Optimal Processing Node Discovery Algorithm for Distributed Computing in IoT

Roman Kolcun*†, David Boyle* and Julie A. McCann†
*Department of Electrical and Electronic Engineering, Imperial College London
†Department of Computing, Imperial College London
Email: {roman.kolcun, david.boyle, j.mccann}@imperial.ac.uk

***Abstract*—The number of Internet-connected sensing and control devices is growing. Some anticipate them to number in excess of 212 billion by 2020. Inherently, these devices generate continuous data streams, many of which need to be stored and processed. Traditional approaches, whereby all data are shipped to the cloud, may not continue to be effective as cloud infrastructure may not be able to handle myriads of data streams and their associated storage and processing needs. Using cloud infrastructure alone for data processing significantly increases latency, and contributes to unnecessary energy inefficiencies, including potentially unnecessary data transmission in constrained wireless networks, and on cloud computing facilities increasingly known to be significant consumers of energy. In this paper we present a distributed platform for wireless sensor networks which allows computation to be shifted from the cloud into the network. This reduces the traffic in the sensor network, intermediate networks, and cloud infrastructure. The platform is fully distributed, allowing every node in a homogeneous network to accept *continuous* queries from a user, find all nodes satisfying the user's query, find an optimal node (Fermat-Weber point) in the network upon which to process the query, and provide the result to the user. Our results show that the number of required messages can be decreased up to 49% and processing latency by 42% in comparison with state-of-the-art approaches, including Innet.**

I. Introduction

As the number of Internet connected devices is increasing and penetrating our day-to-day lives, the amount of data produced by these devices will grow. It is estimated that by 2020 the number of Internet-connected devices will be between 30 billion [1] and 212 billion (including 30.1 billion autonomous things), with a market value of $8.9 trillion [2]. To process these vast amounts of data streams, new techniques are being researched. For example, a recent protocol called Constrained Application Protocol (CoAP) [3] was designed to represent a bridge between a WSN and the Internet. Kovatsch *et al.* presented *Californium* [4] - a scalable cloud service capable of handling hundreds of thousands of concurrent CoAP connections between the cloud and many Wireless Sensor Networks (WSNs).

However, collecting all the data from the network in the cloud may not always be the best solution. Not only may the network itself not be able to transfer all data to a base-station, the link between a base-station and a cloud may not exist, be insufficient, or may be extremely expensive. Additionally, shipping data to the cloud will increase the delay between the occurrence of a phenomenon and its detection. Furthermore, data centres are already responsible for 1.4% of World-wide energy consumption, growing by 12% every year [5], excluding power consumption of the network infrastructure that scales accordingly.

In response to these challenges, companies like Cisco are introducing concepts such as Fog computing, bringing computation from the cloud to the edge of the network. This may lead to shorter delays and reduce the traffic to and/or from the cloud. Edge-network devices through which the sensor nodes connect to the Internet are represented by base-stations, routers, etc. Data from the network are collected at such edge devices, which either (pre-)process the data, and may act upon detected phenomena. The goal is to reduce delays and networking requirements.

However, Fog computing assumes the network consists of large numbers of small devices that communicate with the outside world through base-station(s) that act as gateways. We argue that as the number of deployed networks increases, this type of traditional network with base-stations might not scale and users will start to interact directly with the network. For example, we can think of networks deployed in houses, buildings, streets, shopping centres, etc. Network owners may not be willing to pay for the Internet connection or the associated cloud infrastructure, yet they still want to provide services of their networks to the users.

Consider following scenario: *In the future, every building has several sensor equipped bins used for various types of communal waste, e.g. glass, paper, plastic, etc. Trucks collect only one type of waste. To optimise resources (e.g. fuel consumption, truck utilisation), a truck is instructed to collect waste from a street only if the bins on that street are, on average, more than 50% full.*

Using a cloud approach, all sensor readings would be sent to the cloud at predefined intervals, then grouped by the street, the type of waste, etc. If the average is above the threshold, a message is sent to the company collecting the waste. An alternative approach is, if one sensor is chosen to collect data for given street and type of waste only, the average is computed locally and the company informed only if the average is above the threshold. The node does not require an Internet connection, and may use a passing bus, for example, as a mule to deliver the message.

We present a new platform for efficient in-network data stream processing, where the contributions are as follows:

- Processing Node Discovery (PND) algorithm for continuous queries, which finds an optimal processing node in the network in a fully distributed way (Section III);
- Query Tuple Buffering (QTB) optimisation which further reduces the network traffic;

- Evaluation of the algorithm on networks of various topologies and densities (Section IV).

