Dragon: Data Discovery and Collection Architecture for Distributed IoT

Roman Kolcun and Julie A. McCann
Department of Computing, Imperial College London, United Kingdom
Email: {roman.kolcun08, j.mccann}@imperial.ac.uk

***Abstract*—Wireless Low-powered Sensing Systems (WLSS) are becoming more prevalent, taking the form of Wireless Sensor/Actuator Networks, Internet of Things, Phones etc. As node and network capabilities of such systems improve, there is more motivation to push computation into the network as it saves energy, prolongs system lifetime, and enables timely responses to events or control activities. Another advantage of such edge-processing is that these networks can become autonomous in the sense that users can directly query the network via any node in the network and are not required to connect to gateways or retrieve data via long range communications.**

Dragon is a scheme that efficiently identifies nodes that can reply to user requests based on static criteria that either describes that node or its data and provides the ability to near-optimally route queries or actuation control messages to those nodes. Dragon is scalable and agile as it does not require any central point orchestrating the search. In this paper we demonstrate significant performance improvements compared with state-of-the-art approaches in terms of numbers of messages required (up to 93% less) and its ability to scale to 100s of nodes.

I. Introduction

It is estimated that there are 10 billion wirelessly connected devices routinely used today and this is predicted to grow to 30 billion by 2020 [1]. These devices can be found in various areas including industrial automation, environmental monitoring networks of sensors and actuators, or transportation. In this paper we refer to systems such as sensor enabled Internet of Things (IoT) like phones etc., Wireless Sensor Networks (WSNs), and sensor enabled Wireless Middleboxes as Wireless Low-powered Sensing Systems (WLSS). It is expected that such systems will generate vast amounts of data streams to be analysed. Traditional methods that assume all data is communicated to server systems are becoming outdated due to the physical upper bounds to communication capacities, systems resilience concerns and scalability issues.

An alternative is where any node (including mobile nodes in close proximity) interacts with the network to generate either queries or actuation messages. This essentially supports an autonomous highly-distributed sensor data system, in which all processing is carried out in the sensor nodes. In parallel we are seeing a convergence between sensing systems and control systems; control may be messages to affect a change such as switch a water valve, close a gate, or even update systems WLSS code. However, the vast majority of WLSS work assumes that data is either periodically collected and routed to base-stations, or queries and actuation messages are flooded into the network and the response is processed off-line (in servers/base-stations). Other schemes ship data to known nodes for efficient retrieval [2], [3]. However, all these approaches require a significant amount of communication. Dragon, provides distributed data-centric node identification and point-to-point routing to support actuation and in-network query processing aiming to reduce the amount of communications required, removing single points of failure and improving response-times to events. Let's explore data stream processing in detail using the following simple but illustrative scenario:

Every shop on a high street has a set of sensors providing various readings. Customers are able to query the network in order to find an open shop or a pub with a free space.

A traditional approach is to periodically collect data from every sensor, send them via a base-station to a server which then processes the data and relays answers to the user. This approach requires the network to be equipped with a node capable of long range communication (e.g. using GPRS). Users then retrieve data from the main server using the long-range communication. However, owner of the network may not be willing to pay mobile and server fees, yet still wants to provide its services to users.

In order to be able to answer queries such as "which pub on this street has most free spaces" any node in the network must be able to perform the following tasks: i) identify a set of sensor nodes fulfilling the given criteria (sensors counting free spaces in premises that are pubs), ii) request data from all participating nodes, and iii) report the result back to the user.

Dragon tackles 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:

- We present a *Distributed Data Table* (DDT) which is 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 with only near neighbourhood.
- We present a new *peer-to-peer routing protocol* for WLSS able to route messages via near-optimal routes without the need to search for a path beforehand.

The rest of the paper is structured as follows: In Section II we state the problem in more detail and in the Section III we describe our solution which we experimentally evaluate in Section IV. We conclude our findings in Section V.

