to \( c_{3T \ CHS} = 2594 \text{ B} \). To store DSAT **DRAGON** requires \( 25 \times 6 = 150 \text{ B} \) and 506 B to store the RT.

**DRAGON**’s ability to answer snapshot queries is evaluated by executing two queries based on the scenario described in Section 4. In the first case, an engineer wants to retrieve a minimum, a maximum, and an average flow from sensors on the specific pipe. Each node on that pipe has a flow sensor, and can provide the flow attribute. The engineer
issuses the following: Query 1: SELECT MIN(S.flow), MAX(S.flow), AVG(S.flow) FROM Sensors S
WHERE S.x = @val, where @val is a random number.

Two metrics are evaluated: i) the number of messages sent, and ii) the time it takes to receive the result. The results for Query 1 are shown in Figure 7. Because Query 1 uses the equality operator, the algorithms based on summaries use Bloom filters to direct the search to the correct parts of the network. The query resulted in requesting data from 3 – 11 nodes, depending on the network. Because static attributes were generated for each network randomly, it is important to note that the results are comparable only between different approaches within the same network, not between different network topologies or densities.

As can be seen from Figure 7a (and the summary in Table IV) DRAGON significantly outperforms all other approaches based on summaries in terms of network traffic. DRAGON can decrease the network traffic by as much as 88% with an average over 80%, depending on the network density and the approach with which DRAGON is compared. Surprisingly, there is not much difference between algorithms using various numbers of trees and various numbers of summaries. This suggests that the Bloom filter is effective in finding correct source nodes, with few false positives.

The comparison of time taken for the initiating node to retrieve the result from the network is shown in Figure 7. Similarly to network traffic, DRAGON greatly outperforms other approaches in terms of response time. The response to the query could be as much as 84% faster, with an average of 64%. The network response time is very important in actuation networks where a node should act as soon as possible to a detected event.

Once the engineer receives the results, it may be necessary to check the average flow only on segments from the current position downstream. To do this, the following Query 2: is submitted: SELECT AVG(S.flow) FROM Sensors S
WHERE S.z > @val, where @val is a random number.

The query finds all flow sensors whose z attribute is higher than the given value @val. In the case of DRAGON there is no difference whether the equality or inequality operator is used as it operates over raw data stored in the DSAT. DRAGON is influenced only by the number of parts the DSAT is split into. However, in the case of summaries, the situation is different. Bloom filters cannot be used as they can only check whether a given value was or was not added previously to the filter. Therefore, for this query, the count histogram is used.

The results for Query 2 are shown in Figure 8. This query results in requesting data from 2 – 18 nodes, depending on
the network. The results are grouped by network topology and density. Figure 8a and Table VII show that DRAGON outperforms all other approaches in terms of network traffic. The saving ranges between 30 – 81% with an average of 61%. It is shown that the approach based on three trees and just one summary for all the children (marked as “3T TS”) struggles, and sends significantly more messages than other approaches. This suggests that having too many nodes in just one summary can have a negative impact on false positives of histograms.

Considering latency, DRAGON outperforms all other approaches almost all the time, as shown in Figure 8b and Table VII. The only occasion when DRAGON is slightly slower than approaches based on three summary trees is in a sparse network. This can occur when all the source nodes are relatively close to the initiating node. In that case, searching the DSAT takes longer than searching in a close neighborhood using summaries. The maximum saving could be as high as 77%, while on average DRAGON is 31% faster.

V. Conclusion

Finding a list of nodes with a given set of static attributes, without flooding the whole network, is very challenging in WSN. The nodes are constrained in terms of computational power and, more importantly, memory. Therefore, it is not practical to store global information about the whole network on a node. Most WSN routing protocols do not support point-to-point communication, and where they do, paths among the nodes are either far from optimal, the cost of finding these paths is very high, or both.

This paper presented DRAGON - a platform allowing any node in the network to easily and efficiently find a list of nodes with given static attributes, request data from these nodes, and return the result to the user. The attributes are stored in a distributed way throughout the network using a Distributed Static Attribute Table. Any node in the network can easily search this table while communicating within a close neighborhood. Also presented was a distributed algorithm for Routing Table discovery and its update in the case of node failure. The Routing Table is stored at every node allowing point-to-point communication among any pair of nodes without the need to find or establish the path. DRAGON is comparatively evaluated with the state-of-the-art approaches, where it was shown to achieve network traffic reductions of up to 88%, and the response time improvement up to 84% over comparable methods.