II. Rationale & Related Work

Contemporary WSN designs consist of one (or more) base-station(s) and many sensing nodes. Data are delivered to the base-station via multi-hop communication using CTP, RPL, etc. The base-station serves as a gateway to the network, and may offer network's capabilities to a user. Users usually communicate with the base-station via some long-range communication link. The base-station may also represent a single point of failure, i.e. when it fails the network cannot send data to the cloud and users cannot query the network.

As wireless communication is typically extremely expensive in terms of energy (more than 80% of energy is used on communication) many techniques have been proposed to reduce the number of messages required in the network to deliver a result to a user.

When query processing is optimised, several important requirements must be considered. The query is either *universal*, i.e. requires data from every node in the network, or a *subset query*, i.e. requires data from a subset of the nodes only. We focus our research on *subset queries* only, as they offer wider possibilities for query optimisation techniques. We must consider whether user requires current readings, or can be satisfied with an approximate reading with given a degree of confidence [6]. In this case the query can be optimised using various summary structures like histograms or Bloom filters. However, this is not the focus of our research and we assume that a user requires fresh readings every time the query is evaluated. We must consider whether the platform will accept aggregation queries only (like AVG, MAX, MIN [6]) and/or projection queries. The final requirement that must be considered is the duration of the query. A *snapshot* query is executed only once and therefore offers very small room for query execution optimisation. The optimisation is focused on inexpensive identification of relevant nodes and retrieving data from them as the query optimisation overhead could easily exceed the gain of this optimisation [7]. On the other hand, *continuous* queries allow much wider query optimisation possibilities as the query is executed many times over a specified period of time (possibly indefinitely). Therefore, the query optimisation overhead is mitigated by executing the query many times.

A. Related Work

Several approaches have been proposed to support continuous queries. The simplest variant is processing data *at-the-base*, where only the nodes contributing to the query send data to the base-station which processes them. Another approach, proposed by Chowdhary and Gupta [8], targets special cases where data from exactly two non-overlapping regions need to be processed and the network supports geographical routing. In this case, a node in the *geographical centre* of a triangle formed by two regions and the base-station is chosen as the processing processing node. Similarly, if the data sources are scattered throughout the network, Pandit and Gupta propose to choose a random node using the *hash of the join key*, as the processing node [9]. Stern *et al.* propose *Continuous Join Filtering (CJF)* - a two-phase approach where first summaries of static and dynamic attributes of all nodes in the network are collected, then the base-station uses these summaries to choose candidates for the join [10]. In the second phase, filters generated by the base-stations are pushed back into the network, and only nodes whose sensed data passes this filter send data back to the base-station [10]. The final join of data streams is performed at the base-station. CJF focuses on optimising join queries which join data based on *dynamic attributes*, i.e. sensed data. Mihaylov *et al.* propose a framework based on *pairwise joins* which splits the processing into pairwise joining, and for each join pair, finds a node on the path between them which processes data [11]. This approach can significantly reduce the number of messages but only where the selectivity of the pairwise join, i.e. the percentage of tuples fulfilling the join prediction, is very low. The approach also assumes that the computation can be split into pairwise joins. Pairwise join operates on exactly two streams of data and produces a partial result only. The final join is carried out at the base-station. Mayer *et al.* proposed a solution for searching in a web-based infrastructure for smart things [12]. Their approach relies on a strictly hierarchical tree-like structure, introducing a caching optimisation that uses a top-down approach where a node requests data from its children and waits until all children reply. Abrams and Liu present a *Greedy is Good (GIG)* algorithm for finding a processing node inside the network by controlled flooding from each source of the network [13]. This approach is expensive in terms of messages required to find the processing node, where the discovered processing node could be eight times more expensive than the optimal one. Chatzimilioudis *et al.* propose *distributed Fermat node search (dFNS)* algorithm, which is also based on flooding the network, however, the flooding is more localised when compared to GIG and it is able to find processing nodes which are closer to the optimal one [14]. Their approach is, however, optimised for uniform networks and for joining small number of sources that are close to each other. SNEE, a framework described by Galpin *et al.* [15], is capable of in-network processing, but the base-station must have global knowledge of the network to assign operators to the nodes.

The disadvantages of these approaches are that they either heavily rely on the base-station (i.e. traditional at-the-base processing, e.g. the two-phase approach, SNEE, and the second half of the pairwise join) or on geographical routing (e.g. geographical and hash), which has many known disadvantages. Other approaches either support only special types of queries, are optimised for special types of networks and/or scenarios (GIG, dFNS), find non-optimal processing nodes, or have an expensive set-up phase. Our work attempts to deliver a flexible, scalable, decentralised, and efficient way to find an optimal processing node in a network of any topology and density.

III. Computational Platform

Our platform for in-network processing of continuous queries extends our previous work on DRAGON [7]. We briefly review the main features of DRAGON and explain its functionality.