II. Current Approaches & Related Work

Nowadays, WLSS typically consist of several tens or hundreds of nodes and one or more base-station(s). The base-station is usually powered, has higher computational power compared to a general sensor node, and has some form of long-range network connection which is used to report results to a server. Therefore it represents a gateway to the WLSS and provides WLSS's capabilities to the outside world and thus is a vulnerable failure point in the system.

Alternative approaches dispense with the base-station but are faced with the following three challenges: i) locating the

nodes that fulfil the static criteria in the query, ii) processing data inside the network, and iii) allowing point-to-point communication to ensure that data is routed throughout the network in an efficient way.

A. Searching by Attribute

Each WLSS device in the network has assigned to it several *static attributes* e.g. node ID, its position, type of sensor data it is providing, or the area where the node is deployed in (e.g. on Street #233). Information about all the static attributes found in the network can be seen as a table, where each attribute is represented by a column and each node represents one row in the table. Having the possibility to search in this table, without flooding the whole network with a request, every node can easily retrieve a list of nodes fulfilling given static criteria, e.g. any node can find all nodes from the given area, or all nodes monitoring the same street. A recent survey on in-network processing [4] states the need for a platform that permits nodes to search WLSS networks by attribute and identifies this as one of the subject's largest challenges.

Several approaches have been proposed to solve this problem. The easiest solution is to *flood* the network with the request where only the nodes fulfilling the criteria reply to the request [5]. This approach is simple, yet extremely expensive in terms of time and amount of network traffic. Another common approach exploits the routing tree structures by storing *summaries* (e.g. a Bloom filter, histogram, or R-Tree) of static attributes at every node in the tree. Here, each intermediate node stores summaries for the sub-tree rooted in given node. When a request is received, a node can probabilistically decide whether the subtree contains node(s) that satisfy a given static attribute and decide whether to forward the request or not. Stern et al. rely on building one tree [6] while Mihaylov et al. build three trees rooted in different parts of the network [7] in order to speed-up the search and to find shorter paths between nodes. The problem of using summaries is that the search is probabilistic, hence the confirmation from the destination nodes is required. Further, different types of summaries are optimal for different types operations (e.g. Bloom filters are optimal for equality search while histograms are optimal for range queries). Keeping more summaries requires larger memory.

In the third approach, Ratnasamy et al. propose a *geographic hash table* which stores attributes on a node closest to the hash of an attribute key [2]. Greenstein et al. propose an extension of this approach which supports range queries over the stored attributes [3]. Both of these approaches place data randomly in the network, not taking proximity to other nodes into account, they rely on a geographical routing, which cannot cope with obstacles in a network, and they assume rectangular uniform network topology.

B. In-network Processing

Performing the computation inside the network has several advantages: i) network traffic can be lowered, ii) a single point of failure is removed, and iii) computation latency is lowered. Minimising network traffic is especially important for WLSS where as much as 80% of the overall energy consumption is attributed to the radio [8], hence, by decreasing the network traffic it is possible to significantly increase the lifetime of the network. Decreasing the latency is especially important in networks with actuation capabilities where 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 which processes them. Ciciriello et al. propose an abstractions of *virtual nodes* where a node collects and processes data from its neighbourhood [9]. 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. propose a *two-phase approach* where first summaries are 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 [6]. 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 which processes data [10]. 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. 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.

The disadvantage of all of the aforementioned approaches is that their set-up phase is very expensive and in some cases they heavily rely on the base-station (i.e. traditional base-station processing, the two-phase approach, and the second half of the pairwise join). On contrary, Dragon allows any node to find all relevant nodes and request data from them in an efficient way.

C. Routing

The most common routing mechanisms used in WLSS are CTP [11] and RPL [12]. Both exploit a *tree structure* rooted at the base-station. From these two protocols, 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 [10] which exploits three summary trees rooted in different parts of the network to establish a path between two nodes. 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, therefore it is not suitable for ad-hoc communication. AODV [5] is an *ad-hoc* 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 reply a distance vector is built which 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*, which is not always possible. Routing protocols like GPSR [13] cannot cope with obstacles or voids in the network, resulting in a node not being able to find a path to another node. Further, the last group of protocols are referred to as *hierarchical routing protocols* [14]. Here, those that support P2P routing cannot do so in an optimal way in terms of the length of the discovered paths.