Ongoing work concerns improving existing limitations with the fixed number into which the DSAT is split. We are also considering scenarios whereby increasing network dynamics, e.g. regularly adding and removing of nodes to the network, may be more effectively supported. We have extended DRAGON towards supporting in-network processing of continuous queries in homogeneous constrained networks [30] and our ongoing work includes implementing support for heterogeneous constrained networks.

Acknowledgement

The authors wish to thank the authors of [14] for providing InNet’s source code, which allowed our comparison to be full and fair, and acknowledge the support of EIT Digital (15171) and RCUK/EPSRC (EP/I038837/1).

References


Roman Kolcun received M.Eng. in Computer Science at Technical University of Kosice, Slovakia, in 2007, M.Sc. in Advanced Computing at Imperial College London, United Kingdom, in 2008, and Ph.D. in Computing at Imperial College London, United Kingdom, in 2015.

Since 2015, he works as a Research Associate at Department of Electrical and Electronic Engineering at Imperial College London. His research interest includes distributed algorithms, distributed query processing, and communication protocols in wireless sensor networks with the focus on energy efficiency, reliability, and low latency.

David Boyle (M’07) received the PhD degree in Electronic and Computer Engineering from the University of Limerick, Ireland, in 2009, following his B.Eng. in Computer Engineering in 2005. He joined the Department of Electrical and Electronic Engineering at Imperial College London in 2012. His research interests span numerous topics within sensor networks, cyber-physical systems and IoT technologies with industrial application.

Julie A. McCann (M’99) is currently a Professor in computer systems with Imperial College London, London, U.K., where she leads both the Adaptive Embedded Systems Engineering Research Group and the Intel Collaborative Research Institute for Sustainable Cities and is currently working with NEC and others on substantive smart city projects. Her research centres on highly decentralised and self-organising scalable algorithms for spatial computing systems, e.g., wireless sensing networks. She has received significant funding through bodies such as the U.K.’s Engineering and Physical Sciences Research Council (EPSRC), Technology Strategy Board, and Natural Environment Research Council, as well as various international funds. She is an Elected Peer for the EPSRC.

Prof. McCann is a member of the Association for Computing Machinery and was elected as a Fellow of the British Computer Society in 2013. She has actively served on and chaired many conference committees. She is currently an Associative Editor of the ACM Transactions on Autonomous and Adaptive Systems. She is a Chartered Engineer in the U.K.
APPENDIX A

ILLUSTRATION OF ROUTING TABLE UPDATE PROCEDURE

Fig. A.1: Routing Table Update Algorithm. The figure shows how routing tables (RT) are updated upon node failure. Only partial RT of four nodes: $n_2$, $n_4$, $n_5$, and $n_7$ are shown.

a. Initial state, just before node $n_1$ fails. Partial routing tables for node $n_2$, $n_4$, $n_5$, and $n_7$ are shown.

b. Neighbours of node $n_1$ notice the failure and update their routing tables. Node $n_1$ is marked as failed and the destinations where $n_1$ was the next hop as unreachable.

c. Message about the failure is propagated further into the network. Node $n_7$ broadcasts those records for which it knows a path.

d. Nodes $n_7$ and $n_2$ broadcast updated records from their routing tables.

e. Node $n_4$ learns a new path to node $n_0$ via node $n_7$.

f. The algorithm converges. All nodes have updated routing tables.
APPENDIX B
ILLUSTRATION OF DISTRIBUTED STATIC ATTRIBUTE TABLE PROCEDURE

![Diagram](image)

a. The process is initiated by $n_1$.
b. Timer expires at node $n_3$.
c. Timer expires at node $n_6$. Node $n_1$ re-broadcast its $dsat$ as it was updated in previous epoch.
d. Timer expires at nodes $n_2$ and $n_8$. Node $n_3$ re-broadcast its updated $dsat$.
e. Timer expires at node $n_4$ and $n_{10}$. Nodes $n_1, n_5, n_6,$ and $n_7$ re-broadcast their updated $dsat$. Almost every node has its $dsat$ full.
f. Timer expires at node $n_5$ and $n_9$. Nodes $n_2, n_3,$ and $n_8$ re-broadcast their updated $dsat$.
g. The last node to choose its part is node $n_7$. After node $n_1$ re-broadcast its $dsat$ the process is finished.

Fig. B-1: Assignment of DSAT parts to nodes.