DRAGON is a platform supporting peer-to-peer communication between any two nodes in the network via near-optimal routes. Each node stores a *routing table* (RT) containing a distance and a next hop neighbour to every other node in the network. A table, which contains a list of all static attributes describing every node in the network is distributed throughout the network. Each node stores only a fraction of this table. These fractions are assigned to nodes in such way that every node can reach all nodes storing the other parts of the table by communicating within a close neighbourhood. Thus, every node can search this table (referred to as *Distributed Data Table* (DDT)), i.e. find the list of all nodes in the network satisfying given static attributes, with low overhead. No central node is needed, making the system fully decentralised, thus robust against node failures. Therefore, DRAGON is optimised to answer snap-shot queries while capturing the current readings and state of the network. The node that receives the query processes the data and reports the result to the user.

One disadvantage of DRAGON is its inability to efficiently evaluate *continuous* queries. For the scenario in Section I, a user needs to repeatedly measure "fullness" of bins of specific types at specific streets and be notified once, on average, the bins are more than 50% full. In this case, processing data on the node which accepted the query may not be the most energy efficient, as the sources of the data for the query may be far away from the node through which the user submitted the query. In this case, processing on a node whose distance to all the sources of data streams is much shorter than the distance from all the sources to the initiating node, might be more energy efficient. The energy savings will be much higher than the energy required to find such a node. By allowing a user to submit a query via any node we eliminate the need for a central node, and achieve a fully distributed solution without a single point of failure. The rest of this section describes the process of finding a processing node with the shortest distance to every node producing data satisfying a given query.

A. Processing Node Discovery (PND) Algorithm

Assume that a user can communicate with any node in the network and submit queries. The node that receives a query from the user is referred to as the *initiator* or the *initiating node*. As soon as the initiator receives a query, it uses DRAGON to identify all the nodes participating in the given query. The initiating node searches the DDT for other nodes which satisfy static attributes of the query. These nodes are referred to as *sources*. Each source produces data at a certain rate. This rate depends on the sampling rate and the dynamic condition specified in the query. For example, the dynamic condition may look like: `WHERE capacity > 75`. Here, the sensor node sends the data tuple only if the sensed remaining capacity of the bin is higher than 75%. Let us define *selectivity*, denoted σ as:

$$\sigma = \frac{\text{tuples sent}}{\text{tuples sampled}} \tag{1}$$

Because selectivity has an impact on the position of the processing node, the initiator sends the list of dynamic conditions of the query to all of the sources. This list of dynamic conditions is used by the source node to compute its selectivity for given query. Where a source can compute its selectivity (e.g. using stored historic data or a histogram), it reports the selectivity to the initiator. Otherwise, the source node assumes that the selectivity for given query is $\sigma = 1$. After collecting selectivity from every source, the initiator starts a search for a node that can process the data streams. We refer to this node as the *processing node*.

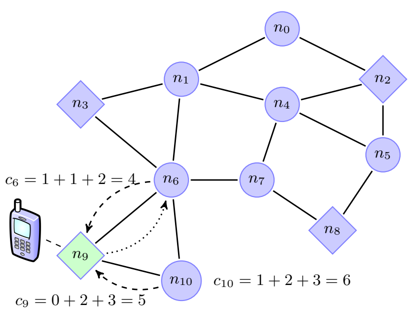
We define the *cost* of processing all sources S with selectivity σ at node i as

$$c_i = \sigma_S r_i + \sum\nolimits_{j \in S} \sigma_j d_{ij} \tag{2}$$

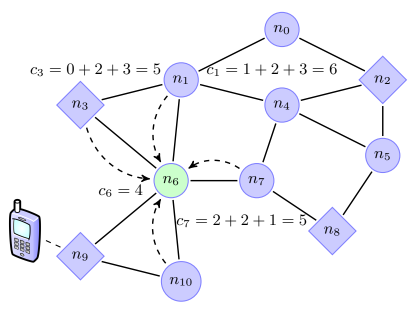
where r_i is the number of hops between node i and the node to whom the final result should be reported (referred to as *report node*), d_{ij} is the number of hops between nodes i and j, σ_j is the selectivity of the node j, and σ_S is the selectivity of the processing node. The lower the cost is, the fewer messages are sent within the network in order to process data streams from all sources.