III. Computational Platform

In this section we present Dragon - a platform for WLSS, which supports efficient and reliable peer-to-peer communication in a multi-hop environment. Additionally, it efficiently

Algorithm 1 Routing Table Discovery

```
1:  procedure RECEIVERECORD(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
        localRecord.nextHop = senderId then
7:          localRecord.hops ← record.hops
8:          localRecord.nextHop ← senderId
9:          markUpdated(localRecord)
10:     else if localRecord.hops ≤ record.hops − 2 then
11:         markUpdated(localRecord)
12:     end if
13: end procedure

14: procedure SENDRT(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
```

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 we will present each subsystem separately 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 (DDT), and fills the DDT with Static Attributes (SA). Once bootstrapping has finished *every* node in the network is ready to receive queries or send actuation messages. For each query or actuation command the node finds all participating nodes and either requests data from them or sends actuation 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 we have chosen the number of hops as the simplest, yet representative metric; but any other kind of additive metric could be used (e.g. energy spent by nodes to deliver a packet from one node to another).

During the *bootstrapping* phase each node runs an algorithm (Alg. 1) inspired by Netchage [15]. Netchange was designed for wired distributed computer networks with no broadcast capability and which assumes reliable packet delivery. Our algorithm is optimised for wireless networks which are by their nature unreliable but with real broadcast capabilities where one packet is received by all nodes within broadcasting distance.

At the beginning of the algorithm, 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 $\langle destination, distance \rangle$ 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 learnt 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" so during the next iteration the record is broadcast to all neighbours.

Due to the unreliability of the 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_1 assumes that the node n_3 has not received its previous message, therefore the path to n_x is rebroadcast so the node n_3 can learn a better path via n_1.

Once the node has not updated its RT for some predefined time Δt it assumes that the RT is complete and it switches to the *stable* phase. 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.

In case a node detects a node failure it executes a failure recovery procedure. The node which detected the node failure marks the failed node and all destinations where the failed node is set as the "next hop" as unreachable. Then it broadcasts the request which contains the failed node, list of unreachable nodes, and the node's distance to the failed node. Upon receiving the message a node waits for a random delay which increases with the distance to the failed node. The receiving node collects messages from all nodes closer to the failed node. Then for all unreachable nodes the node checks its RT. If for an unreachable node the "next hop" is set a node from which the failed message was received, the record is marked as unreachable. Otherwise the record is marked as updated and it will be broadcast in the next round. Finally, a message containing the failed node along with the list unreachable nodes is broadcast.

By requiring the distance from the failed node to increase we avoid loops and repeated re-broadcasting of the same message. By increasing the delay with the distance from the failed node we assure that a nodes closer to the failed node broadcasts their unreachable nodes sooner.

In order to store the routing table the node stores for each neighbour a list of $\langle destination, distance \rangle$ pairs where the neighbour is the next hop. The cost to store the RT is computed as $c_i = 2N + nb$, where N is the size of the network and nb is number of neighbours of the node n_i.

B. Distributed Data Table

Most of the current platforms for WLSS 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.

Each node in a network may have a set of *static attributes* assigned (e.g. id, type, room id). All static attributes in a network can be represented as a table where each column represents an attribute and each row represents a node. In summary, currently the strategies 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

Algorithm 2 Distributed Data Table

```
1:  procedure RECEIVEDDT(ddt)
2:      updateDdtCounter(ddt)
3:      updateDdt(ddt)
4:      if partId = null then
5:          ddtTimeSend ← currentTime+randomDelay()
6:      end if
7:  end procedure

8:  procedure SENDDDT(packet)
9:      partId ← partWithLeastNodes()
10:     ddt[partId] ← this.id
11:     broadcast(ddt)
12: end procedure
```

from all nodes, 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 got the specific attribute, iii) store information on a set of predefined node.

We take a form of the third approach, i.e. we store information about static attributes on nodes. Because WLSS nodes are very limited in terms of memory, fitting the whole table of static attributes on a single node may not be possible. However, if the table is split into p equally sized *parts*, each node needs only to store one part. In case some nodes have a larger memory, they may store several parts of the table. We refer to this distributed table as the *Distributed Data Table* (DDT).