Our platform relies on a reliable end-to-end communication. One hop reliable communication requires an acknowledgement sent by the receiver to the sender. Therefore sending a message h hops away leads to exchange of $2h$ packets. However, the receiver can merge the acknowledgement and the forwarded message into a single packet. The sender receives the acknowledgement by overhearing the receiver's communication. Only the destination node, which does not forward the message sends a separate acknowledgement packet. Hence, sending a message h hops away leads to exchange of only $h + 1$ packets. The cost defined above does not take into account this additional acknowledgement packet sent by the last hop, therefore in addition to the *cost* we also define the *real cost*:

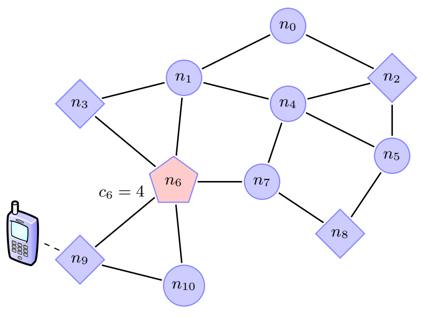
$$rc_i = \sigma_S rr_i + \sum\nolimits_{j \in S} \sigma_j dr_{ij} \tag{3}$$



(a) The search is initiated by an engineer over a cell phone. The first coordinator is the node that received the query.



(b) The search follows the cost gradient. The next coordinator becomes the node with the lowest cost (n_6).



(c) Because there is no neighbour with a lower cost, node n_6 declares itself as the processing node.

Figure 1: Processing Node Discovery Algorithm. The search follows the steepest cost gradient. Once a node whose cost is lower than the cost of all its neighbours, the node declares itself as the processing node.

where

$$dr_{ij} = \begin{cases} 0, & \text{if } i = j \\ d_{ij} + 1, & \text{if } i \neq j \end{cases}$$

and rr_i is the number of messages required to reliably deliver the result to the reporting node. In cases where the processing node is detecting a rare event (σ_S is close to 0), the $\sigma_S rr_i$ is negligible and does not contribute to the overall cost.

The difference between the *cost* and the *real cost* is that the real cost prefers source nodes to non-source nodes, therefore several nodes with the same cost may have different real costs. Assume there are two source nodes s_1, s_2. If we do not take the distance to the *report node* into account, all nodes on the shortest path between s_1 and s_2 (including the source nodes) will have the same minimal *cost*. However, the *real cost* of the source nodes will be lower than the *real cost* of the nodes on the path between these two source nodes. Unfortunately, the *real cost* deforms the search space which can lead to creating local minima at the source nodes. Therefore we use the *cost* in our Processing Node Discovery algorithm and the *real cost* during evaluation in Section IV as it better reflects the real traffic in the network.

The objective of the algorithm is to find a node whose sum of weighted distances to all source nodes is minimised. From geometry, this problem is known as the *geometric median* or *Fermat-Weber* problem. The geometric solution is known only for three nodes. There is no general solution for this problem for n ($n > 3$) nodes, only numerical or symbolic approximations are possible.

Approximations are based on the fact that, since the distance to a single point is a convex function, the sum of distances from a single point to all source nodes remains a convex function. If the algorithm decreases the cost in each step it will eventually reach the global minimum.

Algorithm 1 iteratively decreases the cost in each iteration by following the cost gradient towards the node with the lowest *cost*. The functioning algorithm is graphically

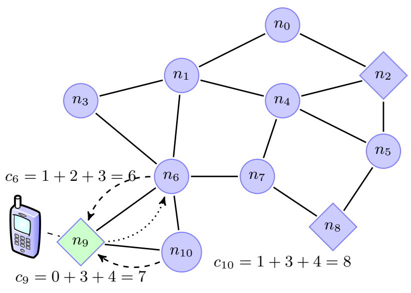
Algorithm 1 Processing Node Discovery

```
1: procedure ReceiveAssignment(query)
2:     compute the cost for query
3:     request a cost from every neighbour
4:     if local cost is the lowest or this node was a coordinator then
5:         declare this node as the join node
6:     else
7:         from the list of nodes with the lowest cost randomly choose
   one node and send an assignment to the node
8:     end if
9: end procedure
```

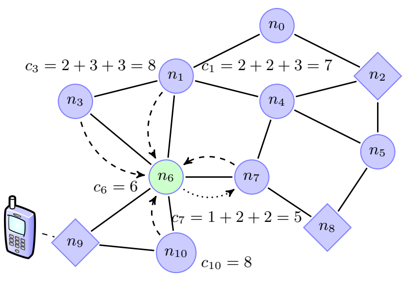
illustrated in Figure 1. Source nodes are diamond shaped, regular nodes are circles, and the processing node is a polygon. The algorithm consists of rounds, each of which is led by one *coordinator*, shown in green in the figure. At the beginning the node which received the query from a user, the *initiator*, becomes the first coordinator (Figure 1a). The coordinator computes its cost (Alg. 1, line 2) and broadcasts the cost to all its neighbours, which reply with their cost for the query. The cost is computed by looking up the distances to every source node in the routing table stored at every node. In Figure 1a the replies from nodes n_6 and n_{10} are shown with dashed arrows. Once the coordinator receives a reply from every neighbour, it compares its cost with all the received costs (line 4). If there is a node with a lower cost, the coordinator sends it an *assignment* message (line 7) and the receiver becomes a coordinator for the next round. In Figure 1a, node n_9 with $c_9 = 5$ sends an assignment message to node n_6 with $c_6 = 4$ as its cost is lower. The assignment message is depicted with a dotted arrow. If there are several nodes with the same lowest cost the next coordinator is chosen randomly.

If all coordinator's neighbours' costs are higher, the coordinator declares itself as the *processing node* (line 5). In Figure 1b, node n_6 receives costs from all neighbours. Because all received costs are higher than 4, the node declares itself the processing node, illustrated in Figure 1c.

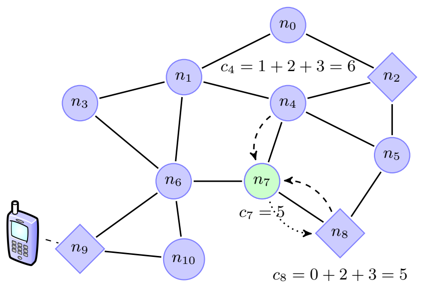
It may happen that the cost gradient is lost when the search hits an area of nodes with the same lowest cost. This



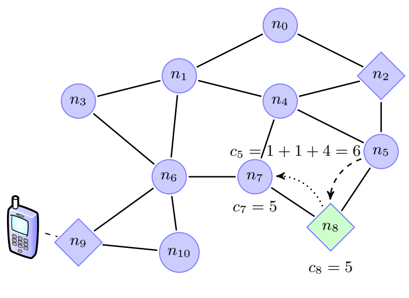
(a) The search is initialised by an engineer via a cellphone.



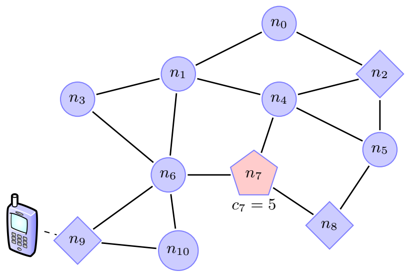
(b) Node with the lowest cost (n_6) is the new coordinator.



(c) The search follows the cost gradient to node n_7.



(d) The node with the same cost (n_8) becomes a new coordinator.



(e) The search is returned back to node n_7 which declares itself as the processing node.

Figure 2: Processing Node Discovery (PND) Algorithm. Sometimes the search hit a neighbourhood of nodes with the same cost and the gradient is lost. In this case the search by a random walk is executed. Data sources are diamond shaped while the processing node is polygon shaped. Coordinator in given round is showed in red.

situation is shown in Figure 2. The search follows the cost gradient from node n_9 to node n_6, and then to node n_7. At this point, shown in Figure 2c, the coordinator cannot find a node with a lower cost. However, node n_8 has the same cost as node n_7, therefore an assignment message is sent to node n_8. The same situation happens when node n_8 is the coordinator, and it assigns node n_7 as the coordinator again, as depicted in Figure 2d. On receiving an assignment, the node checks whether it has already been a coordinator for the given query before (line 4). If so, it means that there is at least one node with the same cost which was delegated as a coordinator but it was unable to find a node with a lower cost, therefore the assignment was returned back to the previous coordinator. In this case the node declares itself as the processing node (line 5) in order to avoid loops in the search. This process can be seen in Figure 2e.

In rare cases it may happen that the search reaches a neighbourhood of nodes with the same cost and the search terminates before the node with the lowest cost in the network is found. It is partially compensated for by the fact that a new coordinator is chosen also in the case where the neighbour's cost is the same, i.e. the requirement that the cost of the new coordinator must be lower than the current one is relaxed. However, if there are more neighbours with the same cost, the new coordinator is chosen randomly and it may not be on the path to the node with the lowest cost.

This problem could be solved by performing an exhaustive search, i.e. choosing multiple nodes with the lowest cost as coordinators. Unfortunately, this approach would have a drawback of significantly increasing the number of messages exchanged during the discovery phase.

Once a node decides to declare itself as a processing node, it informs all sources participating in the query of its ID and its cost. Because each round is led by only one coordinator, each source can receive a notification from one processing node only.

The numbers of messages exchanged and coordinators during the discovery phase mainly depends on the the number of coordinator's neighbours, as the coordinator requires a reply from every neighbour.

The number of coordinators depends on the sparsity of the network, i.e. the maximum distance between two nodes in the network $d_{max} = \lceil N/Nb \rceil$, where N is the number of nodes in the network and Nb is the average number of neighbours. On average, the number of coordinators is $cd = d_{max}/2$. The number of messages is proportional to the number of coordinators, computed as $cd \times Nb$.