When a node receives a query, e.g. `S.x > 25`, it first looks at its local DDT and then forwards the query to $p-1$ nodes which contain the rest of the table. These nodes search in their local copy of the table and reply with the *result only*. DDT introduces a challenge how to assign parts of the table to nodes 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. The minimum number of messages a node needs to send is $(p-1)$, however, only if the node has at least $p-1$ neighbours and each neighbour holds different part of the table. Obviously, this cannot be always achieved, especially in sparse networks where nodes have less than $p-1$ neighbours. In that case we also want to minimise the number of messages by having the nodes with the missing part of the table a minimal number of hops away.

If we think about each part as a colour, the objective of assigning DDT parts to nodes is similar to a graph colouring problem with two main differences: i) a node can have a neighbour with the same colour and ii) each node wants to reach all other colours within minimum number of hops.

Let p be the number of parts the DDT is split into and $partId \in \{0, \ldots, p-1\}$ be the ID of a DDT part. The basic idea of the algorithm is shown in Algorithm 2. Each node stores the following global variables: p - how many parts the DDT is split into, $partId$ - which part of the DDT the node stores (initially NULL), ddt - a vector of size p storing which node stores given part of the DDT, $ddtCounter$ - a vector of size p which stores how many neighbours store a given part of the table (initially all zeros), $ddtTimeSend$ - a time at which the node will choose its $partId$ and broadcast its ddt vector.

The algorithm is initiated by a random node which calls procedure SENDDDT. The key idea of the algorithm is that a node upon receiving a ddt (a list of $\langle partId, nodeId \rangle$) from a neighbour, waits for a random delay (line 5) during which it collects ddt from other neighbours. After this delay the node chooses the $partId$ which has been chosen least times (ties are decided randomly) by its neighbours (line 9). Next, the node broadcasts to all neighbours its ddt.

Algorithm 3 Static Attributes Propagation

```
1:  procedure RECEIVESA(staticAttr, senderId)
2:      sa ← retrieveFromBuffer(staticAttr)
3:      if sa = null then
4:          sa ← insertIntoBuffer(staticAttr)
5:          sa.sentAt ← currentTime+ randomDelay()
6:          sa.receivedBy ← getListOfNeigh(this.id)
7:          sa.sent ← False
8:          if mapToDDT(sa) = this.partId then
9:              insertIntoDDT(sa)
10:         end if
11:     end if
12:     removeFromList(sa.receivedBy, senderId)
13:     senderNeighbours ← getListOfNeigh(senderId)
14:     removeFromList(sa.receivedBy, senderNeighbours)
15: end procedure

16: procedure SENDSA(sa)
17:     if sa.receivedBy ≠ null then
18:         broadcast(sa)
19:     end if
20:     sa.sent ← True
21:     removeFromBuffer(sa)
22: end procedure
```

Currently, dynamic scalability is not supported and p has to be chosen at deployment time. It cannot be changed without re-initialising the bootstrapping phase.

C. Static Attribute Propagation

Dragon stores all Static Attributes (SA) in the DDT. However, the DDT has to be filled with data prior to using it. Using a traditional data dissemination protocol like Drip [16] to disseminate SA about every node to every other node leads to exchange of at least N^2 messages, where N is the size of the network.

However, this number could be significantly reduced using algorithm described below (Alg. 3). Every node stores for each of its neighbour a list of common neighbours. When a receiving node n_r receives a list of static attributes sa from a sending node n_s for the first time (line 3), 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 random delay after which the n_r will broadcast the received sa. Now, we can assume that all n_s's neighbours have also received the sa, so we can remove n_s and all common neighbours with n_s from the ln. If n_r has already received the sa before, n_r just removes n_s and all n_s's neighbours from the ln (lines 12-14).