The number of messages can be significantly decreased by *snooping* on neighbours as they reply to a coordinator. Each node stores these replies, and in the case that it becomes a coordinator for the next round the node requests costs only from neighbours for which it is missing the cost. In Figure 2b node n_6 may request the cost only from nodes $n_1, n_3,$ and n_7. Node n_6 has already received the cost of node n_{10} previously, when node n_9 was the coordinator and node n_{10} reported its cost to node n_9 (Figure 2a).

We assume that every node in the network is capable of processing data streams from all the source nodes. Memory requirements of a processing node may be computed as $m = w \times vs \times ns$, where w is the size of the window (i.e. number of values stored for each node), vs is the number of bytes required to store the value, and ns is the number of bytes required to store the node ID.

B. Query Tuple Buffering Optimisation

Each epoch every source node senses the required value and sends the query tuple to the processing node. As the messages are being forwarded towards the processing node, they may pass via a common node. This node can, instead of forwarding each message separately, merge two or more query tuples into a single message. We refer to this node as a *merging node*. We have implemented and evaluated the Query Tuple Buffering (QTB) optimisation which merges several query tuples into a single message, hence reducing network traffic.

As the distance from the merging node to the source nodes varies, the merging node may need to wait for all the query tuples it is merging to arrive. The *maximum delay*, i.e. the largest difference between arriving of the first and the last query tuple in the same epoch, is continuously monitored, and if a query tuple does not arrive within this delay all query tuples in the merge buffer are sent to the processing node. The merging node must act as the last hop in the forwarding chain and send the acknowledgement to every node from which it received a query tuple. As the merging node can also merge several acknowledgements into a single packet, it is more energy efficient to wait for all the query tuples to arrive, or the maximum delay to expire, before sending acknowledgements. However, increasing the waiting period before the message is re-sent may have a negative impact on the overall delay of the query processing. The sending node has no way to know whether the receiving node is waiting for other tuples to arrive or it has not received the message, in which case the message must be re-sent.

IV. Evaluation

A. Setup

We evaluated the PND algorithm in the TinyOS simulator TOSSIM [16], which was chosen because of its reasonable accuracy in the simulation of real WSNs, and its popularity in the WSN research community. Furthermore, the platform we build upon, Dragon , and the appropriate platform with which we comparatively evaluate our work, Innet, are also programmed in TinyOS and evaluated using TOSSIM. We use the in-built radio and noise models, assume the nodes are synchronised, and can operate at 15% duty cycle. The packet size was set to 30 bytes.

We evaluate our platform on two different types of topologies: a uniform and a random topology. For each type of topology, networks with 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 network density, three different 250-node networks were generated. On each network, 10 different experiments were executed. Results are grouped by network topology and network density.

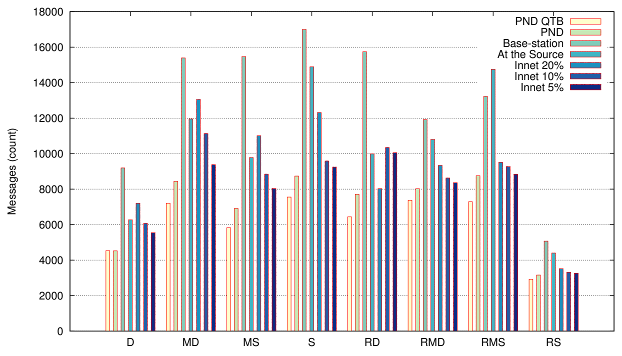
To evaluate in-network data stream processing capabilities of our platform, we revisit the scenario described in Section I. Each bin is uniquely identified by ID, and has a sensor measuring the free capacity of the bin. Additionally, each bin has two static attributes assigned - the ID of the street the bin is on (attribute `x`) and the type of waste the bin is for (attribute `y`). Then, the query which will compute average free capacity of the bins of specific type on the same street and reports it to user if the average capacity is less than 50% may look as follows:

```
SELECT AVG(S2.capacity) AS avg_capacity
FROM Sensors S1, Sensors S2
WHERE S1.x = S2.x AND S1.y = S2.y
AND S2.capacity < 100
HAVING avg_capacity < 50
EVERY 60 SECONDS}.
```

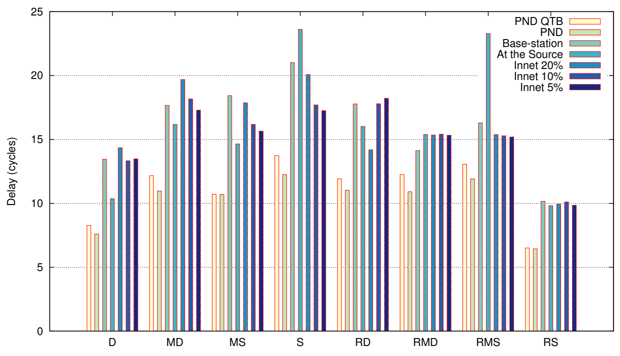
In our evaluation we focus on two metrics: *i*) the overall number of messages and *ii*) the processing delay. While the first metric is a proxy for energy efficiency, the latter shows how fast the network can react to the monitored phenomenon. The sooner source nodes can deliver data to the processing node, the faster the processing node can react to and act upon the input.

We evaluate two versions of our platform (with and without QTB optimisation) against three different approaches: *i*) process at-the-base, *ii*) process at the source node, and *iii*) pair-wise joining with three different join node selectivities. Processing at the base-station is the simplest, and therefore most commonly used solution. We have included this approach to show a baseline for other approaches. Processing data at a source is similar to the processing at-the-base, with the difference that data streams are processed at one of the source nodes. This strategy may decrease the network traffic because one of the source nodes is not required to transfer data to any other node and can process them locally. An alternative is processing on a random node in the network. The node could be chosen using, for example, a hash function [8], [9]. However, it has been shown that processing data on a random node leads to higher network traffic than processing at-the-base [11].

The last approach, and the state-of-the-art algorithm for distributed in-network data stream processing, is an implementation of the pair-wise join algorithm, Innet [11], [17]. The pair-wise join, as its name suggests, joins exactly two streams of values, i.e. one of the sources joins its data with all other data sources. The location of the join node depends on selectivity of the two source nodes and the



(a) Number of messages sent.



(b) Delay in tuple processing.

Figure 3: Comparison of Processing Node Discovery (PND) algorithm with and without the Query Tuple Buffering (QTB) optimisation with various in-network processing algorithms. The x-axis shows various network topologies.

selectivity of the join node. Pair-wise join can reduce the network traffic only if the selectivity of the join node is low. Innet periodically compares the cost of in-network pair-wise processing with processing at the base, and chooses the one with the lower cost.

It is important to note that pair-wise joining produces only partial results. If the join condition is not met, the pair is discarded, otherwise it is sent to a base-station (or any other common node) which collects all joining pairs from the whole network and performs final processing. For comparison we used pair-wise joining with three different selectivities of the pair-wise join nodes: $5\%, 10\%,$ and 20%. Where the selectivity of the join node is higher, Innet automatically switches to processing at-the-base. Using the pair-wise selectivity, it is possible to compute the overall selectivity σ of the processing node as:

$$\sigma = \sigma_p^{|S|-1} \tag{4}$$

where σ_p is the pair-wise selectivity, and $|S|$ is number of sources participating in the query.

B. Results

During the evaluation, each source node sampled and sent a value every 60 seconds for the overall duration of 12000 cycles. As a result, every node produced 200 values which were sent to the processing node.

A comparison of an average number of messages sent in networks of various topologies and densities is shown in Figure 3a. Interestingly, processing data at the source outperformed processing at-the-base in all but the random medium sparse topology. Savings ranged from $9-37\%$ with an average of 23%. While these two techniques may appear similar, significant savings can be achieved when data are processed at a source. The saving is achieved because the processing source node does not have to send data to another node but only receives data and processes them locally with its own data stream. Additionally, Chatzimilioudis *et al.* (2013) showed that under certain circumstances the optimal Fermat-Weber node is often one of the source nodes [14].

Comparing the pair-wise join with processing at-the-base shows that savings up to 49% can be achieved, and on average, range between 28% and 38% depending on the selectivity of the join node. The lower the selectivity, the greater savings can be achieved. One of the reasons why pair-wise join can effectively reduce the network traffic is the fact it can exploit multicast trees, when a value from one source node is delivered to several processing join nodes. A multicast tree can perform this delivery with a very small overhead. Additionally, where the join node selectivity is low, it is most energy efficient to perform join at the sources. Thus, only one source node uses multicast tree to deliver its sensed value to every other source node, while other source nodes join the received value with the value sensed locally.

Using PND to find one central processing node at an optimal position (Fermat-Weber point) can further reduce the traffic on average by 10% when compared with Innet with 5% join node selectivity. Furthermore, if QTB optimisation is used, the network traffic is reduced even more leading to the overall average savings of 20% when compared to the best performing Innet algorithm.

Next we compare the delay of in-network data stream processing. We evaluate the duration of a period starting when the first source node in given epoch senses the data and ending when the processing node receives the data from the last source node within the same epoch. Shortening this delay is especially important in actuator networks where an action needs to be taken as soon as possible once a phenomenon is observed, e.g. a valve needs to be closed as soon as a leak is detected. From Figure 3b it can be seen that our platform decreases the network delay on average by $33-36\%$ depending on the algorithm it is compared to. As expected, the Query Tuple Buffering version increases the network delay on average by 7% when compared to the PND without QTB, yet still outperforms all other approaches. The increase is caused by increasing the acknowledgement timeout period in the forwarding algorithm. We leave the decision which version of the algorithm to use to the engineer,

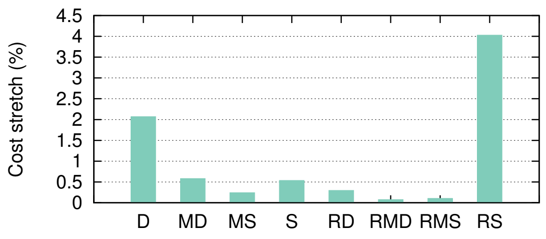


Figure 4: Percentage increase of cost of the processing node discovered by PND algorithm vs. the optimal processing node.

depending on what is more important for the implemented application: reducing the network traffic or decreasing the network delay.