Once a random delay has expired, n_r is ready to broadcast the sa by calling SENDSA. Prior to broadcasting, n_r checks the ln (line 17). If the ln is empty, i.e. all n_r's neighbours have received the sa from other nodes, the n_r removes the sa from the buffer without broadcasting the sa. If the ln is not empty the node broadcasts the sa (line 18). The delay is chosen randomly in order to avoid all nodes broadcasting at the same time.

Due to the unreliability of wireless communication it may happen that a node (n_r) does not receive the list of SA for some node n_x. In this case, the node asks for the missing data from the nearest node storing the same part of DDT. If that node also missed the data, n_r requests data directly from n_x.

IV. EVALUATION

We have evaluated various parts of our platform in the TinyOS simulator TOSSIM [17]. TOSSIM was chosen because

of its high accuracy in the simulation of real WSNs and it is used by many researchers. We have used the built-in radio and noise model. We assume the nodes are synchronised and operate with duty cycle of 15%.

All of the experiments presented in this section were run 10 times on each of 10 different networks for each network density, and the results are grouped by the network density. The packet size was set to 21 bytes which is the standard packet size used in TinyOS. We evaluated our experiments using 100 and 250-node networks of various densities: i) dense (D, with 12 neighbours on average), ii) medium dense (MD, 10 neighbours), iii) medium sparse (MS, 7 neighbours), and iv) sparse (S, 5 neighbours). For each density ten random networks were generated.

A. Routing Table Discovery

In the evaluation of the Routing Table Discovery algorithm we focus on the *routing stretch*, i.e. deviation of the discovered paths from the optimal in terms of length. We compare four versions of routing protocols: using one routing tree (e.g. RPL), using several routing trees (e.g. Innet), hierarchical routing, and Dragon.

Experiments were executed on 100 and 250-node networks of various densities. In case of hierarchical routing we use figures from Iwancki and van Steen [14] who claim that the average routing stretch of the hierarchical routing algorithm is 25%. The routing stretch of algorithms relying on one routing tree is $50-105\%$, depending on the density. For this we used an implementation which assumes that every node in the network is a router, so in implementation where the router is only the base-station, the stretch would be even bigger. The routing stretch of the algorithm based on more routing trees (three in case of Innet [10]), differs from 5% for a dense network to as much as 46% for a sparse one. Dragon has achieved the routing stretch of only 0.5% in every density and is therefore close to the optimal.

It is important to note, that after the bootstrapping Dragon and Hierarchical routing can start routing packets immediately, while the paths in the platforms based on trees must be first discovered. The discovery phase requires additional messages being sent, hence these platforms are not suitable for ad-hoc communication.

B. Distributed Data Table and Static Attribute Propagation

We compare the heuristic algorithm presented in Section III-B with random assigning parts to each node. As data processing will shift from servers and base-stations into the network, looking up nodes with specific static attributes will be a significant part of the overall network traffic. Therefore, any improvement can lead to a significant reduction of the overall traffic. We compare the average number of messages a node has to send in order to search in the whole table while the DDT is split into $5-10$ parts. Nodes in a network using heuristic algorithm to assign parts to nodes need to send up to 22% less messages when compared with random assignment.

We also compare propagation of Static Attributes (SA) using the algorithm presented in III-C with an implementation of a traditional small data dissemination protocol Drip [16]. Unlike Drip, which re-broadcast the message several times, our implementation re-broadcast every message only once. Even after this optimisation our dissemination protocol achieves savings between $35-59\%$ in terms of messages and $16-35\%$ in terms of time, depending on the network density. Savings are smaller for sparse networks as there are less common neighbours, hence a higher probability that a node has to re-broadcast the message.