Finally, we evaluate cost stretch, i.e. percentage increase in the cost of the discovered processing node versus the optimal processing node. For this comparison we used the *real cost*, which better reflects the real network traffic. In Figure 4 we can see the average cost stretch grouped by network topologies and density. As it can be seen, the average cost increase varies between 0.1% and 4%. The overall average cost increase is less than 1%.

V. Conclusion

In-network data processing has proven to be a very challenging problem in WSNs. Choosing the right strategy can significantly decrease the number of messages transmitted within the network, hence increasing its lifetime. Current approaches choose sub-optimal in-network processing strategies which lead to unnecessarily high traffic. Additionally, these approaches do not focus on reducing the processing delay, i.e. the time it takes from sensing the data to retrieving the result.

In this paper we presented a platform for discovering a processing node at, or near, the optimal (Fermat-Weber) node. The platform can find processing nodes whose cost is on average less than 1% higher than the cost of the optimal processing node. Processing data on the node discovered by the Processing Node Discovery algorithm leads to decrease in the network traffic by up to 38% while decreasing the processing delay by as much as 42%. We presented an optimisation of our algorithm based on buffering which can decrease the network traffic by another 11%, leading to the overall savings as high as 49%. However, the optimisation slightly increase the processing delay by 7% when compared to the non-optimised version.

In this work we assume the network is homogeneous from the computational point of view. In the future work we plan to investigate heterogeneous networks where only a subset of nodes is capable of processing data streams.

Furthermore, we will adapt our algorithm to ultra-low energy implementations, e.g. for MAC protocols where overhearing communication is not permissible.

Acknowledgement

We thank the authors of [11] for sharing source code to allow fair and thorough comparison, and acknowledge the support of EIT Digital (15171) and RCUK/EPSRC (EP/I038837/1).

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] IDC, "Worldwide internet of things (IoT) 2013-2020 forecast: Billions of things, trillions of dollars," *Doc 243661*, p. 22, October 2013.

[3] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap), rfc 7252," 2014.

[4] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *Internet of Things 2014 (IoT 2014)*, 2014.

[5] A. Uchechukwu, K. Li, and Y. Shen, "Energy consumption in cloud computing data centers," *International Journal of Cloud Computing and services science*, vol. 3, no. 3, 2014.

[6] M. Umer, E. Tanin, and L. Kulik, "Opportunistic sampling-based query processing in wireless sensor networks," *GeoInformatica*, vol. 17, no. 4, pp. 567–597, 2013.

[7] R. Kolcun and J. A. McCann, "Dragon: Data discovery and collection architecture for distributed IoT," in *Internet of Things 2014 (IoT 2014)*, Cambridge, USA, Oct 2014.

[8] V. Chowdhary and H. Gupta, "Communication-efficient implementation of join in sensor networks," in *In Proceedings of 10th International Conference on Database Systems for Advanced Applications*, 2005, pp. 447–460.

[9] A. Pandit and H. Gupta, "Communication-efficient implementation of range-joins in sensor networks," in *In Proceedings of 11th International Conference on Database Systems for Advanced Applications*, 2006, pp. 859–869.

[10] M. Stern, K. Böhm, and E. Buchmann, "Processing continuous join queries in sensor networks: A filtering approach," ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 267–278.

[11] 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, Sep. 2010.

[12] S. Mayer, D. Guinard, and V. Trifa, "Searching in a web-based infrastructure for smart things," in *Internet of Things (IOT), 2012 3rd International Conference on the*, Oct 2012, pp. 119–126.

[13] Z. Abrams and J. Liu, "Greedy is good: On service tree placement for in-network stream processing," 2006, p. 72.

[14] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, and N. Mamoulis, "A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement," *Journal of Computer and System Sciences*, vol. 79, no. 3, pp. 349 – 368, 2013.

[15] I. Galpin, C. Brenninkmeijer, A. Gray, F. Jabeen, A. Fernandes, and N. Paton, "Snee: a query processor for wireless sensor networks," *Distributed and Parallel Databases*, vol. 29, no. 1-2, pp. 31–85, 2011.

[16] 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.

[17] 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.