C. Sources Discovery

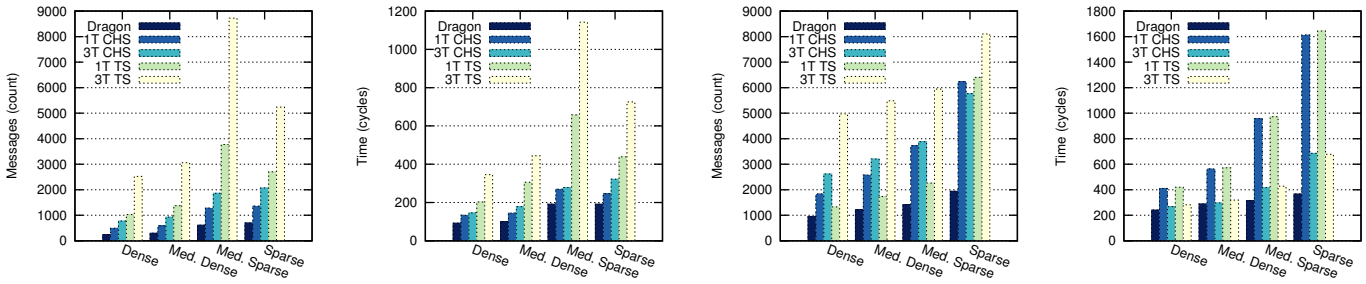
Each node in the network has six static attributes assigned: id - a unique identifier, x - a random uniformly distributed variable, $x \in (0, 10)$, y - an exponential variable with $\lambda = 0.05$, z - an exponential variable with $\lambda = 0.1$, and $coord_x, coord_y$ - virtual coordinates of the node.

We evaluate the ability of any node in the network to find a list of nodes with certain static attributes and request data from them. Two approaches are compared: using DDT and summaries. Using summaries, unlike approaches based on flooding the network (e.g. [5], [9]), can significantly lower the traffic by pointing the search towards the nodes with given static attributes. During the bootstrap phase, static attributes of every node have been propagated throughout the network and stored in the DDT. In the case of summaries, attributes id, x, y, z were stored using Bloom filters and count histograms, while $coord_x, coord_y$ were stored using an R-Tree. Using both the Bloom filter and histogram summary allows nodes to better answer a wider range of queries. As the Bloom filter can only be used to check whether a given value was added to the filter, it is useful for queries using equal comparison. In case a user submits a query with a range of values Bloom filters are ineffective and histograms have proven to be a better option. However, this comes at the price of higher memory requirements.

It is also important to decide whether a node stores one summary for all children or a separate summary for every child. Again, storing only one summary saves memory but leads to a higher network traffic as the message is forwarded to all children, not only to a subset of them. In our case, having nine 16 byte summaries for every child (with 6 children on average) in all of the three trees will require $9 \times 16 \times 6 \times 3 = 2592$ bytes of memory. Storing data in the DDT with 25 records per part will require only $25 \times 6 = 150$ bytes. Additionally, the routing table (storing for every neighbour a list of node id along with a number of hops to given node) will require $2 \times 250 + 6 = 506$ bytes. Therefore, unlike Innet, Dragon is able to scale down to the Berkley-size nodes.

In the evaluation we compare Dragon with a platform using one or three summary trees (in Figures marked as "1T", "3T" respectively). Each node in the tree stores either one summary (which includes all children) or a separate summary for each child (marked as "CHS" for "child summary" and "TS" for "tree summary"). Using one tree imitates behaviour of networks where the base-station has a global knowledge of the network and can deliver the query only to relevant nodes only [6]. Using three trees was shown to be an optimal balance between memory requirements and optimality of discovered paths [10].

We evaluate Dragon on 250-node networks using following scenarios. At a random node the following *Query 1* is executed: `SELECT MAX(S.freespace) FROM Shop S WHERE S.x = 1`. This is a type of query a user would submit if she wanted to find a pub on the high street which has the most free space. The query resulted in finding a variable sized list of nodes ranging from 0 to 11. In case the list of nodes is not empty, every node in the list is requested to send current sensor readings and the maximum is found at the node



(a) Query 1 (b) Query 2

Fig. 1: Message and Time Complexity

which initiated the query. In the second scenario we evaluate a *Query 2:* `SELECT MAX(S.freespace) FROM Shop S WHERE S.x = 2 AND S.z > @val`, where `@val` is a random number. This is a type of query a user would submit in case she wanted to find a restaurant with the most free space which is still open. The query resulted into receiving data from $4-16$ nodes. It is important to note that in case of the platform based on summaries the buffers had to be doubled due to congestion around some of the nodes leading to packets being lost.

Evaluation of these queries is shown in Figure 1. The y-axis in the graphs represents the sum of averages for each network with given density. Our focus is on two metrics: i) number of messages sent and ii) time it takes to find relevant nodes. As it can be clearly seen from the figures comparing messages, the improvement of Dragon does not depend on the network density and remains approximately same when compared with the same alternative method. In the case of Query 1 (Query 2 respectively) savings between $48-93\%$ ($28-81\%$) are achieved. When the search time is compared it is possible to see that Dragon performs better as the network gets more sparse. In the case of Query 1 we can see significant decrease in terms of search time where Dragon is $22-83\%$ faster while in the case of Query 2 the Dragon decreases delay by $3-78\%$.

V. Conclusion

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

In this paper we presented Dragon - a platform allowing any node in the network to easily and efficiently find a list of nodes with given static attributes while requiring only a very limited amount of memory. These attributes are stored in a distributed way throughout the network using a Distributed Data Table. Any node in the network can easily search in this table while communicating only with a close neighbourhood. We also present a distributed algorithm for Routing Table discovery. 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. We compared Dragon with the state-of-the-art approaches and achieved messages reductions of up to 93% and the response time improvement up to 82%.

In the future we will investigate possibilities of in-network processing of continuous queries in homogeneous as well as heterogeneous networks.

Acknowledgement

The authors would like to thank the authors of Innet [10] for providing us with the source code of Innet so our comparison is full and fair.

References

[1] "Abi research: More than 30 billion devices will wirelessly connect to the internet of everything in 2020," May 2013, https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne.

[2] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, "Ght: A geographic hash table for data-centric storage," ser. WSNA '02. New York, NY, USA: ACM, 2002, pp. 78–87.

[3] B. Greenstein, S. Ratnasamy, S. Shenker, R. Govindan, and D. Estrin, "Difs: a distributed index for features in sensor networks," *Ad Hoc Networks*, vol. 1, no. 23, pp. 333 – 349, 2003, sensor Network Protocols and Applications.

[4] H. Kang, "In-network processing of joins in wireless sensor networks," *Sensors*, vol. 13, no. 3, pp. 3358–3393, 2013.

[5] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," ser. WMCSA '99, Feb 1999, pp. 90–100.

[6] M. Stern, E. Buchmann, and K. Böhm, "Towards efficient processing of general-purpose joins in sensor networks," ser. ICDE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 126–137.

[7] S. R. Mihaylov, M. Jacob, Z. G. Ives, and S. Guha, "A substrate for in-network sensor data integration," ser. DMSN '08. New York, NY, USA: ACM, 2008, pp. 35–41.

[8] F. Zhao and L. Guibas, *Wireless Sensor Networks: An Information Processing Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[9] P. Ciciriello, L. Mottola, and G. P. Picco, "Building virtual sensors and actuators over logical neighborhoods," ser. MidSens '06. New York, NY, USA: ACM, 2006, pp. 19–24.

[10] S. R. Mihaylov, M. Jacob, Z. G. Ives, and S. Guha, "Dynamic join optimization in multi-hop wireless sensor networks," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1279–1290, Sep. 2010.

[11] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 1–14.

[12] O. Gaddour and A. KoubíA, "Survey rpl in a nutshell: A survey," *Comput. Netw.*, vol. 56, no. 14, pp. 3163–3178, Sep. 2012.

[13] B. Karp and H. T. Kung, "Gpsr: greedy perimeter stateless routing for wireless networks," ser. MobiCom '00. New York, NY, USA: ACM, 2000, pp. 243–254.

[14] K. Iwanicki and M. van Steen, "On hierarchical routing in wireless sensor networks," ser. IPSN '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133–144.

[15] W. D. Tajibnapis, "A correctness proof of a topology information maintenance protocol for a distributed computer network," *Commun. ACM*, vol. 20, no. 7, pp. 477–485, 1977.

[16] G. Tolle and D. E. Culler, "Design of an application-cooperative management system for wireless sensor networks." in *EWSN*, vol. 5, 2005, pp. 121–132.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 126–137.